Triage: Diagnosing Production Run Failures at the User's Site

Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos and Yuanyuan Zhou
Department of Computer Science
University of Illinois at Urbana Champaign
{tucek, shanlu, chuang30, xanthos2, yyzhou}@uiuc.edu

ABSTRACT

Diagnosing production run failures is a challenging yet important task. Most previous work focuses on offsite diagnosis, i.e. development site diagnosis with the programmers present. This is insufficient for production-run failures as: (1) it is difficult to reproduce failures offsite for diagnosis; (2) offsite diagnosis cannot provide timely guidance for recovery or security purposes; (3) it is infeasible to provide a programmer to diagnose every production run failure; and (4) privacy concerns limit the release of information (e.g. coredumps) to programmers.

To address production-run failures, we propose a system, called *Triage*, that automatically performs onsite software failure diagnosis at the very moment of failure. It provides a detailed diagnosis report, including the failure nature, triggering conditions, related code and variables, the fault propagation chain, and potential fixes. Triage achieves this by leveraging lightweight reexecution support to efficiently capture the failure environment and repeatedly replay the moment of failure, and dynamically—using different diagnosis techniques—analyze an occurring failure. Triage employs a failure diagnosis protocol that mimics the steps a human takes in debugging. This extensible protocol provides a framework to enable the use of various existing and new diagnosis techniques. We also propose a new failure diagnosis technique, *delta analysis*, to identify failure related conditions, code, and variables.

We evaluate these ideas in real system experiments with 10 real software failures from 9 open source applications including four servers. Triage accurately diagnoses the evaluated failures, providing likely root causes and even the fault propagation chain, while keeping normal-run overhead to under 5%. Finally, our user study of the diagnosis and repair of real bugs shows that Triage saves time (99.99% confidence), reducing the total time to fix by almost half.

Categories and Subject Descriptors

D.4.5 [**Operating Systems**]: Reliability; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Experimentation, Reliability

1. INTRODUCTION

1.1 Motivation

Software failures are a major contributor to system down time and security holes. As software has grown in size, complexity and cost, software testing has become extremely expensive, frequently requiring hundreds of programmers dedicated to testing. Additionally, this has made comprehensive software testing far more difficult. Consequently, it has become inevitable that even production software packages contain a significant number of bugs, which result in software failures (e.g. crashes, hangs, incorrect results, etc.) during production runs at end users' sites. Since these errors directly impact end users, software vendors make them the highest priority and devote extensive time and human resources to releasing timely patches.

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 provide a security defense against fast internet worms. (3) Programmers cannot be provided onsite to debug every end-user failure. (4) Privacy concerns prevent many users from sending failure information, such as coredumps or execution traces, back to programmers. 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, 45, 48]. Typically programmers still need manual debugging with the aid of other diagnosis techniques to collect enough information to thoroughly understand a failure.

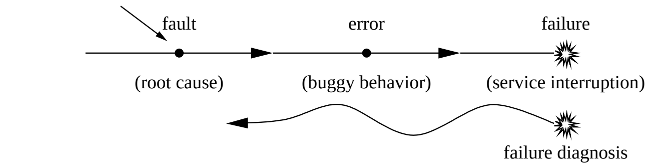


Figure 1: Failure diagnosis is driven by occuring failures and tries to understand the whole fault chain

Following the definition used in software dependability [33], comprehensive knowledge of a software failure includes the fault, error, and trigger (shown in Figure 1). Consequently, failure diagnosis targets three things: (1) what execution misbehaviors caused the failure, (2) where the misbehaviors came from, 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 *occurring* problem.

1.2 Current State of the Art

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 [10], program slicing [1, 45, 48], and offline partial execution path constructors from a coredump such as PSE [19]. Almost all of 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 Microsoft's Dr. Watson [20] and the Mozilla Quality Feedback Agent [22], 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 [7], system call traces [43], traces of predicated values [16], etc. Furthermore, some deterministic replay tools [9, 14, 41] have also been developed for uniprocessor systems.

While these techniques are helpful for in-house analysis of production run failures, they are still limited because (1) the data sent back are still raw, leaving the majority of the diagnosis task for programmers to conduct manually; and (2) they may not be applicable due to end users' privacy concerns.

1.3 Challenges for Onsite Diagnosis

Unfortunately, providing onsite diagnosis is not simply a matter of slapping together a bunch of diagnosis techniques. We feel that this is because 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**: We cannot provide a programmer 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**: We have no knowledge of what failures are about to happen. So any failure-specific techniques (e.g. memory bug monitoring) are a total waste during normal execution, prior to failure.

1.4 Our Contributions

In this paper, we propose *Triage*, the first (to the best of our knowledge) 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-reexecution 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, modifying variables) can be repeatedly applied to multiple iterations of reexecution focusing only on the moment leading up to the failure. In this scheme, our diagnosis has most failure-related information at hand; meanwhile both normal-run overhead and diagnosis times are minimized. In combination with system support for reexecution, heavy-weight bug detection and analysis tools become feasible for onsite diagnosis. Furthermore, we can relive the failure moment over and over. We 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 automatically conducts *delta analysis* to narrow down the failure-related code paths and variables.

(3) An automated, top-down, human-like software failure diagnosis protocol. As we will show in Figure 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 reexecution 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 reexecution to find the likely type of program misbehavior that caused the failure. This includes 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 the 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 reexecution with instrumentation and the guidance of the TDP diagnosis protocol allow Triage to synergistically use previous failure analysis techniques. We have implemented 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.

We evaluate Triage using a real system implementation and experiments on Linux with 10 real software failures from 9 applications (including 4 servers: MySQL, Apache, CVS and Squid). Our experimental results show that Triage, including our 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, we performed a user study with 15 programmers. The results show that the diagnostic information provided by Triage shortens the time to diagnose real bugs (statistically significant with $p < .0001$), with an average reduction of 44.6%.

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 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 reexecution (simple replay or reexecution with controlled execution perturbation and variation) for failure diagnosis. Second, the *control* group deals with deciding how the subcomponents should all interact, and implements the Triage Diagnosis Protocol (see Section 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 and 5.

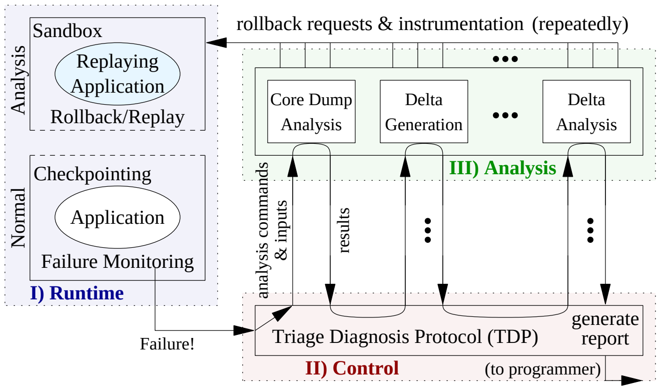


Figure 2: Triage architecture overview

Checkpoint and Reexecution.

In order to allow repeated analysis of the failure, Triage requires checkpoint and reexecution support. There are many ways to implement reexecution, such as Time Traveling Virtual Machines [14] or Flashback [41]. Triage leverages the lightweight checkpoint and reexecution runtime system provided in our previous work, Rx. We briefly describe it here; details can be found it [32, 41].

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 [17, 41] 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 reexecution. This replay may be potentially modified to suit the current reexecution 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 reexecution 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 reexecution, 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 or 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 (input-related manipulations, see § 4), and briefly considers some radical changes (patching the code).

Lightweight Monitoring to detect failure and global execution history.

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 [31, 25] 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 histories or system call traces that will be useful for onsite diagnosis upon a failure. Previous work [4] has shown that branch history collection imposes less than 5% overhead. In our current implementation we rely on assertions and exceptions as our only normal-run monitors.

Control Layer.

The process of applying diagnosis tools through multiple reexecutions is guided by the control layer, which implements the Triage

Diagnosis Protocol described in Section 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 to the off-site programmers for them to use in fixing the bug.

Analysis (Failure Diagnosis) Layer.

Figure 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 reexecution with perturbation and variation. Dynamic backward slicing, although previously proposed, is made much more feasible in post-hoc application during reexecution. Finally, delta analysis is newly proposed. The details of these techniques are presented in Section 4, 5, and 6.

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. We first present the default protocol using some representative fault analysis techniques, and then discuss extensions and customizations to the default protocol.

```
Block  Line   Code
        1   char *
        2   get_directory_contents
  A     3   (char * path, dev_t device){
        4     char * dirp = savedir(path);
        5     char const * entry;
  B     6     /*More variable declarations*/
        7     if(!dirp)
  C     8       savedir_error(path);
        9     errno = 0;
  D    10     /*More code omitted*/
       11     if(children != NO_CHILDREN)
  E    12       for(entry = dirp;
       13           (len = strlen(entry));
       14           entry += len + 1;){
  F    15         /*Omitted*/
       16       }
  G    17     /*Omitted*/
       18   }

       19   char *
       20   savedir
  H    21   (const char *dir){
       22     DIR *pdirp;
       23     /*More variable declarations*/
       24     dirp = opendir(dir);
  I    25     if(dirp == NULL)
  J    26       return NULL;
  K    27     /*Omitted*/
       28   }
```

Figure 4: Simplified excerpt of a real bug in tar-1.13.25 as a running example to explain the diagnosis protocol.

3.1 Default Protocol

Figure 3 shows a flow chart of the default diagnosis protocol after a failure is detected. Triage uses different diagnosis techniques (some new and some modified from existing work) to automatically collect different types of diagnostic information (as described in the Introduction) 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 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 shows a simplified version of a bug in tar, the common Unix archive program, which we will use 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 reexecuted from a previous checkpoint, each time with a bug detection technique dynamically attached, 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 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 reexecution, during diagnosis, *after* a failure has occurred. For tar, we find that 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, we use a technique we call *delta generation* (§ 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 our previous Rx work [32] that varies execution environments to bypass deterministic failures for recovery, our 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* (§ 5) and with a modified dynamic backward slicing technique [45]. From the delta generation, Triage obtains many failed replays as well as successful replays from previous check-

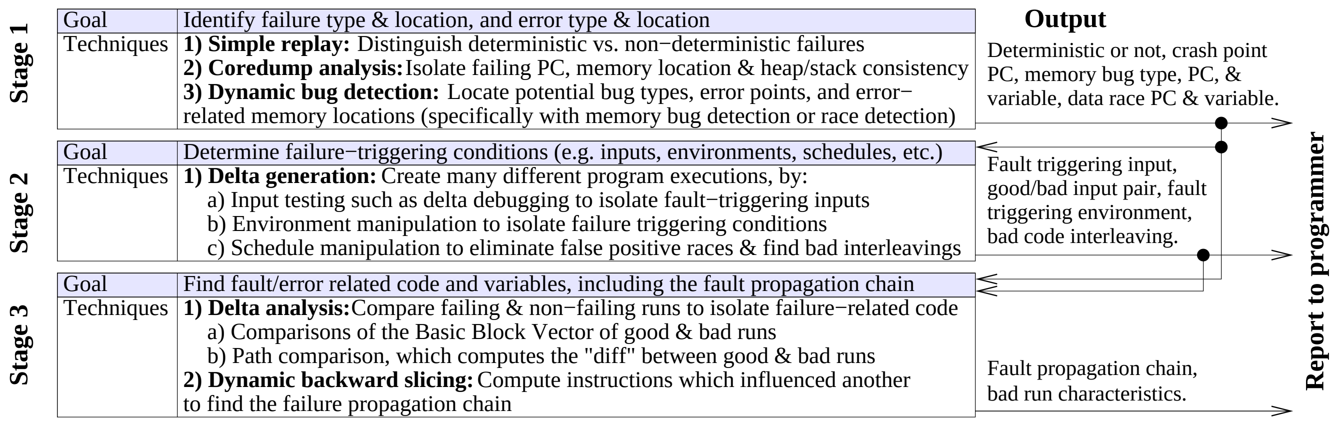

Stage 1
Goal
Identify failure type & location, and error type & location
Techniques
1) Simple replay: Distinguish deterministic vs. non−deterministic failures
2) Coredump analysis: Isolate failing PC, memory location & heap/stack consistency
3) Dynamic bug detection: Locate potential bug types, error points, and error−related memory locations (specifically with memory bug detection or race detection)
Output
Deterministic or not, crash point PC, memory bug type, PC, & variable, data race PC & variable.
Stage 2
Goal
Determine failure−triggering conditions (e.g. inputs, environments, schedules, etc.)
Techniques
1) Delta generation: Create many different program executions, by:
a) Input testing such as delta debugging to isolate fault−triggering inputs
b) Environment manipulation to isolate failure triggering conditions
c) Schedule manipulation to eliminate false positive races & find bad interleavings
Fault triggering input, good/bad input pair, fault triggering environment, bad code interleaving.
Stage 3
Goal
Find fault/error related code and variables, including the fault propagation chain
Techniques
1) Delta analysis: Compare failing & non−failing runs to isolate failure−related code
a) Comparisons of the Basic Block Vector of good & bad runs
b) Path comparison, which computes the "diff" between good & bad runs
2) Dynamic backward slicing: Compute instructions which influenced another to find the failure propagation chain
Fault propagation chain, bad run characteristics.
Report to programmer

Figure 3: Diagnosis protocol (TDP) diagram (This figure illustrates the Triage diagnosis protocol, including the failure diagnosis components we have implemented. These separate analysis components are run in one or more iterations of reexecution, during which all side-effects are sandboxed. Later stages are fed results from earlier stages as necessary.)

points. By comparing the code paths (and potentially data flows) from these replays, Triage finds the differences betweed 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; our 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 our user study) can be seen in Table 5.

3.2 Protocol extensions and variations

The protocol and diagnostic techniques discussed above provide good results for diagnosis, and indeed represent what we have 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 unproposed, which could be added. For instance, information flow tracking [28, 36] 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 (we use cost to determine 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). We are currently exploring the possibilities of extending and varying the protocol, as well as reducing the amount of effort it takes to add in new components.

The dynamic backward slice and the results from delta analysis can be combined through intersection. That is, we can consider to be more relevant those portions of the backward slice which are also in the path delta (see Section 5). This will not only identify the code paths which are possibly in the propagation chain, but highlight those which differ from normal execution. We are considering ways in which this information may be best presented to the programmer while at the same time considering that portions of the propagation chain may not be in the path delta.

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, we can begin automatically filtering failure-triggering inputs (as in [5, 13, 42]), to avoid triggering the bug in the future. With a higher degree of risk, we may be able to generate a patch. Currently we have preliminary results of identifying heap-buffer overflows and repairing them, by instrumenting the calculation for how large the buffer must be allocated. Finally, since our goal is merely to gather diagnostic information, we can attempt quite "risky" fixes, such as dynamically deleting code or changing variable values, in an attept to see which changes will prevent failure during replay. Such speculative techniques that were proposed for recovery, such as failure oblivious computing [34], or forcing the function to return an error value as proposed in STEM [38], 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 diagnosis process. We are further considering automatic patch generation, as well as using "unsafe" patches for diagnostic purposes.

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 reexecution) 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, 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 [8], Autograph [13], Sweeper [42] and others [39, 15, 30], or avoiding failure triggering execution environments like in the Rx recovery system [32].

To achieve the above goals, our delta generator automatically and repeatedly replays from a recent checkpoint, with controlled changes in input and execution environment (including dropping messages, modifying inputs, changing allocation sizes, and modifying scheduling). 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 [47], 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 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 reexecution from a previous checkpoint. If the failure is caused by combinations of requests, Triage finds minimal triggers by applying hierarchical delta debugging [21].

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 [40], "twiddling" the request, to create a non-failing request with the "minimum distance" from the failing one. Triage focuses on deleting characters to create smaller inputs, but it will also randomly change characters. For well-structured inputs, like HTTP requests, Triage will make protocol-aware changes; specific examples include changing "http:" to "ftp:", and dropping semantically consistent portions of the request (that is, whole HTTP directives). For some bugs, 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 (which particular portion of a protocol), we 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 (e.g. a different memory layout of thread scheduling) then execution could also be different. This can be done artificially by modifying the execution environment during reexecution. There are several known techniques proposed by previous work such as Rx [32] and DieHard [3]. Triage will pad or zero-fill new allocations, change message order, drop messages, manipulate thread scheduling, and modify the system environment. 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 our previous Rx work, we already have some idea about the failure based on earlier failure analysis steps. We can target our perturbations directly at the expected fault. For example, for a deterministic failure, Triage does not attempt different thread schedules. Similarly, given we know that 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, as described below.

Speculative changes (preliminary).

Our execution perturbation during replay can be speculative since all side-effects during replay are sandboxed and discarded. For example, during replay, we can force the control flow to fall through a non-taken branch edge. We can also forcefully change some key data's value during replay. The new value can 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 can be useful for diagnosis. We are currently exploring which sorts of changes are likely to produce most fruitful for diagnosis.

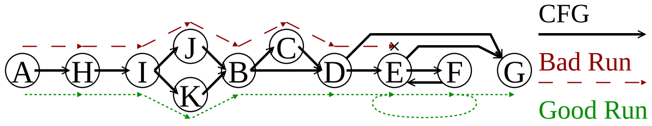


Figure 5: Control flow graph and two executions of our running tar bug example shown in Figure 4

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, we could increase the granularity to the instruction level or reduce it 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, we consider the basic-block level to be a good trade-off.

Figure 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}.

5. DELTA ANALYSIS

Based on the detailed information from many failing and non-failing replays produced by the delta generator, the delta analyzer examine 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 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. But we can also tradeoff accuracy for performance since we do not necessarily need to find the minimum pair—as long as a pair of failing and non-failing replays are reasonably similar, it may be sufficient.

In our running example shown on Figure 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 [44] would allow us to answer a key question–is there a statistically significant difference between the exercise count of each individual basic block? Currently, our implementation does not consider such tests.

Path Comparison.

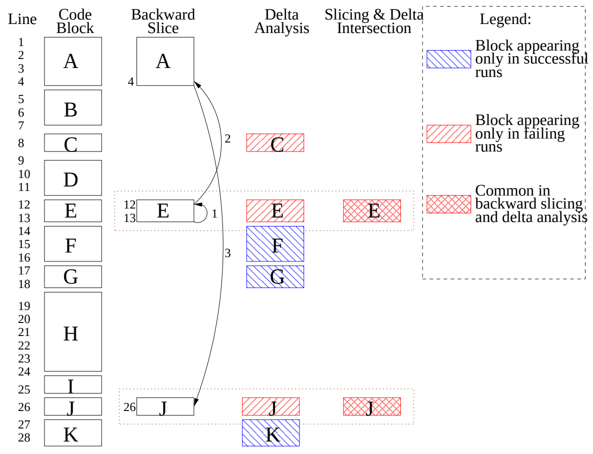
While the BBV difference is helpful to identify basic blocks unique to failing runs, it does not consider basic block execution order and sequence. This limitation is addressed by our path comparison. Our goal with the path difference is to identify those segments of execution where the paths of the 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; we use the $O(ND)$ approach found in [23]. The path comparison algorithm also records these edit operations that give the minimum edit distance between the two path sequences.

In our running example, we would be finding the minimum edit distance between AHIJBCDE (the failing run) and AHIKBDEFEF-...EG (the non-failing run). In `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, an extra block C, and is truncated prior to the EF loop.



Line
Code Block
Backward Slice
Delta Analysis
Slicing & Delta Intersection
Legend:
Block appearing only in successful runs
Block appearing only in failing runs
Common in backward slicing and delta analysis
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
A
B
C
D
E
F
G
H
I
J
K
12
13
26
1
2
3

Figure 6: An example presenting backward slicing, delta analysis results and their intersection. It is derived from Figure 4.

Backward Slicing and Result Intersection.

To further highlight important information and eliminate noise, we extract those path differences which are related to the failure, i.e. in the fault propagation chain. 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 [45]. The intersection results can help us focus 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 6, for a given instruction (the starting point of a backward slice), its data or control dependent code lines are extracted and a lot of irrelevant code lines (shown in Figure 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, we also report the whole backward slice results in the Triage report.

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 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 reexecution 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 [48] we 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, we use just one dependency, "return value depends on input arguments". This greatly reduces the overhead for some commonly-called library functions. Our experiments show that the resulting total analysis overhead is acceptably low.

Returning again to our example, the difference between our two replays lies in the blocks {+J, -K, +C, +E, -F, -G}, *+J* meaning that block *J* either appears only in the failing run or contains the failing instruction and *-K* meaning that *K* appears only in 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. If we consider these {E, J, C, K, F, G}, we can rank the key differences 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 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 we 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. Also it is hard to statistically compare data of various types such as floating point variables, etc. Therefore, our tool does not yet perform any data delta analysis. This remains as future work.

6. OTHER DIAGNOSIS TECHNIQUES

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

Core dump analysis.

The Triage coredump analyzer considers 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, we generate 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[1]. Currently, Triage employs two types of dynamic common-bug detectors, a memory bug detector and a race detector. These are only used during reexecution via dynamic instrumentation [18] to address overhead concerns. Triage's **memory bug detector** (MD) detects memory misbehaviors during reexecution to search for four types of memory errors (stack smashing, heap overflow, double free, and dangling pointers) using techniques similar to previous work [11, 26]. 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 [27] by instrumenting memory accesses with PIN; other techniques [37] would also certainly work.

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

7. LIMITATIONS AND EXTENTIONS

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[2] is "transparent"– comprehending what is being sent in a Triage report, and verifying that nothing confidential is being leaked, is much easier than understanding a memory image.

Automatic patch generation.

Triage provides a wealth of information about occurring failures; we have attempted to use this information to automatically generate patches. However, without an understanding of the semantics of the program, our success has been limited. For heap buffer overflows, we can identify the allocation point of buffer which overflows. Similarly to Rx [32], we can apply padding; unlike Rx we have identified one particular allocation point. As we are not blindly extending every buffer, we can apply a permanent padding. We try a linear buffer increase up to a cutoff, and then we try 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 9) which addresses all possible triggers. Currently, we are unable to create correct patches for any 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 [12], to help the heavy-weight instrumentation used during reexecution. 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 reported as rare by previous work [24], 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, assuming sufficient disk space. The challenge lies in quickly replaying long windows of execution to reduce diagnosis time. Solving this challenge is future work.

Reproduce nondeterministic bugs on multiprocessor architectures.

The current prototype of Triage supports deterministic replay of single-threaded applications, and mostly determinstic replay of multithreaded applications on uniprocessor architectures. It is very difficult to support this functionality with low overhead in multi-

[2]See, for example, Tables 3, 4, and 5

Name	Program	App Description	#LOC	Bug Type	Root Cause Description
Apache1	apache-1.3.27	a web server	114K	Stack Smash	Long alias match pattern overflows a local array
Apache2	apache-1.3.12	a web server	102K	Semantic (NULL ptr)	Missing certain part of url causes NULL pointer dereference
CVS	cvs-1.11.4	GNU version control server	115K	Double Free	Error-handling code placed at wrong order leads to double free
MySQL	msql-4.0.12	a database server	1028K	Data Race	Database logging error in case of data race
Squid	squid-2.3	a web proxy cache server	94K	Heap Buffer Overflow	Buffer length calculation misses special character cases
BC	bc-1.06	interactive algebraic language	17K	Heap Buffer Overflow	Using wrong variable in for-loop end-condition
Linux	linux-extract	extracted 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	GNU tar archive tool	27K	Semantic (NULL ptr)	Directory property corner case is not well handled

Table 1: Applications and real bugs evaluated in our experiments.

processor architectures. addressing this problem would require advanced, recently proposed, hardware support such as Flight Data Recorder [46] and BugNet [24] to achieve deterministic replay.

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. A second 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 our experiments we have never encountered any cases that Triage reports misleading information, it is conceivable that in some rare cases Triage may report some wrong 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 incorporating the accuracy of each technique into the result confidence ranking.

8. EVALUATION METHODOLOGY

To evaluate Triage, we conduct 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 [18]. After a failure occurs, Triage dynamically attaches PIN to the target program in the beginning of every reexecution attempt.

Machine environment and parameters.

Our experiments are 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 1 shows the 9 applications (4 server and 5 open source utilities) and 10 bugs we 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, we conducted a user study. We 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 with a full suite of programming tools, including Valgrind. To balance the difficulty of the bugs, we randomly gave the programmers the error reports produced by Triage for half of the bugs, and, as a control, we denied them the error reports for the other half of the bugs. Aside from formatting, Table 5 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[3]. The time limits were necessary for practical purposes; participants averaged approximately two and a half hours of total time each.

We ran our experiments with 15 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 [35]. This test compares each subject against themselves, to help account for individual programmer skill; the variation of individual programmer skill do still appear in the overall means. To improve our results, we are still continuing our user study with more participants.

9. EXPERIMENTAL RESULTS

Table 2 presents a summary of Triage's diagnosis results for each failure. For the four deterministic server bugs, we present results from input testing/delta generation, delta analysis, and backward

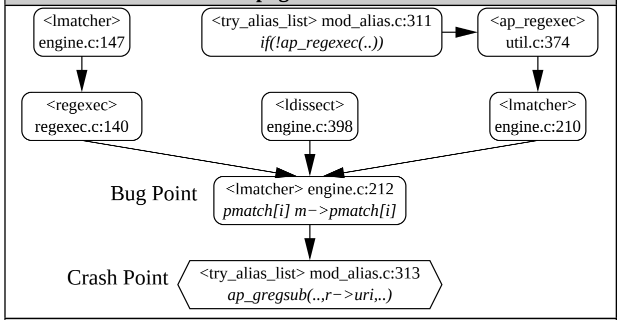
[3]These time limits artificially show Triage in a bad light because they bound the maximum time; this improves the performance of the non-Triage cases.

Server Applications			
App	**Results - Stages 2 & 3**		
	Method	**Results**	**Useful**
Apache1	Input	*GET /trigger/crash.html ...*	√
	Testing	Key part: */trigger/crash.html*	√
	Backward	Found root-cause ***line***	√
	Slicing	8 instructions from crash	√
	Delta	Edit distance is 79089	√
	Analysis	Removes 12% of dynamic blocks	√
Apache2	Input	*GET ... Referer:1.2.3.4*	√
	Testing	Key part: *Referer:*	√
	Backward	Found root-cause ***line***	√
	Slicing	3 instructions from crash	√+
	Delta	Edit Distance is 5964	√
	Analysis	Removes 69% of dynamic blocks	√
CVS	Input Testing	*Stream of requests...*	√
	Backward	Found root-cause ***function***	√
	Slicing	4 functions from crash	√
	Delta	No result	
	Analysis	Not applicable	
MySQL	Schedule Manipulate	Bad interleaving pair: 0x8132fa8 – 0x8128c4b	√+
		Found root-cause	√+
Squid	Input	*ftp://user\ *30:p@...*	√
	Testing	Key parts: *ftp, user*	√
	Backward	Found root-cause ***line***	√
	Slicing	6 instructions from crash	√
	Delta	Edit distance is 54310	√
	Analysis	Removes 71% of dynamic blocks	√
Other Open-Source Applications			
BC	Backward	Found root-cause ***line***	√
	Slicing	3 instructions from crash	√+
	Delta	Edit distance is 5381	√
	Analysis	Removes 98% of dynamic blocks	√+
Linux-extr.	Backward	Found root-cause ***line***	√
	Slicing	6 instructions from crash	√
	Delta	No result	
	Analysis	Not applicable	
MAN	Backward	Found root-cause ***function***	√
	Slicing	9 functions from crash	√
	Delta	No result	
	Analysis	Not applicable	
NCOMP	Backward	Found root-cause ***line***	√
	Slicing	5 instructions from crash	√
	Delta	No result	
	Analysis	Not applicable	
TAR	Backward	Found root-cause ***line***	√
	Slicing	6 instructions from crash	√
	Delta	Edit distance is 83564	√
	Analysis	Removes 68% of dynamic blocks	√

Table 2: Diagnosis results. √+ indicates exceptionally good results. For all of the bugs, Stage 1 (identify failure/error types and locations) works as well as similar tools in existing literature.

slicing. As the nondeterministic bug isn't reproduced during simple replay, we instead perform schedule manipulation. Finally, for the five application bugs, there is no input stream and hence we do not provide input testing; further, our delta generation only worked for BC and TAR of the application bugs, so we only provide delta analysis for BC and TAR.

In all cases, Triage correctly diagnosis the nature of the bug (deterministic or nondeterministic), and in all 6 applicable cases Triage correctly pinpoints the bug type, buggy instruction, and memory location. Hence, Table 2 omits detailed listing of Stage 1 results. To summarize Stage 2 and 3 results, for all the 5 server applications,

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 ...
Close non-failing inputs:	GET / HTTP/1.1 ...

<lmatcher> engine.c:147
<try_alias_list> mod_alias.c:311 if(!ap_regexec(..))
<ap_regexec> util.c:374
<regexec> regexec.c:140
<ldissect> engine.c:398
<lmatcher> engine.c:210
Bug Point
<lmatcher> engine.c:212 pmatch[i] m−>pmatch[i]
Crash Point
<try_alias_list> mod_alias.c:313 ap_gregsub(..,r−>uri,..)

Table 3: Triage report for Apache-1.3.27 version

Triage successfully captures and reproduces the fault-triggering input. Also, for the cases where delta analysis is applied, it reduces the amount of execution (as measured by dynamic basic blocks, from the checkpoint) that must be considered by 63%; for the best case (BC) it reduces it by 98%. In 8 of the 10 cases, the root cause instruction appears 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 finds an example interleaving pair which is the trigger for the failure.

9.1 Triage Report Case Studies

In this section, we use three case studies to show how Triage reports can help developers understand failures.

Case 1: Apache.

The bug in Apache 1.13.27 is a stack related, difficult to reproduce and diagnose bug. As shown in Table 3, 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 more 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.

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*).

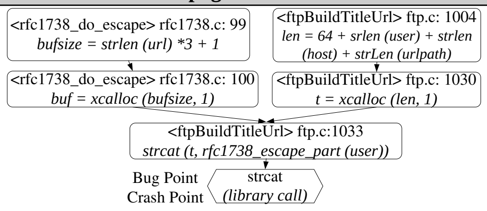
Triage Report for Application — Squid	
Failure Point Information	
Segment Fault (at) Stack/Heap?	Instruction: 0x4f0f0907 (in lib. $strcat$) called from ftp.c:1033 OK/**Corrupt**
Bug Detection Information	
Deterministic Bug? Heap Overflow (at)	Yes lib. $strcat$ called from ftp.c:1033
Fault Propagation Information	
 <rfc1738_do_escape> rfc1738.c: 99 *bufsize = strlen (url) *3 + 1* <ftpBuildTitleUrl> ftp.c: 1004 *len = 64 + srlen (user) + strlen (host) + strLen (urlpath)* <rfc1738_do_escape> rfc1738.c: 100 *buf = xcalloc (bufsize, 1)* <ftpBuildTitleUrl> ftp.c: 1030 *t = xcalloc (len, 1)* <ftpBuildTitleUrl> ftp.c:1033 *strcat (t, rfc1738_escape_part (user))* Bug Point Crash Point strcat *(library call)*	
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: Triage report for Squid 2.3-Stable5 version

Case 2: Squid.

As shown in Table 4, 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. We can be 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 `str-len(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. Hence multiplying the allocation size of the `t` buffer by 4 is sufficient to avoid triggering the bug.

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.

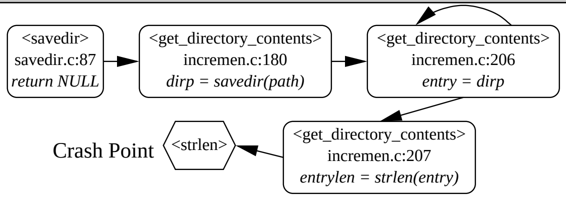
Triage Report for Application — Tar	
Failure Point Information	
Segment Fault (at) Stack/Heap?	Instr: 0x4f0f1e13 (in lib. $strlen$) called from increment.c:207 OK/OK
Bug Detection Information	
Deterministic Bug?	Yes
Null Pointer (at)	lib. *strlen*, called from incremen.c:207
Fault Propagation Information	
 <savedir> savedir.c:87 *return NULL* <get_directory_contents> incremen.c:180 *dirp = savedir(path)* <get_directory_contents> incremen.c:206 *entry = dirp* Crash Point <strlen> <get_directory_contents> incremen.c:207 *entrylen = strlen(entry)*	

Table 5: Triage report for tar-1.13.25

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

Case 3: tar.

Case study three is our running example (see Figure 4). Briefly, Table 5 shows the output of Triage on this bug. Since we have discussed this bug in previous sections, we do not explain the results further. Of note is that the figure shows exactly the same information provided in the user study.

9.2 Normal Execution Overhead

Triage imposes negligible overhead during execution; it should be nearly indistinguishable from the overhead of the underlying checkpoint system [32]. Figure 7 shows the results for three applications: Squid (network bound), BC (CPU bound), and MySQL (both). To explore the effects of checkpoint interval, we also run squid 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 we only run analysis *after* a failure has occurred.

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 6 lists the diagnosis time break down for three representative applications: an IO and 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 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 [23]. 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.

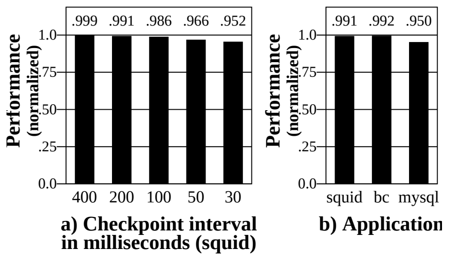


Figure 7: Normalized performance during normal-time execution. a) shows squid at checkpoint intervals from 400 ms to 30 ms, while b) shows squid, MySQL, and bc at 200ms intervals.

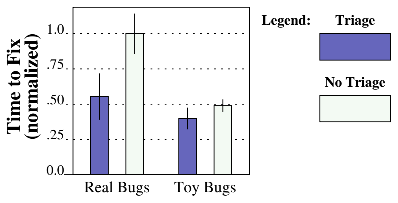


Figure 8: Results from the user study, with error bars showing 95% confidence intervals. Normalization is to real bugs without Triage

9.4 User Study

Our user study (described in Section 8) has demonstrated very positive results. As shown in Figure 8, on average programmers took 44.6% less time debugging the real bugs when they had the diagnostic information provided by Triage (13.468 ± 3.984 minutes versus 24.298 ± 3.458 minutes). A paired T-test shows that this is significant at the 99.99% confidence interval ($p = .0000511$), 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 18.4% (9.690 ± 1.876 minutes versus 11.877 ± 1.096 minutes), with significance at 95% confidence ($p = .0486$); although Triage still helped, the effect was not as large since the toy bugs are very simple and straightforward to diagnose even without Triage.

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 goose chases through auto-generated parser code which, although close in time to the bug, was unrelated. In contrast, one participant said that "[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 a user study (sample representativeness, sample set size, etc.), we believe that these results still provide strong evidence about the usefulness of Triage in helping programmers diagnose software failures.

10. RELATED WORK

Software Failure Offline Diagnosis.

As discussed in Section 1, most existing software failure diagnosis focus 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 [10], program backward and forward slicing [1, 45, 48], deterministic replay tools [14, 41], and delta debugging techniques [21, 47] (described briefly in Section 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 the Introduction. 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 complements the above work because it provides 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 effectively use of all failure information without violating the end user's privacy concerns.

Dynamic Software Bug Detection.

Our work is related to and well complemented by dynamic software bug detection tools, such as Purify [11]. 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 our Triage framework, many of them 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, 45, 48].

Checkpointing and Reexecution.

Triage is related to previous checkpointing system such as Zap [29], FlashBack [41], Rx [32], and TTVM [14], just to name a few, most of which are used for recovery or interactive debugging. Triage uses checkpoint, rollback, and reexecution 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 2, even for checkpoint and reexecution, our proposed project has different requirements, namely all side effects are sandboxed, no need to deal with output commit problems and allowing speculative reexecution such as forcefully skipping code and modifying variables.

Distributed Systems Fault Localization.

Recently some research efforts have been devoted to pinpointing faults (failures [6] 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.

11. CONCLUSIONS AND FUTURE WORK

This paper presents an innovative approach for diagnosing software failures. By leveraging lightweight checkpoint and reexecution techniques, Triage captures the moment of a failure, and *automatically* performs diagnosis at the end user's site. We propose a *failure diagnosis protocol*, which mimics the debugging process a programmer would take. Additionally, we propose 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.

While Triage provides an important first step towards onsite failure diagnosis, there is more work to be done, as discussed in detail in Section 7. Currently we are improving Triage to deal with false positives, through improved confidence ranking and cross correlation of results. Also, we are extending Triage with additional bug-detection, fault analysis tools, and lightweight fault-based sensors, as well as refining the delta generation and delta analysis techniques. Finally, we are considering how to extend Triage to support diagnosis of distributed applications.

12. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their useful feedback, as well as the shadow PC reviewers. We also thank our shepherd, Peter Chen, for his guidance in preparing the final version. Thank you to all of the volunteers for the human study, for all of the time collectively spent debugging.

This material is based upon work supported in part by the National Science Foundation under Grant Nos. 0325603, 0347854, and 0615372, by the Department of Energy under Grant No. DE-FG02-05ER25688, and by Intel Gift grants. Joseph Tucek was awarded an SOSP student travel scholarship, supported by Hewlett-Packard, to present this paper at the conference.

13. REFERENCES

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

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

[3] E. D. Berger and B. G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, 2006.

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

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

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

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

[8] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, 2005.

[9] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay debugging for distributed applications. In *Proceedings of the 2006 USENIX Annual Technical Conference*, 2006.

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

[11] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the 1992 USENIX Winter Technical Conference*, 1992.

[12] M. Hauswirth and T. M. Chilimbi. 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*, 2004.

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

[14] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the 2005 USENIX Annual Technical Conference*, 2005.

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

[16] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, 2003.

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

[18] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, 2005.

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

[20] Microsoft Corporation. Dr. Watson overview.

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

[22] mozilla.org. Quality feedback agent.

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

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

[25] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the*

29th Annual ACM SIGPLAN - SIGACT Symposium on Principloes of Programming Languages, 2002.

[26] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 2003.

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

[28] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, 2005.

[29] S. Osman, D. Subhraveti, G. Su, and J. Nieh. 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.

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

[31] F. Qin, S. Lu, and Y. Zhou. Safemem: Exploiting ECC-Memory for detecting memory leaks and memory corruption during production runs. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, 2005.

[32] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies — A safe method to survive software failures. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, 2005.

[33] B. Randell. Facing up to faults. *The Computer Journal*, 2000.

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

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

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

[37] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4), 1997.

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

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

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

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

[42] J. Tucek, J. Newsome, S. Lu, C. Huang, S. Xanthos, D. Brumley, Y. Zhou, and D. Song. Sweeper: A lightweight end-to-end system for defending against fast worms. In *Proceedings of the 2007 EuroSys Conference*, 2007.

[43] C. Verbowski, E. Kiciman, A. Kumar, B. Daniels, S. Lu, J. Lee, Y.-M. WAng, and R. Roussev. Flight data recorder: Monitoring persistent-state interactions to improve systems management. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, 2006.

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

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

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

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

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