ADDRESSING PRODUCTION RUN FAILURES DYNAMICALLY

BY

JOSEPH A. TUCEK

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Doctoral Committee:

Associate Professor Darko Marinov, Chair
Professor Yuanyuan Zhou, UCSD, Director of Research
Professor William Sanders
Assistant Professor Samuel King
Associate Professor Dawn Song, UC Berkeley

Abstract

The high complexity of modern software, and our pervasive reliance on that software, has made the problems of software reliability increasingly important. Yet despite advances in software engineering practice, pre-release testing, and automated analysis, reports of high-profile production failures are still common. This dissertation proposes several run-time techniques to analyze and alleviate software failures dynamically, during production runs.

The first technique is low overhead checkpoint, rollback, and re-execution. By allowing a window of time in which a period of execution can be relived, low overhead checkpointing allows expensive analytical steps to be saved for only when they are needed. The second technique is a collection of dynamically insertable run-time analysis tools, which can use information gleaned over multiple analytical runs of the same execution to incrementally build picture of a production run failure more completely than any individual analysis could. Finally, based on my experience with the behavior of programs under failure, and the underlying causes of said failures, this dissertation introduces the concept of, and provides a run time which supports, delta execution. Delta execution (or Δ execution) is the process of running more than one instance or version of a program, while sharing the majority of issued instructions and state. This dissertation uses Δ execution specifically to validate software patches at production run time.

These three techniques have been demonstrated in three implemented systems supporting various end-level reliability goals. The first system, called Sweeper, is a run-time defensive system against security bugs. Low overhead checkpointing captures system state until an intrusion tripwire notices an anomaly. The system can then roll back to perform a more

thorough (and expensive) analysis of past execution to determine the nature of the exploit. Because of the low overhead of the initial checkpointing, the barriers to widespread deployment are low. Further, because Sweeper can still perform more complex analysis, there is the opportunity to generate strong protective measures, like vulnerability specific execution filters (or VSEFs), which can effectively stop a worm infestation. The implemented Sweeper system imposes only 1% overhead in ordinary operation, and can generate an effective protective measure in only 60 milliseconds. From an analytic model, this is sufficient to minimize the spread of a fast worm to only 5% of the susceptible hosts, even for a worm which spreads 10,000 times faster than any previously observed in the wild.

The second system is called Triage. Rather than improving reliability by improving security, Triage attempts to enable the improvement of the underlying code by automating failure diagnosis of production run systems. Production run failures are difficult to address. Such failures commonly are irreproducible in a development environment due to workload or scale issues. As they occured in a production run, these are clearly faults which were not caught and fixed by the developer's standard pre-release testing. Finally, production runs have stringent restrictions on overhead and privacy. Hence giving the programmer enough insight into the failure to implement a patch is challenging. Triage addresses this by performing failure diagnosis post-hoc at the end-user's site. Low overhead checkpointing allows the capture of a failing execution, so expensive analysis can be deferred until it is definitely needed. Repeated replays allows the incremental application of a variety of failure analysis techniques, similar to the process a human programmer may undertake. For analysis which generally takes direction from a human, Triage substitutes the results of previous analytical steps. Overall, Triage imposes only 5% overhead in failure free execution, and, if a failure occurs, all of the analysis which requires re-execution is complete within about 5 minutes. In a study with human programmers, the output of Triage analysis reduced the time to patch real software faults by 45%.

The third system presented in this dissertation deals with the problems introduced when

programmers make changes. Despite testing before release, a large number of software patches are released buggy. Indeed, software patches are generally of such poor quality that to optimize uptime it is better to delay applying even security patches while others act as "volunteer" beta testers uncovering the faults which made it though the vendor's quality control. However, as Triage's novel delta analysis diagnostic tool shows, the difference between correct and buggy execution can be minimal. Indeed, a manual study of software patches described in this dissertation shows that many patches should not create large changes in the underlying execution. Hence this dissertation proposes Δ execution. If the execution (in terms of instruction streams and data) of the patched and unpatched versions of a program are mostly identical, then it is possible to run both versions mostly in one instruction stream. Only rarely, when the executions do differ, is it necessary to run two sets of instructions. By only running the differing, or delta, segments separately, Δ execution allows low overhead production run patch validation which is 12% faster than side-by-side patch validation. Further (and perhaps more important), many of the effects which make patch validation difficult (multithreading, timing sensitivity, and system level nondeterminism) are nullified as they effect the two logical executions inside the one physical execution identically. This dissertation shows that, of ten applications tested, Δ execution can validate all of the patches, while traditional side-by-side validation only manages to validate 2.

To Noi, and also Tutu.

Acknowledgments

There are many co-authors, collaborators, family, friends, groupmates, and/or mentors who deserve some amount of blame and/or credit for their guidance, help, inspiration, patience, and/or tolerance. In alphabetical order, they are Anna, Bill, Bill, Brad, Chengdu, Craig, Darko, Dave, David, Dawn, Deepak, Eric, Feng, Fred, Jagadeesan, James, Jason, Jay, Jim, Katie, Marcelo, Mark, Mehul, Nitish, Patrick, Pin, Rob, Ryan, Sam, Sandeep, Shan, Spiros, Steve, Sylvia, Weiwei, Will, William, and YY.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Modern software is plagued by failures. The growing size and complexity of modern software systems has increased the difficulty of finding and fixing bugs, with dozens to hundreds of programmers frequently devoted to quality control. It is inevitable that production software will contain a significant number of bugs. These bugs contribute to 26-30% of system failures [80], and cause costly downtime. Worse, many of these bugs are security vulnerabilities. Worms such as Blaster [20] and SQL Slammer [22] take advantage of such vulnerabilities to rapidly do billions of dollars of damage [69].

Clearly it is important to react to these bugs as quickly as possible. As they represent cases which are causing actual (rather than theoretical) harm, production run failures should be our highest priority. In the case of security flaws, even waiting for an administrator to manually patch the system is too slow, since by the time an alert goes out, it is already too late [130]. For both normal software failures and for security vulnerabilities, automating as much of the response as possible would be ideal. If we could automate the initial diagnostic steps, programmers could more quickly generate patches; if we could automate the security response processes we could stand a chance against fast worms [129]. Unfortunately, current techniques for dealing with software failures are incapable of addressing the problem of production run failures.

Many failure diagnosis techniques do not provide enough information for automated use. Some are simply too manual for automation to be applicable. While interactive debug-

gers [53] or even reverse debuggers [128] are clearly powerful, they require a human programmer to actually perform diagnosis. Others, such as bug detectors, are not as manual, but still are insufficient. Although bug detectors such as buffer overflow detectors [30, 94] or race detectors [95] are more automatic, they do not isolate root causes. Instead, they notify programmers of *errors*, or incorrect program states. It is up to the programmer to identify the underlying root cause. For security vulnerabilities, tools such as StackGuard [34] or ProPolice [48] are incapable of detecting many classes of attacks [148]. Address space randomization [102, 50], can detect (and temporarily prevent) many attacks, but doesn't give much information about the attack. At best execution is halted at the vulnerable instruction, and at worst the attack succeeds anyway. None of these techniques give enough information for an automatic response.

There are techniques which can give sufficient information for a response, and which do not rely on human guidance, such as program slicing [2, 147, 156], path reconstruction as in PSE [79], or taint analysis [99]. However, these techniques tend to impose crippling overheads in the range of 10 to 1000x. Clearly 1000x overhead for dynamic slicing is impractical for production run deployment. Even the more limited case of dealing with security flaws suffers from this problem. Tools like DIRA [125], DACODA [35], or Vigilante [33] are too expensive (30-40x slowdowns) for widespread use, and instead must rely on a canary-like deployment. A small number of sentinel machines provide very thorough analysis; the bulk of hosts, however, are unmonitored and open to attack. For both normal failures and for security vulnerabilities buggy executions cannot be adequately analyzed while maintaining acceptable performance.

A further problem is how to verify the correctness of any proposed response. Released patches are often buggy [7, 90]. Pre-release testing is useful, but it is a poor substitute for live workloads. Further, extensive periods of pre-release testing can lengthen the window of vulnerability for security related bugs. Yet it is because patches are buggy that administrators are wary of immediately deploying patches, even delaying the application of critical security

patches to allow for more testing [7]. While there is work in patch validation, the current efforts have shortcomings. Off-line validation, where a test workload is played against a test system and compared against a known correct result, may not exhibit all of the characteristics of the production workload. On-line validation, which compares the test system against the production system while running the same workload, is too resource intensive; it may either interfere with the production instance, or require additional resources beyond what is reasonable. Both types also incur additional and costly administrator labor, and, as pointed out in [88], non-determinism can make it very difficult to actually compare the output of the new code to the old code without excessive false positives. Hence we lack an effective mechanism to properly vet patches for correctness.

1.2 Contributions

This dissertation works towards developing run-time techniques to address software failures during production runs.

- **Use system support for checkpoint and re-executing to capture the failures.** Checkpointing has often been used for failure recovery [108]; I propose using such techniques for capturing production run failures. Because the propagation chain from trigger to failure is typically short in software failures, the amount of state which must be captured is small, and generally does not need to be captured to persistent storage. Hence the run-time overhead can be especially low, allowing production deployment. Further, by using re-execution rather than deterministic replay, we can relive the failure as if it had been instrumented for analysis the first time around. Finally, the flexibility for re-execution allows the use of "what-if" scenarios; this results in further insights into the failure, as well as allowing post-failure recovery.

- **Adapt and systematically apply previously proposed analysis tools for post-hoc analysis of failures.** Since we can relive the failure as needed, it is not necessary

to use expensive analysis during normal runs. *After* a failure has occurred, we can focus any analysis technique we desire on that segment of execution which contains the failure. Even the most expensive analysis techniques are feasible, because they only need be applied for brief time periods. I propose diagnosis protocols in order to select which analysis is appropriate at which time. For security vulnerabilities, this protocol focuses on generating results quickly, in order to beat fast worms. For normal failures, our protocol traces back from the failure to find the entire fault propagation chain, and suggest the actual fault itself.

- **Allow new analysis techniques and new responses.** Due to the flexibility of our re-execution, I propose *delta analysis*, a new analysis technique. Delta analysis involves generating many possible runs of the program, some failing and some not, using flexible re-execution. Subsequently diffing the most similar failing/non-failing pair highlights those aspects of the execution which separate success from failure. This dissertation also proposes a new vulnerability-specific execution filter (VSEF [96]) to respond to security vulnerabilities. By detecting the specific buffer overflow responsible for the vulnerability, all future exploit attempts can be easily protected against, without imposing the overhead of bounds checking throughout the entire program. Further, because the system has checkpoints available, it can use hits against the VSEF filter to trigger recovery; refining vulnerability signatures based on the input which triggered the VSEF and then re-executing catches all exploit attempts without ceasing execution.

- **Allow low-overhead comparison of patched and unpatched production runs.** From comparison of various failing and non-failing runs, it becomes apparent that the differences in various executions are surprisingly small. Due to this observation, I propose that it is possible to run both the original and the patched version of a program *within one execution*. That is, for the majority of the execution, it is possible

to run both old and new program version inside of one execution stream, and to run two execution streams only for those portions of the execution which differ. This *delta execution* allows users to validate the correctness of a patch on their own workloads, without the full expense of running their applications twice. Additionally, many of the difficulties in performing a live validation of two versions of a system come from small non-determinisms in the execution. Even two identical program versions run side by side may exhibit different output due to differences in thread interleaving, message order, timing, and random number generation. However, because both the original and modified program are running in the same physical execution stream most of the time, such non-determinism effects both logical streams identically. This greatly reduces the spurious differences that can frustrate on-line validation.

1.3 Summary of Results

These techniques are, as shown in this dissertation, quite effective. They demonstrate feasible overheads and good functional results. In more concrete terms, this dissertation presents measurements of the ordinary-case runtime overhead, the time to generate a result, and the end functional result.

For both the automatic debugging and security use-cases, the overhead in the common case is the continuous checkpointing. This overhead depends on the application and the checkpoint interval. For a 200 ms checkpoint interval, the overhead can range from 1 to 5% (see Chapters 3 and 4). This is much lower than the overhead of the software analysis tools used after an issue is detected, which emphasizes the advantages of using checkpointing to defer analysis. In the patch validation use case, delta execution (Chapter 5) imposes 12% less overhead than side-by-side validation, despite the overhead imposed by Pin [77].

Both the debugging and security use-cases generate a result. For security purposes, the system should generate the initial result as soon as possible; as described in more detail in

Section 3.5.3, this can be as quick as 40-60 milliseconds. This low turnaround time minimizes the amount of time a worm has to get ahead of an antibody. For automatic debugging, the on-line portion of analysis is the important limiter; this can be completed (see 4.9.2) in about 5 minutes.

Finally, even if the overhead is trivial, expending that overhead is only worthwhile something is gained in return. First, for security, we care about what the extent of infection is for a fast-spreading worm. Sweeper, the system presented in Chapter 3, limits the extent of infection to only 5% for susceptible hosts, for a worm with a contact rate (β) of 1000. For comparison, the fastest worm to date, Slammer [22] only had a β of 0.1. Second, the automatic debugging system in Chapter 4 (Triage) can be measured by improvement in the time to fix a bug. The results of a human study (see 4.9.4) show a 44.7% reduction in the time to generate a fix. Finally, for the patch validation system in Chapter 5 (Δ execution), we care about the ability to validate fixes. Delta execution can validate all ten of the attempted patches, while traditional side-by-side validation only managed to validate two.

1.4 Outline

The remainder of this dissertation is organized as follows. Chapter 2 covers background material and related work. Chapter 3 discusses using run-time analysis to prevent fast worms with Sweeper. Chapter 4 presents techniques for automatically diagnosis of production run failures, and an implementation of those techniques embodied in the Triage system. Chapter 5 presents Δ execution, a method for on-line patch validation. Chapter 6 briefly presents potential future work, and Chapter 7 concludes the dissertation.

Much of the material in this dissertation has been published in journals, conference proceedings, and workshops. The checkpointing system has been presented previously in [108, 109, 110]. The security aspects of this work have been presented in [137]. [135, 136] present the failure diagnosis work, and [159, 138] presents the patch validation work.

Chapter 2

Background

2.1 Checkpointing for Software Reliability

Checkpoints have long been used to increase software reliability [12]. In [54], 3 of the 5 schemes described to survive software failures are some variety of checkpointing. The principle behind these checkpoints is the assumption that most failures are "Heisenbugs", which depend on specific conditions in order to be triggered. If the checkpoint system doesn't capture those failure triggering conditions, they may not be present on replay. Hence, the natural variations in execution when replaying from a checkpoint will prevent the fault from manifesting. In general, these checkpoint systems assume that the entire system is vulnerable to failure; hence they store their checkpoints to disk or to a separate system. This greatly increases the overhead involved. Further, a study of software faults by Chandra and Chen [24] shows that "only 4-14% of the faults were triggered by transient conditions ... that naturally fix themselves during recovery", implying that Heisenbugs are not as common as once thought.

Another completely different use of checkpoints in software reliability is time traveling or reverse debugging [49, 114, 128, 64]. Rather than trying to make the failure go away, in reverse debugging the goal is to replicate a failure so that a programmer can inspect it. To do this, such systems attempt to capture as much of the relevant environment as possible, including memory access interleavings [114], and the complete results of system calls [128]. TTVM [64] performs checkpoints at the virtual machine level, allowing even kernel bugs to be reliably duplicated and inspected. With varying levels of overhead, such systems do

allow programmers to easily step backward through a software failure, greatly easing the debugging process compared to "cyclic debugging" [64]. However, the diagnosis itself is still performed by the programmer manually. If a failure is difficult to trigger in the first place (one of the key motivations for such debuggers), the failure must be triggered at least once in front of the programmer. This does not address the problem of production run failures.

All of the previously discussed checkpoint systems attempt a faithful reproduction of the execution. Even for failure recovery, that the re-execution is successful is merely accidental. In contrast, the Rx system [108] purposefully disturbs the environment during re-execution in the hopes of preventing the failure from reoccurring. While Chandra and Chen find that only a small proportion of faults are truly Heisenbugs, they do find that up to 56% of failures depend on some system or environmental condition [24]. By purposefully modifying the environment (for a wide definition of environment) during re-execution, Rx can avoid otherwise deterministic bugs during re-execution. Unlike the deterministic (or mostly deterministic) re-execution and replay used by previous systems, Rx uses a *semi-deterministic* re-execution. This re-execution is capable of reproducing a failure, and is also capable of producing an execution that could have happened, if some environmental condition had been different.

A common criticism of using continuous checkpoints is the perceived expense. While older checkpoint systems did impose non-trivial expense, more recent checkpoint systems have demonstrated very cheap checkpointing. Rx itself imposes under 5% overhead for a 200 millisecond checkpoint interval [108]. Flashback (which Rx was derived from) imposes under 10% overhead for a web-server workload [128]. TTVM, which checkpoints an entire virtual machine, imposes 16-33% for a 10 second checkpoint interval [64]. Recently, the DejaView system imposes approximately 5% overhead for "execution capture" checkpoints [66]. Overall, using copy-on-write, minimizing logging, and avoiding disk allow frequent, cheap checkpoints.

This work builds on the Rx checkpoint/re-execution system. Specifically, the Rx system

has two properties which make it especially suited for the purposes of this dissertation. First, it has especially low overhead. Most of the interesting things that happen between activating a fault and a failure occuring happen within a small time window immediately before the failure. The fact that Rx does not attempt to maintain checkpoints for a long period (checkpointing only to memory and then rapidly discarding them) is consequently of little importance. Second, Rx is not restricted to exactly replicating a previous execution, as some other checkpointing systems are. Rx explicitly attempted to force execution to happen differently. For the systems presented in this dissertation, what is desired is mostly to replicate the failure, but with analysis tools (e.g. bug detectors) which will perturb the reexecution from a purely faithful replay. Between these two properites, Rx-style checkpointing is an especially good match for addressing production-run failures.

2.2 Bug Detection & Diagnosis

There is plentiful work in techniques to automatically detect bugs, and to help diagnose them. Especially for common bugs, such as memory bugs or data races, there is no shortage of tools to detect them. Most of these techniques suffer from unacceptably high overhead, and cannot be used in production runs. This section gives a brief overview of various bug detection and diagnosis techniques.

2.2.1 Memory Bug Detection

Unlike type safe languages such as Java, programming languages such as C and C++ are vulnerable to what are called memory bugs; these are bugs which result from the improper or incorrect use of pointers. Memory bugs are an important class of bugs, as they not only contribute to a large portion of failures overall, but cause half of the most severe security vulnerabilities [23].

One way to prevent such faults is to retrofit type-safety onto C and C++. Jones &

Kelley [60], CRED [115], and CCured [92, 30] all do this. The JK checker assumes that all pointers point to correct objects; they then enforce that pointer arithmetic never creates an incorrect pointer. However, this cannot deal with out of bounds pointers which will later be manipulated to be valid. Further, it can impose overheads of up to 12x [40]. The other techniques explicitly track what the original object was. They all modify the compiler so that the compiled code uses some variant of "fat" [1] pointers. A fat pointer is a pointer which has been augmented with information about what object it is pointing to. Pointer assignment updates which object the pointer is referencing; other pointer arithmetic merely changes the offset within that object. Since we know the sizes of all objects, we can easily tell if a fat pointer is dereferenced while it points outside of the bounds of the object it should point at. C and C++, however, are decidedly not type safe. Correctly inferring all pointer arithmetic is difficult. Furthermore, fat pointers involve non-trivial overhead in both time and space, and the runtime bounds checks are expensive as well. The original CCured [92] can impose up to 150% overhead. In later work, CCured [30] attempted to prove that many types are only used in a type-safe way, and hence do not require any dynamic checks. Depending on the extent to which type safety can be statically proven, the overheads range from nearly zero to 87%. This can still be unacceptably expensive. Further, these techniques all require that the programs be compiled with the special CCured compiler.

More recently, advances in compile-time analysis and compiler-provided runtimes (e.g. as in LLVM [67]) have brought down the costs of retrofitting type-safety, both in terms of runtime overhead and code changes. In [40], Dhurjati and Adve show extreme reductions in the overhead necessary for JK-style bounds checking. Specifically, they show that by using automatic pool allocation, which isolates different types into separate heap areas automatically, they are able to much more efficiently look up the bounds of a pointer. They further combine this with static analysis to eliminate runtime checks where possible, and with compiler optimizations tuned at reducing the overhead of the runtime checks that remain.

[1] Alternatively known as "chubby" pointers".

The result is an overhead of 12% overall across a selection of benchmarks, with a peak of 69% in one case. Further, it worked with unmodified C code, rather than a restricted subset of C. In slight contrast, this work was later used by the Secure Virtual Architecture [37], or SVA, to provide run-time memory safety guarantees to the Linux kernel. SVA did require changes to the kernel code, primarily for two reasons. First and foremost, operating system kernels are not generally written in pure C, but include substantial assembly code. SVA required this to be redone using provided primitives in a virtual architecture. Second, OS kernels are notoriously type unsafe, freely and frequently casting between dissimilar types[2]. Further, SVA showed much higher overhead than [40], with typical values of 50% and ranging up to 4x for kernel-heavy workloads. This implies that type-safe C/C++ is close to feasible for general case use, but not quite.

It is also possible to use instrumentation to detect memory bugs. This can potentially be the only choice if the program's source code is unavaiable, or if it is impractical to recompile using a special compiler. Purify [56] and Valgrind [94] are to popular tools for performing memory safety checking. As Annelid shows, fat pointers can be used when using instrumentation [93], but in general instrumentation-based memory bug detectors use redzones. Redzoning involves monitoring for accesses to memory locations which should not be touched. This is done by adding padding before and after each buffer. Any access to these redzones is invalid, and can be flagged as a bug. Although this will not catch truly wild pointers, it will detect most buffer overflows. While these techniques can apply to already compiled programs, the flexibility comes at a cost. Valgrind imposes approximately 22x overhead, while Purify imposes 5.5x. This is clearly much too expensive for production run use.

A more limited case of memory bugs is stack smashing. Stack smashing occurs when a buffer overflow overwrites the return address on the stack. This will generally allow

[2]There is a joke that kernel programmers only belive in three types: integers, bytes, and arrays of bytes. The authors of [37] make a note of the prediliction of Linux kernel code to declare a variable as an integer when it is used nearly exclusively as a pointer.

an execution hijacking attack. StackGuard [34] and ProPolice [48] are two tools which defend against such stack smashing attacks. By placing canary values around the return address on the stack, both StackGuard and ProPolice can detect stack smashing attacks. A canary is a value put before and/or after some important variable. Since most overflows will overwrite in sequence, any buffer overflow which modifies the important variable will also modify the canary. By verifying that the canary values haven't changed prior to accessing the important variable, writes to the important can be detected. Both StackGuard and ProPolice make minor, non-intrusive modifications to the compiler; gcc currently includes a version of ProPolice. While the overheads are reasonable, they are limited to detecting stack smashing attacks on the return address; further, they give minimal information about how the return address was corrupted, only that it was.

Another alternative is to use hardware to support memory bug detection. The Intel iAPX 432 [100] did not support directly accessing memory by a raw address, but instead required the use of a segment and an offset. The segments were bounds checked, and also imposed a hardware enforced type mechanism; a segment could either point to data or segment descriptors, preventing arbitrary data from being treated as a pointer. Mondrian memory [149] allows read/write/execute permissions to be specified on arbitrarily small segments of memory; small non-accessible segments before and after each buffer can easily act as hardware enforced redzones. Similarly, iWatcher [158] allows inexpensive notification when specified address are accessed, and SafeMem provides similar notification at the granularity of a cache line [107]. While quite efficient, hardware based bug detection is only possible on special hardware. For programs running on commonly available hardware, it is not an option.

2.2.2 Data Race Detection

A data race occurs when two different threads access the same memory location without any synchronization between them. Data races are the classic example of a hard to debug Heisenbug; they are highly dependent on the precise interleaving of memory accesses, and are

hence difficult to reproduce. There are two dominant techniques for detecting data races: the happens before algorithm [41, 95, 105] and the lockset algorithm [27, 117, 118]. The happens before algorithm produces a partial order among memory access between threads. Within a thread, each access occurs after the ones prior to it. Between threads, access to a synchronization primitive is necessary to order accesses. For example, if several threads reach the same barrier, then everything that happens after the barrier in each thread is ordered after all accesses from before the barrier in the other threads. Similarly, everything which happens after a lock acquire also happens after the last release of the same lock. If there are two accesses to the same memory location which do not have a well defined order, then a data race is reported.

The lockset algorithm instead looks at locks. The set of locks each thread holds is noted for every shared variable access. If two threads ever access a shared variable without holding a consistent set of locks, then it is assumed that a data race between those accesses is possible, and a race is reported. Put another way, when accessing a particular shared variable, if the intersection between set of locks held by thread A and the set of locks help by thread B is empty, then the locks did not well protect the variable, and a race could be possible. This has the advantage that the race need not actually occur during a run to be detected.

Both the happens before and lockset algorithms require many checks for shared memory accesses. Therefore, they both impose high runtime overheads; the Valgrind implementation of lockset imposes 694x overhead [76]. Further, both of them require knowledge of the synchronization primitives used. If a programmer uses their own primitive (e.g. a while flag), both may report false positives. Also, it may be impossible for a race to actually occur for control-flow reasons, causing even more false positives.

A new approach is to look for invariants in memory accesses. Both SVD [153] and Avio [76] do this. The programmer assumes that certain pairs of accesses will happen atomically; bugs occur when this invariant is not enforced by the program and the assumption is violated. SVD extracts invariants from static analysis, while Avio uses training runs.

As runtime techniques, both impose large overhead (65x for SVD, 25x for the software implementation of Avio). Furthermore, they require the exact data race to actually occur.

2.2.3 Crash Reports

One of the earliest forms of debugging information was the core dump: a snapshot of the memory state of the computer at the time a failure occured. A somewhat more modern variant of this is the crash report. Crash reports are data (e.g. core dumps, but also stack traces, symbol tables, log files, hardware information, etc.) about the state of a computer at the time of a failure which are automatically collected at a remote site and sent back to the programmer. There are many crash-reporting frameworks, such as the Mozilla Quality Feedback Agent [86], Dr. Watson [43], and Microsoft's Windows Error Reporting [52]. Typically they do not send full core dumps, but stack information, version information, PC values at the time of the fault, and occassionaly program specific information (e.g. a web-browser crash-report tool may send the URL that caused the crash). The programmers can use the crash reports to try to identify root causes, but also to prioritize fixing the most common bugs.

Microsoft's Windows Error Reporting, or WER, is by far the most common crash-reporting framework (with an installed base of one billion clients), and the most well-described in the literature. As described in Glerum et al's paper on WER [52], when a failure occurs, the user is prompted to submit error information back to the programmer. If the user is asked every time, 40 to 50% of failures are reported; changing the prompt to allow one time opt-in and simplifying the wording of the prompt increased the submission rate to 70 to 80% in Windows Vista. These submission rates are surprisingly high, given that there is some risk of sensitive or personally identifiable information being included in the crash report.

One of the most important uses of WER is prioritizing which bugs to fix. Hence, the WER backend has extensive support for heuristically assigning crash reports to "buckets". The

ability of the backend server to automiatically catagorize the bug also allows programmers tracking down a specific bug to request additional information. For instance, they can request a full memory dump, or the contents of a specific file. Further, they can request that the specific client turn on further debugging support to get more details if the same failiure occurs in the future.

Unfortunately, while crash reports are extremely useful for prioritizing bugfixing work, they still only provide information about a failure after it has occured. They give information about the state at the time of failure, but don't give any explicit information about the fault propagation chain. There are some tools which can extract more useful infomation from a core dump. For example, a tool such as PSE [79] can infer possible execution paths from static analysis and symbolic exection. However, without runtime information, the task of tracing from the failure state to the root cause is more difficult.

2.2.4 Other Techniques

Invariant Based Bug Detection

The bug detectors described previously target generic behaviors; for example, for most programs it is probably incorrect to overwrite the return address. These generic invariants are easy to develop detectors for. However, many buggy behaviors are specific to a particular program, and cannot be captured by a generic detector. Instead, invariants must be developed automatically. Daikon [47] takes as its input many training runs; based on variable values during these runs it guesses as to what may be invariants for the values of those variables. For instance, if a particular variable only every has values between 1 and 10 during training, Daikon may decide that this bounds range is an invariant, and that violations of the invariant indicate a bug. Similarly, DIDUCE [55] examines a program as it runs, and hypothesizes as to invariants. As they are violated, it reports the violations and refines its hypothetical model. AccMon [157] proposes program counter based invariants. By analogy,

a credit card company monitors where a credit card is used; if it is used in a new place, they may flag that transaction as suspicious. Similarly, particular memory locations are typically only accessed by a small subset of PC values. If a new PC value accesses the memory location, it is suspicious, and can be reported as a bug.

Slicing

[147] notes that "Programmers use slices when debugging". A slice is the portion of a program which propagates values to (or from) a particular statement. Slices show how statements influenced one another, and leave out statements which don't matter (relative to the statement of interest). Slices can be either forward or backward, and either static or dynamic. A forward slice from statement s consists of all of the statements *after* s which have a data or control dependency on s. Similarly, a backward slice from s is all of the statements *before* s on which s has a data or control dependency. A static slice is one computed from the source code, while a dynamic slice is one computed from an actual execution trace.

Although slices are quite useful in debugging, they are inconvenient to generate. Static slices suffer from imprecision; without specific information from a particular run, poor alias analysis and ambiguity as to control flow causes the slices to be overly large. Dynamic slices are small, but are expensive to compute [156]. Even though they are imprecise, static slices have seen more use simply because they are more practical to compute [127].

Delta Debugging

Delta debugging [155, 84] examines how changes in input can turn a failing execution into a success. Specifically, delta debugging repeatedly mutates an input known to cause a failure for two purposes. The first is to find a minimally sized failure-triggering input. By eliminating those portions of the input which are extraneous to the failure, delta debugging makes determining what causes the failure easier. Second, delta debugging searches for a maximally similar input which succeeds. In the best cases, delta debugging can find

a successful/failing pair separated by as little as one character. Seeing this difference is helpful to the programmer in determining why the program failed.

Model Checking

Model checking seeks to prove that a program will exhibit or not exhibit a particular behavior, e.g. to prove that there will be no null pointer exceptions. One method of model checking is exhaustive state space exploration, in which the model checking software explores many many inputs to check if the desired properties hold in all reachable program states. As I mention in previous work [159], delta execution (presented in Chapter 5) can be used to reduce the effort of such model checking, by attempting multiple inputs simultaneously. This idea is developed in [38]. Modifications to the Java Path Finder model checker allow thousands of executions to run while sharing large amounts of state and execution. Although the modifications to JPF impose a large amount of overhead, in the tested instances there are many thousands of simultaneous executions, and hence [38] can model check more quickly than the base model checker.

2.3 Worm Response

Ever since the Morris worm [46] struck the internet in 1988, the threat of automatically spreading worms has existed. In more recent years, worms such as Blaster [20], Code Red [21], and SQL Slammer [22] have demonstrated that this threat is not academic. Although these worms did not have particularly malicious payloads, they still caused billions of dollars in damage [69]. Distressingly, the Witty worm [121] incorporated a purposely damaging payload: in between attempts to spread it would randomly overwrite disk blocks, until the infected host was too damaged to continue. Witty targeted a relatively obscure piece of software; a worm with a similarly malicious payload which targeted a commonly deployed core service (such as a popular web server) could be devastating. Further, these worms are

much faster than we can rely on a manual response. Slammer, currently the fastest spreading worm known, infected 90% of susceptible hosts within 10 minutes minutes [85]. Recent academic work [129, 130] has shown the possibility of far more efficient worms. By avoiding scanning, building hit lists, and carefully ordering which hosts will infect which others and in what order, it is possible to build worms of frightening speed; 95% infection rates are possible in 510 ms for UDP and 1.3 seconds for TCP [129].

In order to protect against network-based exploits (automatic worms or otherwise), a significant amount of research effort has gone into developing exploit signatures. Such signatures provide a template against which incoming messages can be checked; if there is a match, the input is considered malicious and can be dropped. Work such as Earlybird [124], Honeycomb [65], and Autograph [63] all provide signatures for use as input filters. Unfortunately, these filters are contiguous strings. In order to bypass them, a worm author merely needs vary the specific message string they send to trigger the exploit. Such polymorphic and metamorphic attacks are much more difficult to detect. Although creating polymorphic signatures (e.g. Polygraph [98]) is possible, recent work [98, 104] shows that misleading such generators is possible. The result is a signature with much higher false negative and false positive rates. By fooling the filter into dropping perfectly legitimate traffic, malicious training makes such polymorphic signatures much less useful; conversely a high false negative rate allows evil traffic to pass by unmolested. An alternative to training based on detecting exploit attempts is to derive a signature from the vulnerability itself; Shield [142], Vigilante [33], Bouncer [32], DACODA [35], and Brumley et. al. [18, 19] all do this. Although such techniques can prove a zero false positive rate, they suffer from false negatives. In general, these techniques will generate signatures with a sensitivity to the execution path taken during exploitation. Bouncer [32] and [19] are much less sensitive to path, but still have false negatives. Extracting a "perfect" signature from such analysis is equivalent to the halting problem, and so is impossible in the general case.

A completely different approach is to target the vulnerability directly. Vulnerability spe-

cific execution filters, or VSEFs [96], define a detector based on the execution of the software itself. By specifying the path by which an observed exploit propagates to incorrect behavior (e.g. execution of attacker provided data), one knows exactly those instructions which need to be monitored in order to detect future exploit attempts. This specific technique, however, is sensitive to variation in the execution path, and so may be defeated by metamorphism which varies the vulnerable software's execution path. Also, detection occurs after the program has consumed the malicious input; the state of the program can therefore not be trusted, and the only recourse is to restart the service.

A further problem with both signatures and VSEFs is that they have to be derived and distributed. While this is feasible for known exploits and known vulnerabilities, zero-day attacks (which focus on previously unknown security vulnerabilities) are much harder to respond to. Given the potential speed of worms, it is clear that we must respond automatically. Vigilante [33] is such an automatic defense. In Vigilante, a subset of nodes will monitor their execution (or the execution a subset of requests). If an exploit attempt is detected, Vigilante will generate a self certifying alert (SCA). The SCA includes sufficient information to generate a filter, and to verify (certify) that the vulnerability is real. This SCA can then be shared with all other potentially vulnerable hosts. This sort of reactive antibody system is effective against such worms as Slammer. However, there are weaknesses through which a worm author could defeat Vigilante. First, since monitoring is only performed on a subset of hosts (or requests), if the worm can avoid such sentinel hosts it can avoid detection. Second, and most important, any method one can use to quickly spread an antibody, a worm author can use to spread a worm. Vigilante relies on the ability to notify hosts far faster than the worm can spread; a top-speed worm is far too fast, and gets a head start [129].

2.4 Patches

The long term solution to a security vulnerability, and the end-goal of bug analysis techniques, is a patch for the underlying fault. In the case of a security vulnerability, patching quickly is required to minimize the window of vulnerability [15, 61]. Unfortunately, reports of buggy patches are all too common [89, 90, 68]. System administrators loath to be the first ones to apply a new patch, for fear that the cure will be worse than the disease. Their fears are not unfounded; [8] shows that even for security patches, administrators are better off waiting. Even a month after release, 6% of patches are still buggy. Due to poor patch quality, vendors invest much effort into testing, which delays the availability of security fixes [5].

2.4.1 Patch Validation

To protect against bugs in patches, a patch can be validated against the behavior of the unpatched software. Most patch validation efforts focus on offline regression testing [62]. [42] discusses various regression testing techniques. Selecting which test cases to run, how to generate test cases, and how to test the integration of the entire system are all key issues in regression testing. However, there is no substitute for testing on production runs [88]. On-line workloads exercise the entire system, can cover extreme or unusual conditions, and are more useful for the individual administrator. What order and which test cases a vendor runs is not of interest to a system administrator deciding whether or not to apply a patch; what matters is merely "will it break *my* workload?"

Because of this, on-line testing is greatly preferred [31, 88]. In general, this requires setting up a separate test machine, which doubles the hardware costs and requires extra administration. This can be difficult; further, the upgrade needs to be done once for the testing and again for the actual production run, doubling the chances of an operator error. [75] considers using a "devirtualizable virtual machine" to ease maintenance while only using a single machine. Once validation is complete, the workload is switched to the "testing"

machine and the other VM is destroyed. Unfortunately, this still results in one half of machine resources to be dedicated to the patched version. Deciding whether a patch is good can also be problematic; predicates can be used to validate correct behavior [61].

2.4.2 Dynamic Software Update

In dynamic software update (e.g. [59, 78]), a program may be patched while it is still running. Of particular interest are procedure-based dynamic update systems, where individual procedures within a program may be updated. The PODUS system is an example [119]. Although "any program can be so poorly written that it cannot be dynamically updated", most well-structured systems are suitable. During the update, two different versions will be resident at once. Typically the old version will be resident as long as there is a stack frame which points to within the old code.

Another technique similar to dynamic software update is band-aid patching [122]. Band-aid patching will run the old and new versions of patched code in sequence. It then must immediately determine which version to use. The unused instance is then squashed; unfortunately this prevents dealing with faults with even short latent periods. Band-aid patching only handles cases where the change is isolated to within one function.

Chapter 3

Sweeper

3.1 Introduction

Security vulnerabilities are one particularly critical class of software bug. This chapter considers a technique to address such vulnerabilities, specifically those exploited by fast self-spreading worms, and describes an implementation of the technique in a system called Sweeper[1]

Self-propagating worms are malicious programs which use software vulnerabilities to spread from computer to computer. Even if the author of the worm is trying to avoid negative consequences, they can cause much damage by overloading infected machines and requiring administrator effort to clean up [46]. An attacker who is out for maximum effect can do up to 50 to 100 billion dollars worth of damage in one incident [146]. Worms which have been seen in the wild, like Blaster [20], Code Red [21], and SQL Slammer [22] have not approached this upper bound, but have still been quite expensive. SQL Slammer alone cost 1.2 billion dollars, while Code Red cost 2.6 billion dollars [69].

Further, these worms can do their damage in very little time: only 10 minutes from starting SQL slammer had reached a 90% infection rate [85]. Again, a potential for far worse exists; [129] estimates a 95% infection rate in only 1.3 seconds for a worm exploiting a TCP-based vulnerability, and only 510 milliseconds for a UDP vulnerability. Even at much slower rates, a manual response is inadequate. If a patch were made available the very

[1]This work is based on an earlier work: Sweeper: a lightweight end-to-end system for defending against fast worms, in ACM SIGOPS Operating Systems Review - EuroSys'07 Conference Proceedings, Volume 41, Issue 3, June 2007 (c) ACM, 2007. http://doi.acm.org/10.1145/1272998.1273010

moment the infection started, by the time an administrator found out about it the systems they mange would likely have already been infected [130].

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 detects the attack. It then analyzes the attack attempt to find the underlying vulnerability. Without 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 traces of the worm's existence are log messages and new antibodies.

Essentially, what is needed 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 damage and future attacks exploiting the same vulnerability.

- *Low overhead for universal deployment*: The defense system has to have low overhead to enable practical production system deployment, especially in server scenarios where performance is important.

- *Efficient recovery*: It is also highly desirable for the defense system to recover from an attack as efficiently as possible to provide non-stop service, especially for applications that demand high availability.

As discussed in Chapter 2, existing defenses are insufficient. For instance, some existing solutions such as PaX [102], StackGuard [34], LibSafe [134], and ProPolice [48] add reason-

ably low overhead (22%-0%) so that they can potentially deploy universally. Unfortunately, they only detect some types of attacks, as shown by a prior work [148]. Address space randomization [102, 50] 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 by who. 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.

In contrast, those techniques which provide reasonably accurate attack detection and analysis incur too much overhead (up to 30-40X slowdowns [99]) to be practical to deploy universally. Example of these tools include DIRA [125], DACODA [35], Vigilante [33], or TaintCheck [99]. The techniques which can best detect and analyze an attack (e.g., TaintCheck or DACODA) impose the highest overheads. To provide detailed analysis of the exploit, they instrument most of the instructions, and record many details about what happens. Due to 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 of hosts are unmonitored, open to attack.

A partial remedy proposed in Vigilante [33] is to, once caught at a sentinel machine, analyze the attack and automatically generate antibodies (called Self-Certifying Alerts or SCAs in Vigilante's parlance), to quickly distribute to other hosts against infection. Unfortunately fast hit-list worms can, if unimpeded, infect every vulnerable host in milliseconds [129]; 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., they fail to provide accurate and fast detection and analysis of Internet attacks without incurring high normal execution overhead.

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, the original implementation of TaintCheck cannot undo the bad effects; any overrun buffers will remain overrun. TaintCheck merely stops the attack, delegating recovery to restart. Unfortunately, restarting a system or an application usually takes up to several seconds [140]. 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 [13, 141].

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.

3.1.1 Contributions

This chapter describes techniques to defend against self-spreading worms which address these issues and achieves the three goals. The techniques described are implemented in the demonstration system Sweeper[2]. Sweeper does this in three ways:

First, by cleverly leveraging a lightweight checkpointing and monitoring support, Sweeper can postpone heavyweight monitoring until absolutely necessary — after being attacked. In other words, during normal execution, the system takes only lightweight checkpoints and runs lightweight monitoring. The checkponts allow allow re-execution and recovery in case of an attack, while the monitoring detects a wide range of suspicious requests. Unknown exploits can be detected using generic detectors (such as address randomization), while known exploits can be detected through Sweeper's automatically generated antibodies. Both the checkpoinging and monitoring impose very low overhead, making near universial deployment practical.

Second, after an attack is detected, Sweeper "goes back in time" (i.e., rolls back) and *dynamically* adds heavy-weight instrumentation and analysis during replay to conduct com-

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

prehensive and thorough attack analysis. This includes dynamic memory bug detection, dynamic program slicing, memory state analysis, and dynamic taint analysis, as well as automatically generating antibodies such as input signatures and vulnerability-specific execution filters (VSEF) [96]. Antibodies are further discussed in Section 3.3. Doing such allows sophisticated and detailed analysis to be performed only for those recent messages and execution period that are relevant to the occurred attack—server initialization and long runs of harmless inputs and normal execution need not suffer expensive monitoring and information recording. This novel use of checkpoint and rollback provides both low overhead and thorough analysis.

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

These ideas are implemented in a real system. The functioning prototype is implemented in Linux, building on a modified version of the previous Rx framework [108]. Sweeper uses address space randomization for lightweight detection, backed by post-exploit analysis tools such as dynamic memory bug detection, dynamic taint analysis [99], and backward slicing [147]. Sweeper uses the PIN [77] dynamic instrumentation tool to add these analysis tools on-demand. Sweeper also has an implementation of both input based filtering and VSEFs for defense on both hosts performing analysis and those hosts which choose not to.

Sweeper is tested 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, this chapter presents analytical results showing that even when partially deployed, Sweeper is capable of containing even fast hit-list worms. To summarize, Sweeper has the following unique 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 [26, 50, 150] or lightweight dynamic bug detection [39, 107, 157] impose reasonable amount of overhead (nearly zero for address space randomization), feasible for production run deployment. In-memory checkpointing, such as the previous Flashback and Rx systems [108, 128], also impose only marginal amounts of overhead (e.g., 1-5%). As demonstrated in the experimental results (Section 3.5.1), 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 analysis power. When the light-weight monitoring trips, we can roll back and re-execute with heavyweight analysis. Sweeper then dynamically uses binary instrumentation tools (e.g., PIN [77]) to insert analysis such as dynamic taint analysis [99] or backward slicing [147] after the fact. Therefore, Sweeper does not pay for expensive analysis for requests which 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; Sweeper must also restore the server to a safe state. Once an attack is detected, Sweeper uses rollback/re-execution to re-execute without the attacker's input. Rollback removes the corruption the attacker may have left, while re-execution allows the program complete servicing concurrent and further valid requests without restarting, thus achieving fast recovery.

4. **It provides a partial deployment option to hosts that demand even lower overhead.** Although the overheads involved are low, there may be hosts which do not wish to deploy the analysis tools. Sweeper does not leave such hosts completely defenseless. As shown in Section 3.6, Sweeper also provides an effective community

defense option which can protect most hosts even in a hit-list worm attack when only a fraction deploy the Sweeper analysis mechanisms.

3.2 Architecture

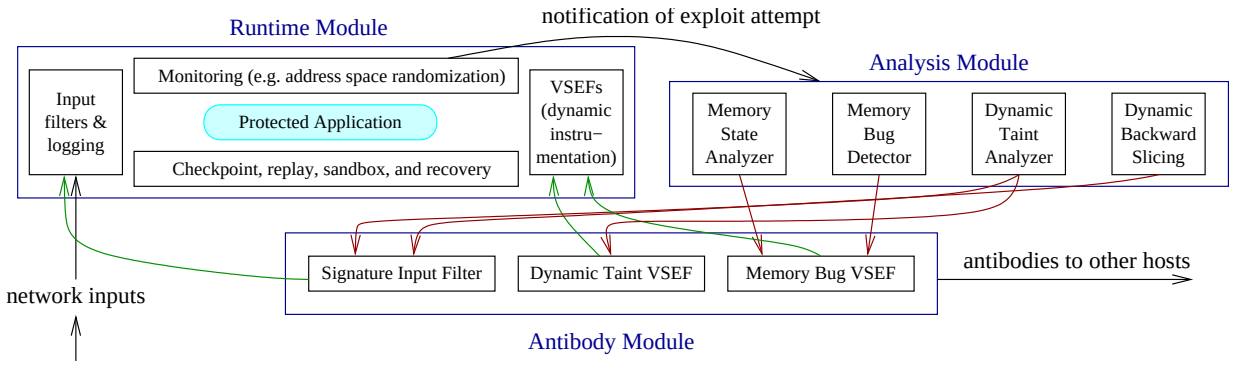


Figure 3.1: Architecture diagram of Sweeper

3.2.1 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 3.1 shows the architecture of Sweeper. The above four functions are provided by three modules: runtime, analysis and antibody. Section 3.3 describes the details of each component; this section discusses their overall function and interactions.

Runtime module The run time module supports (1) light-weight 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 3.3 to detect suspicious requests. 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 light-weight monitoring, the run time module also takes periodic light-weight, in memory checkpoints similar to Rx [108] and FlashBack [128] 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, the checkpoints cannot be touched. Further, the analysis tools are applied *after* an attack is detected. They take control of the execution path, and can 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 analyses. To support re-execution from a previous checkpoint, Sweeper needs to replay all or a selected subset of incoming network messages received since that checkpoint based on the type of analysis performed. During the re-execution, all side effects such as outgoing network messages are sandboxed and silently dropped.

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 my previous work, Rx. These issues are briefly discussed in Section 3.4, but more details can be

found in [108].

Analysis module The thorough 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 monitors in the runtime module. 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 which could be applied; the actual implementation (see Section 3.3 for details) performs a static analysis of the memory state, dynamic memory bug detection similar to Valgrind [94] and Purify [56], dynamic taint analysis similar to TaintCheck [99], and dynamic backward slicing [147]. 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 and derive antibodies to detect future exploits to the same vulnerability. There are two types of antibodies supported by Sweeper: input signature filters, and VSEFs [96]. Given the input responsible for the exploit, an input signature for filtering can be generated [63, 65, 98, 124]. Also, given the instructions involved in the exploit (especially for buffer overflows), we can generate a VSEF. In the case of a memory bug (e.g. stack smashing), the VSEF consists of monitoring

the instruction which 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. Nevertheless, it is possible, and still beneficial, to run only a partial set. This is further discussed in Section 3.6.

3.2.2 Process

To clarify how the system works, this subsection presents a concrete walk-though of a real vulnerability. Figure 3.2 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 3.3 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. In response,Sweeper begins its attack analysis. The execution is rolled back to the previous checkpoint, and heavier weight analysis techniques are performed. In the 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

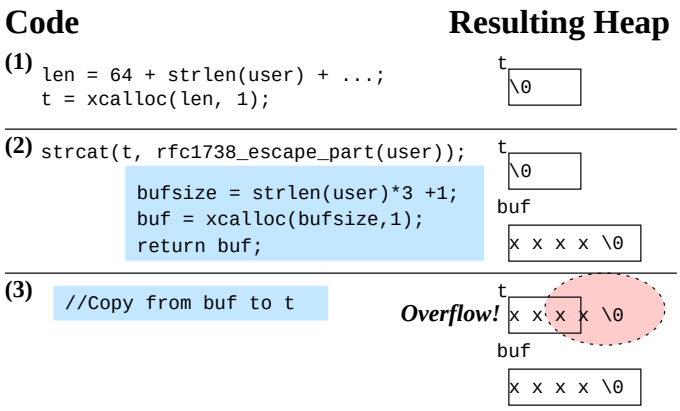


Figure 3.2: A buffer overflow in Squid (CVE-2002-0068).

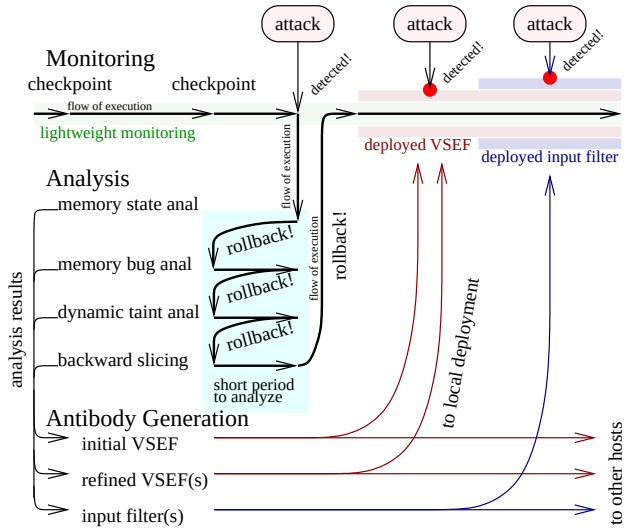


Figure 3.3: Sweeper defense process.

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 the Squid example, the heap is inconsistent, and memory bug detection points out instruction 0x4f0f0907 in `strcat` as the source. This confirms earlier results, and takes around 37 seconds.

The next step is dynamic taint analysis [99]. 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 [63, 65, 98, 124]. 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 Sweeper to verify the results of the previous analyses: any identified issue which is not in the slice is a false positive. 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 which 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 coredump, and Sweeper 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, Sweeper achieves something better than either one; fast but potentially weak results from static analysis augments slow but thorough results from dynamic analysis.

3.3 Design and Implementation

As discussed in Section 3.2, Sweeper has three components, one for runtime support, one for post-attack exploit analysis, and one for dealing with antibodies. This section further describes the details of each individual components.

3.3.1 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 [102, 10, 11, 26, 50, 50, 150], although there are many other mechanisms

which could be used [102, 30, 34, 50], including some of our own previous work such as SafeMem [107], LIFT [106]. 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 which 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 [77]. PIN allows the efficient addition of instrumentation to an already running process. However, any instrumentation tool which allows dynamically attaching to a running process would be feasible (e.g. dynInst [45]); we choose PIN due to familiarity and efficiency. Since only a minute portion of the execution needs to be monitored (generally only the instruction which 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. Sweeper uses a modified version of the Rx [108] 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 Sweeper keeps them for a short time (a few minutes at most) and then discards them. The advantage is much lower overhead than present in systems which write checkpoints to disk.

Similar to Rx [108], 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 user level*, even though some of their virtual pages may point to the same physical pages due to copy-on-write. Under the assumption that the operating system kernel is secure, an attack that corrupts the monitored process is unlikely to affect any checkpoint state because 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 to ensure that the shadow process's memory state is not affected.

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 [74, 128] 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 can also implement signature-based input filtering. The re-execution runs faster than the original, since there are no network delays or disk cache misses and hence IO costs are lower. More details can be found in the Rx paper [108].

Recovery As mentioned, the identification of the original input responsible for the attack also allows fast recovery: Sweeper simply rolls back the process and re-executes it without the malicious input, and thus brings the process back to a safe state. In the implementation, rollback is accomplished by reverting to a previously saved system checkpoint. Sweeper then restarts the system and replays legitimate (non-malicious) requests received after the checkpoint. Further issues related to recovery of stateful services are discussed in Section 3.4.

3.3.2 Exploit Analysis

After the lightweight monitors have triggered, Sweeper performs a more thorough analysis of the attack. Sweeper uses 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, Sweeper cannot achieve highly precise results. It is possible that an exploit may trigger the monitors and leave memory in a seemingly consistent state. Hence, Sweeper 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 desirable (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 [50]. Detecting the misbehaving memory instruction usually gives an important clue to find the exploited instruction. Furthermore, detecting a memory bug gives a straightforward VSEF: simply 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 [157]). 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, Sweeper *dynamically* attaches 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* in x86). The second memory misbehavior Sweeper detects is *heap overflow*. Sweeper uses a modified red-zone technique which 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 demonstrated in [99], 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. Sweeper uses a reimplementation of 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, Sweeper can trace back to the responsible input, identifying the instructions that passed it along the way. For more details, please see [99].

Dynamic Backward Slicing A backward program slice is the set of instructions which affected the execution of a particular instruction [147]. 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.

Sweeper implements dynamic backward slicing in a way similar to [156]. It tracks the last dynamic instruction to write to each register and memory location, as well as the last

instruction 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. Sweeper constructs 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: the implementation used in Sweeper 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. Although Sweeper can compute such a slice from the dependence tree it generates, it currently does not do so.

3.3.3 Antibodies

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

Input Signatures Input filters are commonly used to eliminate known exploits before they reach vulnerable servers [63, 65, 99, 98, 124]. Based on the input which 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, it can start by generating signatures as exact matches. This has the benefit of very low false positives, and being impervious to malicious training [97]. Polymorphic signatures are also feasible; see [18] for details.

VSEFs Vulnerability specific execution filters [96] 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 polymorphism and metamorphism. 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 which 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 which 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 [96].

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 which 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.

3.4 Issues and Discussion

3.4.1 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 be different on re-execution. An alternative to Rx is to use a Flashback [128] based checkpointing system. Flashback *logs* all of the system calls made by the process, in order to allow *deterministic* re-execution. This allows Sweeper 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 Sweepwer was 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) which changes the output, Sweeper can abort the re-execution and resort to restart. In practice this is a rare case, however, for those instances where the execution is sensitive to small changes, this alternative exists.

A further issue would be the 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 Sweeper would be unable to apply a correct rollback and re-execution. To prevent this, the same checkpointing techniques could be applied to the whole OS through a virtual machine (e.g. as is done in Time Traveling Virtual Machines [64]). This allows rollback of an entire software stack, including the OS, any helper applications, and even disk state. Although the OS is unlikely to be corrupted prior to the lightweight monitoring registering an attack, it is nearly a sure thing that the VM hypervisor will not become corrupted by a network-based attack on one of its guests.

3.4.2 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 that 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.

3.4.3 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, or 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 3.6 discusses in much greater detail the performance of Sweeper as a whole under varying conditions.

3.5 Experimental Results

3.5.1 Experiment Setup

Name	CVE ID [139]	Bug Type	Security Threat Description
Apache1	CVE-2003-0542	Stack Smashing	Local exploitable vulnerability enables unauthorized access
Apache2	CVE-2003-1054	NULL Pointer	Remotely exploitable vulnerability allows disruption of service
CVS	CVE-2003-0015	Double Free	Remotely exploitable vulnerability provides unauthorized access and disruption of service
Squid	CVE-2002-0068	Heap Buffer Overflow	Remotely exploitable vulnerability provides unauthorized access and disruption of service

Table 3.1: List of tested exploits

Implementation Sweeper is implemented 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 [77]. All of the tools are integrated together except for taint analysis; it is implemented stand alone although it could be integrated. Hence this sectin provides functionality results but not performance numbers for taint analysis. In lieu of taint analysis performance, this section presents the time to isolate the exploit input by sending the potentially suspicious requests one at a time. Both approaches provide the exploit input as a result, but, based on experience with Valgrind-based TaintCheck, taint analysis is expected to be faster.

Experiment Environment and Parameters The experiments were 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 The evaluation of Sweeper is on four *real* vulnerabilities in three server applications, as shown in Table 3.1. All of the vulnerabilities are recorded by US-CERT / NIST [139].

Experimental Design In the experiments, the functionality of Sweeper, as well as the efficiency of exploit and vulnerability analysis, is evaluated. This section also reports the normal overhead of checkpointing for various checkpoint intervals.

3.5.2 Functionality Evaluation

Table 3.2 presents the details of what Sweeper's functionality returns for four exploits. For all four exploits, Sweeper detects the attack, generates a VSEF, and identifies the original input which triggered the fault. Somewhat more specifically, the end results for the four exploits are:

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

Table 3.2: Overall Sweeper results

- Apache 1 - Correct detection of buggy instruction and memory location, correct VSEFs, and correct configuration-specific triggering input.
- Apache 2 - Correct identification of NULL pointer dereference, correct VSEFs, and correct triggering input.
- CVS - Correct detection of buggy instruction and memory location, correct VSEFs, and correct triggering input.
- Squid - Correct detection of buggy instruction and memory location, correct VSEFs, and correct triggering input.

The detailed results in Table 3.2 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[3]; 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 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 was 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.

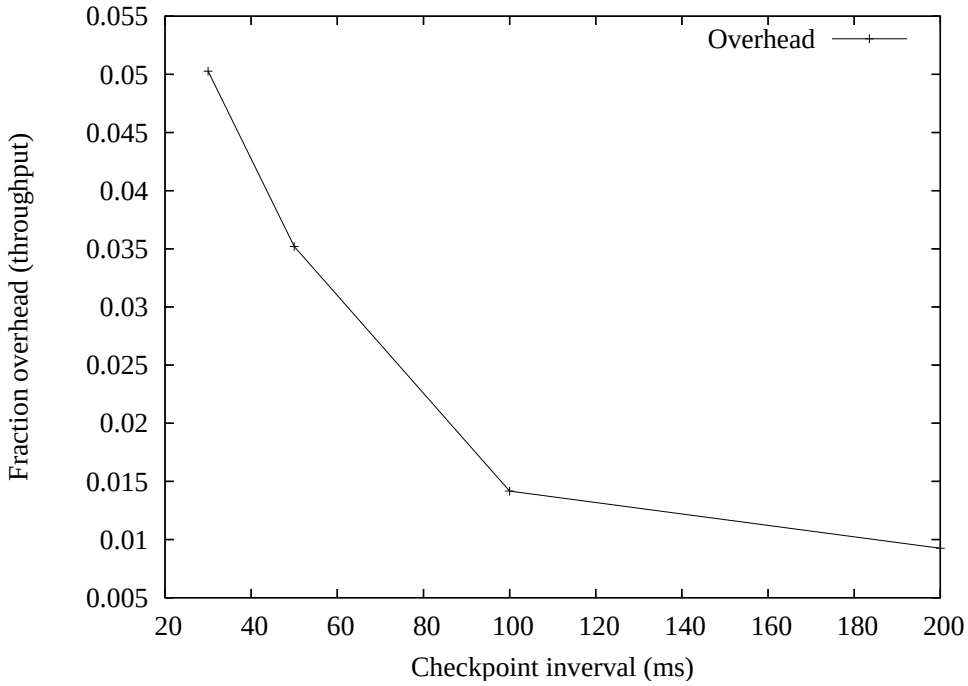


Figure 3.4: Performance at varying checkpoint intervals for Squid

3.5.3 Performance Evaluation

Checkpointing Since Sweeper is intended for widespread deployment, overhead is an important concern. As demonstrated by Figure 3.4, the performance overhead of checkpointing and network logging is low; at a 200ms checkpoint interval, Sweeper only degrades performance by -.925% —throughput drops from 93.5 Mbps to 92.6Mbps. 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 [108].

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, the VSEF checks for a heap buffer overflow at 0x4f0f0907 (in `strcat`), and then only when `strcat` is called by 0x804ee82 (in

[3]This particular buffer overflow does *not* have such multiple methods of exploitation.

	Apache 1	Squid
Time to first VSEF	60 ms	40 ms
Time to best VSEF	14 sec	40 ms
Initial Analysis Time	24 sec	38 sec
Total Analysis Time	68 sec	145 sec
Component Diagnosis Time		
Memory State Analysis	60 ms	40 ms
Memory Bug Detection	14 sec	30 sec
Input/Taint Analysis	9 sec	7 sec
Dynamic Slicing	45 sec	108 sec

Table 3.3: 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, Sweeper 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.

`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 which have wild exploits will experience higher overheads. Users who wish to avoid such overhead should apply 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 3.6, fast antibody generation is important for dealing with the fastest of worms; 60 ms is more than fast enough. Table 3.3 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 immediately upon availability (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.

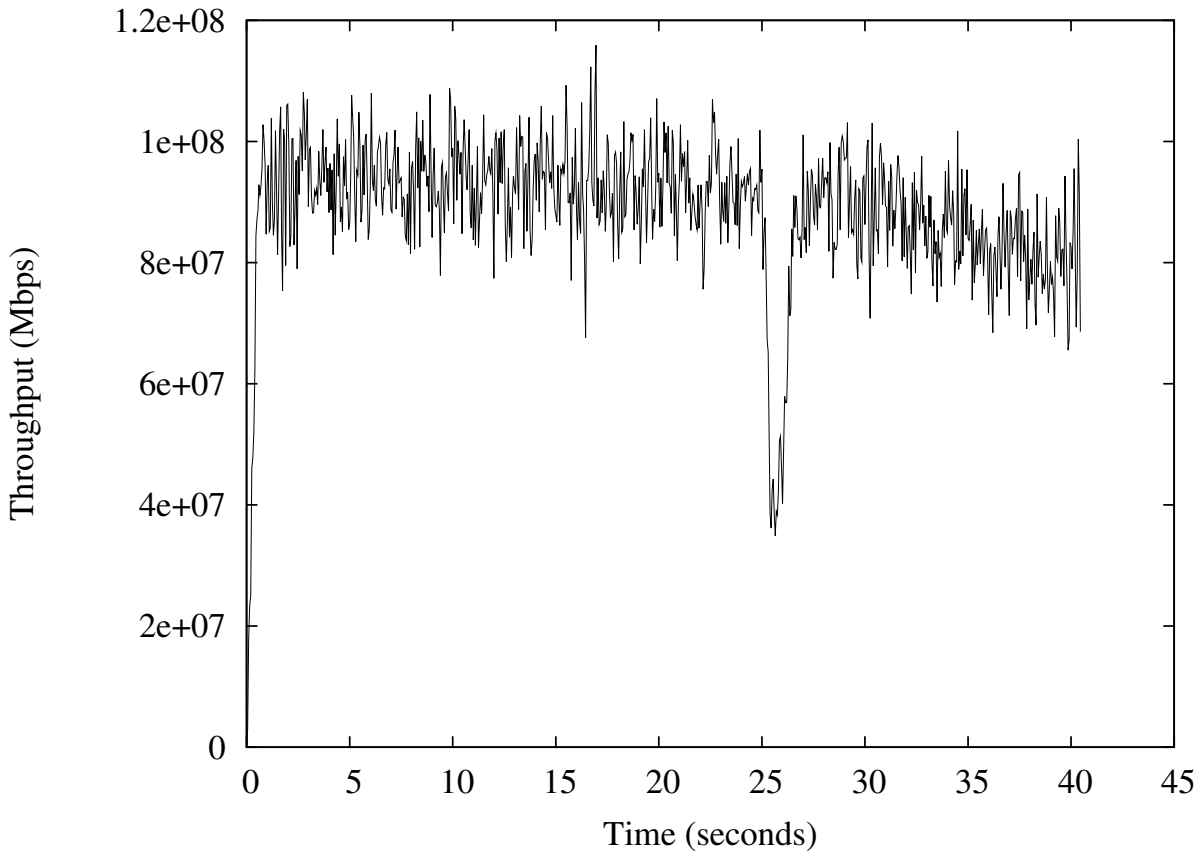


Figure 3.5: Throughput during a single attack against Squid

Recovery Once VSEFs are applied, Sweeper performs recovery. Figure 3.5 shows the client-perceived 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. For more details about the recovery performance of Sweeper please see [108].

3.6 Community Defense Against Fast-Spreading Worms

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

This section shows how a Sweeper *community* can protect even those who do not deploy Sweeper from new exploit attacks, including fast-spreading worms. In this community, hosts who deploy the complete Sweeper system are called *Producers*. When a Producer detects a new attack and generates the corresponding antibodies, it shares those antibodies with *Consumers* (and all other Producers), 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; this section considers Producer deployment ratios far below expectations.

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

3.6.1 Community Model

Worm propagation can be well described with the classic Susceptible-Infected (SI) epidemic model [58]. 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 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:

$$\frac{dI(t)}{dt} = \beta I(t)(1 - \alpha - I(t)/N) \tag{3.1}$$

$$\frac{dP(t)}{dt} = \alpha\beta I(t)(1 - P(t)/(\alpha N)) \tag{3.2}$$

The time at which at least one Producer has received an infection attempt, and hence can begin generating and distributing antibodies, is called T_0. By this definition, $P(T_0) = 1$. The above equation is solvable 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 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.

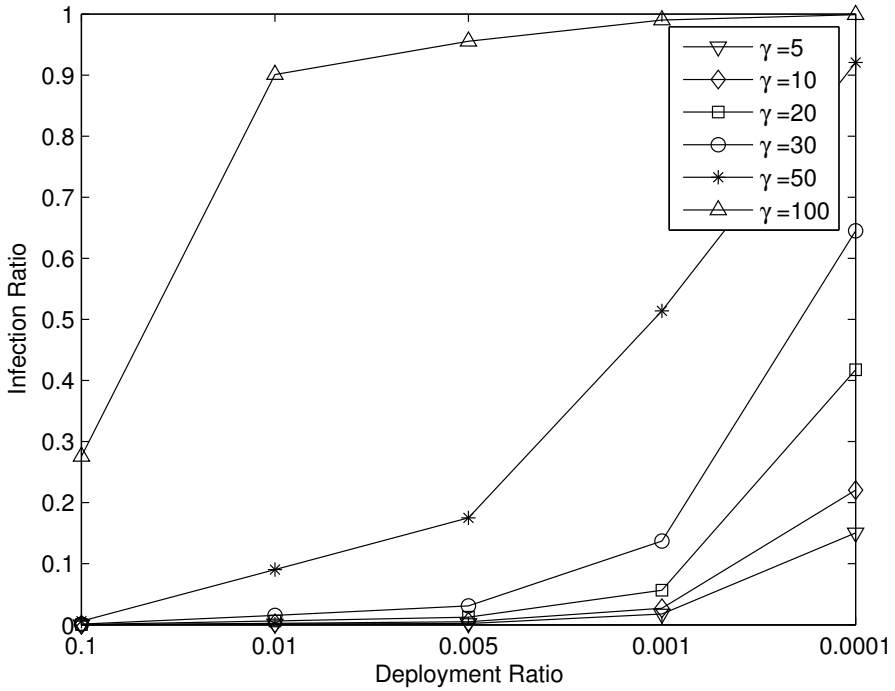


Figure 3.6: Sweeper defense against Slammer ($\beta = 0.1$)

3.6.2 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 [22].

Figure 3.6 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.

3.6.3 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 [129, 130].

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 [17].

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. The spread of a hit-list worm where vulnerable hosts use proactive protection can be modeled with:

$$\frac{dI(t)}{dt} = \beta\rho I(t)(1 - \alpha - I(t)/N) \tag{3.3}$$

$$\frac{dP(t)}{dt} = \alpha\beta I(t)(1 - P(t)/(\alpha N)) \tag{3.4}$$

Sweeper combined with proactive protection can protect against even hit-list worms with contact rate $\beta = 1000$ (Figure 3.7), and with contact rate $\beta = 4000$ (Figure 3.8). Here, the probability that an infection attempt succeeds is set to $\rho = 2^{-12}$, which many address

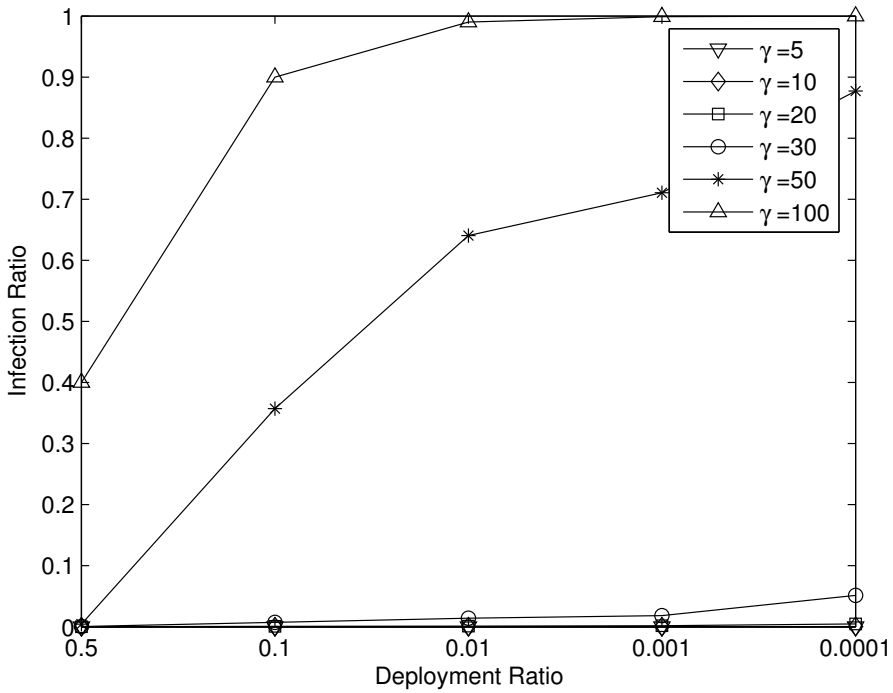


Figure 3.7: Sweeper with proactive protection against hit-list ($\beta = 1000$). Note that $\gamma = 50$ is much worse than $\gamma = 30$.

randomizations achieve [120]. 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%) in 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, an automated defense such as Sweeper is still required.

These 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 the previously presented experiments (Section 3.5.1) show that detection and analysis are almost instantaneous, and the total time it takes to create an effective VSEF is well

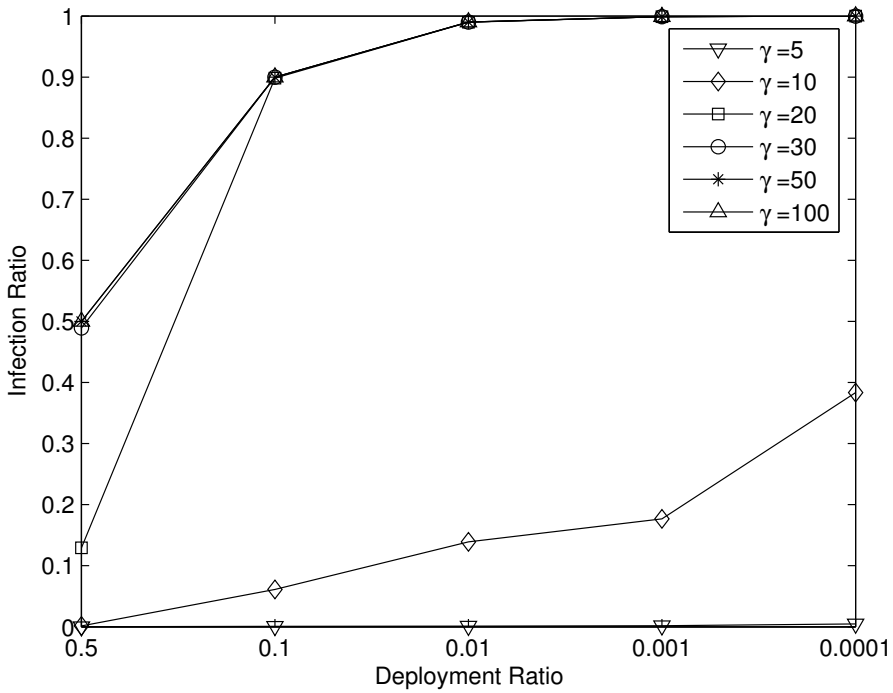


Figure 3.8: Sweeper with proactive protection against hit-list ($\beta = 4000$). Note that $\gamma = 20$ is much worse than $\gamma = 10$.

under 2 seconds. Vigilante shows that the initial dissemination of an alert could take less than 3 seconds [33]. Thus Sweeper achieves an $\gamma = 2+3 = 5$. By impeding the spread of the worm, Sweeper can effectively defend against effectively defend against even hit-list worms which are thousands of times faster than the fastest observed worm, even for low values of α.

3.7 Related Work

Chapter 2 provides a more through understanding of the related work; this section briefly covers closely related work and how it relates to Sweeper.

3.7.1 Checkpoint and Rollback

While Sweeper leverages a lightweight checkpoint and re-execution support similar to Flash-Back or Rx [108, 128], it could use other checkpoint systems like the Time-traveling Virtual Machines [64], or ReVirt [44]. ReVirt also deals in a security setting: specifically, post-mortem analysis. However, ReVirt is intended as an offline forensic tool, and does not target on-line systems.

3.7.2 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 [50], provides an almost free detection mechanism, however, it can be probabilistically bypassed [120]. 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 [107] may also be used widely. Other monitors exist which trade runtime overhead for greater protection. Those such as StackGuard [34] or CCured [30, 92] require source code, while Purify [56] or Valgrind [94] are applicable in the binary-only case. Shadow honeypots [4] consider moving the attack detectors to a separate machine or separate process in order to allow better overhead management.

Dynamic taint analysis has been variously proposed in [99, 36, 132]. Taint analysis detects the usage of data "tainted" by untrusted input in various dangerous ways (for example, using the input as a return address). Taint analysis is quite similar to dynamic slicing [147, 156], but focuses on detecting exploit attempts rather than debugging. Although taint analysis is a powerful technique to detect security exploits it tends to impose infeasibly high overheads.

A recent work in dynamic binary instrumentation, LIFT [106], reduces the overehad to potentially manageable levels (2-4x); this may be deployable for a decent fraction of hosts.

3.7.3 Attack Response

A considerable amount of research effort [33, 63, 65, 99, 98, 124] has been devoted to automatically generating attack signatures. Earlybird [124], Honeycomb [65] and Autograph [63], 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 [98] generate signatures that consist multiple disjoint content substrings. However, recent work [97, 104] shows that such polymorphic signature generators can be mislead into generating bad signatures: specifically signatures with high false negative rates.

There have been various approaches to repair buggy programs. Some techniques are applied before hand; for instance CCured [30] retrofits memory bounds checking at the source code level, while DieHard [9] applies whole-program probabilistic memory safety through replication and library interposition. Rx [108], DIRA [125], and STEM [123] make attempts at repair post-hoc; Rx through environmental perturbations, DIRA by FIXME, and STEM through forcibly returning from a failing function. [72] modifies STEM by spreading the monitoring load out among all instances of an application; such space-wise sampling will reduce per-instance overhead at the expense of lower per-instance effectiveness.

FLIPS [73] is another automated worm defense, contemporaneous with Vigilante; it uses emulated execution and instruction set randomization to shepherd the execution of a potentially vulnerable program and to isolate the exploit input. Like Sweeper, FLIPS applies subsequent isolated exploit inputs to refine input filters. However, unlike both Sweeper and Vigilante, FLIPS does not consider fast worms, as it does not provide for sharing of protection among various hosts, and can take several exploit attempts and up to a full second before filters can be generated.

Vigilante [33] 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 honeypot hosts. Hosts which are sampling only have a small chance to analyze an exploit attempt, while honeypot 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 which 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 [70] and Xu et al. [151] 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 [35]. 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.

3.8 Conclusions

This chapter presents new techniques for defending against self-spreading exploits. By leveraging checkpointing and replay, continuous lightweight monitoring can be combined with heavy-weight analysis. The resulting system has low overhead (1%) during normal execution, which allows more wide-spread 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.

These techniques are implemented in Sweeper. Against 4 real exploits in 3 different server applications, Sweeper generates effective antibodies quickly (no slower than 60 ms). This chapter also provides analytical results demonstrating how effective Sweeper would be against a fast worm outbreak. These results show how sophisticated vulnerability-specific execution filters can be deployed while maintaining performance suitable for widespread production deployment.

Chapter 4

Triage

4.1 Overview

This Chapter presents the Triage[1] system. Triage builds on Sweeper (Chapter 3), extending it to attempt to automatically diagnose failures in production runs.

4.1.1 Motivation

As discussed in the introduction, software failures are a major contributor to system down time and security holes. Although vendors test their products before release, some bugs will inevitably be experienced by end users. Since these production run failures are directly causing pain to their customers, it is these failures that vendors most want to diagnose. While much work has been conducted on software failure diagnosis, most previous work focuses on *offsite* diagnosis (i.e. diagnosis at the development site with the involvement of programmers). This is insufficient to diagnose production run failures for four reasons:

1. It is difficult to reproduce the user site's failure-triggering conditions in house for diagnosis.

2. Offsite failure diagnosis cannot provide timely guidance to select the best online recovery strategy or security defense against fast internet worms (e.g. as discussed in Chapter 3).

[1]This work is based on an earlier work: Triage: diagnosing production run failures at the user's site, in ACM SIGOPS Operating Systems Review - SOSP '07, Volume 41, Issue 6, December 2007 (c) ACM, 2007. http://doi.acm.org/10.1145/1323293.1294275

3. Programmers cannot be provided to debug every end-user failure.

4. Privacy concerns prevent many users from sending failure information such as core-dumps or stack traces back to programmers[2].

Unfortunately, today's systems provide limited support for this important task: *automatically diagnosing software failures occurring in end-user site production runs.*

Unlike software bug detection, which is often conducted "blindly" to screen for possible problems, software failure diagnosis aims to understand a failure that has actually occurred. While errors detected by a bug detector provide useful information regarding a failure, they are not necessarily root causes—they could be just manifestations of other errors [1, 147, 156]. Typically programmers will still need to manually debug the fault, utilizing many different diagnostic techniques, before they have collected enough information to thoroughly understand a failure.

Following the definition used in software dependability [111], comprehensive knowledge of a software failure includes three components (shown in Figure 4.1) beyond the failure itself. The first of these is the fault: the underlying incorrect code which is responsible for the failure. The second is the trigger: the input or environmental condition which caused the fault to activate or whatever it was that "tickled the bug". The third is the error and the error propagation chain: the incorrect system state which, potentially propagated through multiple stages, lead to the failure. Consequently, failure diagnosis targets three things: (1)

[2] Even if sent back, such information is just a snapshot of system state after the failure, and does not on its own provide an understanding of the fault.

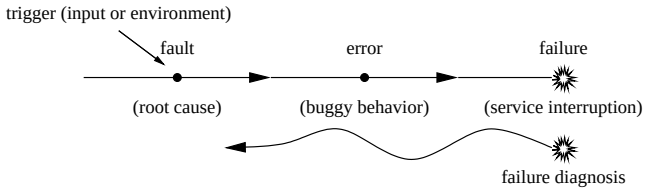


Figure 4.1: Failure diagnosis is driven by failure and tries to understand the whole fault chain.

what execution misbehavior caused the failure; (2) where the misbehavior originated; and (3) how the fault was triggered. So while software bug detection is like disease screening, failure diagnosis is more like disease diagnosis: it is a focused, problem-driven analysis based on an *actual* problem.

4.1.2 Current State of the Art

There are many well known bug detection tools. Dmalloc [145] only gives information about heap corruption at free time, which can be substantially delayed from the corruption itself. ProPolice [48] and StackGuard [34] can identify stack smashing, although the runtime overhead is a little high given the limited information they return. The instrumentation suite Valgrind [94] is well known for its bug detection tools. Memcheck can identify a variety of memory bugs, while helgrind detects data races through the lockset algorithm. While providing much more information than, say, dmalloc or StackGuard, Valgrind based tools are much too expensive to run all of the time. Regardless, all of these "bug detectors" are actually error detectors; they detect bad runtime states which could possibly cause a failure. While having the error in hand can be useful for discovering the fault, the actual incorrect code could be well separated from a detectable error.

Existing failure diagnosis work mostly focuses on *offsite* diagnosis; hence although they provide some automated assistance, they rely heavily on programmers to manually and interactively deduce the root cause. Examples of such offsite tools include interactive debuggers [53], program slicing [1, 147, 156], and offline partial execution path constructors from a coredump such as PSE [79]. Almost all these tools either impose overheads too large (up to 100x) to be practical for production runs, or heavily rely on human guidance.

The current state of the art of *onsite* software failure diagnosis is primitive. The few deployed onsite diagnosis tools, like Dr. Watson [43] and the Mozilla Quality Feedback Agent [86], only collect simple raw (unprocessed) failure information (e.g. coredumps and environment information). Recently proposed techniques extract more detailed information

such as traces of network connections [28], system call traces [154], traces of predicated values [71], etc. Furthermore, some deterministic replay tools [51, 64, 128] have also been developed for uniprocessor systems.

One tool of particular note is the Windows Error Reporting service, or WER [52]. WER is crash reporting service built into modern versions of Windows, and has an installed base of one billion clients. According to [52], WER is very useful in prioratizing which failures to fix, by their rate of occurance. Most of the power of WER comes from the backend server's ability to automatically catagorize crash occurances into "buckets", which fairly faithfully represent occurances of the same bug. For bugs which have been addressed, WER can tell the user what update to install or workaround to use to prevent future recurrance of the bug. Further, WER manages a 70 to 80% submission rate, thanks in part to the wording of the dialog ("Check online for a solution and restart the program?", or "Do you want to send more information about the problem?" depending on the version) and the feature to opt-in once rather than asking every time. In contrast, earlier versions of WER which had a more detailed message and asked every time had a 40 to 50% submission rate.

While all of these techniques are helpful for in-house analysis of production run failures, they are still limiting. Many of the more useful techniques (such as slicing) impose run-time overhead sufficiently high to eliminate production-run use from consideration. The low overhead techniques (e.g. returning core dumps) are limited in the amount of aid given to programmers.

4.1.3 Challenges for Onsite Diagnosis

Unfortunately, providing onsite diagnosis is not simply a matter of slapping together a bunch of diagnosis techniques. In order to achieve this goal one must address several major challenges:

1. **Efficiently reproduce the occurred failure**: Since diagnosis usually requires many iterations of failure executions to analyze the failure, an onsite diagnosis needs an effective way to automatically reproduce the failure-triggering conditions. Moreover, the diagnosis tool should be able to reproduce the failure quickly, even for failures that occur only after a long setup time.

2. **Impose little overhead during normal execution**: Even moderate overhead during normal execution is unattractive to end-users.

3. **Require no human involvement**: A programmer cannot be provided for every end-user site. Therefore, various diagnosis techniques should be employed automatically. Not only does each individual step need a replacement for any human guidance, but the overall process must also be automated.

4. **Require no prior knowledge**: Knowledge of what failures are about to happen does not exist. So any failure-specific techniques (e.g. memory bug monitoring) are a total waste during normal execution, prior to failure.

4.1.4 Summary of Contributions

This chapter describes *Triage*, the first automatic onsite diagnosis system for software failures that occur during production runs at end-user sites. Triage addresses the above challenges with the following techniques:

1. **Capturing the failure point and conduct just-in-time failure diagnosis with checkpoint-re-execution system support**. Traditional techniques expend equal heavy-weight monitoring and tracing effort during the whole of execution; this is clearly wasteful given that most production runs are failure-free. Instead, Triage takes lightweight checkpoints during execution and rolls back to recent checkpoints for diagnosis after a failure has occurred. At this moment, heavy-weight code instrumentation,

advanced analysis, and even speculative execution (e.g. skipping some code or modifying variables) can be repeatedly applied to multiple iterations of re-execution focusing only on the moment leading up to the failure. In this scheme, the diagnosis has most failure-related information at hand; meanwhile both normal-run overhead and diagnosis times are minimized. In combination with system support for re-execution, heavy-weight bug detection and analysis tools become feasible for onsite diagnosis. Furthermore, the failure moment can be relived over and over. Triage can study it from different angles, and manipulate the execution to gain further insights.

2. **New failure diagnosis techniques — delta generation and delta analysis — that effectively leverage the runtime system support** with extensive access to the whole failure environment and the ability to repeatedly revisit the moment of failure. The delta generation relies on the runtime system support to speculatively modify the promising aspects of the inputs and execution environment to create many similar but successful and failing replays to identify failure-triggering conditions (inputs and execution environment settings). From these similar replays, Triage conducts *delta analysis* automatically to narrow down the failure-related code paths and variables.

3. **An automated, top-down, human-like software failure diagnosis protocol**. As will be shown in Figure 4.3, the Triage diagnosis framework automates the methodical manual debugging process into a diagnosis protocol, called the TDP (Triage Diagnosis Protocol). Taking over the role of humans in diagnosis, the TDP processes the collected information, and selects the appropriate diagnosis technique at each step to get more information. It guides the diagnosis deeper to reach a comprehensive understanding of the failure. Using the results of past steps to guide future steps increases their power and usefulness.

   Within the TDP framework, many different diagnosis techniques, such as delta generation, delta analysis, coredump analysis and bug detection, are integrated. These

techniques are automatically selected at different diagnosis stages and applied during different iterations of re-execution to find the following information regarding the occurred failure:

- *Failure nature and type*: Triage automatically analyzes the failure symptoms, and uses dynamic bug detection during re-execution to find the likely type of program misbehavior that caused the failure, including both the general bug nature such as nondeterministic vs. deterministic, and the specific bug type such as buffer overflow, data race, etc.

  Failure-triggering conditions (inputs and execution environment): Through repeated trials, Triage uses the delta generation technique to forcefully manipulate inputs (e.g. client requests) and execution environment to identify failure-triggering conditions.

- *Failure-related code/variable and the fault propagation chain*: Triage uses delta analysis to compare failing replays with non-failing replays to identify failure-related code/variables. Then it may intersect the delta results with the dynamic backward slice to find the most relevant fault propagation chain.

4. **Leverage previous failure analysis techniques for onsite and post-hoc diagnosis.** Runtime system support for re-execution with instrumentation and the guidance of the TDP allow Triage to synergistically use previous failure analysis techniques. Triage implements some such techniques, including static coredump analysis, dynamic memory bug detectors, race detectors and backward slicing. However, as they are dynamically plugged in after the failure has occurred, they require some modification. Both information from the beginning of execution and human guidance are unavailable. Either the tools must do without, or (especially in the case of human guidance) the results of previous analysis steps must fill in.

As will be detailed in Section 4.9, this chapter evaluates Triage using real system implementation and experiments on Linux with 10 real software failures from 9 applications (including 4 servers: MySQL, Apache, CVS and Squid). The experimental results show that Triage, including its delta generator and delta analyzer, effectively identifies the failure type, fault-triggering inputs and environments, and key execution features for most of the tested failures. It successfully isolates the root cause and fault propagation information within a short list; under 10 lines of suspect code in 8 out of the 10 failure cases. Triage provides all this while it imposes less than 5% overhead in normal execution and requires at most 5 minutes to provide a full diagnosis.

Finally, a user study with 10 programmers shows that the diagnostic information provided by Triage shortens the time to diagnose real bugs (statistically significant with $p < .01$), with an average reduction of 44.7%.

4.2 Triage Architecture Overview

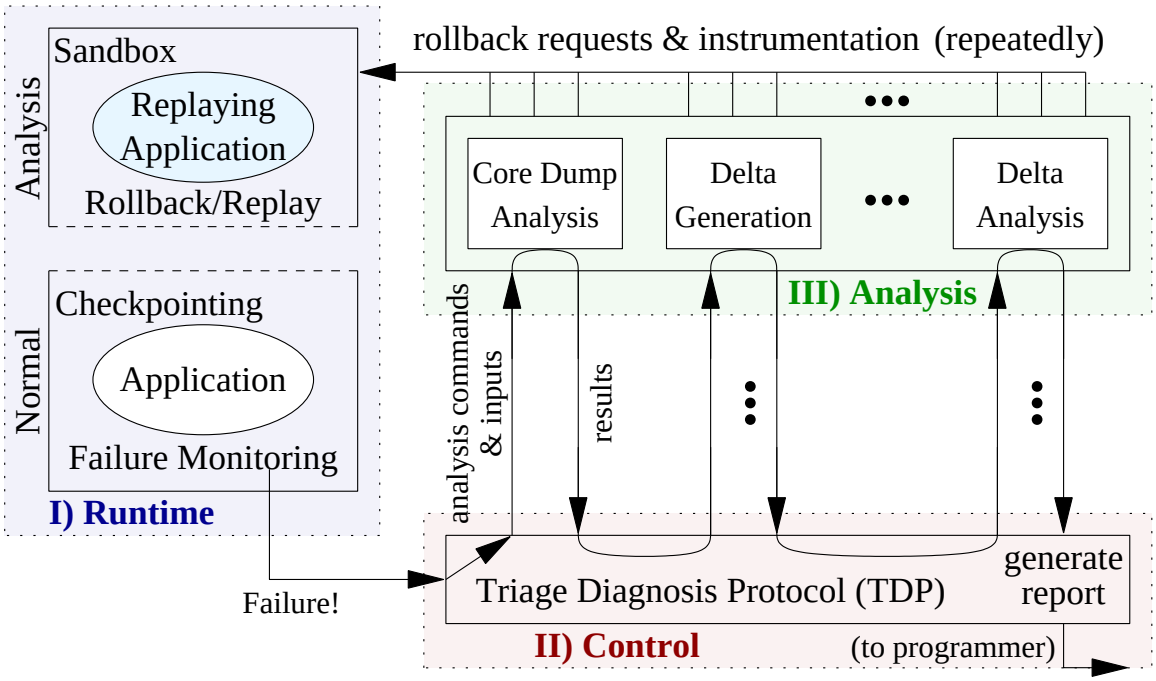


Figure 4.2: Triage architecture overview

Triage is composed of a set of user- and kernel-level components to support onsite just-in-time failure diagnosis. As shown in Figure 4.2, it is divided into three groups of components. First, the *runtime* group provides three functions: lightweight periodic checkpointing during normal execution, catching software failures, and sandboxed re-execution (simple replay or re-execution with controlled execution perturbation and variation) for failure diagnosis. Second, the *control* group deals with deciding how the sub-components should all interact, and implements the Triage Diagnosis Protocol (see Section 4.3). It also directs the activities of the third layer: failure diagnosis. Finally, the *analysis* group deals with post-failure analysis; it is comprised of various dynamic failure analysis tools, both existing techniques and new techniques such as automatic delta generation and delta analysis presented in Sections 4.4 and 4.5.

Checkpoint and Re-execution In order to allow repeated analysis of the failure, Triage requires checkpoint and re-execution support. There are many ways to implement re-execution, such as Time Traveling Virtual Machines [64] or Flashback [128]. Triage leverages the lightweight checkpoint and re-execution runtime system provided in Rx. It is briefly described here; details can be found it [108, 128].

Rx takes checkpoints using a `fork()`-like operation, keeping everything in memory to avoid the overhead of disk accesses. Rollback operations are a straightforward reinstatement of the saved task state. Files are handled similarly to previous work [74, 128] by keeping a copy of accessed files and file pointers at the beginning of a checkpoint interval and reinstating it for rollback. Network messages are recorded by a network proxy for later replay during re-execution. This replay may be potentially modified to suit the current re-execution run (e.g. dropped or played out of order). Triage leverages the above support to checkpoint the target application at runtime, and, upon a failure, to roll back the application to perform diagnosis. Rx is particularly well suited for Triage's goals because it tolerates large variation in how the re-execution occurs. This allows us not only to add instrumentation, but to use

controlled execution perturbations for delta generation.

However, Rx and Triage have vastly different purposes and hence divergent designs. First, Triage's goal is diagnosis, while Rx's is recovery. Triage systematically tries to achieve an understanding of the occurring failure. Such an understanding has wide utility, including recovery, security hot fixes, and debugging. Rx simply tries to achieve survival for the *current* execution–gathering failure information is a minor concern so long as Rx can recover from the failure. That is, Rx considers the "why?" to be unimportant. Second, while Rx needs to commit the side effects of a successful re-execution, Triage must instead sandbox such effects. Fortunately, this allows Triage to completely ignore both the output commit problem and session consistency. Hence Triage can consider much larger and varied execution perturbations than Rx, even those which are potentially unsafe (e.g. skipping code, modifying variable values), with minimal consistency concerns. Triage directly uses some of Rx's existing perturbations (e.g. changing memory layouts), uses others with both a much higher degree of refinement and variety (e.g. input-related manipulations, see Section 4.4), and briefly considers some radical changes (patching the code).

Lightweight Monitoring Also like Rx, Triage must detect that a failure has occurred. Any monitoring performed at normal time cannot impose high overhead. Therefore, the cheapest way to detect a failure is to just catch fault traps including assertion failures, access violations, divide-by-zero exceptions, etc. Unique to Triage, though, is the need to monitor execution history for subtle software faults. More sophisticated techniques such as program invariant monitoring or memory safety monitoring [107, 92] can be employed as long as they impose low overhead. In addition to detecting failures, lightweight monitoring can be also used to collected some global program execution history such as branch history or system call traces that will be useful for onsite diagnosis upon a failure. Previous work [16] has shown that branch history collection imposes less than 5% overhead. The current implementation relies on assertions and exceptions as the only normal-run monitors.

Control Layer The process of applying diagnosis tools through multiple re-executions is guided by the control layer, which implements the Triage Diagnosis Protocol described in Section 4.3. It chooses which analysis is appropriate given the results of previous analysis, and also provides any inputs necessary for each analysis step. After all analysis is complete, the control layer sends the results off to the off-site programmers for them to use in fixing the bug.

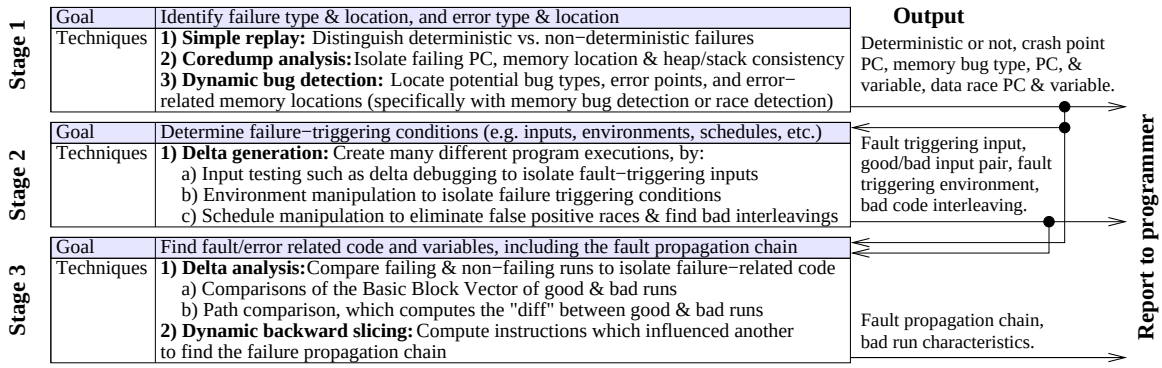


Figure 4.3: Diagnosis protocol diagram (This figure illustrates the Triage diagnosis protocol, including the implemented failure diagnosis components. These separate analysis components are run in one or more iterations of re-execution, during which all side-effects are sandboxed. Later stages are fed results from earlier stages as necessary.

Analysis (Failure Diagnosis) Layer Figure 4.3 provides a brief summary of the different failure diagnosis components in Triage. The stage one techniques are modified from existing work to make them applicable for onsite failure diagnosis. Stage 2 (delta generation) is enabled by our runtime system support for automatic re-execution with perturbation and variation. Dynamic backward slicing, although previously proposed, is made much more feasible in post-hoc application during re-execution. Finally, delta analysis is newly proposed. The details of these techniques are presented in Section 4.4, 4.5, and 4.6.

4.3 Triage Diagnosis Protocol

This section describes Triage's diagnosis protocol. The goal of the protocol is to stand in for the programmer, who cannot be present, and to direct the onsite software diagnosis process automatically. The default protocol using some representative fault analysis techniques is presented first, and then the protocol extensions and customizations to the default protocol are discussed.

Block	Line	Code

A	1
	2
	3
	4
		`char *`
		`get_directory_contents`
		`(char * path, dev_t device){`
		`  char * dirp = savedir(path);`

B	5
	6
	7
		`  char const * entry;`
		`  /*More variable declarations*/`
		`  if(!dirp)`

C	8	`    savedir_error(path);`

D	9
	10
	11
		`  errno = 0;`
		`  /*More code omitted*/`
		`  if(children != NO_CHILDREN)`

E	12 13	`    for(entry = dirp;` `        (len = strlen(entry));`

F	14
	15
	16
		`        entry += len + 1;){`
		`      /*Omitted*/`
		`    }`

G	17 18	`  /*Omitted*/` `}`

H	19
	20
	21
	22
	23
	24
		`char *`
		`savedir`
		`(const char *dir){`
		`  DIR *pdirp;`
		`  /*More variable declarations*/`
		`  dirp = opendir(dir);`

I	25	`  if(dirp == NULL)`
J	26	`    return NULL;`
K	27 28	`  /*Omitted*/` `}`

Figure 4.4: Simplified excerpt of a real bug in tar-1.13.25 as a running example to explain the diagnosis protocol.

4.3.1 Default Protocol

Figure 4.3 shows a flow chart of the default diagnosis protocol after a failure is detected. Triage uses different diagnostic techniques (some new and some modified from existing work) to automatically collect different types of diagnostic information (as described in Section 4.1) including (1) the failure type and nature, (2) failure-triggering input and environmental conditions, and (3) failure-related code/variables and the fault-propagation chain. The diagnosis stages are arranged in a way so that the later stages can effectively use the results produced

by the earlier stages as inputs, starting points, or hints to improve diagnosis accuracy and efficiency. Note that the default protocol is not the most comprehensive. Its purpose is to provide a basic framework that performs a general fault analysis as well as a concrete example to demonstrate the diagnosis process and ideas, and it could be extended and customized with new or application-specific diagnosis techniques.

Figure 4.4 shows a simplified version of a bug in tar, the common Unix archive program, which is used as a running example to explain the diagnosis protocol and new diagnosis techniques. Briefly, the bug occurs when the line 24 call to `opendir` returns NULL; subsequently this value is passed into `strlen` on line 13 without being checked. In the actual source code this bug is spread across thousands of lines in two separate files in separate directories.

In the first stage of diagnosis, Triage conducts analysis to identify the nature and type of the failure. It first mimics the initial steps a programmer would follow when diagnosing a failure: simply retry the execution, without any control or change and without duplicating timing conditions, to determine if the failure is deterministic or nondeterministic. If the failure repeats, it is classified as deterministic; otherwise it is classified as nondeterministic based on timing. Subsequent steps vary depending on this initial classification. For the tar example, this step indicates a deterministic bug.

To find out whether the failure is related to memory, Triage analyzes the memory image at the time of failure, when coredumps are readily available, by walking through the heap and stack to find possible corruptions. For tar, the coredump analysis determines that the heap and the stack are both consistent; the cause of the failure is a segmentation fault at `0x4F0F1E15` in the library call `strlen`.

After coredump analysis, the diagnosed software is repeatedly rolled back and deterministically re-executed from a previous checkpoint, each time with a bug detection technique dynamically plugged in, to check for specific types of bugs such as buffer overflows, dangling pointers, double frees, data races, semantic bugs, etc. Most existing dynamic bug detection techniques can be plugged into this step with some modifications described in Section 4.6.

Additionally, the high overhead associated with these tools becomes tolerable because they are not used during normal execution, but are dynamically plugged in at re-execution, during diagnosis, *after* a failure has occurred. For tar, the segfault was caused by a null-pointer dereference.

The second stage of the diagnosis is to find failure triggering conditions including inputs and execution environment settings, such as thread scheduling, memory layout, signals, etc. To achieve this, Triage uses a new technique called *delta generation* (Section 4.4) that intentionally introduces variation during replays in inputs, thread scheduling, memory layouts, signal delivery, and even control flows and memory states to narrow the conditions that trigger the failure for easy reproduction.

Unlike in Rx, which varies execution environments to bypass deterministic failures for recovery, Triage's execution environment variation can be much more aggressive since it is done during diagnostic replay while side effects are sandboxed and discarded. For example, not only does Triage drop some inputs (client requests), but it also alters the inputs to identify the input signature that triggers the failure.

In the third stage, Triage aims at collecting information regarding failure-related code and variables as well as the fault propagation chain. This stage is done by a new diagnosis technique called *delta analysis* (Section 4.5) and with a modified dynamic backward slicing technique [147]. From the delta generation, Triage obtains many failed replays as well as successful replays from previous checkpoints. By comparing the code paths (and potentially data flows) from these replays, Triage finds the differences between failed replays and non-failing replays. Further, the backward slice identifies those code paths which were involved in this particular fault propagation chain. Both of these are very useful debugging information.

All of the analysis steps end in producing a report. If ranking is desired, results of different stages could be cross-correlated with one another; the current implementation doesn't do this yet. Furthermore, information which is likely to be more precise (e.g. memory bug detection vs. coredump analysis) can be prioritized. The summary report gives the programmer a

comprehensive analysis of the failure. An example of a report (as used in the user study) can be seen in Table 4.6.

4.3.2 Protocol Extensions and Variations

The above protocol and diagnostic techniques discussed above provide good results for diagnosis, and indeed represent what has been implemented for evaluation. However, especially for more specific cases, there could be many potential variations. There are many bug diagnosis techniques, both existing and as of yet not-proposed, which could be added. For instance, information flow tracking [99, 116] can reliably detect "bad behavior" caused by inappropriate use of user-supplied data. Also, the diagnosis order can be rearranged to suit specific diagnosis goals or specific applications. For example, input testing could be done for nondeterministic bugs. Or, for some applications, some steps could be omitted entirely (e.g. memory bug detection may be skipped for programs using a memory safe language like Java). To extend the protocol, all that is necessary is to know what inputs the tool needs (e.g. a set of potentially buggy code lines), what priority it is (e.g. low-cost tools are high priority), and what outputs it generates (e.g. the failing instruction). Alternatively, a protocol may be custom-designed for a particular application and include application-specific tools (say, a log analyzer for DB2).

The dynamic backward slice and the results from delta analysis can be combined through intersection. That is, Triage could consider to be more relevant those portions of the backward slice which are also in the path delta (see Section 4.5). This will not only identify the code paths which are possibly in the propagation chain, but highlight those which differ from normal execution.

Triage may attempt to automatically fix the bug. Quite a large amount of information is at hand after Triage finishes its analysis. In a straightforward manner, Triage can begin automatically filtering failure-triggering inputs (e.g. as described in Chapter 3, or as in [18, 63, 137]), to avoid triggering the bug in the future. With a higher degree of risk it may be

possible to generate a patch. Currently Triage can tentatively identify heap-buffer overflows and repair them, by instrumenting the calculation for how large the buffer must be allocated. More details on these preliminary patches are detailed in 4.4. Finally, since the goal is merely to gather diagnostic information, Triage can attempt quite "risky" fixes, such as dynamically deleting code or changing variable values, in an attempt to see which changes will prevent failure during replay. Such speculative techniques that were proposed for recovery, such as failure oblivious computing [112] or forcing the function to return an error value proposed in STEM [123], can also be borrowed here. While those techniques can be very risky when used for recovery, they are fine for diagnosis purposes, since all side-effects of any replay are discarded during the diagnostic process.

4.4 Delta Generation

A key and useful technique commonly used by programmers when they manually debug programs, is to identify what differs between failing and non-failing runs. Differences in terms of inputs, execution environments, code paths and data values can help programmers narrow down the space of possible buggy code segments to search for the root cause. Triage automates this manual, time-consuming process using a delta generation technique, which (through the runtime system support for re-execution) captures the failure environment and allows automatic, repetitive delta replays of the recent moment prior to the failure, with controlled variation and manipulation to execution environment.

Delta generation has two goals. The first goal is to generate many similar replays from a previous checkpoint, some of which fail and some of which do not. During each replay, much detailed information is collected via dynamic binary instrumentation to perform the next step — delta analysis.

Second, from those similar replays, the delta generator identifies the signatures of failure-triggering inputs and execution environments, which can be used for two purposes: (1) report

to programmers to understand the occurred failure and efficiently find a way to reproduce the failure; and (2) guide the online failure recovery or security defense solutions by actively filtering failure triggering inputs like in Vigilante [33], Autograph [63], Sweeper [137] and others [124, 65, 104], or avoiding failure triggering execution environments like in the Rx recovery system [108].

To achieve the above goals, the delta generator automatically and repeatedly replays from a recent checkpoint, with controlled changes in input and execution environment (including potentially forcefully modifying control flows, variable values, etc). Thus, it obtains closely related failing and non-failing runs.

Changing the Input (Input Testing) If a program is given a different input (client request stream for servers), in most cases it will have a different execution. If it is given two similar inputs, then one would expect that the executions would also be similar. Furthermore, if one input fails and one succeeds, the differences in the executions and in the inputs should hold insights into the failure. It is this idea that motivates the previously proposed delta debugging idea [155], an *offline* debugging technique for isolating minimally different inputs which respectively succeed and fail in applications such as gcc.

So inspired by offline delta debugging, Triage automates this process and applies it to server applications by replaying client requests through a network proxy (see Section 4.2). The proxy extracts requests as they arrive and stores them for future replay. Since Triage is meant for the end-user's site, it can leverage the availability of real workloads. After a failure, the input tester searches for which input triggers the failure by replaying a subset of requests during re-execution from a previous checkpoint. If the failure is caused by combinations of requests, finding minimal triggers can be done by applying hierarchical delta debugging [84].

Besides identifying the bug-triggering request, the input-tester also tries to isolate the part of the request that is responsible for the bug. It does this in a manner reminiscent of data fuzzing [126], "twiddling" the request, to create a non-failing request with the "min-

imum distance" from the failing one. For well-structured inputs, like HTTP requests, the difference between inputs can be as little as one character, and can generate highly similar executions. This maximizes the usefulness of the later delta analysis step. Finally, if the specific triggering portion of the input is known, Triage can create a "normal form" of the input. This can address user's privacy concerns, since their actual input need not be reported.

Changing the Environment If a program is executed in a different environment such as memory layout and thread scheduling, then execution could also be different. This can be done artificially by modifying the execution environment during re-execution. There are several known techniques proposed by previous work such as Rx [108] and DieHard [9]. Some examples include allocating new buffers in isolated locations, padding or zero-filling new allocations, changing scheduling and message orders, etc. Triage applies these techniques to generate different replays even from the same input. Unlike the previous work, Triage is not randomly twiddling with the environment for recovery purposes, but rather to generate more failing and succeeding executions. Further, unlike in Rx, Triage already has some idea about the failure based on earlier failure analysis steps. It can target its perturbations directly at the expected fault. For example, for a deterministic failure, Triage does not attempt different thread scheduling. Similarly, given we know a particular buffer has overflowed, we specifically target its allocation, rather than blindly changing all allocations. Moreover, a non-recovery focus implies correctness is no longer an overriding concern, and Triage can exploit some speculative changes described below.

Speculative Changes (preliminary) Execution perturbation during replay can be speculative since all side-effects during replay are sandboxed and discarded. For example, during replay, Triage could force the control flow to fall through a non-taken branch edge. It could also forcefully change some key data's value during replay. The new value could be some

common value in non-failing runs (generated by the input tester). Such changes clearly violate the semantic correctness requirements of Rx; however, they may be useful for diagnosis.

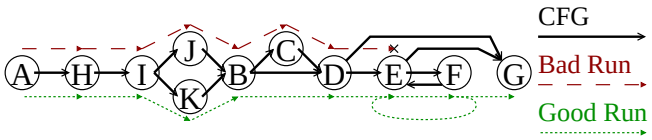


Figure 4.5: Control flow graph and two executions of our running tar bug example shown in Figure 4.4

Result of Delta Generation The result of delta generation is a set of many similar failing and non-failing replays. To feed into the next stage, delta analysis, Triage extracts a vector of the exercise counts of each basic block (the basic block vector) and a trace of basic blocks from each replay. Alternatively, the granularity could be increased to the instruction level or reduced to the level of function calls. Further, both instruction- or function- level granularity could include or exclude data arguments. Finer granularities capture more detail, but also introduce more noise. For general use, the basic-block level is a good trade-off.

Figure 4.5 shows the control-flow graph of our running bug example bug, with a failing run and a non-faining run superimposed. The good run visits basic blocks AHIKBDEFEF...EG, while the bad run visits blocks AHIJBCDE, and then fails. The good run has a basic block vector of {A:1, B:1, D:1, E:11, F:10, G:1, H:1, I:1, K:1}, while the bad run has {A:1, B:1, C:1, D:1, E:1, H:1, I:1, J:1}.

4.5 Delta Analysis

Based on the detailed information from many failing and non-failing replays produced by the delta generator, the delta analyzer examines these data to identify failure-related code, variables and the most relevant fault propagation chain. It is conducted in three steps:

(1) Basic Block Vector (BBV) comparison: Find a pair of most similar failing and non-failing

replay, S and F, using a basic block vector (BBV) comparison and also identify those basic blocks unique to failing runs—suspects for root causes.

(2) Path comparison: Compare the execution path of S and F and get the difference in the control flow path.

(3) Intersection with backward slice: Intersects the above difference with dynamic backward slices to find out those differences that contribute to the failure.

Basic Block Vector (BBV) Comparison For each replay produced by the delta generator, the number of times that each basic block is executed during this replay is recorded in a basic block vector (BBV). This information is collected by using dynamic binary instrumentation to instrument before the first instruction of every basic block.

The first part of the BBV comparison algorithm calculates the Manhattan distance of the BBVs of every pair of failing replay and non-failing replay and then finds the pair with the minimum Manhattan distance. The computation is not expensive for a small number of failing and non-failing replays. If needed, one could also trade-off accuracy for performance since it is not necessary to find the absolute minimum pair—as long as a pair of failing and non-failing replays are reasonably similar, it may be sufficient.

In the running example shown on Figure 4.5 (which only has 2 replays), the BBV difference between the two replays is {C:-1, E:10, F:10, G:1, J:-1, K:1}; the successful replays makes many iterations through the EF loop, and does not execute C or J at all. The Manhattan distance between the two would therefore be 24.

To identify basic blocks unique to failing replays (and thus good suspects for root causes), a more thorough BBV comparison algorithm could compute statistics (e.g. the mean and standard deviation) on each BBV entry. Performing significance tests between the means of the failing and non-failing replays in a way similar to PeerPressure [143] would allow a key question to be answered–is there a statistically significant difference between the exercise

count of each individual basic block? Currently, the implementation does not consider such tests.

Path Comparison While the BBV difference is helpful to identify basic blocks unique to failing runs, it does not consider basic block execution order and sequence. This limitation is addressed by Triage's path comparison. The goal with the path difference is to identify those segments of execution where the paths of the failing and non-failing replay diverge. The pair of failing and non-failing replays is the most similar pair identified by the BBV comparison. Similar to BBV, the execution path information is collected during each replay in the delta generation process. It is represented in a path sequence, a stream of basic block executed in a replay.

Given two path sequences (one from the failing replay and the other from the non-failing replay), the path comparison computes the *minimum edit distance* between the two, i.e. the minimum number of simple edit operations (insertion, deletion and substitution) to transform one into the other. Much work has been done on finding minimum edit distances; Triage uses the $O(ND)$ approach found in [87]. The path comparison algorithm also records these edit operations that give the minimum edit distance between the two path sequences.

In the running example, the minimum edit distance between AHIJBCDE (failing) and AHIKBDEFEF...EG (non-failing) would be found. In a modified sdiff format, this is:

```
A H I K B   D E F E F ... E G
      -   v     ^ ^ ^     ^ ^
A H I J B C D E
```

This demonstrates the difference in program execution: the failing replay takes branch J instead of K, while the non-failing replay takes an extra block C, and is truncated prior to the EF loop.

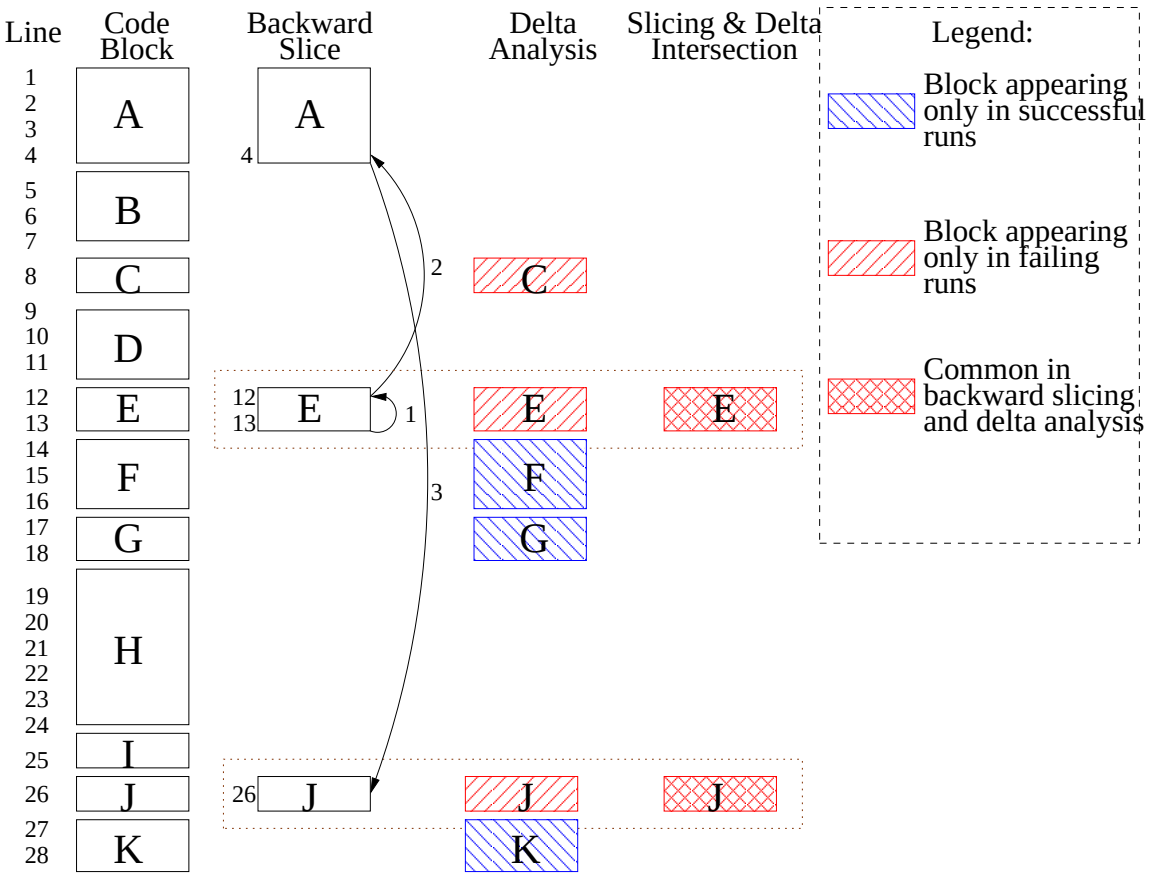


Figure 4.6: An example presenting backward slicing, delta analysis results and their intersection. It is derived from Figure 4.4.

Backward Slicing and Result Intersection To further highlight important information and eliminate noise, those path differences which are related to the failure (i.e. in the fault propagation chain) are extracted. This can be achieved by intersecting the path difference with the dynamic backward slice, which is a program analysis technique that extracts a program slice consisting of all and only those statements that lead to a given instruction's execution [147]. The intersection results can help focus attention on those instructions or basic blocks that are not only in the fault propagation chain but also are unique to failing replays.

As shown in Figure 4.6, for a given instruction (the starting point of a backward slice), its data or control dependent codes are extracted and a lot of irrelevant codes (shown in Figure 4.4), are discarded. This greatly reduces the amount of noisy information that is

irrelevant to the occurred failure. In our tar example, only lines 12, 4 and 26 belong to the dynamic backward slice of the failing instruction (line 13). This information, beyond being useful to refine delta analysis, is useful to the programmer. Therefore, the Triage report includes the the whole backward slice.

Unfortunately, backward slicing is non-trivial to apply to productions runs. First, backward slicing requires a starting point from which to slice; this would usually be supplied by the programmer. In Triage, the results of other stages of analysis (see Figure 4.3) are substituted for this human guidance.

Additionally, backward slicing incurs large time and space overheads, and therefore has seldom been used during production runs. In Triage, the overhead problem is addressed in two ways. First, the re-execution support makes the analysis *post-hoc*: backward slicing is used only during replays after a failure occurs, when the overhead is no longer a major concern. By using forward computation backward slicing [156] Triage can dynamically build dependence trees during replay and need not trace the application from the beginning. As a further optimization, Triage applies a function call summary technique to certain known library calls. For some select library calls just one dependency, "return value depends on input arguments", is recorded. This greatly reduces the overhead for some commonly-called library functions. The experiments show that the resulting total analysis overhead is acceptably low.

Returning again to the running example, the difference between the two replays lies in the blocks {+J, -K, +C, +E, -F, -G}, $+J$ meaning that block J either appears only at the failing run or contains the failing instruction and $-K$ meaning that K appears only at the successful run. In the backward slice, F, G, and K do not appear at all, while J is very close on the potential propagation chain. Considering these {E, J, C, K, F, G}, the key differences can be ranked to the very top: the null pointer dereference in E is the failure, and the `return NULL;` statement in J along with the `entry=disp;` assignment in E are very important factors in the fault. Therefore, the two most relevant to the failure basic blocks,

as shown in Figure 4.6, are E and J. Normal differences caused by accident of execution that are far from the fault are ranked low, as they do not have a close impact on the fault itself.

Data delta analysis (unimplemented) It is conceivable that one could also compare the data values of key variables (e.g. branch variables) to complement the comparison in control flows. However, this method requires collecting too much information and also it is hard to statistically compare data of various types such as floating point variables, etc. Therefore, Triage does not perform any data delta analysis.

4.6 Other Diagnosis Techniques

Delta generation and delta analysis comprise stages 2 and 3 of the TDP (Figure 4.3). For stage 1 Triage also uses other diagnosis techniques. This section briefly describes these techniques.

Coredump analysis The Triage coredump analyzer reports the register state, what signal caused the fault, and basic summaries of the stack and heap state. The stack can be unwound starting from the current `ebp` register value. By checking whether each stack frame is indeed on the stack, and whether the return values point into the symbol table, Triage generates a call-chain signature and detect possible stack corruption such as stack smashing attacks. Heap analysis examines `malloc()`'s internal data structures, walking chunk-by-chunk through the block lists and the free lists. This identifies some heap buffer overflows. If the application uses its own memory allocator, an application specific heap consistency checker is needed. This step is extremely efficient (under 1s), and provides a starting point for further diagnosis.

Dynamic bug detection Triage can leverage many existing dynamic bug detection techniques to detect common bugs[3]. Currently, Triage employs two types of common dynamic bug detectors, a memory bug detector and a race detector, and only during re-execution via dynamic instrumentation [77] to address overhead concerns. Triage's **memory bug detector** (MD) detects memory misbehavior during re-execution to search for four types of memory errors: stack smashing, heap overflow, double free and dangling pointers using techniques similar to previous work [56, 94]. Once simple replay determines that a failure is nondeterministic, Triage invokes the **data race detector** to detect possible races in a deterministic replay. Triage currently implements the happens-before race detection algorithm [95] by instrumenting memory accesses with PIN; other techniques [117] would also certainly work.

4.7 Limitations and Extensions

Privacy policy for diagnosis report After failure diagnosis, Triage reports the diagnosis results back to the programmers. However, for some end-users, results such as failure-triggering inputs may still contain potentially private information. To address this problem, it is conceivable to extend Triage to allow users to specify privacy policies to control what types of diagnosis results can be sent to programmers. Furthermore, unlike a coredump, the information Triage sends back[4] is "transparent"–comprehending what is being sent in a Triage report, and verifying that nothing confidential is being leaked, is much easier than with a memory image.

Automatic patch generation Triage provides a wealth of information about occurring failures; this information has been used to automatically generate patches. However, without an understanding of the semantics of the program, success has been limited. For heap

[3]Note that dynamic bug detectors find errors, *not faults*, and while useful, other techniques (e.g. delta analysis) are needed to find root causes

[4]See, for example, Table 4.6.

Name	Program	App Description	#LOC	Bug Type
Root Cause Description				
Apache1	apache-1.3.27	web server	114K	Stack Smash
Long alias match pattern overflows a local array				
Apache2	apache-1.3.12	web server	102K	Semantic (NULL ptr)
Missing certain part of url causes NULL pointer dereference				
CVS	cvs-1.11.4	version control server	115K	Double Free
Error-handling code placed at wrong order leads to double free				
MySQL	msql-4.0.12	database server	1028K	Data Race
Database logging error in case of data race				
Squid	squid-2.3	web cache server	94K	Heap Buffer Overflow
Buffer length calculation misses special character cases				
BC	bc-1.06	algebraic language	17K	Heap Buffer Overflow
Using wrong variable in for-loop end-condition				
Linux	linux-extract	from linux-2.6.6	0.3K	Semantic (copy-paste error)
Forget-to-change variable identifier due to copy-paste				
MAN	man-1.5h1	documentation tools	4.7K	Global Buffer Overflow
Wrong for-loop end-condition				
NCOMP	ncompress-4.2.4	file (de)compression	1.9K	Stack Smash
Fixed-length array can not hold long input file name				
TAR	tar-1.13.25	tar archive tool	27K	Semantic (NULL ptr)
Directory property corner case is not well handled				

Table 4.1: Applications and real bugs evaluated in our experiments.

buffer overflows, Triage can identify the allocation point of buffer which overflows. Similar to Rx [108], Triage can apply padding; unlike Rx one particular allocation point is clearly identified. As Triage is not blindly extending every buffer, it can apply a permanent padding. Triage tries a linear buffer increase up to a cutoff, and then it tries a multiplicative increase. Although limited to a subset of heap buffer overflows, this technique does provide an adequate patch for the buffer overflow in Squid (see Section 4.9) which addresses all possible triggers. Triage is currently unable to create correct patches for an other bugs.

Bug handling limitations Of course, Triage is not a panacea. For some bugs, it may be difficult for Triage to provide accurate diagnostic information. For example, since Triage does not have information prior to the checkpoint, it is difficult to pinpoint memory leaks, although our coredump analysis may provide some clues about buffers that are not freed and

also no longer accessible. To address this may require new bug detection techniques that do very lightweight monitoring during normal execution such as sample-based monitoring [57] in order to help the heavy-weight instrumentation used during re-execution. Similarly, Triage is ineffective if no failures are detected. While many bugs lead to obvious failures, some bugs, especially semantic bugs, result merely in incorrect operation, sometimes in subtle ways. At the expense of normal run performance, failure detectors can help, but some failures will be quite difficult to automatically detect.

Another class of difficult bugs, although previous work [91] reports they are rare, is bugs that take a long time to manifest. To diagnose such failures, Triage needs to replay from very old checkpoints. Rolling back to old checkpoints is not a problem since Triage can store old checkpoints to disk, given sufficient disk space. The challenge lies in quickly replaying long windows of execution to reduce diagnosis time.

Reproduce nondeterministic bugs on multiprocessor architectures The current prototype of Triage only supports deterministic replay of both single- and multi- threaded applications on uniprocessor architectures. For multiprocessor architectures, replay of multi-threaded applications is on a best-effort basis. Although this works well enough for most bugs, deterministic replay may be necessary for some faults. It is very difficult to support this functionality with low overhead in multiprocessor architectures. Hardware support, such as Flight Data Recorder [152] or BugNet [91], may be necessary to achieve sufficiently low overhead. Alternative techniques, such as output deterministic replay [3], are available to trade off record-time performance for replay time performance, although currently only at a swingeing replay time penalty.

Deployment on highly-loaded machines Triage imposes negligible overhead in the normal-run case. However, it does expend significant resources during analysis. Although the optimal case is to perform diagnosis immediately after the failure in the exact same

environment, there are cases where this is infeasible. To alleviate this, there are several possibilities. First, diagnosis can occur in the background, while normal activities (or even recovery) continue. It may even be deferred until a later time when resources are available. Second, one possibility would be to perform the analysis on a separate machine, albeit one still at the user's site; this would require extending Triage to include process-migration support. Finally, it may be acceptable to skip the more expensive analysis steps; although they are useful, it is better to get something than nothing. Regardless, it is always the intent that analysis should be done at the end-user's site, and that only results should be sent back to the programmers.

Handle false positives Even though in the experiments Triage never reported misleading information, it is conceivable that in some rare cases Triage may report incorrect diagnosis results due to the false positives introduced by some specific diagnosis techniques. This problem can be addressed by performing more sophisticated consistency checks among results produced by different diagnosis techniques and also incorporate the accuracy of each technique into the result confidence ranking.

4.8 Evaluation Methodology

To evaluate Triage, this section presents various experiments using 10 real software failures with 9 applications (including 4 servers) as well as a user study with real programmers. Triage is implemented in the Linux operating system, version 2.4.22. Various diagnosis techniques are implemented on top of a dynamic binary instrumentation tool, PIN [77]. After a failure occurs, Triage dynamically attaches PIN to the target program in the beginning of every re-execution attempt.

Machine environment and parameters The experiments were conducted on single-processor machines with a 2.4GHz Pentium-4, 512KB L2 cache, and 1GB of memory. Server

application experiments use two such machines connected with 100Mbps Ethernet; the server runs on one and the client runs on the other. By default, Triage keeps twenty checkpoints, and checkpoints every 200ms.

Evaluated Applications and Failures Table 4.1 shows the 9 applications (4 server and 5 open source utilities) and 10 bugs that were evaluated. This suite covers a wide spectrum of representative applications and **real** software failures. The software failures are segmentation faults or assertion failures, with the underlying defects belonging to different categories: semantic bugs (2 null pointer and 1 copy-paste), memory bugs (2 stack smashing, 2 heap overflow, 1 static buffer overflow, and 1 double free) and 1 data race bug. The error propagation distances also vary among these applications.

User Study To validate that Triage reduces programmer effort in fixing bugs, I conducted a user study. The study used 5 fail-stop bugs: 3 toy programs with injected bugs, and two real bugs (the bugs in BC and TAR). Participants were asked to fix the bugs as best as they could. They were provide a controlled workstation with a full install of Fedora Core 6 including a full suite of programming tools, including Valgrind. To balance the difficulty of the bugs, we randomly selected 50% of the programmers to diagnose each bug with the error reports produced by Triage, and, as a control, the remaining 50% without the reports. Aside from formatting, Table 4.6 is precisely the report given in the TAR case. All participants were given a coredump, sample good and bad inputs, a prepped source tree, and instructions on how to replicate the bug; although this eliminated the difficulty of replicating the bug for the non-Triage case, it was necessary to bring the task down to an achievable difficulty. Further, there was a half hour time limit per real bug and 15 minute time limit per toy bug; failure to fix the bug resulted in a recorded time of the full limit[5]. The time limits were necessary for practical purposes; participants averaged approximately two hours of total

[5]These time limits artificially show Triage in a bad light because they bound the maximum time, and this improves the reported performance of the non-Triage cases.

time each.

As it took almost 6 months to get legal approval to use human subjects in the study, the experiments include only 10 programmers, drawing from local graduate students, faculty, and research programmers. No undergraduates were used. All of the subjects indicated that they have extensive and recent experience in C/C++. We tested statistical significance using a 1 sided paired t-test [131, 113]. This test compares each subject against themselves, to help account for individual programmer skill; the variation of individual programmer skill still appears in the overall means.

4.9 Experimental Results

Table 4.2 presents a summary of Triage's diagnosis results for each failure for server bugs, while Table 4.3 presents a summary for the non-server bugs. For the four deterministic server bugs, the table presents results from input testing/delta generation, delta analysis, and backward slicing. For the nondeterministic bugs, the table presents the results from schedule manipulation. Finally, for the five application bugs, the results do not provide input testing, and delta analysis is only provided for BC and TAR.

In all cases, Triage correctly diagnosed the nature of the bug (deterministic or nondeterministic), and in all 6 applicable cases Triage correctly pinpointed the bug type, buggy instruction, and memory location. Hence, Table 4.2 and Table 4.3 omit detailed listings of Stage 1 results. To summarize Stage 2 and 3 results, for all the 5 server applications, Triage successfully captured and reproduced the fault-triggering input. Also, for the cases where delta analysis is applied, it reduced the amount of execution that must be considered by 63%; for the best case (the BC application) it reduces it by 98%. In 8 of the 10 cases, the root cause instruction appeared within the top 10 failure-relevant candidate instructions and in the other 2, within the top 10 failure relevant functions. Finally, of note is that for the nondeterministic MySQL bug, Triage found an example interleaving pair which is the

Server Applications			
App	**Results - Stages 2 & 3**		
	Method	**Results**	**Useful?**
Apache1	Input Testing	*GET /trigger/crash.html ...*	Yes
		Key part: */trigger/crash.html*	Yes
	Backward Slicing	Found root-cause ***line***	Yes
		8 instructions from crash	Yes
	Delta Analysis	Edit distance is 79089	Yes
		Removes 12%	Yes
Apache2	Input Testing	*GET ... Referer:1.2.3.4*	Yes
		Key part: *Referer:*	Yes
	Backward Slicing	Found root-cause ***line***	Yes
		3 instructions from crash	**Yes**
	Delta Analysis	Edit Distance is 5964	Yes
		Removes 69%	Yes
CVS	Input Testing	*Stream of requests...*	Yes
	Backward Slicing	Found root-cause ***function***	Yes
		4 functions from crash	Yes
	Delta Analysis	No result	No
		Not applicable	No
MySQL	Schedule Manipulate	Bad interleaving pair: 0x8132fa8 – 0x8128c4b	**Yes**
		Found root-cause	**Yes**
Squid	Input Testing	*ftp://user\ *30:p@...*	Yes
		Key parts: *ftp, user*	Yes
	Backward Slicing	Found root-cause ***line***	Yes
		6 instructions from crash	Yes
	Delta Analysis	Edit distance is 54310	Yes
		Removes 71%	Yes

Table 4.2: Diagnosis results. "U" means whether each piece of failure information is useful—'Y' for useful and "N" otherwise. **Y** indicates exceptionaly good results. For all of the bugs, Stage 1 (identify failure/error types and locations) works admirably.

trigger for the failure.

4.9.1 Triage Report Case Studies

This section uses three case studies to show how Triage reports can help developers understand failures.

Other Open-Source Applications			
App	Results - Stages 2 & 3		
	Method	Results	Useful?
BC	Backward Slicing	Found root-cause ***line*** 3 instructions from crash	Yes **Yes**
	Delta Analysis	Edit distance is 5381 Removes 98%	Yes **Yes**
Linux-extr.	Backward Slicing	Found root-cause ***line*** 6 instructions from crash	Yes Yes
	Delta Analysis	No result Not applicable	No No
MAN	Backward Slicing	Found root-cause ***function*** 9 functions from crash	Yes Yes
	Delta Analysis	No result Not applicable	No No
NCOMP	Backward Slicing	Found root-cause ***line*** 5 instructions from crash	Yes Yes
	Delta Analysis	No result Not applicable	No No
TAR	Backward Slicing	Found root-cause ***line*** 6 instructions from crash	Yes Yes
	Delta Analysis	Edit distance is 83564 Removes 68%	Yes Yes

Table 4.3: Diagnosis results. "U" means whether each piece of failure information is useful—'Y' for useful and "N" otherwise. **Y** indicates exceptionaly good results. For all of the bugs, Stage 1 (identify failure/error types and locations) works admirably.

Case 1: Apache The bug in Apache 1.13.27 is a stack related, difficult to reproduce and diagnose bug. As shown in Table 4.4, the failure occurs at a call to function `ap_gregsub`. Coredump analysis informs us an invalid pointer dereference at variable `r`. However, `r` was correctly dereferenced a few lines before, and is not changed throughout the function. Fortunately, Triage's bug detector catches a stack-smash in function `lmatcher`, `engine.c:212`. This is useful, however, there are some confusing wrinkles: (1) the application fails *before* function `try_alias_list` returns, which means the overwritten return address is NOT the reason for the failure; and (2) there is no obvious connection between `lmatcher` and `try_alias_list`. How `lmatcher` can smash the stack of `try_alias_list` is unclear.

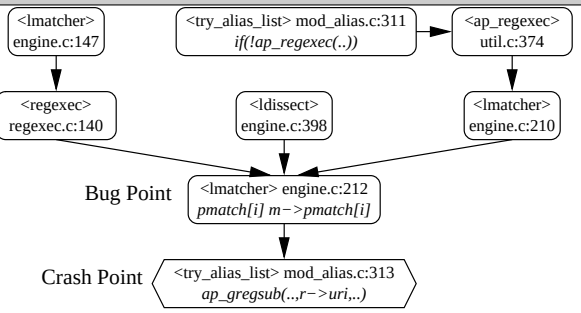
Triage Report for Application — Apache	
Failure Point Information	
Segment Fault (at)	Instr:0x805e33f@ mod_alias.c:313
Stack / Heap?	**Corrupt** / OK
Bug Detection Information	
Deterministic Bug?	Yes
Stack-smash (at)	engine.c:212
Fault Propagation Information	
	
Failure Trigger Information	
Failure-triggering input:	GET /trigger/crash.html HTTP/1.1 ...
Critical part:	GET /trigger/crash.html HTTP/1.1 ...
Normal-form:	GET /trigger/crash.html HTTP/1.1 ...
Close non-failing inputs:	GET /trigger/ HTTP/1.1 ...
	GET / HTTP/1.1 ...

Table 4.4: Triage report for Apache-1.3.27 version

The fault tree and the path differences provided by Triage's delta analyzer and the backward slicer clears up the above confusions. Tracing from the root, the edge from `engine.c:212` to *root* indicates the crashing function call gets pointer variable `r`'s value from the assignment in the stack-smash statement (`engine.c:212`). This explains the failure: the stack-smashing overwrites the stack frame(s) above it, and invalidates pointer variable `r`, an argument of function `try_alias_list`. Tracing further back, we can identify that this function is called by `try_alias_list` via a function pointer. The destination, `pmatch[i]` in `engine.c:212`, is a fixed length stack array declared in `try_alias_list`. It is filled in by function `ap_regexec` without bounds check (`mod_alias.c:311`).

The input testing in Triage's delta generator in this case identifies that the failure is independent of the headers of the request and also that the failure is triggered by requests for a very specific resource (*/trigger/crash.html*).

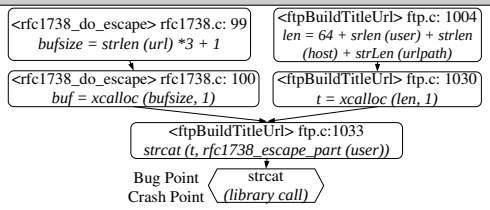
Triage Report for Application — Squid	
Failure Point Information	
Segment Fault (at)	Instruction: 0x4f0f0907 (in lib. *strcat*) called from ftp.c:1033
Stack/Heap?	OK/**Corrupt**
Bug Detection Information	
Deterministic Bug?	Yes
Heap Overflow (at)	lib. *strcat* called from ftp.c:1033
Fault Propagation Information	
	
Failure Trigger Information	
The failure triggering input was: ftp://user\ (repeat 43 times):password@ftp.slackware.com Trigger-critical parts: *protocol*,*username* Normal-form of failure-triggering input: ftp://user\ (repeat 30 times):p@ftp.slackware.com Similar but not-failure-triggering inputs: ftp://user\ (repeat 29 times):password@ftp.slackware.com http://user\ (repeat 43 times):password@ftp.slackware.com	

Table 4.5: Triage report for Squid 2.3-Stable5 version

Case 2: Squid As shown in Table 4.5 coredump analysis indicates that Squid probably has a heap overflow triggered by a call to `strcat` from `ftp.c` line 1003. Triage's memory bug detector confirms this, catching a heap-overflow bug at said point. It is fairly certain that the failure is caused by a heap-overflow of buffer `t` in `ftp.c`, line 1003. The fault propagation tree shows us how this happens: a `strcat` of two buffers, one returned from `rfc1730_escape_part`, and `t` from `ftpBuildTitleUrl`. It also shows how these buffers were allocated; in the left branch we multiply `strlen(url)` by 3 while in the right branch we simply add the length `strlen(user)` (which is passed as `url`) to some other numbers. This is the root cause: it is possible for the buffer returned by `rfc1730_escape_part` to be three times longer than expected (if there are many characters that need escaping), while the `strcat` only can deal with 64 extra characters.

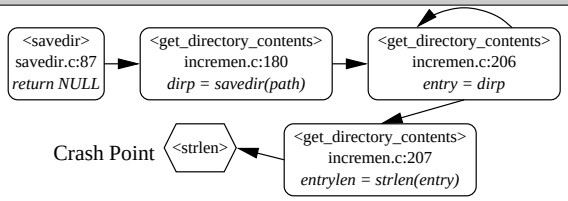
Triage Report for Application — Tar	
Failure Point Information	
Segment Fault (at)	Instr: 0x4f0f1e13 (in lib. *strlen*) called from increment.c:207
Stack/Heap?	OK/OK
Bug Detection Information	
Deterministic Bug?	Yes
Null Pointer (at)	lib. *strlen*, called from incremen.c:207
Fault Propagation Information	
	

Table 4.6: Triage report for tar-1.13.25

Finally, input testing, provides the actual request that triggered the failure. It is an *ftp* request, where the *username* has 43 instances of “

”. Furthermore, it identifies the *normal-form* of the bug triggering request, one with 30 repetitions of “

” in the *username* field, and a minimally different non-failing request, where there are only 29 repetitions.

Case 3: tar Case study three is the running example (see Figure 4.4). Briefly, Table 4.6 shows the output of Triage on this bug. Since previous sections have discussed this bug, it is not explained further here. Of note is that the figure shows exactly the same information provided in the user study; the only difference is that the user study report doesn't suffer from space limitations and hence has a loose format.

4.9.2 Normal Execution Overhead

Triage imposes negligible overhead during execution; it should be nearly indistinguishable from the overhead of the underlying checkpoint system [108]. Figure 4.7 shows the results for three applications: Squid (network bound), BC (CPU bound), and MySQL (both). To

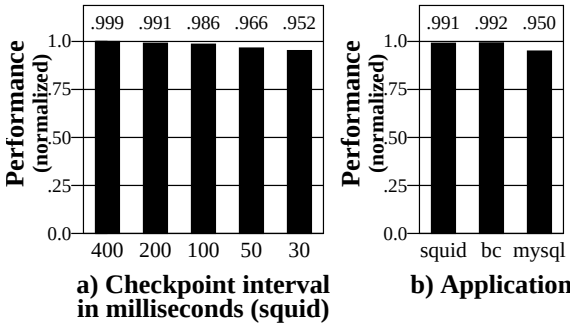


Figure 4.7: Normalized performance during normal-time execution. a) shows squid at checkpoint intervals from 400 ms to 30 ms, while b) shows squid, MySQL, and bc at 200ms intervals.

explore the effects of checkpoint interval, squid is also run at checkpoint intervals from 400 to 30 ms. In no case is the overhead during normal runtime over 5%. For the 400ms checkpoint interval, the overhead drops to 0.1%. Given such low overhead, Triage is acceptable during normal execution. This is because Triage only runs analysis *after* a failure has occurred.

4.9.3 Diagnosis Efficiency

With the exception of delta analysis, Triage's diagnosis is very efficient: all diagnostic steps finish within 5 minutes, when running in the foreground. Table 4.7 lists the diagnosis time break down for three representative applications, i.e. an IO & network-bound application, *apache*; a CPU-bound application, *bc*; and a network-bound application, *squid*. Among the different diagnosis components, delta analysis takes the longest time, because it examines every single basic block. For tasks with very small deltas (like BC), it is efficient. If the edit distance becomes large, the D (edit distance) term in the $O(ND)$ complexity becomes expensive. Also, for the *apache* and *squid* bugs chosen, the larger D causes high memory pressure; more complex implementations of the edit distance algorithm have much better space efficiencies [87]. However, given their expense, the path comparison stage of delta analysis as well as backward slicing are top candidates to be run in the background (or on a different machine) to avoid interfering with foreground tasks.

App.	Total	Component Diagnosis Time				
		Core-Dump	Input Test	Bug Detect	Slicing	Delta Anal.
Apache1	68 s	0.06 s	9 s	14 s	45 s	27 m
BC	303 s	0.03 s	0 s	98 s	205 s	9 s
Squid	145 s	0.04 s	7 s	30 s	108 s	64 m

Table 4.7: Triage failure diagnosis time, in seconds (s) and minute (m). Total excludes delta analysis.

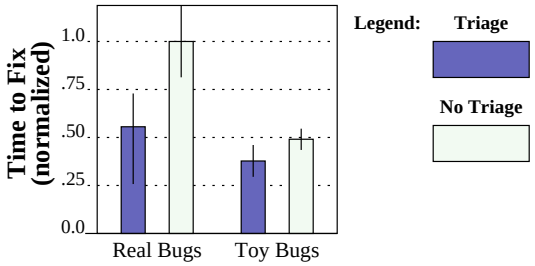


Figure 4.8: Results from user study, with error bars showing 95% confidence intervals. Normalization is to real bugs without Triage

4.9.4 User Study

The user study (described in Section 4.8) has demonstrated very positive results. As shown in Figure 4.8, on average programmers ***took 44.7% less time*** debugging the real bugs when they had the diagnostic information provided by Triage (13.4 ± 5.7 minutes versus 24.1 ± 4.5 minutes). A paired t-test shows that this is significant at the **99%** confidence interval ($p = .00141$), indicating that *the hypothesis that Triage reduces the time to fix bugs is very strongly supported by the data.* The results for the toy bugs are less, as programmers saved 22.9% (9.117 ± 1.991 minutes versus 11.831 ± 1.342 minutes), with significance at 95% confidence ($p = .03830$); although Triage still helped, the effect was not as large since the toy bugs are very simple and straightforward to diagnose without Triage. To verify, a statistical measure for size of impact, Cohen's d [29], is used. Values of this measure of $d = .2$, $d = .5$, and $d = .8$ are considered small, medium, and large effects respectively. For the real bugs, we get $d = 1.29$; for the toy bugs we get $d = .99$. Both of these are considered large effects, although the effect for the toy bugs is not as overwhelmingly large as for the real bugs.

Less formally, the study participants reported that the Triage reports were a significant

aid in helping them understand the bugs. By observation, the BC bug was particularly tricky; several of the control group went on time consuming wild goose chases through auto-generated parser code which, although close in time to the bug, was unrelated. In contrast, one participant commented "[the Triage report] pointed out the error right away. Most of my time was spent in getting the program to compile and run."

Overall, *Triage has a large, statistically significant effect on programmers' diagnosis time.* While there are many factors that can affect the accuracy of the user study (sample representativeness, sample set size, etc.), these results still provide strong evidence about the usefulness of Triage in helping programmers diagnose software failures.

4.10 Related Work

Chapter 2 provides a more thorough coverage of the related work; this section briefly covers closely related work and how it relates to Triage.

Software Failure offline diagnosis As discussed in Section 4.1, most existing software failure diagnosis focuses on offline tools that provide some assistance but still rely heavily on programmers to manually determine the root cause of a failure. Such tools include interactive debugging tools [53], program backward and forward slicing [1, 147, 156], deterministic replay tools [64, 128], and delta debugging techniques [84, 155] (described briefly in Section 4.4).

Triage has a two-fold relation to the above work. First, Triage differs by focusing on *onsite* diagnosis during production runs at *end-user sites*. Therefore, it must be fully automatic and impose low overhead during normal execution; many of the above techniques do not satisfy these constraints. Second, Triage can incorporate many of the above techniques and bypass their high overheads by employing them only during diagnostic replay.

Onsite Failure Information Collection. Most existing work on onsite failure information collection has been discussed in 4.1.2. While these techniques are helpful for postmortem

analysis, they are still limited, leaving the majority of the diagnosis task to programmers. Moreover, these coredumps or execution traces may not be made available by end-users due to privacy and confidentiality concerns.

Triage differs from and also well compliments the above work because it provides *automatic*, onsite failure diagnosis *at the end-user site*. When a failure occurs, Triage automatically follows the human-like, top-down error diagnosis protocol without any user or programmer involvement. Moreover, by performing the diagnosis right after a failure at the end-user site, Triage can make an effective usage of all failure information without violating the end user's privacy concerns.

Dynamic Software Bug Detection This work is related to and well complemented by dynamic software bug detection tools, such as Purify [56]. While these tools effectively detect certain types of bugs during in-house testing, most of them impose large overheads (up to 100X slowdowns) unsuitable for production runs on end-user sites. Fortunately, by using the Triage framework, many of these tools can be dynamically plugged in as needed during diagnostic replay after a failure occurs, when overhead is no longer such a concern.

Moreover, Triage goes beyond dynamic bug detection. It also uses other error diagnosis techniques like the input tester, environmental manipulator, delta generation, delta analyzer, coredump analyzer and backward slicer to collect more diagnostic information. It is important to fully understand a failure since the errors detected by dynamic bug detectors are not necessarily root causes [1, 147, 156].

Checkpointing and Re-execution Triage is related to previous checkpointing system such as Zap [101], FlashBack [128], Rx [108], and TTVM [64], most of which are used for recovery or interactive debugging. Triage uses checkpoint, rollback, and re-execution for a very different purposes—onsite software failure diagnosis. Different design goals lead to several major, important differences in research challenges, design and implementations issues.

Among the many differences, the most significant one is that the proposed project needs to perform various failure analysis to obtain failure information and find clues about production-run failures onsite. As discussed in Section 4.2, even for checkpoint and re-execution, our proposed project has different requirements, namely all side effects are sandboxed, no need to deal with output commit problems and allowing speculative re-execution such as forcefully skipping code and modifying variables.

Distributed Systems Fault Localization Recently some research efforts have been devoted to pinpointing faults (failures [25] and performance problems [2]) in distributed systems. These techniques support onsite diagnosis but the granularity of the fault information provided is much coarser (usually at component level) than what Triage provides. Triage complements these tools to provide more detailed diagnosis.

Advanced Input Filtering Triage's input tester can be further enhanced by using recently proposed advanced techniques in automatically generating input/execution filtering [18, 63].

4.11 Conclusions

This chapter presents Triage, an approach for diagnosing software failures. By leveraging lightweight checkpoint and re-execution techniques, Triage captures the moment of a failure, and *automatically* performs diagnosis at the end user's site. This chapter proposes a *failure diagnosis protocol*, which mimics the debugging process a programmer would take. Additionally, this chapter proposes a new online diagnosis techniques, delta analysis, to identify failure-triggering conditions, related code, and variables. Beyond onsite diagnosis, Triage is also helpful for in-house debugging. By performing the initial steps speedily and automatically, Triage can free programmers from some labor intensive parts of debugging.

Chapter 5

Delta Execution

This chapter presents delta execution[1], or Δ execution. Delta analyis (Section 4.5) showed that two runs of the same program with different input could have very similar traces of instruction streams. Δ execution builds on this, and attempts to leverage the potential of a high level of similary in the instruction streams of two different versions of the same program, for the purpose of online patch validation.

5.1 Overview

Although the rapid diagnosis of software faults is important, we have to wonder why programs are so buggy in the first place. Programs are not like machines or buildings; they do not suffer from rust or physical wear. "Bits don't rot". Hence all of the faults we find in programs are design faults, introduced when the programmer made a change to the code. If we never made changes to software, except to fix faults, we would expect that over time the software would become fault free. However, software is constantly changed to add new features or improve performance. Further, even the changes made to correct faults commonly introduce new ones [7]. This is especially problematic, since patches to fix security issues are common (over 7600 as per CERT in 2007 [23])). Every one of those changes is an opportunity to introduce a new fault, and cause new failures.

Failures caused by even supposedly well-tested patches commonly make the news. In

[1]This work is based on an earlier work: Efficient online validation with delta execution, in ACM SIGPLAN Notices - ASPLOS 2009, Volume 44, Issue 3, March 2009 (c) ACM, 2009. http://doi.acm.org/10.1145/1508284.1508267

2008, Microsoft had to withdraw the first service pack to Windows Vista because it caused "computers to crash or enter an endless cycle of boots and reboots" [81]. Also in 2008, Apple's "Leopard" update to Mac OS X broke applications [103]. Large operating system updates are not the only sort of patches to suffer such problems. In early 2009, Seagate went through a series of firmware patches that first caused drives to sporadically fail, and then to cause drives to fail on reboot, before a fourth firmware version finally addressed the issue [82]. Because patching is such a risky business, administrators do not want to be the first to apply a patch. Indeed, Microsoft's Patch Tuesday, when an entire month's worth of changes are released in one go, was created to give Microsoft additional time for testing and to give administrators a predictable schedule for validation and roll out [83]. While this unfortunately increases the window of vulnerability [61], it allows additional time to test security patches still increases uptime [7]. Even so, unusual end-user configurations can still cause problems, such as the 2010 security update which took some users' machines from being infected with a rootkit (bad) to being unbootable (worse) [14]. It is therefore unsurprising that administrators do not immediately apply patches if they can help it.

5.1.1 Patch Validation

Especially if we want a more rapid cycle between the initial discovery of a fault and the deployment of a fix (as in Chapters 3 and 4), we must have increased confidence that the changes programmers make are correct. As discussed in Section 2.4.1, there are two primary methods for patch validation; off-line or on-line. Off-line validation encompasses traditional regression testing; one sets up a system and sees if it fails. On-line validation involves running a production workload on the new version and comparing it to the old production instance. Although off-line validation is a necessary step, since it allows the vendor to detect problems before a patch is released, the greater variety of conditions a program will experience in the field on users sites suggests that on-line validation is more accurate. Specifically, on-line validation allows users to answer the question that they care about most: "*Does this patch*

*break **my** workload?*"

Despite on-line validation's promise of greater accuracy, it is rarely used. First, there is the trouble and expense of deploying two instances which can both support the full production workload. From a computational resources point of view, either the two instances can share resources (e.g. isolated by a VMM [75]) or on a completely separate set of hardware. Although the additional hardware cost may be affordable, secondary costs (e.g. power, cooling, floorspace, software & support licenses, etc.) can be significant, especially the cost of administrator labor. The administrator must configure a system that is nearly identical to the production instance, but differing just enough so that it doesn't interfere with the production instance. Maintaining a separate testing instance represents an expensive drain of human resources.

Second, and even more problematic, is that performing the comparison between the two instances is non-trivial [88]. Even if we run two identical copies side-by-side, non-determinism from things like thread interleaving, message timing, and random number generation will cause the outputs to differ. These spurious differences can be quite large, and make directly comparing the outputs infeasible. Instead, the comparison must be at a higher semantic level, which is error prone and troublesome. Because of these two problems, the strong evidence that on-line validation gives that a patch won't break when put into production is generally viewed as not worth the trouble.

5.1.2 Multiple Almost Redundant Executions

Although the challenges to on-line validation are large, the changes that typically need validating are not. Figure 5.1 shows a real bug fix in GNU tar. The only difference is in handling the special case of `dirp` being null; if `dirp` is null, then the main body of the loop is skipped. The patch is only two lines long, compared to the 100 lines of the main loop. In practice we expect the main loop to be executed because `dirp` will be non-null. One would therefore expect that the execution of the patched code and the original code will be almost

```
Patch: tar null pointer
tar/src/incremen.c

+ if (dirp) {
     if (children!=NO_CHILDREN) {
        for (entry=dirp; ... ; ...) {
           // main loop
           // 100 lines omitted
        }
     }
     free(dirp);
+ }
```

Figure 5.1: Illustrative bug fix patch extracted from GNU tar. '+' indicates code insertion.

```
Patch: CAN-2004-0493
httpd-2.0/server/protocol.c

+ if ((fold_len-1) > (r->server->limit)) {
+    r->status = BAD_REQUEST;
+    return;
+ }
```

Figure 5.2: Illustrative bug fix patch extracted from Apache httpd. '+' indicates code insertion.

```
Patch: CAN-2004-0811
httpd-2.0/server/core.c

  if (new->satisfy[i] != SATISFY_NOSPEC) {
     conf->satisfy[i] = new->satisfy[i];
+ } else {
+    conf->satisfy[i] = base->satisfy[i];
  }
```

Figure 5.3: Illustrative bug fix patch extracted from Apache httpd. '+' indicates code insertion.

identical almost all of the time. Even at the level of individual instructions, the common case will have the patched version running only two more: a compare and a conditional jump.

Such highly similar executions are called MAREs, for multiple almost redundant executions. That is, MAREs are more than one (multiple) executions that are very nearly identical (almost redundant) to one another, but not quite. These MAREs are common in patch validation, especially for security patches. Figures 5.2 and 5.3 show two security patches in the Apache web server that one would expect to usually not have much effect. Both insert a small amount of conditional code. Since security patches attempt to solve one specific problem, that they are small is unsurprising. Further, in the case of a security patch, the changes are often corner-case handling. Since most of the time the program is not being exploited, the corner cases do not commonly occur, and there is minimal effect on execution. Consider, for example, that adding a bounds check is the easiest way to deal with a buffer overflow. The patch is small, and if there is not an exploit attempt, nothing of note happens. Indeed, this is precisely what Figure 5.2 shows; in this instance, even the bounds to check had already been computed by previously included code. Together, their small size and exception-handling nature suggests that security patches will generally only lead to small changes in the execution.

5.1.3 Summary of Delta Execution

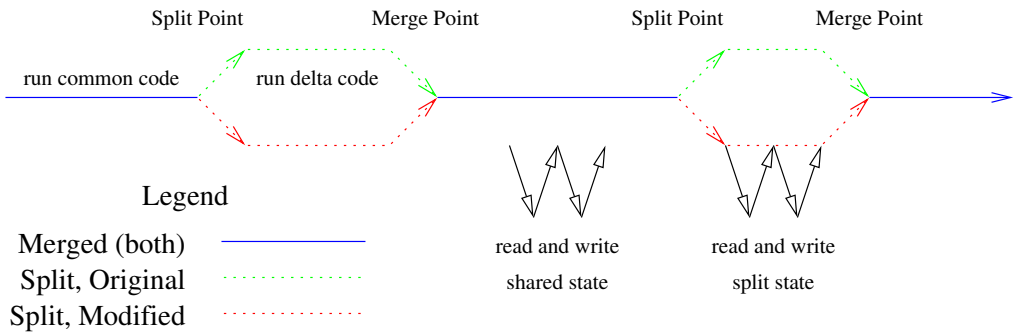


Figure 5.4: Logical diagram of delta execution

This chapter proposes a new technique, called *delta (or Δ) execution*, which exploits MAREs in patch validation to improve the efficiency and effectiveness of on-line patch validation. As illustrated in Figure 5.4, Δ execution runs two separate logical executions as one physical execution, only running those segments of execution that differ (e.g. the *deltas*) separately. For the most part, the one physical *merged execution* is all that is needed. In the segments where the original and the patched versions differ (called Δ code), two separate executions are run (*split execution*). After a time, the two separate physical executions can be joined back into one, resuming merged execution. Further, during split execution state updates are tracked, so that note can be made of program state which differs (that is, the Δ state) between the two logical executions. Further executions of delta code or accesses to delta state will cause another split, with each logical execution behaving as if it had been running separately the entire time.

This should have two advantages. First, Δ execution should have lower overhead. We expect that the bulk of execution between the two logical versions will be identical. Hence, the majority of runtime will be merged execution. Most CPU operations, I/O operations, and system calls will be performed one time, and there will be only one copy of most state, and there won't be contention between instances for resources (e.g. the processor cache). This should greatly improve performance. Further, even during split execution, competition between the logical executions for expensive operations can be minimized. Rather than issue them twice, I/O operations and system calls are monitored by the Δ execution runtime to ensure that if both are identical, they are only issued once. This is especially helpful for operations involving the disk. Since both split and merged execution can be faster in Δ execution than running two instances, one would expect for overall performance to be better.

Second, and perhaps more important, Δ execution reduces the non-determinism between the two instances. As mentioned previously, small sources of non-determinism (e.g. thread interleaving, message order, and random number generation) can cause differences in the results of side-by-side validation even if the changed code is never run. During Δ execution,

however, we expect the two logical instances to run mostly merged. During merged execution any sources of non-determinism will influence both logical executions identically. Only non-determinism during split execution will affect the output of the two versions. By minimizing the window for non-determinism to cause the two versions to vary, Δ execution can mostly eliminate false positives in patch validation.

These advantages result in more efficient and effective patch validation. As detailed in Section 5.5, the reduced resource contention during Δ execution validation allows Δ execution to outperform side-by-side validation by 12% (by 74% if the baseline overhead of instrumentation is ignored). Further, the reduction in non-determinism allows Δ execution to validate all 10 sample patches, while side-by-side validation flags 8 of the test cases as false positives. Δ execution should also be widely applicable; a manual examination of patches shows that 77% should be straightforward to run under Δ execution. These results suggest that Δ execution could greatly increase the practicality of on-line patch validation.

The rest of the chapter is organized as follows. Section 5.2 presents a characteristic study of patches, validating that many patches are appropriate for Δ execution. Section 5.3 describes the overall design of Δ execution. Section 5.4 goes into further detail about implementing Δ execution. Section 5.5 presents experimental results with Δ execution, while Section 5.6 sums up the chapter.

5.2 Characteristic Study of Patches

Delta execution is predicated on the idea that for many patches, the differences between the original execution and the patched execution is small. The three patches shown in Figures 5.1, 5.2, and 5.3 are all indeed small; the largest patch (Figure 5.2 is only 4 lines including the closing brace. Indeed, it is not surprising that most patches are small and unobtrusive. Work in binary differencing [6] and binary matching [144] shows that executables can be highly similar between versions. At the development side, large and intrusive patches

Category	Description	Δ-EXE
Refactor	Changing the names of variables, functions, etc.	all
Rare path	The code that is changed is almost never run	all
Stack effect	The changes are expected to be isolated to the stack	most
Side effect	The changes are expected to have global effects, e.g. manipulating the heap or different I/O.	some
Conditional	Adding or changing a conditional, e.g. adding a buffer overflow check.	most
Synchronization	Changes related to concurrency, e.g. data race prevention or deadlock elimination.	most
Data structure	Changing a structure size	none
Macro	Patches that involve modifying a macro	most W/C
Polymorphic	Data type changes that don't affect memory layout	most W/C
Complex changes	Changes that are too complex to segregate into small patches, e.g. 100s of lines changed.	none

Table 5.1: The ten patch categories. The last column indicates if patches in the category can be dealt with using the implementation used in the evaluation (Section 5.5). "W/C" indicates that although the implemented version cannot, with small amounts of support from the compiler it would.

are hard to understand; hence they are (or, least, are perceived as) more likely to introduce bugs. Because of this, some software projects, such as the Linux kernel, explicitly discourage large patches [133], and encourage changes to be provided as small, separate pieces which can be verified in isolation. Hence, it is intuitive that most patches will be small, and suitable for Δ execution.

To validate this intuition, this section presents a study of real world patches. 60 patches each from four representative open source applications (Apache, MySQL, OpenSSL, and Squid) were manually evaluated, and categorized according to how they would behave under Δ execution. Further, in the course of examining all of these patches, 10 general categories which the patches fall into based on distinguishing characteristics were identified.

5.2.1 Ten Categories of Patches

In examining the 240 patches, it became apparent that many patches had similarities. Many of them involved changing a conditional statement. Another common sort had its effects

limited to just one function. From these commonalities, several overarching categories that patches could fall into can be identified. Table 5.2 shows theese ten categories of patches, and Figures 5.2.1 through 5.2.1 show concrete examples of 9 of these 10, drawn from the applications used in the experiments and in the characteristic study[2]. These categories summarized essential qualities of the patches as they relate to Δ execution, and are not necessarily an exhaustive taxonomy of patch qualities. However, all of the changes studied fall into one or more of these categories. Table 5.2 also shows how the current implementation of Δ execution handles each category, based on the 240 patches examined.

More specifically, the categories are defined as follows:

```
Patch: OpenSSL function rename
openssl/crypto/des/des_enc.c

- void des_encrypt(DLONG * data, k_sched ks, int enc) {
+ void des_encrypt1(DLONG * data, k_sched ks, int enc) {
```

Figure 5.5: Example of the "Refactor" category, extracted and simplified

Refactor "Harmless refactoring" is intended to improve the readability or organization of the code, and is not intended to change the behavior of the program at all. As shown in Figure 5.2.1, this can be as simple as renaming a function. If the changes really don't change anything, as expected, then the execution should be identical. Only if the refactoring was buggy (e.g., the name change caused a conflict in name due to scoping) will split execution be necessary. This is an ideal case for Δ execution.

Rare path Some code is not run very often. Patches to such code can be categorized simply by frequency of execution. For example, the changed segfault handler in Figure 5.2.1 should almost never be run, and hence is a good candidate for Δ execution. Other cases of rare path patches include uncommonly used functionality and error handlers. For most

[2]The tenth category, "complex", does not have an example. Complex patches are by their nature large, invasive, and difficult to summarize.

```
Patch: MySQL segfault handler tweak
sql/mysqld.cc

   sig_handler handle_segfault(int sig) {
+    curr_time = time(NULL);
+    localtime_r(&curr_time, &tm);
```

Figure 5.6: Example of the "Rare Path" category, extracted and simplified

users and for most runs, rare path changes will not be executed, and so Δ execution will work very well.

```
Patch: ATPHttpd buffer overflow
atphttpd/sockhelp.c

   int sock_gets(int sockfd, char * str, int count) {
+    --count;
```

Figure 5.7: Example of the "Stack Effect" category, extracted and simplified

Stack effect Some patches are expected only to have local effects, limited to the scope of the function that they are in. The patch may vary how a function works, but is not intended to changes its external behavior (e.g. the return value or any side-effects). An example of this is shown in Figure 5.2.1; the change is to a local variable and generally won't effect the output of the function. In such cases, the need for split execution will be limited to the single function in question, and Δ execution should be effective most of the time.

```
Patch: Apache error log tweak
httpd/server/scoreboard.c

   if (rv != APR_SUCCESS) {
     ap_log_error(APLOG_MARK, ...,
-      "unable to create scoreboard\"%s\"",
+      "unable to create or access scoreboard\"%s\"",
```

Figure 5.8: Example of the "Side Effect" category, extracted and simplified

Side effect While some patches are expected only to have local effect, some patches are expected only to have an effect elsewhere. For instance, the patch shown in Figure 5.2.1 varies the message printed to the log. Asides from the additional work done, the program should vary little at the time of the difference. Instead, a differing side effect in the heap is introduced. This will likely cause further splitting due to differing data. Whether or not Δ execution will work well depends on how confined the delta data remains. In some cases, such as this change to the log, the change in heap state will have only minor further effects. However, in other cases, the heap state change may cause further heap changes, which accumulate until Δ execution is spending most of its time splitting due to delta data.

```
Patch: Apache regular expression matching
http/modules/proxy/mod_proxy.c

   if (strcmp(e[i].sch, "*") == 0 ||
       (e[i].regex &&
-       regexec(e[i].regex, url, 0, 0 0)) ||
+       regexec(e[i].regex, url, 0, 0 0) == 0) ||
```

Figure 5.9: Example of the "Conditional" category, extracted and simplified

Conditional Changes to condition statements are the most common class of patch. Most of these, like the patch to Apache illustrated in Figure 5.2.1, are intended to better deal with a corner cases. In the common case, the overall value of the conditional will be the same. Hence, Δ execution will work well; it will compute the different logic statement, get the same final answer, and continue on identically.

Synchronization Changes to synchronization code (e.g. like adding a lock as Figure 5.2.1 illustrates) are similar to rare path changes or conditional changes. Such changes are meant to deal with the rare case of data races or deadlocks. Hence, like rare path changes and conditional changes, most of the time execution will happen identically with or without the lock. Only under the unusual case where event ordering comes out wrong will Δ execution

```
Patch: OpenSSL data race fix
openssl/crypto/rsa/rsa_eay.c

   int helper(RSA *rsa, BTX *ctx) {
+    CRYPTO_r_lock(CRYPTO_LOCK_RSA);
     if (rsa->flags & RSA_NO_BLINDING)
       ret = 1;
+    CRYPTO_r_unlock(CRYPTO_LOCK_RSA);
```

Figure 5.10: Example of the "Synchronization" category, extracted and simplified

experience much trouble.

```
Patch: Apache data structure change
httpd/..../mod_deflate.c

   struct delfate_ctx_t {
     //....omitted....
+    int inflate_init;
   };
   //....omitted...
-  if (!inflate_init++) {
+  if (!ctx->inflate_init++) {
```

Figure 5.11: Example of the "Data Structure" category, extracted and simplified

Data structure Not all patches work well with Δ execution. Changes to data structures do not work well at all in the current implementation. Adding (as illustrated in Figure 5.2.1, a patch in Apache) or removing a field in a data structure can change the entire layout of memory. Hence a naive comparison of raw bytes cannot identify whether two sets of data are semantically equivalent. This is especially true for languages such as C or C++, which are not type safe. Even for type-safe language like Java, such a change means a direct comparison of the raw bits cannot identify that two structures are semantically identical. Δ execution cannot deal with such changes without a large amount of support from the compiler, language, and/or runtime environment.

```
Patch: Apache object cache macro
httpd/modules/cache/mod_mem_cache.c

-  #define DEFAULT_MIN_CACHE_OBJECT_SIZE 0
+  #define DEFAULT_MIN_CACHE_OBJECT_SIZE 1
```

Figure 5.12: Example of the "Macro" category, extracted and simplified

Macro Changes to macros, as in Figure 5.2.1, are another case which the current implementation of Δ execution does not deal with well. Although the change in the source code is isolated, after the preprocessor runs it can result in many diverse changes. However, if the compiler (or the preprocessor) could label where the changes in the code are, Δ execution could be applied as normal.

```
Patch: MySQL typedef
sql/log_event.h, and sql/log_event.cc

   class Log_event {
+    typedef unsigned char Byte;
   }

-  void get_strlen_and_ptr(const char **src, ....
+  void get_strlen_and_ptr(const Log_event::Byte **src, ....
```

Figure 5.13: Example of the "Polymorphic" category, extracted and simplified

Polymorphic Changes involving polymorphism exhibit similar effects as macro changes: a small change in the source code can lead to many small changes throughout the binary. Or, as illustrated in Figure 5.2.1, it may lead to many trivial textual changes which have minimal real effect. If the compiler could indicate to Δ execution where the changes in the binary were, the current implementation would likely be applicable.

Complex Finally, although the vast majority of patches are small and simple, there are a few which are large and intrusive. Such patches can involve many changes involving hundreds of lines of code scattered across multiple modules, hence we neither provide an example, nor

	Apache	MySQL	OpenSSL	Squid	Total (%)
Category	**Number of Patches**				
Refactor	5	4	6	2	17 (7%)
Rare Path	8	4	7	6	25 (10%)
Stack Effect	16	12	14	11	53 (22%)
Side Effect	10	11	7	9	37 (15%)
Conditional	31	34	28	32	125 (52%)
Synchronization	0	2	3	0	5 (2%)
Data Structure	7	7	1	7	22 (9%)
Macro	5	2	3	5	15 (6%)
Polymorphic	0	2	0	1	3 (1%)
Complex Changes	3	3	4	5	15 (6%)
Support	**% Patches Supported**				
Yes	76.7%	78.3%	81.7%	73.3%	77.5%
With Compiler	83.3%	83.3%	88.3%	78.3%	83.3%
No	16.7%	16.7%	11.7%	21.7%	16.7%

Table 5.2: Distribution of patches among the 10 categories. Some patches involve more than one category, and are counted twice.

do we claim the Δ execution can deal with such patches. However, because such big patches are difficult even for humans to deal with [133], they consist of only a minority of patches (6% in this study).

5.2.2 Distribution of the Categories

Table 5.2.1 summarizes the distribution of patch types among the ten categories. The two most common types of patches, conditional and stack effect, are also of types well suited to Δ execution. Overall, 77% of the patches are of the sort that Δ execution should be able to handle. A further 6% of the patches are either macro patches or polymorphic patches; these should be feasible for Δ execution to verify with minor additional compiler support. Only 6% of patches are of the complex sort, and approximately 9% involve changing the layout of a data structure. Overall, this implies that the general technique of Δ execution should be applicable to a wide variety of patches.

5.3 Delta Execution Design

The patch study implies that most execution during patch validation is redundant. Hence there seems little reason to actually have two separate physical executions. Instead, one physical execution, which applies state updates to both logical executions, would be sufficient. Further, since most of the state is also identical, only one physical set of state, for the most part, needs to be stored. The small portions of execution where two execution streams and two set of state are needed can be treated as a special case, rather than keeping two streams and sets of state for the entire validation run.

Figure 5.4 illustrates Δ execution at a high level. Initially, there is only one physical execution and one set of state. Both logical executions are captured by one set of memory state, CPU state, instruction counter, etc. Runtime monitoring will tell us when the program eventually accesses patched code (or *delta code*). This causes a split; the execution continues with two contexts, one running the old version and one running the new. During this split execution, the runtime can monitor I/O operations, both to validate that the two versions are behaving the same and to eliminate unnecessary work. Further, during split data any state updates must be tracked by the runtime. State updates that differ will result in *delta state*. After the runtime merges the execution, the runtime must monitor not only for executing delta code, but also for access to delta state, since by operating on different data the two logical executions may diverge.

Eventually the two separate physical executions should be merged back into one. Any time that the register state and instruction pointers of the two physical executions are the same is a potential time to merge the two executions together. This is because computers are deterministic state machines — once two contexts are identical they will remain in lockstep until something external (e.g. accessing memory with different values or fetching different instructions) causes them to diverge. That is, we can return to having one physical execution which will perform the work of both logical executions. Specifically, when each instance

reaches the end of the patched code, we block them to compare them to one another. If they match, then merged execution can continue, with the runtime environment monitoring for further splits (accesses to delta code or delta data).

Running merged most of the time saves the cost of performing computation and IO twice for validation. Further, any nondeterminism that occurs during merged execution will have identical effects on both the original and validation instances. Hence, there will be fewer harmless differences in the output, making the validation itself much easier.

Of course, these benefits do not come for free. There are certain patches for which delta execution is not suitable . Further, the underlying delta execution mechanism must cost something. Finally, Δ execution is not trivial to actually implement. Specifically, any implementation of Δ execution must address the following challenges:

- **Splitting.** Whenever the execution would diverge between the original and the patched versions, there must be a split. An implementation of Δ execution must detect when a divergence will occur, begin two instruction streams, and arrange for eventually merging the split instruction streams together.

- **Running split.** To the extent possible, any implementation should minimize the overhead of running split. Further, an implementation must track any changes in program state that differ between the two versions. Any writes that are different must be tracked, and any reads of differing data must return the value of the correct version.

- **Merging.** Eventually, the split execution must be merged. An implementation of Δ execution must detect when the executions have become identical, identify the data which is shared between the two instances, join the two instruction streams together again, and arrange for any further splits.

- **System calls and I/O.** It is likely that both versions will perform system operations. Whenever possible an implementation should arrange to share the work performed be-

tween the two versions. Further, the implementation must prevent the non-production instance from becoming externally visible.

- **Minimizing split execution.** The benefits of delta execution mostly come from the merged execution. Consequently it is desirable to maximize merged execution and minimize split execution. This means that an implementation should prevent the introduction of trivial or moot differences between the two versions, and attempt to prevent small differences from exploding into larger ones.

- **Threads.** As multi-core processors have become common, it is important for any system to support multithreading. However, in the context of delta execution, threads impose special challenges, as detailed in Subsection 5.4.3.

5.4 Implementation of Delta Execution

The previous section describes the overall idea of Δ execution: that it is possible to run two logically different versions of a program execution together in one physical execution, minimizing most duplicated effort and minimizing the exposure of the validation run to non-deterministic effects. This section describes the practical issues involved in implementing the Δ execution mechanism.

5.4.1 Basic Delta Execution

There are three basic issues a Δ execution implementation must face. These are splitting, running split, and then merging again. Related to these, an implementation must also decide when splitting and merging are appropriate to attempt.

Splitting There are two reasons the execution of the two versions could diverge: they could diverge because they are running different code, or they could diverge because they

are operating on different data. To monitor execution of delta code, the implementation takes a list of patched functions, and then use dynamic instrumentation to insert a split at the beginning of them. For delta data, the implementation uses `mprotect` to deny read access to pages containing delta data. Using the page protection hardware avoids the expense of instrumenting every memory access; the tradeoff is that there be "false sharing" (see Subsection 5.4.2).

Actually spliting, uses the `fork()` system call. Calling `fork()` creates two copies of the current program (the parent and child) and arranges for copy-on-write (COW) sharing of memory. COW is precisely what delta-execution calls for: copying only pages that are potentially modified while sharing identical pages. By arbitrary convention, consider the child instance to be the patched version, and the parent instance to be the original unpatched execution. If the split was caused by delta code, the implmentation sets the instruction pointer of the child instance to the patched version of the function. Otherwise it leaves the instruction pointer unchanged. For all splits, whether due to delta code or delta data, the implementaiton restores any delta data which was saved by a previous merge into the child instance; again by arbitrary convention, it leaves the unpatched version's delta data in place during merges. Once the instruction pointers have been modified (if needed) and delta data has been moved into place, there will be two instances of the execution, one running as the original version and one as the patched version.

Running split Most of the work of running split is handled by the semantics of the `fork()` system call. The two instances have their own instruction pointers and logically separate memory spaces. Our responsibility is to track delta state. Before allowing execution to continue, the implementation first uses `mprotect` to restrict access to memory pages. Delta pages are given both read and write access, since a read to these pages cannot cause another split, and the pages are already tainted from the previous write. Non-delta pages are restricted from writing. Any attempt to write to a non-delta page will cause the operating

system to deliver a segmentation fault to the process. At program initialization, the delta execution runtime adds a signal handler to deal with these faults. If it receives one to a non-delta page while the program is running split, the Δ execution runtime can mark the page as containing delta-state and then re-enable write access. Since we called `fork()` previously, the kernel will then copy the page for us.

Merging After a period of time running split, it is likely that the two executions will return to executing identical instructions on identical data. There are two things to consider with respect to merging: when and how. When can be addressed with a heuristic. The Δ execution runtime attempts to merge whenever it returns to a lower stack level than where it split. This is based on the results of the patch characteristic study: most patches are contained within one function. At a worst case, this heuristic will never trigger (e.g. a patch to `main`) although this doesn't happen in practice.

As for how to merge, the runtime first forces synchronization among the split instances; it blocks until the other instance reaches the same stack level. If the processor state (e.g. register files) of the two instances are the same, the runtime begins to merge. First, it compares all of the pages flagged as containing delta data. If a pair of pages actually differ between two instances, it saves the child's version off as a Δ page. After all of the Δ pages are saved, it can terminate the child. In the single remaining process, the Δ execution runtime uses `mprotect` to restore read and write access to the non-delta pages, and to remove permission to access the delta pages (hence restoring the monitoring for access to delta state). Finally, it continues with merged execution.

5.4.2 Advanced Delta Execution

The basic implementation of delta execution will work well for simple patches in simple programs (e.g. trivial changes to a SPEC benchmark). However, basic Δ execution is insufficient to handle either complex patches or complex server programs. There are issues

relating to real systems (e.g. I/O), as well as issues relating to efficiency. This subsection discusses the further features needed for such cases.

I/O During split execution, I/O writes from the child instance must be verified and sandboxed. Further, some I/O operations are non-idempotent. For instance, network socket data read by one process won't be there for another process to read. The Δ execution runtime must instrument all calls to `read` and `write` to manage I/O on behalf of the program. The runtime 1) compares the writes made by the child to those of the parent for validation purposes, and 2) performs all I/O operations once, sharing the results between both processes. By forwarding the results of all I/O reads, the runtime not only deals with non-idempotent operations, but it also reduces the overhead of repeating expensive I/O calls. Finally, because an I/O operation could target delta data, even in merged mode the runtime monitors I/O calls.

One issue that remains unaddressed is RPC. If the verification instance sends an RPC which the production instance does not, the runtime has no way to continue the verification instance. The Δ execution runtime can't allow the verification instance to send the request, because that would make the verification instance visible to the outside, and it can't share the production instance's request because that request doesn't exist. One way to address this is to maintain a verification instance of any possible RPC targets [88]. However, maintenance of such a "mirror world" is difficult; it may be better to run delta-execution targets of RPC instances, although doing so is well beyond the scope of this work. Communication with the outside world, rather than within RPC targets under a common administrative domain, is an issue that frustrates any on-line validation system, and there is currently no satisfactory solution.

Minimizing delta state The advantage of Δ execution over traditional validation is the merged execution. We get the maximum benefit if we minimize the portion of the execution

```
void ftpBuildTtileUrl(FtpStateData *ftpState) {
   size_t len;  char *t;
   len = 64
-        + strlen(ftpState->user)
-        + strlen(ftpState->password)
+        + strlen(ftpState->user) * 3
+        + strlen(ftpState->password) * 3
         + strlen(ftpState->request->host)
         + strlen(ftpState->request->urlpath);
   t = xcalloc (len, 1);
   ...
   strcat(t, do_escape(ftpState->user));
   strcat(t, do_escape(ftpState->password);
   ...
}
```

Figure 5.14: Simplified patch to buffer overflow bug in Squid 2.4

that is run split. One avoidable source of split execution is false delta data. False delta data is state which is literally different and will be detected as different by the runtime, but is actually semantically identical. If the runtime can make the two semantically identical copies also physically identical, then it can avoid some needless splits. The simplest example of false delta data is the dead area of the stack. An original and patched function may differ greatly in how they use the scratch space of their activation record, but once they are complete, that state is moot. By "scrubbing" the state, so that both stacks are filled with zeros, the Δ execution runtime can avoid situations where top of the stack appears to be in a delta page.

Another source of false delta data is from memory allocation. If the original and modified executions allocate different amounts of memory, the heaps will become different, and subsequent allocations will be shifted between the two. This can cause substantial divergence in the amount of detected delta data, for two reasons. First, pointer values to the misaligned heaps will be different, even though they are pointing to semantically the same data. Second, and more importantly, as we are only detecting identical data if there are in the same place in the address space, the misalignment will likely cause all of the data

to appear different, even though it has only been shifted over. A further source of trouble is `malloc`'s internal data structures. As they diverge, future allocation and deallocation actions will cause access to delta data, and hence a split. Figure 5.14 shows a patch in Squid which suffers particularly badly from memory allocation issues.

The bug is that the `user` and `password` fields are escaped before being copied into the temporary buffer `t`, which can expend their length by a factor of 3. However, in the original code, their unescaped length was used to decide how big of a temporary buffer to allocate. Increasing the size of the allocation avoids the buffer overflow, but causes all future memory operations to become misaligned. This causes the entire heap to quickly becomes delta data.

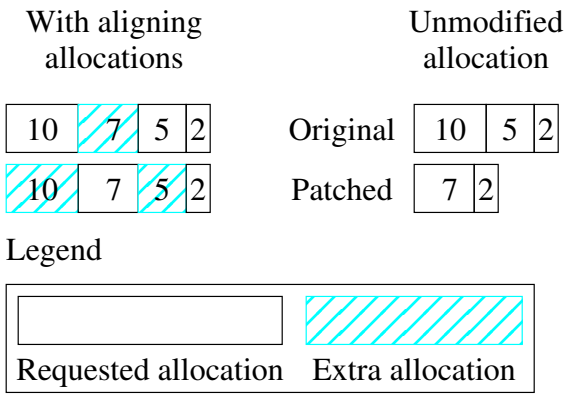


Figure 5.15: How Δ execution improves memory alignment

To reduce the amount of delta data caused by differing heap allocations, the Δ execution runtime instruments all calls to `malloc` and `free` that occur during split execution. As shown in Figure 5.15, if either instance makes an allocation different from the other, the runtime will make a false allocation for both. For example, suppose the original program will allocate 10 bytes, 5 bytes, and then 2 bytes, while the patched version allocates 7 bytes and then 2 bytes. To force the heaps to line up, the runtime can insert a fake allocation for 7 bytes in the original execution, and insert fake allocations for 10 and 5 bytes in the patched execution. If it further forces the allocations to occur in the same order, then the two executions will have the same heap state, and `malloc`'s internal data structures will be

identical between the two.

The mechanism for the order forcing is to maintain a queue of allocation sizes requested by the execution which is "ahead" of the other. Allocations by the "ahead" execution keep getting added to the queue. If the "slower" execution requests an allocation size which is not at the head of the queue, the Δ execution runtime allocates a fake allocation of that size anyway. It will drain allocations from the queue until it either finds a matching execution, or the queue is empty. When it finds a matching size, the runtime satisfies the slower execution's request and dequeues that request. If the queue is empty, then both executions are considered to be at the same place in their allocation streams, and so whichever execution next requests allocation becomes "ahead", gaining ownership of the queue. Finally, to minimize the delta state during merged execution, when merging the runtime will make fake allocations as necessary to empty the queue.

Calls to `free` also need to be intercepted. If one execution calls `free` and the other does not, then the heap state will also diverge. When the program calls `free`, we defer actually deallocating the memory until 1) the other execution deallocates the same memory block too, and 2) the allocation metadata state is otherwise identical. The first condition prevents an area of memory from appearing to be available in one execution and not the other. In effect, it converts the earlier freed memory into a fake allocation. If we were freeing a fake allocation, then we treat it as if the other execution had already freed it. The second condition preserves the total order of allocation operations. It will be met whenever the allocation queue is empty; this at a minimum will occur whenever we merge.

Together, these changes to memory management minimize the impact of differing memory management between executions on the rest of the program. For Squid, these changes are mandatory for efficient Δ execution. Unfortunately, this sacrifices execution fidelity. Suppose a patch was attempting to fix a memory leak. That the Δ execution runtime defers freeing memory until both executions have done so means it will have duplicated the memory leak in both executions. Further, these manipulations of memory allocation changes the layout

of the heap, which means that some memory bugs will occur differently under Δ execution than under unmodified execution. For type-unsafe C programs, this is probably the best trade-off that can be managed. However, for a type-safe environment like Java, one would not be limited to comparing state by direct byte-for-byte comparison. Instead, one could track if the underlying objects are the same.

Reducing split/merge overhead Some patches lie on commonly taken execution paths, which could force rapid splits and merges. Although `fork()` is an inexpensive mechanism for splitting, merging is expensive, due to the need to synchronize between two separate processes. The patch in ATPhttpd shown in Figure 5.2.1, for example, is run multiple times per http request.

To reduce the overhead of splitting and merging, there is a possible alternative to `fork()` based splitting and merging using instrumentation. First, the instrumentation needs to save the processor context, and then run the unmodified version. Rather than use the copy-on-write mechanisms of `fork()`, the signal handler used to monitor delta writes can copy the pages as needed. When a merge point is hit, the runtime can save any modified pages as if it were doing a usual merge, but then it can use the state saved by the signal handler to roll back to the split point. From there, it can run the modified version. When the modified version reaches the merge point, the Δ execution runtime can perform a merge as usual, except that it is already in a single process and hence does not need any cross-process synchronization operations.

Handling small deltas in large functions Sometimes a patch only involves one branch of a large function. If the the whole function is flagged as delta code, the runtime will incur unnecessary splits. To avoid this, the implementation supports a mechanism to support smaller segments of delta code. Instead of labeling entire functions, the areas of delta code are labbeled, and what was added/subtracted, by adding macros to the source. Figure 5.16

```
Patch: Small delta version of CAN-2004-0493
httpd-2.0/server/protocol.c

void ap_get_mim_headers_core(/*...*/) {
   /* 55 lines omitted * /
   DELTA_START();
   D_MOD_START();
   if ((fold_len-1) > (r->server->limit)) {
      r->status = BAD_REQUEST;
      return;
   }
   D_MOD_END();
   DELTA_STOP();
   /* 92 lines omitted */
}
```

Figure 5.16: Apache httpd patch implemented using small delta support.

illustrates this. It shows the same patch as in Figure 5.2, but using small delta support. `DELTA_START()` and `DELTA_END()` markers wrap the entire area which should be run split, with the start being a split point and the end being a candidate for merging. `D_ORIG_START()` and `D_ORIG_END()` markers would wrap statements which were removed in the batch (e.g., the "-" lines in the diff), had there been any in the patch. The new lines are wrapped in `D_MOD_START()` and `D_MOD_END()` markers. Given the diff of a patch, it is trivial to decide where to place the markers, and this could be easily automated. This allows 157 of lines be run under Δ execution rather than split.

Competitive analysis for worst case situations There are situations where Δ execution is not suitable. A change in data could cause data differences throughout the memory, or the two executions could follow drastically different paths. In this case, the two versions actually would be different, and there would be few opportunities for merged execution. Another situation where Δ execution would work poorly occurs when the differences occur in a hotspot. The executions could well be highly similar, but there would be too much overhead repeatedly splitting and merging. In either case, the implementation has a fallback

mechanism to avoid worst-case behavior. It divides the execution into epochs, and monitors how much time is spent split, how much is spent merged, and how much time is spent either splitting or merging. Given an estimate of the overhead of running two instances side-by-side, the runtime can calculate if the epoch would have been more efficient under Δ execution or under side-by-side validation. Specifically, Δ execution is more efficient that side-by-side validation if and only if

$$\frac{\epsilon \cdot time\_merged + time\_split}{epoch\_length} \geq 1$$

where ϵ is the slowdown imposed by side-by-side execution. Although ϵ varies from program to program, in general 2 is a good estimate (see 5.5 for more details). The numerator is an estimate of how long this epoch would have taken under side-by-side validation. If the ratio is less than 1 for 3 epochs in a row, the Δ execution runtime environment can remove itself and fall back to side-by-side validation. Later, if the load on the system is reduced, Δ execution can be dynamically re-enabled to allow high-fidelity validation.

Type changes A final issue which the current implementation is incapable of addressing is type changes. A change in types, e.g. inserting a new field into a structure, can change the layout of memory. Changing the layout of the stack in one or two functions is fine, since the difference will be discarded after the function returns. Changing the layout of the heap, however, is not easy to deal with. Even adding a single new member to a structure completely breaks the implementation of Δ execution presented here. Although sufficient language or compiler support [119] could potentially allow such patches to be supported in the future, such patches are outside the scope of current support.

5.4.3 Threads

Given the current trend of increasing numbers of multicore processors, it would be remiss to ignore the issues raised by threads. Beyond the increased complexity of writing thread safe instrumentation, threads raise several difficulties. First, threads are more difficult to split because the `fork()` system call doesn't duplicate threads. Further, there is the issue of thread creation or destruction during split execution.

There are 3 ways to deal with `fork()` not duplicating threads. One can create a modified `fork()` call, which does duplicate threads. This has the advantage of supporting the widest variety of changes and programs. The disadvantage is requiring a modified system call; this is fairly intrusive. Alternatively, one can use instrumentation to recreate the threads. By directing a signal at each specific thread, one can trap them in a barrier. Then, the splitting thread can copy their execution contexts and *then* fork. The parent can then resume the stopped threads, while in the child process, we recreate all of the threads before continuing. Although this does not require modifications to the kernel, it is much more expensive. Finally one can temporarily disable other threads for the duration of split execution. Again, a signal can be used to pause the other threads so the don't have to be recreated. This is simple, inexpensive, and minimally intrusive. The downside is the potential for deadlock if the thread which caused the split needs a resource which has been locked by one of the suspended threads. Such deadlock situations can be avoided by temporarily merging execution, executing the thread that holds the resource, and then splitting again. The primary implementation, uses this option. In the experiments such deadlock has never been observed, and the implementation doesn't require changes to the kernel.

Thread creation and destruction during delta execution is more complex. If both versions create (or destroy) a thread, then issues are minimized. However, if only one version does this, then there will be a mismatch in the number of threads. The number of live threads is in some ways state intrinsic to the entire process, and may prevent merging. However, aside

from contrived cases (e.g. changing a `MAX_THREADS` variable) such a situation is extremely rare; if it were to happen side-by-side validation is a fall-back.

5.5 Experimental Evaluation

5.5.1 Methodology

For the evaluation, both Δ execution and traditional side-by-side on-line validation are implemented. For Δ execution, Pin [77], a dynamic instrumentation tool, is used to insert code implementing the individual functions of Δ execution:

- **Delta code split and merge** Before (and after) functions which contain delta code, the implentation inserts calls to functions which performed splitting (and merging). This code called `fork()` (or arranged a join), set up delta state (or saved it), modified the page tables via calls to `mprotect()`, signaled other threads to stall their execution, called `fork()`, and dispatched to the proper version of the delta function.

- **Signal handlers** For handling delta state, instrumentation to add signal handlers for `SIGSEGV` was added.

- **System calls** All system calls which could perform IO or access delta state were instrumented to check if a split was necessary or if only a single call is needed.

In these experiments, traditional side-by-side validation, is implemented either with a network proxy in front of two separate instances for networked applications, or with a script wrapping two calls and a verification for command line applications.

Table 5.3 shows the tests cases which were used, summarizing the programs used and the changes to be validated. The table presents 10 different applications, 7 of which are server programs. Further, 5 of the programs can utilize multiple cores. 6 of the changes are bug fixes, 2 add functionality, 1 is a refactoring, and 1 is a change which unintentionally introduced a bug. The workloads used for testing were as close to standard benchmark workloads

Benchmark	Program	Change Description
crafty	Chess Program	Code refactoring
raytrace	Raytracer	Fixed bug in result reporting
tar	Archive Utility	Fixed incremental archiving
Apache 1	Web Server	Fixed overflow in mod_alias
Apache 2	Web Server	Fixed overflow in mime parser
ATPhttpd	Web Server	Fixed overflow in HTTP parsing
DNSCache	DNS Cache	Behavior change
MySQL 5.0	Database Server	Extra permission checks
OpenSSL	Security Library	Regression - unintentional bug in TLS handling
squid	Web Cache	Fixed overflow in FTP parsing

Table 5.3: Test applications and the change between the two versions.

as possible. For instance, since crafty is a SPEC CPU benchmark, the experiments used the workload specified by SPEC. However, because the versions of the programs specified by the benchmarks do not necessarily contain bugs, and because the benchmark workloads are not necessarily bug-triggering, it cannot be said that these are precisely the same as the benchmarks. It should also be noted that these are not all the same patches as used to illustrate the patch categories in Section 5.2.1, because of the need to have inputs which can cause the patches to be executed.

All of the experiments were run on identically configured machines. The processors used were 2.4 GHz Pentium-4 based Xeons configured in 2-way SMP. The machines all had 2.5 GB of memory, and gigabit network. For the server programs, the clients were identically configured as the servers, and were connected to the same gigabit switch.

For the performance evaluation, three cases are compared:

- Traditional validation by running two copies side-by-side.
- Traditional validation (as above), but also with a "null" pintool. A null pintool imposes the overhead of Pin's recompilation and code cache, but does not add any actual instrumentation.
- Validation using Δ execution

In all three cases, the results presented normalize the performance to the performance of running a single instance without any instrumentation or validation.

5.5.2 Functionality of Patch Validation

Benchmark	Δ Execution	Baseline	Reason baseline failed
crafty	pass	fail	"Kibitzes" differ
raytrace	pass	fail (large)	Nondeterminism from threads
tar	pass	pass	(N/A)
Apache 1	pass	fail	Randomized etags are output
Apache 2	pass	fail	Randomized etags are output
ATPHttpd	pass	fail (minor)	Timestamps differ
DNSCache	pass	fail	Randomized Tx IDs & timing sensitivity
MySQL 5.0	pass	fail (large)	Nondeterminism from threads
OpenSSL	pass	fail **(major)**	Random nonce for key exchange
squid	pass	pass	(N/A)

Table 5.4: Functionality of validation for baseline side-by-side validation and for Δ execution. The baseline validation fails 8 of the 10 applications when they should pass; 3 of these (MySQL, OpenSSL, & raytrace) fail badly enough to be considered unvalidatable.

Table 5.4 shows the function effectiveness of Δ execution versus side-by-side validation. A pass occurs when the validation correctly identified the runs as matching, while a fail indicates that the validation technique incorrectly identified the runs as differing. The baseline validation has trouble with 8 of the 10 runs, only passing tar and squid cleanly. Five of the runs fail validation in small ways. The crafty chess benchmark periodically prints what moves it is considering (called "kibitzes"); this varies slightly from run to run due to timing. For both runs of Apache, the "Etag" field in the HTTP response varies from run to run, and fools the baseline validation into thinking that the patch has changed the output. ATPHttpd occasionally fails because requests will occasionally have different timestamps, although most requests are the same. Finally, of the more minor failures, DNSCache uses random transaction ID as a nonce for security purposes, and also is sensitive to timing (because of cache entry expiry). This causes small differences in the replies. All of these five failures could be overcome by writing a more sophisticated validation engine, which has more

understanding of which output fields are important and which are not.

The baseline validation does poorly for two of the threaded benchmarks, raytrace and MySQL. Both have slightly variable output based on thread interleaving. From run to run raytrace renders the same thing, but the image differs in low-significance bits. For MySQL, although the transactions may all finish atomically and correctly, they may occur in different orders. Correctly identifying that the output for the two runs is correct, even though they differ, is difficult.

The baseline validation is completely broken for OpenSSL. As part of the SSL handshake, OpenSSL generates a random nonce to exchange with the client. Since the client can only see the nonce from one of the two instances, the replies it generates will be nonsense to the test instance. The test instance, therefore, reports an error and stops, even though the bug is not triggered. To the test instance, the proxy is creating what looks like a poorly-done man-in-the-middle attack, which SSL is supposed to detect. Because of the non-determinism in random number generation, and the high dependence of the output on the random numbers, the baseline validation does not work at all.

In contrast with the baseline validation, Δ execution correctly identifies all of the runs as passing. Since the bulk of the execution occurs only once for "both" copies, Δ execution ensures that the non-determinism doesn't cause the outputs to diverge. For the kibitzes in crafty, they are computed and called during merged execution. The calls to get the time, and to generate random numbers, which makes Apache, ATPHttpd, and DNSCache to differ for the baseline validation occur only once in Δ execution, and so both copies have the same data to work with. In raytrace and MySQL, the data races occur the same way for "both" copies, and so the outputs are identical. Finally, OpenSSL uses the same nonce for both the production and test instance, and so can proceed through the entire SSL handshake while producing identical output. The lack of non-deterministic effects causing the outputs between instances to diverge makes validation with Δ execution simple.

5.5.3 Performance

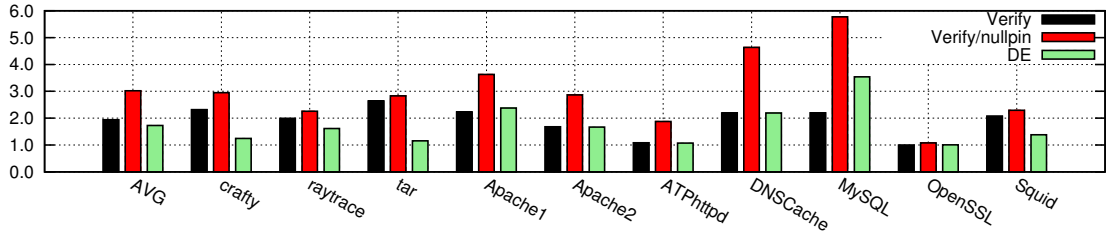


Figure 5.17: Overhead of validation with Δ execution.
Note that 1x is normalized to the performance of a single, unvalidated and uninstrumented instance of the application.

Figure 5.17 shows the performance of Δ execution, compared to side-by-side validation, and side-by-side validation running a null pintool. For four of the benchmarks (crafty, raytrace, tar, and Squid), Δ execution outperforms side-by-side validation. This is on top of the overhead that Δ execution must pay for using Pin for dynamic instrumentation. tar stands out as a particularly good result; the high cost of IO interference between the test instance and the production instance make side-by-side validation particularly poor, and Δ execution is 128% faster. Overall, Δ execution manages to be 12% faster than side-by-side validation, averaged over all of the benchmarks.

The "black sheep" performance-wise is MySQL. Δ execution is 37% slower than the side-by-side validation. However, Δ execution is still 63% faster than side-by-side with nullpin. It turns out that Pin imposes a very large overhead even without instrumentation for MySQL, running 3.23 times more slowly even with nullpin. Δ execution is only 9.6% slower than this.

Another interesting thing is the high efficiency of validation of ATPhttpd and OpenSSL. Both applications are CPU bound, yet neither program is capable of taking advantage of the multiple cores they had available. Hence, when running two copies side-by-side, there is an unusually low level of overhead. Indeed, because OpenSSL detects the network proxy as a man in the middle attack, *the second copy doesn't even perform any work*, and aborts the connection prematurely.

	splits/sec	% merged	% split	% splitting	% merging
crafty	.005	99.996	.001	.011	.035
raytrace	.037	99.134	.696	.130	.055
tar	5.360	45.400	10.500	3.080	40.600
Apache1	.368	94.500	.002	.072	2.420
Apache2	6.560	72.900	.081	1.800	25.200
ATPhttpd	19.100	12.100	3.850	16.300	67.600
DNSCache	9.638	55.691	2.164	17.178	24.917
MySQL	.520	87.827	5.122	.469	7.440
OpenSSL	11.700	59.500	.229	6.290	33.900
squid	.903	88.200	.358	.896	10.500

Table 5.5: Detailed accounting of where time was spent in each application. Rows may not sum to 100% due to rounding.

5.5.4 Detailed Performance Characteristics

Table 5.5 shows a detailed listing of where each program spent its time while running under delta execution. 8 of the 10 applications spend most of their time merged. The exceptions are tar (45% merged) and ATPhttpd (only 12% merged). Tar is especially interesting because it in fact performs better under Δ execution than any other application. Yet that tar spends 10.5% of its runtime split, and 40.6% of its runtime merging would imply that tar should have terrible performance. It turns out that most of the contention in tar is due not to the CPU but due to I/O. Although a large amount of CPU cycles are spent in merging, most of that time would have otherwise been spent waiting for the disk. Further, although tar has the most time spent split, most of that time is composed of waiting for I/O. Even when split, delta execution will issue only one I/O operation to the system if both versions are issuing the same operation. This allows tar to perform well under delta execution despite its large amount of split execution and merge overhead.

ATPhttpd, on the other hand, has the highest number of splits per second, at 19.1. On average among all of the benchmarks, a split takes 9.81 ms and a merge takes 105.9 ms. At 19 split/merges per second, ATPhttpd should spend 2.2 seconds per second either splitting or merging. Clearly it would make no forward progress at all if it weren't for the fact that it takes far less time per merge than average, only 35.3 ms. Since ATPhttpd is completely

serial, it does not suffer from scheduling contention while merging. Even so, ATPhttpd still spends over two thirds of its running time in merging back together. Unfortunately, although the patch in ATPhttpd does not change the path of execution in nearly all cases, the patch is called a minimum of *three times per request*. Cases like ATPhttpd motivate working on more efficient methods of merging.

5.6 Summary

In summary, Δ execution is a workable technique for allowing online validation of patches. It can allow system administrators to validate the patches that they currently view with suspicion by testing them with their own production workloads. Based on a study of real software, it is clear that Δ execution can be successfully applied to most patches. The basic implementation of Δ execution presented here imposes reasonably low overhead, allowing validation to occur without consuming undue resources. More importantly, compared to traditional on-line validation, Δ execution greatly improves the quality of the validation. Differences in output caused by thread interleavings, timing events, random number generation, and other sources of spurious influence do not cause Δ execution to falsely claim that the patch is causing different output.

Chapter 6

Future Work

This dissertation does not address all issues regarding dynamically addressing production run failures; nor does it completely address the specific areas it explores. The trend of fast worms which motivated Sweeper has given way to a threat of careful, slow, persistent attacks. The ability to react within milliseconds is less important for such attacks; the ability to retrospectively analyze suspicious segments of execution should still be valuable. Triage shows that there are many analyses that can be used for automatically diagnosing failures; beyond delta analysis there should be other undiscovered techniques which would ordinarily be intractable if not applied to finely focused execution segments. Further, Triage is tantalizingly close to being able to automatically fix some software bugs; automatic bug fixing seems to be a promising area to explore.

With respect to Δ execution, there are several issues that need to addressed in order to make it sufficiently practical such as to be commercially deployed. Although the study of patches in Section 5.2 is a start, it would greatly increase our confidence that Δ execution can address many patches if the study were extended. With a formal taxonomy of patches, it would be possible to more precisely describe the sorts of patches which Δ execution can and cannot work with. Further, the overheads of Δ execution are still high enough that end-users may be concerned about the impact. It seems likely that there is room for improvement, through reducing the overhead of Δ execution primitive operations (e.g. merging), reducing the number of times these primitives occur (e.g. via finer granularity detection of delta code and further reductions in false delta data), and by reducing the overhead of the framework Δ execution is implemented in (e.g by using static instrumentation rather than dynamic,

or by extending the operating system kernel). The compiler could also be enlisted. Given both the original and patched versions of a program, the compiler can, as mentioned before, identify the locations where code was changed due to either changes to macros or due to a type changing. Further, the compiler could automatically identify shorter segments of delta code, and either use the support for short segments, or if the change supports it, hoist the differing segment into another sub-function. Finally, Δ execution would benefit from enlisting the aid of software vendors and of the programmers who make the patches. The vendor can ship binaries which are already annotated to support Δ execution, and programmers can structure their changes such that they are smaller and more likely to be well suited for Δ execution. As implemented, Δ execution can only be applied if the source code of the patched and unpatched versions are available. If a vendor is unwilling to provide the source code or to provide a binary already annotated with split points and delta code, then Δ execution must rely on binary differencing, which is more difficult and currently not implemented. Although Δ execution as it currently exists is an interesting proof of concept, to be made practical for commercial deployment would clearly require additional effort.

Chapter 7

Conclusions

This dissertation demonstrates that it is advantageous to address software reliability *dynamically*, during production runs, as opposed to before deployment, in post-failure debugging, or during dedicated testing or validation runs. The wealth of information available during a production run, and the "live-fire" nature of such runs, are qualitatively invaluable. Further, this dissertation demonstrates that the overheads of leveraging production runs for software reliability purposes can be made low enough for continuous use in production. In the past, production runs have been underutilized as a way to improve software quality, due to overheads which excluded widespread use. This dissertation demonstrates that we can now take advantage of production runs and the real-world workloads and failures they experience.

In demonstrating that production run failures can be leveraged to improve reliability, this dissertation highlights five key insights. The first insight is that many software reliability tools are wasteful, especially when applied to production runs. Online patch validation, for example, runs two nearly identical sequences of instructions and system calls with nearly identical data. Whenever the two sequences are identical, the second run is wasted. When running a buffer overflow detector on a production run, every bounds check that passes is also wasted. Similarly wasted is all of the effort spent taint tracking, data-race or deadlock detecting, or slicing during failure free execution. The overheads would be much lower if the effort could be focused only on the portion of production run execution where it is useful: on failures and potential failures.

The second insight is that checkpointing can help provide this focus. Recently developed modern checkpointing techniques can capture the behavior of production runs at fairly low

execution cost: 10 to 20%. For many software reliability purposes, however, this can be reduced further, based on the insight that the checkpoints are only needed for a very short time period (a few seconds) and that perfect fidelity during replay is unnecessary. Indeed, it is not so much replay as reexecution that is needed. It is possible to take ephemeral checkpoints for the purposes of reexecution with overheads of 2 to 10%.

The third insight is that failure analysis tools can be applied after the fact. Dynamic instrumentation tools allow instrumentation to be added to an already running program. Even with these tools, however, in the usual case it is not known when to apply a failure analysis, since the future cannot be foreseen. However, with checkpointing and reexecution, when a failure is about to occur is easy to predict: while it is being reexecuted. Hence the overhead of analysis can be focused on the short segments of execution which are expected to have failures; this reduces the overhead for moderately expensive analysis and allows for extremely expensive analysis to be actually run. This makes it worthwhile to figure out how to make existing analysis techniques work when they're starting without information from the very beginning of execution. As a further benefit, since one can immediately apply the analysis to the failure, one immediately gains the benefit of the analysis result. Having immediate access to the analysis results allows security problems to be addressed, where quick results can be more important.

Fourth, given the ability to apply failure analysis tools repeatedly to the same failure, the outputs of various tools can be combined, use the output of one tool can be used as an input of another, to greatly improve the results of analysis. A buffer overflow detector is useful, but feeding it's output to backward slicing gives the programmer much more information about why the buffer overflowed. This can result in a much quicker time to generate a patch.

Fifth and finally, failing and successful runs can be highly similar, as can runs from different versions of the same program. This can be used for patch validation, where the normal behavior is expected to be identical. One can reduce the overhead of validation by sharing a single instruction stream for the original and changed instance except for the

rare times that they do differ. Furthermore, by sharing a single instruction stream, the original and patched executions will be identically influenced by exigent non-determinism (e.g. the timing of system calls, random number generation, and thread interleaving), which will result in a higher fidelity of validation.

Beyond presenting these insights, this dissertation also demonstrates the practicality of using the resulting techniques, through implementations of functional systems acting on real programs and real software reliability concerns. Chapters 3 and 4 present the Sweeper and Triage systems, which use production run checkpointing to capture failures for immediate post-failure analysis. Sweeper shows how analysis tools such as taint tracking and memory bug detection can be used while re-executing an exploit attempt. Chapter 3 also shows how the quickly available results of such analysis could be used to stop fast-spreading worm infestations.

Triage, in Chapter 4, demonstrates using similar post-hoc failure analysis to attempt to automatically debug production run faults. Triage uses the outputs of repeated runs of various failure analysis tools as the inputs to further runs, to walk backward from the failure to the underlying root cause. Furthermore, because Triage can focus analysis effort on the small segments of execution, it allows much more expensive analysis tools to be used than would otherwise be realistic for a production run. For example, backwards slicing, which ordinarily imposes a 100 to 1000x overhead, is completed by Triage within a minute or two of a failure. Chapter 4 also presents a new failure analysis, *delta analysis*, which computes an edit script (i.e. a diff) between a failing and successful run. Such a diff can highlight why a failure occurred, but computing it can impose an overhead of up to 20000x. Such an expensive analysis is only remotely feasible if it is focused on promising segments of execution; reexecuted segments of a failing execution counts as promising. The result of all of Triage's analysis is a report, which can appreciably (about 45%) reduce the time taken to produce a working patch.

Finally, in Chapter 5, this dissertation shows how the high degree of similarity between

a patched and unpatched program can be used to improve online validation. By running only one instruction stream whenever the two versions are running identical instructions on identical data, the new technique of *delta execution*, or Δ execution, can reduce the overhead of online validation. More importantly, because the two logical executions mostly occur in one instruction stream, they are equally influenced by non-determinism. This greatly increases the similarity between the outputs, since things like differing random number draws occur identically in both logical executions. Hence Δ execution improves the fidelity of validation.

References

[1] Agrawal, H., DeMillo, R. A., and Spafford, E. H. An execution-backtracking approach to debugging. *IEEE Software 8*, 3 (1991), 21–26.

[2] Aguilera, M. K., Mogul, J. C., Wiener, J. L., Reynolds, P., and Muthitacharoen, A. Performance debugging for distributed systems of black boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (2003).

[3] Altekar, G., and Stoica, I. Odr: output-deterministic replay for multicore debugging. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP'09)* (2009).

[4] Anagnostakis, K. G., Sidiroglou, S., Akritidis, P., Xinidis, K., Markatos, E., and Keromytis, A. Detecting targeted attacks using shadow honeypots. In *Proceedings of the 2005 USENIX Security Symposium* (2005).

[5] Associated Press. Microsoft to test patch for flaw in windows, 2006. http://www.latimes.com/business/la-fi-briefs4.6jan04,1,72544.story?coll=la-headlines-business.

[6] Baker, B. S., Manber, U., and Muth, R. Compressing differences of executable code. In *ACM SIGPLAN 1999 Workshop on Compiler Support for System Software (WCSSS'99)* (May 1999).

[7] Beattie, S., Arnold, S., Cowan, C., Wagle, P., Wright, C., and Shostack, A. Timing the application of security patches for optimal uptime. In *LISA* (2002), pp. 233–242.

[8] Beattie, S., Arnold, S., Cowan, C., Wagle, P., Wright, C., and Shostack, A. Timing the application of security patches for optimal uptime. In *LISA '02: Proceedings of the 16th USENIX conference on System administration* (Berkeley, CA, USA, 2002), USENIX Association, pp. 233–242.

[9] Berger, E. D., and Zorn, B. G. Diehard: probabilistic memory safety for unsafe languages. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2006), ACM, pp. 158–168.

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

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

[12] Borg, A., Baumbach, J., and Glazer, S. A message system supporting fault tolerance. In *SOSP '83: Proceedings of the ninth ACM symposium on Operating systems principles* (New York, NY, USA, 1983), ACM, pp. 90–99.

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

[14] Bort, J. Microsoft fixes troublesome patch, 2010. http:// www.networkworld.com/ community/node/58100.

[15] Browne, H. K., Arbaugh, W. A., McHugh, J., and Fithen, W. L. A trend analysis of exploitations. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2001), IEEE Computer Society, p. 214.

[16] Bruening, D. L. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2004. Supervisor-Saman Amarasinghe.

[17] Brumley, D., Liu, L.-H., Poosankam, P., , and Song, D. 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).

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

[19] Brumley, D., Wang, H., Jha, S., and Song, D. Creating vulnerability signatures using weakest preconditions. In *CSF '07: Proceedings of the 20th IEEE Computer Security Foundations Symposium* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 311–325.

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

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

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

[23] CERT. US-CERT vulnerability notes database.

[24] Chandra, S., and Chen, P. M. Whither generic recovery from application faults? a fault study using open-source software. In *International Conference on Dependable Systems and Networks (DSN'00)* (2000), pp. 97–106.

[25] Chen, M., Kiciman, E., Fratkin, E., Fox, A., , and Brewer, E. Pinpoint: Problem determination in large, dynamic systems. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks* (2002).

[26] Chew, M., and Song, D. Mitigating buffer overflows by operating system randomization. Tech. rep., Carnegie Mellon University, 2002.

[27] Choi, J.-D., Lee, K., Loginov, A., O'Callahan, R., Sarkar, V., and Sridharan, M. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation* (New York, NY, USA, 2002), ACM, pp. 258–269.

[28] Clarke, G. How to diagnose and solve software errors. *PC World* (1999).

[29] Cohen, J. *Statistical power analysis for the behavioral sciences*. Lawrence Earlbaum Associates, 1988.

[30] Condit, J., Harren, M., McPeak, S., Necula, G. C., and Weimer, W. CCured in the real world. In *PLDI* (2003).

[31] Cook, J. E., and Dage, J. A. Highly reliable upgrading of components. In *ICSE '99: Proceedings of the 21st international conference on Software engineering* (Los Alamitos, CA, USA, 1999), IEEE Computer Society Press, pp. 203–212.

[32] Costa, M., Castro, M., Zhou, L., Zhang, L., and Peinado, M. Bouncer: securing software by blocking bad input. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles* (New York, NY, USA, 2007), ACM, pp. 117–130.

[33] Costa, M., Crowcroft, J., Castro, M., Rowstron, A., Zhou, L., Zhang, L., and Barham, P. Vigilante: End-to-end containment of internet worms. In *SOSP'05* (2005).

[34] Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., and Hinton, H. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *the 7th USENIX Security Symposium* (1998).

[35] Crandall, J. R., Su, Z., Wu, S. F., and Chong, F. T. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *CCS '05* (2005).

[36] Crandall, J. R., Wu, S. F., and Chong, F. T. Minos: Architectural support for protecting control data. *ACM Transactions on Architecture and Code Optimization 3* (December 2006), 359–389.

[37] Criswell, J., Lenharth, A., Dhurjati, D., and Adve, V. Secure virtual architecture: A safe execution environment for commodity operating systems. In *Proceedings of the Twenty First ACM Symposium on Operating Systems Principles (SOSP'07)* (October 2007).

[38] d'Amorim, M., Lauterburg, S., and Marinov, D. Delta execution for efficient state-space exploration of object-oriented programs. In *Proceedings of the 2007 international symposium on Software testing and analysis* (New York, NY, USA, 2007), ISSTA '07, ACM, pp. 50–60.

[39] Dhurjati, D., and Adve, V. Backwards-compatible array bounds checking for c with very low overhe ad. In *ICSE* (2006).

[40] Dhurjati, D., and Adve, V. Backwards-Compatible Array Bounds Checking for C with Very Low Overhead. In *Proceedings of the 2006 International Conference on Software Engineering (ICSE'06)* (Shanghai, China, May 2006).

[41] Dinning, A., and Schonberg, E. An empirical comparison of monitoring algorithms for access anomaly detection. In *PPOPP '90: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming* (New York, NY, USA, 1990), ACM, pp. 1–10.

[42] Do, H., and Rothermel, G. An empirical study of regression testing techniques incorporating context and lifetime factors and improved cost-benefit models. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering* (New York, NY, USA, 2006), ACM, pp. 141–151.

[43] Documentation, W. X. P. P. Dr. Watson overview.

[44] Dunlap, G. W., King, S. T., Cinar, S., Basrai, M., and Chen, P. M. Revirt: Enabling intrusion analysis through virtualmachine logging and replay. In *OSDI'02* (2002).

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

[46] Eichin, M. W., and Rochlis, J. A. A. With microscope and tweezers: An analysis of the internet virus of november 1988. In *Proceedings of the 1989 IEEE Computer Society Symposium on Security and Privacy* (Oakland, Ohio, 1989).

[47] Ernst, M., Czeisler, A., Griswold, W. G., and Notkin, D. Quickly detecting relevant program invariants. In *ICSE* (2000).

[48] Etoh, H. GCC extension for protecting applications from stack-smashing attacks. http://www.trl.ibm.com/projects/security/ssp/.

[49] Feldman, S. I., and Brown, C. B. Igor: a system for program debugging via reversible execution. In *PADD '88: Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging* (New York, NY, USA, 1988), ACM, pp. 112–123.

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

[51] Geels, D., Altekar, G., Shenker, S., and Stoica, I. Replay debugging for distributed applications. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference* (Berkeley, CA, USA, 2006), USENIX Association.

[52] Glerum, K., Kinshumann, K., Greenberg, S., Aul, G., Orgovan, V., Nichols, G., Grant, D., Loihle, G., and Hunt, G. Debugging in the (very) large: Ten years of implementation and experience. In *Proceedings of the Twenty Second ACM Symposium on Operating Systems Principles (SOSP'09)* (October 2009).

[53] GNU. Gdb: The gnu project debugger.

[54] Gray, J. Why do computers stop and what can be done about it? In *Symposium on Reliability in Distributed Software and Database Systems* (1986), pp. 3–12.

[55] Hangal, S., and Lam, M. S. Tracking down software bugs using automatic anomaly detection. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering* (New York, NY, USA, 2002), ACM, pp. 291–301.

[56] Hastings, R., and Joyce, B. Purify: Fast detection of memory leaks and access errors. In *Usenix Winter Technical Conference* (1992).

[57] Hauswirth, M., and Chilimbi, T. M. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'04)* (2004).

[58] Hethcote, H. W. The mathematics of infectious diseases. *SIAM Rev. 42*, 4 (2000), 599–653.

[59] Hicks, M., Moore, J. T., and Nettles, S. Dynamic software updating. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation* (New York, NY, USA, 2001), ACM, pp. 13–23.

[60] Jones, R. W. M., and Kelly, P. H. J. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the International Workshop on Automatic Debugging* (May 1997).

[61] Joshi, A., King, S. T., Dunlap, G. W., and Chen, P. M. Detecting past and present intrusions through vulnerability-specific predicates. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles* (New York, NY, USA, 2005), ACM, pp. 91–104.

[62] Kandogan, E., and Bailey, J. Usable autonomic computing systems: The administrator's perspective. In *ICAC '04: Proceedings of the First International Conference on Autonomic Computing (ICAC'04)* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 18–26.

[63] Kim, H.-A., and Karp, B. Autograph: Toward automated, distributed worm signature detection. In *Proceedings of the 13th Usenix Security Symposium* (2004).

[64] King, S. T., Dunlap, G. W., and Chen, P. M. Debugging operating systems with time-traveling virtual machines. In *USENIX* (2005).

[65] Kreibich, C., and Crowcroft, J. Honeycomb: Creating intrusion detection signatures using honeypots. *SIGCOMM Computer Communication Review* (2004).

[66] Laadan, O., Baratto, R. A., Phung, D. B., Potter, S., and Nieh, J. DejaView: a personal virtual computer recorder. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles* (New York, NY, USA, 2007), ACM, pp. 279–292.

[67] Lattner, C., and Adve, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)* (Palo Alto, California, Mar 2004).

[68] Lemos, C. R. Microsoft patch freeses some systems, 2003. http://news.com.com/2100-1002-993515.html.

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

[70] Liang, Z., and Sekar, R. Fast and automated generation of attack signatures: a basis for building self-protecting servers. In *CCS '05* (2005).

[71] Liblit, B., Aiken, A., Zheng, A. X., and Jordan, M. I. Bug isolation via remote program sampling. In *PLDI* (2003).

[72] Locasto, M., Sidiroglou, S., and Keromytis, A. Software self-healing using collaborative application communities. In *Proceedings of NDSS 2006* (2005).

[73] Locasto, M., Wang, K., Keromytis, A., and Stolfo, S. FLIPS: Hybrid adaptive intrusion prevention. In *Proceedings of RAID 2005* (2005).

[74] Lowell, D. E., and Chen, P. M. Free transactions with Rio Vista. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles* (1997).

[75] Lowell, D. E., Saito, Y., and Samberg, E. J. Devirtualizable virtual machines enabling general, single-node, online maintenance. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2004), ACM, pp. 211–223.

[76] Lu, S., Tucek, J., Qin, F., and Zhou, Y. Avio: detecting atomicity violations via access interleaving invariants. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2006), ACM, pp. 37–48.

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

[78] Makris, K., and Ryu, K. D. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *EuroSys '07: Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (New York, NY, USA, 2007), ACM, pp. 327–340.

[79] Manevich, R., Sridharan, M., Adams, S., Das, M., and Yang, Z. PSE: Explaining program failures via postmortem static analysis. *SIGSOFT Software Engineering Notes 29*, 6 (2004), 63–72.

[80] Marcus, E., and Stern, H. *Blueprints for High Availability*. John Willey & Sons, 2000.

[81] McDougall, P. Microsoft pulls buggy Windows Vista SP1 files. InformationWeek, Feb 2008. http://www.informationweek.com/ story/ showArticle.jhtml?articleID=20680 0819.

[82] Mellor, C. Seagate promises second fault fix in 24 hours. The Register, Jan 2009. http://www.channelregister.co.uk/ 2009/ 01/ 21/ seagates_second_fault_fix/.

[83] Microsoft. Revamping the microsoft security bulletin release process, Oct 2003. http://www.microsoft.com/ technet/ security/ bulletin/ revsbwp.mspx.

[84] Misherghi, G., and Su, Z. HDD: Hierarchical delta debugging. In *Proceedings of the 28th International Conference on Software Engineering* (2006).

[85] Moore, D., Paxson, V., Savage, S., Shannon, C., Staniford, S., and Weaver, N. Inside the slammer worm. In *IEEE Security & Privacy* (2003).

[86] mozilla.org. Quality feedback agent. http://www.mozilla.org/quality/qfa.html.

[87] Myers, E. W. An O(ND) difference algorithm and its variations. *Algorithmica 1*, 2 (1986), 251–266.

[88] Nagaraja, K., Oliveira, F., Bianchini, R., Martin, R. P., and Nguyen, T. D. Understanding and dealing with operator mistakes in internet services. In *OSDI '04* (2004), pp. 61–76.

[89] Naraine, R. Another black eye for microsoft patch creation process, 2005. http://www.eweek.com/ article2/0,1759,1879088,00.asp.

[90] Naraine, R. Faulty microsoft update rekindles patch quality concerns, 2005. http://www.eweek.com/ article2/0,1895,1815956,00.asp.

[91] Narayanasamy, S., Pokam, G., and Calder, B. BugNet: Continuously recording program execution for deterministic replay debugging. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA'05)* (2005).

[92] Necula, G. C., McPeak, S., and Weimer, W. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th Annual ACM SIGPLAN - SIGACT Symposium on Principloes of Programming Languages* (2002).

[93] Nethercote, N., and Fitzhardinge, J. Bounds-checking entire programs without recompiling. In *Informal Proceedings of the Second Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE 2004)* (Jan. 2004).

[94] Nethercote, N., and Seward, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2007).

[95] Netzer, R. H. B., and Miller, B. P. Improving the accuracy of data race detection. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming* (1991).

[96] Newsome, J., Brumley, D., and Song, D. Vulnerability-specific execution filtering for exploit prevention on commodity software. In *NDSS* (2006).

[97] Newsome, J., Karp, B., and Song, D. Paragraph: Thwarting signature learning by training maliciously. In *RAID* (Sept. 2006).

[98] Newsome, J., Karp, B., and Song, D. X. Polygraph: Automatically generating signatures for polymorphic worms. In *IEEE Symposium on Security and Privacy* (2005).

[99] Newsome, J., and Song, D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS* (2005).

[100] Organick, E. I. *A programmer's view of the Intel 432 system.* McGraw-Hill, Inc., New York, NY, USA, 1983.

[101] Osman, S., Subhraveti, D., Su, G., and Nieh, J. The design and implementation of Zap: A system for migrating computing environments. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (2002).

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

[103] Pegoraro, R. Apple updates Leopard–again. The Washington Post, Feb 2008. http://blog.washingtonpost.com/ fasterforward/ 2008/ 02/ apple_updates_leopard-again.html.

[104] Perdisci, R., Dagon, D., Lee, W., Fogla, P., and Sharif, M. Misleading worm signature generators using deliberate noise injection. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy* (May 2006).

[105] Perkovic, D., and Keleher, P. J. Online data-race detection via coherency guarantees. In *OSDI '96: Proceedings of the second USENIX symposium on Operating systems design and implementation* (New York, NY, USA, 1996), ACM, pp. 47–57.

[106] Qin, F., Chen, H., Li, Z., Zhou, Y., seop Kim, H., and Wu, Y. LIFT: A low-overhead practical information flow tracking system for detecting general security attacks. In *MICRO* (Dec 2006).

[107] Qin, F., Lu, S., and Zhou, Y. SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs. In *Proceedings of the 10th International Symposium on High-Performance Computer Architecture (HPCA'05)* (Feb 2005).

[108] Qin, F., Tucek, J., Sundaresan, J., and Zhou, Y. Rx: Treating bugs as allergies—A safe method to survive software failures. In *SOSP* (2005).

[109] Qin, F., Tucek, J., and Zhou, Y. Treating bugs as allergies: a safe method for surviving software failures. In *HOTOS'05: Proceedings of the 10th conference on Hot Topics in Operating Systems* (Berkeley, CA, USA, 2005), USENIX Association, pp. 19–19.

[110] Qin, F., Tucek, J., Zhou, Y., and Sundaresan, J. Rx: Treating bugs as allergies—a safe method to survive software failures. *ACM Trans. Comput. Syst. 25*, 3 (2007), 7.

[111] Randell, B. Facing up to faults. *The Computer Journal* (2000).

[112] Rinard, M., Cadar, C., Dumitran, D., Roy, D. M., Leu, T., and Beebee, Jr., W. S. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation* (2004).

[113] Rosander, A. C. *Elementary Principles of Statistics*. D. Van Nostrand Company, 1951.

[114] Russinovich, M., and Cogswell, B. Replay for concurrent non-deterministic shared-memory applications. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation* (New York, NY, USA, 1996), ACM, pp. 258–266.

[115] Ruwase, O., and Lam, M. S. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Annual Netwrok and Distributed System Security Symposium (NDSS)* (Feb. 2004).

[116] Sabelfeld, A., and Myers, A. Language-based information-flow security. In *IEEE Journal on Selected Areas in Communications* (2003).

[117] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. Eraser: a dynamic data race detector for multi-threaded programs. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles* (New York, NY, USA, 1997), ACM, pp. 27–37.

[118] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst. 15*, 4 (1997), 391–411.

[119] Segal, M. E., and Frieder, O. On-the-fly program modification: Systems for dynamic updating. *IEEE Softw. 10*, 2 (1993), 53–65.

[120] Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., and Boneh, D. On the effectiveness of address-space randomization. In *CCS* (2004).

[121] Shannon, C., and Moore, D. The spread of the witty worm. *IEEE Security and Privacy 2*, 4 (2004), 46–50.

[122] Sidiroglou, S., Ioannidis, S., and Keromytis, A. D. Band-aid patching. In *HotDep'07: Proceedings of the 3rd Workshop on Hot Topis in System Dependability* (2007), USENIX Association.

[123] Sidiroglou, S., Locasto, M. E., Boyd, S. W., and Keromytis, A. D. Building a reactive immune system for software services. In *Proceedings of the 2005 USENIX Annual Technical Conference* (Apr 2005).

[124] Singh, S., Estan, C., Varghese, G., and Savage, S. Automated worm fingerprinting. In *Proceedings of the 6th USENIX Symposium on Operating System Design and Implementation* (2004).

[125] Smirnov, A., and cker Chiueh, T. Dira: Automatic detection, identification and repair of control-hijacking attacks. In *NDSS* (2005).

[126] So, B., Miller, B. P., and Fredriksen, L. An empirical study of the reliability of unix utilites. http://www.cs.wisc.edu/~bart/fuzz/fuzz.html.

[127] Sridharan, M., Fink, S. J., and Bodik, R. Thin slicing. *SIGPLAN Not. 42*, 6 (2007), 112–122.

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

[129] Staniford, S., Moore, D., Paxson, V., and Weaver, N. The top speed of flash worms, 2004.

[130] Staniford, S., Paxson, V., and Weaver, N. How to 0wn the internet in your spare time. In *USENIX Security Symposium* (2002).

[131] Student. The probable error of a mean. *Biometrika 6*, 1 (1908), 1–25.

[132] Suh, G. E., Lee, J., and Devadas, S. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'04)* (October 2004), pp. 85–96.

[133] Torvalds, L. Re: [rant] linux-irda status. Linux Kernel Mailing List, Nov. 2000.

[134] Tsai, T. K., and Singh, N. Libsafe: Transparent system-wide protection against buffer overflow attacks. In *DSN* (2002), p. 541.

[135] Tucek, J., Lu, S., Huang, C., Xanthos, S., and Zhou, Y. Automatic on-line failure diagnosis at the end-user site. In *HOTDEP'06: Proceedings of the 2nd conference on Hot Topics in System Dependability* (Berkeley, CA, USA, 2006), USENIX Association, pp. 4–4.

[136] Tucek, J., Lu, S., Huang, C., Xanthos, S., and Zhou, Y. Triage: Diagnosing production run failures at the user's site. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)* (October 2007).

[137] Tucek, J., Newsome, J., Lu, S., Huang, C., Xanthos, S., Brumley, D., Zhou, Y., and Song, D. Sweeper: a lightweight end-to-end system for defending against fast worms. In *EuroSys '07: Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (New York, NY, USA, 2007), ACM, pp. 115–128.

[138] Tucek, J., Xiong, W., and Zhou, Y. Efficient online validation with delta execution. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, March 2009), ACM, pp. 193–204.

[139] US-CERT. Common vulnerabilities and exposures.

[140] Vogels, W., Dumitriu, D., Agrawal, A., Chia, T., and Guo, K. Scalability of the Microsoft Cluster Service. In *USENIX Windows NT Symposium* (Aug 1998).

[141] Vogels, W., Dumitriu, D., Birman, K., Gamache, R., Massa, M., Short, R., Vert, J., Barrera, J., and Gray, J. The design and architecture of the Microsoft Cluster Service. In *FTCS* (Jun 1998).

[142] Wang, H. J., Guo, C., Simon, D. R., and Zugenmaier, A. Shield: vulnerability-driven network filters for preventing known vulnerability exploits. In *SIGCOMM '04* (2004).

[143] Wang, H. J., Platt, J., Chen, Y., Zhang, R., and Wang, Y.-M. Peerpressure for automatic troubleshooting. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)* (2004).

[144] Wang, Z., and Pierce, K. Bmat – a binary matching tool for stale profile propagation. *Instruction-Level Parallelism* (2000).

[145] Watson, G. DMalloc. http://dmalloc.com/.

[146] Weaver, N., and Paxson, V. A worst-case worm. In *Proceedings of the Third Annual Workshop on Economics and Information Security (WEIS04)* (2004).

[147] Weiser, M. Programmers use slices when debugging. *Communications of the ACM 25*, 7 (1982), 446–452.

[148] Wilander, J., and Kamkar, M. A comparison of publicly available tools for dynamic buffer overflow prevention. In *NDSS* (2003).

[149] Witchel, E., Cates, J., and Asanović, K. Mondrian memory protection. In *ASPLOS* (2002).

[150] Xu, J., Kalbarczyk, Z., and Iyer, R. K. Transparent runtime randomization for security. Tech. rep., Center for Reliable and Higher Performance Computing, University of Illinois, May 2003.

[151] Xu, J., Ning, P., Kil, C., Zhai, Y., and Bookholt, C. Automatic diagnosis and response to memory corruption vulnerabilities. In *CCS '05* (2005).

[152] Xu, M., Bodik, R., and Hill, M. D. A "Flight Data Recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th Annual International Symposium on Computer Architecture* (2003).

[153] Xu, M., Bodík, R., and Hill, M. D. A serializability violation detector for shared-memory server programs. In *PLDI* (2005).

[154] Yuan, C., Lau, N., Wen, J.-R., Li, J., Zhang, Z., Wang, Y.-M., and Ma, W.-Y. Automated known problem diagnosis with event traces. In *EuroSys* (2006).

[155] Zeller, A. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering* (2002).

[156] Zhang, X., Gupta, R., and Zhang, Y. Precise dynamic slicing algorithms. In *Proceedings of the 25th International Conference on Software Engineering* (2003).

[157] Zhou, P., Liu, W., Long, F., Lu, S., Qin, F., Zhou, Y., Midkiff, S., and Torrellas, J. AccMon: Automatically detecting memory-related bugs via program counter-based invariants. In *MICRO* (2004).

[158] Zhou, P., Qin, F., Liu, W., Zhou, Y., and Torrellas, J. iWatcher: Efficient Architecture Support for Software Debugging. In *ISCA* (2004).

[159] Zhou, Y., Marinov, D., Sanders, W., Zilles, C., d'Amorim, M., Lauterburg, S., Lefever, R. M., and Tucek, J. Delta execution for software reliability. In *HOTDEP'07: Proceedings of the 23rd Workshop on Hot Topics in System Dependability* (2007).