First-Aid: Surviving and Preventing Memory Management Bugs during Production Runs

Qi Gao Wenbin Zhang Yan Tang Feng Qin

Dept. of Computer Science and Engineering
Ohio State University, Columbus, OH, 43210, U.S.A.
{gaoq,zhangwen,tangya,qin}@cse.ohio-state.edu

Abstract

Memory bugs in C/C++ programs severely affect system availability and security. This paper presents First-Aid, a lightweight runtime system that survives software failures caused by common memory management bugs and prevents future failures by the same bugs during production runs. Upon a failure, First-Aid diagnoses the bug type and identifies the memory objects that trigger the bug. To do so, it rolls back the program to previous checkpoints and uses two types of environmental changes that can *prevent* or *expose* memory bug manifestation during re-execution. Based on the diagnosis, First-Aid generates and applies runtime patches to avoid the memory bug and prevent its reoccurrence. Furthermore, First-Aid validates the consistent effects of the runtime patches and generates on-site diagnostic reports to assist developers in fixing the bugs.

We have implemented First-Aid on Linux and evaluated it with seven applications that contain various types of memory bugs, including buffer overflow, uninitialized read, dangling pointer read/write, and double free. The results show that First-Aid can quickly diagnose the tested bugs and recover applications from failures (in 0.084 to 3.978 seconds). The results also show that the runtime patches generated by First-Aid can prevent future failures caused by the diagnosed bugs. Additionally, First-Aid provides detailed diagnostic information on both the root cause and the manifestation of the bugs. Furthermore, First-Aid incurs low overhead (0.4-11.6% with an average of 3.7%) during normal execution for the tested buggy applications, SPEC INT2000, and four allocation intensive programs.

Categories and Subject Descriptors D.4.5 [*Operating Systems*]: Reliability

General Terms Design, Experimentation, Reliability

Keywords Memory Bug Diagnosis, Software Reliability, Software Failure, Error Prevention

EuroSys'09, April 1–3, 2009, Nuremberg, Germany.

1. Introduction

1.1 Motivation

Memory management bugs, such as buffer overflows and dangling pointers, are a major category of common software defects and severely affect system availability and security. During production runs, these types of bugs can corrupt memory data, leading to program crashes or hangs. Furthermore, malicious users often launch security attacks by exploiting these bugs. According to the US-CERT Vulnerability Notes Database [US-CERT], memory management bugs dominate recent security vulnerability reports.

Furthermore, it is a challenging task for developers to diagnose these bugs and release timely fixes because of ever-increasing software complexity and lack of on-site failure information [Tucek 2007a]. Previous studies [Arbaugh 2000, Symantec 2006] have shown that it takes several weeks on average to diagnose bugs and generate patches. During this long time window, users have to either continue running the software with bugs and tolerate problems such as intermittent crashes and potential attacks, or stop running the software and experience costly system downtime. Neither option is desirable.

Therefore, it is critical to be able to quickly recover programs from software failures caused by memory bugs, protect programs from future failures due to the same bugs, and provide useful on-site failure information for developers to fix the bugs.

While several recent proposals help programs survive failures caused by memory bugs, they suffer one or more of the following limitations: unsafe speculation on programmers' intentions [Rinard 2004, Sidiroglou 2005], inability to prevent subsequent failures due to the same bugs [Berger 2006, Qin 2005b], little diagnostic information to developers [Lvin 2008], and large time and space overheads [Berger 2006, Novark 2007]. For example, failure oblivious comput-

This is a minor revision of the work published in EuroSys 2009, http://doi.acm.org/10.1145/1519065.1519083

ing [Rinard 2004] discards out-of-bound writes and manufactures arbitrary values for out-of-bound reads. While this approach may survive failures for certain types of applications, its speculation on programmers' intention could easily lead to program misbehavior. Based on a novel randomized memory allocator, DieHard [Berger 2006] can probabilistically prevent memory errors and Exterminator [Novark 2007] can locate and fix memory errors in a high precision. However, large time and space overheads prevent them from being adopted for production runs.

Our previous work Rx [Qin 2005b] can quickly recover programs from failures by re-executing a program from previous checkpoints and applying *environmental changes* to program execution. An example of such an environmental change is adding padding to *all* memory objects during recovery to avoid buffer overflow bugs. While effectively and safely avoiding the memory bugs, Rx has to disable the environmental changes after surviving a failure due to their potentially large overhead. Therefore, Rx cannot prevent future failures caused by the same bug. Furthermore, the on-site failure recovery information provided by Rx (i.e., whether and what environmental changes can work) is quite limited and may mislead developers because failure-surviving environmental changes may not directly relate to the bugs that have occurred.

1.2 Our contributions

This paper makes three contributions in order to address the limitations of previous work:

Contribution 1: A low-overhead method for surviving and preventing memory management bugs. We propose a system, called First-Aid, that can quickly recover programs from failures caused by memory bugs, prevent future failures due to the same bugs, and provide useful on-site diagnostic information to developers. The main idea is to diagnose the bug type and identify the memory objects that trigger the bug when a failure occurs. Based on results of the diagnosis, First-Aid generates and applies *runtime patches* (environmental changes for bug-triggering memory objects) to a small set of memory objects that can potentially trigger the bug, during both program recovery and future program execution. As a result, First-Aid not only helps programs survive the current failure but also prevents future failures caused by the same bug. Furthermore, First-Aid validates the runtime patches to make sure their effects are consistent and generates a bug report to help developers fix the bug.

Contribution 2: Environmental change based failure diagnosis. We propose an online failure diagnosis method that leverages efficient checkpointing and re-execution mechanisms as well as execution environment changes. Specifically, when a bug causes a failure or an error at runtime, First-Aid rolls back the program to previous checkpoints and re-executes the program. During each re-execution, First-Aid dynamically applies two types of environmental changes: *exposing changes*, which force a certain type of bug to manifest itself, and *preventive changes*, which prevent a certain type of bug from manifesting itself. Based on bug manifestation after re-execution, First-Aid can conclude whether a certain type of bug is occurring. For example, to determine whether a buffer overflow bug is occurring, First-Aid applies the exposing change for buffer overflow, i.e., padding each memory object and filling the padding with canary values, and the preventive changes for all other memory bug types. The term *canary* refers to certain memory content patterns that are unlikely to appear during normal program execution. If the canary values in some padding are corrupted, First-Aid knows that the bug is likely to be a buffer overflow. Exposing changes also help First-Aid identify the bug-triggering memory objects. In the same example, based on the corrupted padding, First-Aid can identify the memory objects that are overflowed.

Contribution 3: Evaluation with real-world applications. We have implemented First-Aid on Linux and evaluated its functionality and performance with seven applications: three server applications (Apache, Squid, and CVS) and four desktop applications (Pine, Mutt, M4, and BC). These applications contain various types of memory bugs, including buffer overflow, dangling pointer read/write, double free, and uninitialized read. Additionally, we evaluate First-Aid's performance with the SPEC INT2000 benchmark [SPEC] and four allocation intensive benchmarks [Berger 2000]. Our experimental results show that, compared to previous approaches, First-Aid has the following advantages:

- **Fast diagnosis and failure recovery.** First-Aid can quickly identify the bugs and recover programs from failures using its diagnosis algorithm. Our evaluation with the seven tested applications shows that the time for failure diagnosis and recovery ranges from 0.084 to 3.978 seconds with an average of 0.887 seconds.
- **Prevention of bug reoccurrence.** First-Aid applies the runtime patches after a program failure and thereby prevents future failures due to the reoccurrence of the same bug. Furthermore, First-Aid stores the generated patches persistently to prevent the bug from occurring on subsequent runs or on other processes running the same program. This improves the overall reliability of the system.
- **Low normal-run overhead.** First-Aid incurs low runtime overhead during normal program execution, i.e., without bugs being triggered. Our evaluation with applications and benchmarks shows that the runtime overhead ranges from 0.4 to 11.6% with an average of 3.7%.
- **Informative bug reports.** First-Aid provides programmers with accurate information about the bug: the bug type, the memory objects triggering the bug, their allocation or deallocation sites, and the relevant illegal memory accesses. Such diagnostic information helps programmers understand both the root cause and the manifestation of the bug.

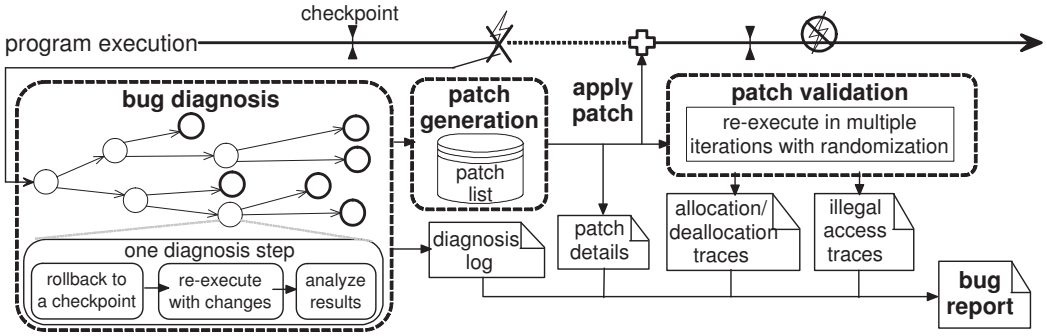


Figure 1. Working scenario of First-Aid

Bug type	Common reason(s) for the bug	Preventive change/ Runtime patch	Exposing change (Bug manifestation)	Patch application point
buffer overflow	1. length underestimation 2. offset miscalculation	add padding to objects	pad objects with canary values (canary corruption)	allocation
dangling pointer read	1. premature buffer free 2. forget to set NULL	delay free	fill objects with canary values (failure)	deallocation
dangling pointer write			fill objects with canary values (canary corruption)	
double free			check parameters (freed twice)	
uninitialized read	1. assume zeros in buffers	fill objects with zeros	fill objects with canary values (failure)	allocation

Table 1. Memory bug types and corresponding environmental changes

The rest of the paper is organized as follows. Section 2 gives an overview of First-Aid and Section 3 describes the design and components of First-Aid. Sections 4 and 5 introduce the diagnosis algorithm and validation algorithm, respectively, followed by discussion of issues and limitations in Section 6. The evaluation methodology and results are presented in Section 7. Section 8 discusses related work, and Section 9 concludes the paper.

2. Overview of First-Aid

Figure 1 shows the working scenario of First-Aid. During normal execution, First-Aid periodically takes checkpoints of a running program. Upon a failure, First-Aid diagnoses the bug and generates the corresponding runtime patches. Then it applies the patches to allow the program to recover from the failure and to prevent future failures due to the same bug. After recovery, First-Aid validates the patches and generates a detailed bug report.

Bug diagnosis. First-Aid diagnoses the bug by re-executing the failed process from previous checkpoints for multiple iterations. In each iteration, First-Aid rolls back the program to a previous checkpoint, tentatively applies one or more environmental changes to all or a subset of memory objects, and re-executes the program. Based on the results of execution, it narrows down the possible causes of the failure. If First-Aid can identify the bug type and the bug-triggering memory objects, the diagnostic process stops. Otherwise, First-Aid continues the above process until it identifies the bug or times out.

To accurately diagnose the bug, First-Aid uses a combination of two types of environmental changes: *exposing changes* for forcing a certain type of bug to manifest itself and *preventive changes* for preventing a certain type of bug from manifestation. More specifically, to check the occurrence of a bug type *b*, First-Aid applies the exposing change for *b* and the preventive changes for all other bug types during re-execution. In this way, First-Aid ensures that only the bug type *b* can manifest itself. If the manifestation of bugs with type *b* is observed, First-Aid concludes that a bug of type *b* has occurred before the failure. Otherwise, First-Aid rules out the bug type *b*. Furthermore, based on the bug manifestation, First-Aid can identify the memory objects that potentially trigger the bug.

Table 1 describes both types of environmental changes for each bug type, including buffer overflow, double free, dangling pointer read/write, and uninitialized read. For example, adding padding to both ends of each newly-allocated memory object can prevent buffer overflow bugs, while adding canary-filled padding can manifest buffer overflow bugs as canary corruption. Delaying the recycling of freed memory objects can prevent dangling pointer read bugs from accessing meaningless data as well as prevent dangling pointer write bugs from corrupting useful data. On the contrary, filling delay-freed memory objects with canary values can manifest dangling pointer read bugs as failures and dangling pointer write bugs as canary corruption. Furthermore, zero-

filling new objects can prevent uninitialized read bugs, while canary-filling new objects is likely to manifest uninitialized read bugs as failures.

At the end of the diagnostic process, there are three possible results. First, the failure can be caused by non-deterministic bugs (as indicated by a successful re-execution with only timing-based changes and no memory management changes). In this case, First-Aid lets the program continue the normal execution. Second, the failure can also be caused by deterministic memory management bugs. In this case, First-Aid passes the diagnostic results to the next step for patch generation. The third case is deterministic bugs that First-Aid cannot handle. Examples of such bugs include non-memory bugs and some types of memory bugs (details in Section 6). In this case, First-Aid times out and resorts to other recovery schemes.

Patch generation and application. Based on the bug diagnostic information, First-Aid generates runtime patches for recovering the program from a failure and preventing future failures caused by the same bug. A runtime patch is a pair of a preventive change corresponding to the identified bug type and a patch application point. As shown in Table 1, the preventive change for buffer overflows is to add padding to both ends of bug-triggering objects when these objects are allocated; the preventive change for dangling pointers and double frees is to delay recycling of deallocated bug-triggering objects for a long time until the memory occupied by these objects reaches a customizable threshold; the preventive change for uninitialized read is to fill the contents of bug-triggering objects with all 0's.

A patch application point is the allocation or deallocation *call-site* of the bug-triggering memory objects. In this paper, the *call-site* is defined as the return addresses of the most recent three functions on the stack. It can serve as the "signature" of the bug-triggering memory objects because memory objects with the same call-site of allocation or deallocation often have similar characteristics such as overflow [Novark 2007, Qin 2005a].

First-Aid stores the generated patches and applies them to the running process. More specifically, a patch takes effect when a later memory allocation or deallocation call-site matches the patch application point. By applying the identified preventive change at these points, First-Aid protects programs from future failures caused by the same bug. Furthermore, since the patches are specific to the program executable (not only the running process), First-Aid applies them to the subsequent runs of the same program and other processes running the same executable. This effectively protects the programs from future failures caused by the same bug and improves the overall system reliability.

Some of the runtime patches generated by First-Aid increase the application's memory usage. Filling new objects with 0's incurs no space overhead. Space overhead for the added padding depends on the sizes of padding and the number of padded memory objects. Delaying frees increases memory usage by accumulating delay-freed objects. For our evaluated applications, First-Aid's runtime patches incur a small memory space overhead (Section 7.6.1). This is because the patches are only applied to a small number of memory objects whose allocation or deallocation sites match call-sites specified in the patches.

Although the situation never arose during our experiments, the extra memory usage incurred by First-Aid's runtime patches may theoretically exhaust the memory space of a process. To address this issue, First-Aid can disable runtime patching and even start deallocating the oldest delay-freed objects when the memory usage reaches a user-defined threshold. Although deallocating very old delay-freed objects is usually safe for dangling pointer and double free bugs, it can potentially undermine patch effectiveness – the program may fail again. First-Aid allows users to decide how much extra memory space they are willing to pay for better system reliability.

Patch validation and bug reporting. After a runtime patch is generated and applied for a quick recovery, First-Aid performs a further step to validate the patch and collect detailed information for the bug report. This step can be done in parallel on a different processor core based on a snapshot of the program so that it does not delay the failure recovery.

To validate that the applied patch is consistently effective, First-Aid re-executes the buggy program region in multiple iterations with randomized memory allocation, and collects detailed traces on memory management operations, patch triggering, and *illegal memory accesses*, e.g., accesses through a dangling pointer, reads before initialization, etc. Then it checks the traces to validate that the patch has consistent effects on the program execution, e.g., neutralizing the same number of illegal accesses. If the validation fails, First-Aid removes the corresponding patch and logs the event of validation failure.

Furthermore, First-Aid assists developers in diagnosing and fixing the bugs by providing detailed on-site bug information. For example, the call-sites of bug-triggering memory objects in the diagnosis log and patch information can help developers locate the memory management code related to the bug. Additionally, the execution traces on memory management operations and illegal accesses can help developers understand the process of bug manifestation.

3. First-Aid System Design

Figure 2 shows the software architecture of First-Aid. It consists of six main user-level and kernel-level components: (1) a lightweight memory allocator extension for assisting bug diagnosis, patch validation, and patch application, (2) error monitor(s) for detecting errors or failures in applications, (3) a checkpoint/rollback component for taking snapshots of running programs and performing rollbacks for diagnosis and recovery, (4) a diagnostic engine for diagnosing the

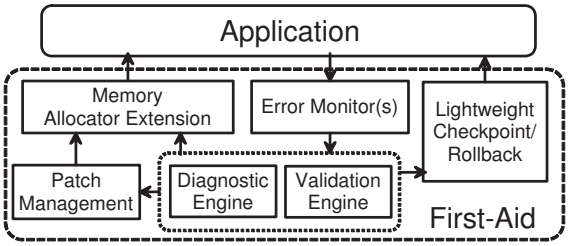


Figure 2. Software architecture of First-Aid

bugs and generating runtime patches (details in Section 4), (5) a patch management module for managing the generated runtime patches and controlling patch application, and (6) a validation engine for validating the consistent effects of the patches and collecting on-site diagnostic information (details in Section 5).

Memory allocator extension. The memory allocator extension collects memory object information and applies preventive and/or exposing changes for diagnosing bugs and surviving failures. It operates in one of three modes: *normal* mode, *diagnostic* mode, or *validation* mode.

In normal mode, the memory allocator extension checks whether the current call-site of a memory allocation or deallocation request matches any patch application point in the available patches. If so, it applies the preventive change in the patch to the memory object. Note that the memory allocator extension relies on the underlying memory allocator for fulfilling memory management requests.

In diagnostic mode, the memory allocator extension performs three functions during re-execution. First, it applies preventive and/or exposing changes, as instructed by the diagnostic engine, to all or a subset of memory objects when they are allocated or deallocated. Second, it collects multi-level call-site information for each memory object allocation and deallocation. Such information is used for bug diagnosis and future patch application. Third, for each deallocation request, it checks whether the memory object has been freed previously, to detect double free bugs.

In validation mode, as controlled by the validation engine, the memory allocator extension introduces randomization in the memory allocation algorithm and keeps traces of memory allocation and deallocation as well as the patch triggering information.

Error monitor(s). The error monitors detect errors or failures at runtime and notify the diagnostic engine upon detection. The cheapest way to detect an error or failure is to catch assertion failures as well as exceptions (e.g., access violation) raised from the kernel. Additionally, one can deploy more sophisticated error detectors such as AccMon [Zhou 2004] if they incur low overhead. Our current implementation is based on assertion failures and exceptions.

Lightweight checkpoint/rollback. First-Aid leverages the lightweight checkpointing and re-execution runtime system provided in Rx [Qin 2005b]. More specifically, it takes in-memory checkpoints using a `fork`-like operation and rolls back the program by reinstating the saved task state. For handling files, it applies ideas similar to previous work [Lowell 1998, Srinivasan 2004] by keeping a copy of each accessed file and file pointers at the beginning of each checkpoint and reinstating it for rollback. Copy-on-write (COW) is also applied here to reduce the overhead. Additionally, First-Aid leverages a network proxy to record network messages during normal execution and replay them during re-execution. More details can be found in the work on Rx [Qin 2005b] and Flashback [Srinivasan 2004].

Instead of using fixed checkpointing intervals as in Rx, First-Aid dynamically adjusts the checkpointing intervals for balancing the low normal execution overhead and quick recovery time. It does so by monitoring the copy-on-write (COW) page rate, which directly affects the runtime overhead. If the runtime overhead is higher than the threshold $T_{overhead}$ specified by the user, i.e., the COW page rate is too high, First-Aid gradually increases the checkpointing interval to control the overhead. On the other hand, the recovery time becomes longer when the checkpoint interval is larger. Thus, once the checkpoint interval reaches the user-specified maximal interval $T_{checkpoint}$, First-Aid stops increasing it.

Patch management. This component manages the patches and makes them available to all the processes that are running the same program. Once the diagnostic engine generates a patch, the patch management component stores it in a central patch pool based on the call-site information. First-Aid maintains a patch pool for each program so that the patches do not mix for different programs. During normal execution, the memory allocator extension queries the patch pool at each allocation or deallocation request.

4. Bug Diagnosis

The diagnostic engine uses two phases to diagnose the bug. The first phase (Section 4.1) searches for the best checkpoint from which the patch should be applied for surviving the bug. The second phase (Section 4.2) performs in-depth diagnosis to identify the bug type and the patch application points, i.e., allocation or deallocation call-sites of the bug-triggering memory objects. In Section 4.3, we compare the First-Aid bug diagnosis with that of Rx.

4.1 Phase 1: Identify the checkpoint for patching

In order to be both effective and efficient, the patch should take effect from the latest checkpoint before the bug is triggered. To identify that checkpoint, First-Aid rolls back the program to previous checkpoints in reverse chronological order and re-executes the program. It first re-executes the program without any memory management changes. If the program succeeds, then the failure is likely caused by a non-deterministic bug and First-Aid only logs this event and lets the program continue execution. If the program fails, First-

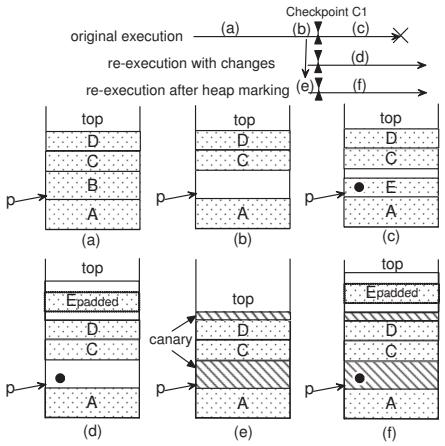


Figure 3. A potential misidentification of the checkpoint for patching and the heap marking technique

Aid again re-executes the program from the checkpoint with *all* the preventive changes on *all* subsequently allocated or deallocated memory objects. If the program succeeds this time, i.e., some preventive change is effective, First-Aid stops searching and reports this checkpoint as the latest one before the bug-triggering point. Otherwise, First-Aid continues searching with the checkpoint right before this one. After trying a certain number of previous checkpoints, First-Aid times out and logs the bug as non-patchable. Note that the criterion of failed or successful re-execution in First-Aid is based on whether the program execution can pass the original failure region. Its end point is conservatively defined as a certain number (3 in our experiments) of checkpoint intervals after the failure point.

One challenge in this scheme is the possible misidentification of the latest checkpoint before the bug-triggering point. In some cases, when being applied to a checkpoint that is *after* the bug-triggering point, the preventive changes can appear to be effective by temporarily avoiding the failure. This is because the manifestations of some memory bugs may rely on the heap layout, which could be disturbed when the preventive changes are applied later.

Figure 3 shows one such example. During the original execution, from (a) to (c), the dangling pointer p appears at (b) *the bug-triggering point*, when the object B is prematurely freed. After (b), a checkpoint $C1$ is taken, and then the freed space of B is re-allocated to another object E. At the end, a failure occurs because a write via de-referencing p corrupts the data in the object E, illustrated as the black dot in Figure 3(c). When the program re-executes from the checkpoint $C1$, the preventive changes avoid the failure. This is because the preventive change for buffer overflow adds padding to E, denoted as E_{padded}, making it larger than the original B. As a result, shown in Figure 3(d), the freed space of B is not reused by E_{padded}, which avoids the memory corruption caused by the dangling pointer write.

To address this issue, we devise a technique called *heap marking* to verify that the bug indeed occurs *after* the checkpoint. The key idea is to expose the bugs that occur before the checkpoint. To this end, First-Aid marks the old heap region before re-executing the program from the checkpoint. More specifically, as shown in Figure 3(e), First-Aid marks all the free chunks in the heap by filling their contents with canary values. Additionally, it adds padding filled with canary values after the last memory object in the heap. This heap marking technique exposes previously triggered dangling pointer write or buffer overflow bugs as canary corruption even though the failure can be accidentally avoided due to heap layout disturbance. For dangling pointer read bugs, the heap marking technique makes the failure still occur during re-execution due to the canary.

4.2 Phase 2: Identify the bug type and patch application points

After identifying the latest checkpoint before the bug triggering point, First-Aid starts phase 2 for more in-depth analysis. It first identifies the types of the bugs and then identifies the allocation or deallocation call-sites of bug-triggering memory objects. Note that, First-Aid takes into consideration the case where multiple types of bugs are triggered and the program will not survive unless all of them are avoided. Therefore, the algorithm carefully separates each bug type.

The basic procedure is to diagnose each bug type one by one using a combination of preventive changes and exposing changes. We define two sets of bug types, the *undecided set* S_u and the *identified set* S_i. Initially, S_u contains all the bug types and S_i is empty. For each bug type b from the undecided set S_u, First-Aid applies the following changes to all subsequently allocated or deallocated memory objects: the exposing change for the bug type b, and the preventive changes for all other bug types in the set $S_u \cup S_i - \{b\}$. This way, the bugs with the type b will manifest themselves during re-execution and the potential interferences from other types of bugs will be prevented. If the bugs of the type b manifest themselves during re-execution, First-Aid will move the bug type b from the undecided set S_u to the identified set S_i; otherwise, First-Aid will simply remove it from the undecided set S_u. At the end of this procedure, the identified set S_i will contain all the types of the bugs that have occurred.

After each new bug type is identified, First-Aid checks whether the current identified set S_i covers all the types of bugs that have occurred. To do so, First-Aid performs one iteration of re-execution, applying preventive changes for bug types in the identified set along with exposing changes for the undecided set. If no bugs manifest themselves, First-Aid stops searching for more bug types; otherwise, it continues.

After identifying all the bug types, the next step is to identify the call-sites of the bug-triggering memory objects. For buffer overflow and dangling pointer write, First-Aid can directly identify the bug-triggering memory objects by look-

ing for canary corruption in padding and delay-freed memory objects, respectively. For double frees, it identifies the bug-triggering memory objects by checking the parameters passed to memory deallocation operations.

For uninitialized read and dangling pointer read, it is more challenging to identify the call-sites through the bug-triggering memory objects themselves, because these bugs only cause incorrect content reads. To address this problem, First-Aid uses a binary search algorithm to identify such call-sites. Specifically, starting with a search range covering all N call-sites after the checkpoint, in each iteration of re-execution, First-Aid applies the exposing change to half of the call-sites in the search range and the preventive change to the rest of call-sites. Depending on whether the bug is exposed, i.e., whether the program fails, it narrows down the search range by half and starts another iteration until the search range contains only a single bug-triggering call-site. The number of iterations for this algorithm is $O(log\ N)$.

By also applying preventive changes in each iteration, First-Aid can prevent interference from undiagnosed bug-triggering call-sites that are outside of the current search range. This is critical for handling the case where multiple call-sites need to be patched at the same time to prevent a failure. In this case, one bug-triggering call-site is guaranteed to be identified by each round of the above binary search, and First-Aid needs to conduct multiple rounds of search and remove the identified call-site from the whole search range after each round. If there are M bug-triggering call-sites, the search algorithm uses $O(M * log\ N)$ re-executions in total.

4.3 Comparison of First-Aid and Rx bug diagnosis

Our previous work, Rx [Qin 2005b], can also provide quick recovery for failures caused by memory bugs. However, it (intentionally) does not perform in-depth diagnosis since it aims for fast recovery. First-Aid's goals, on the other hand, are not only to quickly recover programs from failures, but also to prevent reoccurrence of the same bug and provide on-site diagnostic information to developers. Therefore, First-Aid performs accurate diagnosis on memory bugs in the following two respects.

Correctness: First-Aid will not misdiagnose one type of memory bugs as another. It determines one bug type by observing both failure symptoms and possible bug manifestations such as memory content corruption imposed by the exposing change. In contrast, Rx makes its decision based on whether the program survives or fails after applying preventive changes only. It could thus misdiagnose one type of bugs for another. For example, Rx may end up using padding, which is for avoiding a buffer overflow bug, to cure a dangling pointer write bug when the memory write through the dangling pointer happens to corrupt some padding instead of useful data. This cannot happen in First-Aid because when diagnosing buffer overflow, the "delay free" change is also applied to prevent dangling pointer writes from corrupting any other places, including the padding.

Exactness: First-Aid identifies a small set of memory objects that potentially trigger the bug, while Rx applies environmental changes to all memory objects during re-execution. For example, Rx stops diagnosis if padding all the new objects can avoid the bug during re-execution. In contrast, First-Aid pinpoints the exact objects where the buffer overflows occur by checking canary values in the padding of all the objects.

5. Patch Validation and Bug Reporting

Although First-Aid's diagnosis algorithm will not misdiagnose one type of memory bugs as another, it may diagnose other types of bugs (e.g. semantic bugs) as memory bugs if the bug manifestation depends on memory layout. For example, if a memory write due to a semantic bug happens at the address right after a newly allocated object, it may be diagnosed as a buffer overflow. Even though the chance of such misdiagnosis is small, it undermines the safety and reliability of the program execution in the long run and can mislead developers when they fix the bug in the source code.

To rule out the possibility of such misdiagnosis, the random side-effects of a patch must be distinguished from the desired effects. First-Aid does this by checking that the effect of a runtime patch is consistent under memory layout randomization. During validation, First-Aid re-executes the buggy region of the program for three more iterations with a randomized allocation algorithm. In each iteration, the memory allocator's activities are logged by First-Aid's allocator extension, and illegal memory accesses are traced using the dynamic instrumentation tool Pin [Luk 2005]. Specifically, for each memory allocation or deallocation, First-Aid logs the object address and whether a patch is triggered at this operation. If a patch is triggered, First-Aid also traces the illegal accesses corresponding to the patch: the memory writes to the padding, the memory reads and writes to delay-freed objects, and the reads before initialization. With these traces, First-Aid checks whether the effects of the patch are consistent among multiple re-executions, based on the following criteria: a) the patch is triggered for the same number of times; b) there are the same number of total illegal accesses prevented by the patch; and c) each illegal access is made by the same instruction at the same offset in the corresponding memory object (the memory object's address is randomized though). If the consistency check fails, First-Aid removes the runtime patch and logs the validation failure.

The traces collected in the above validation process are organized into a bug report for developers. Specifically, besides the usual bug report (e.g., core dump, event log), First-Aid provides four pieces of information: a) a diagnosis log, which helps developers understand the diagnostic process; b) the runtime patch information (i.e., the bug type and call-sites of the relevant allocation or deallocation operations),

which points developers to the critical source code section related to the bug; c) allocation and deallocation traces in the bug region, which show clearly when the runtime patch takes effect and what memory objects are affected; and d) illegal memory accesses in the bug region, which show the instructions that have made illegal memory accesses. The above information about both the root causes of the bug and the bug manifestation process can help developers fix the bug.

6. Discussion

Assumptions on memory bugs: First-Aid's diagnosis algorithm is based on several assumptions about the common characteristics of memory bugs. These assumptions include: a buffer overflow bug must corrupt data within a neighboring region of the memory object; canary values must not be coincidentally used as normal values in buggy memory objects; programmers intend to initialize the newly allocated buffers with zeros in the case of an uninitialized read bug. These assumptions cover most common cases in real memory bugs and are also used in previous work [Novark 2007, Qin 2005b], although exceptions can happen theoretically and result in failed attempts to patch.

Bug coverage: Although First-Aid covers common memory bugs, there are some types of bugs that First-Aid cannot fix. Specifically, First-Aid cannot deal with memory leak bugs, whose negative effects are cumulative and cannot be reverted by simply rolling back to a recent checkpoint. One way to survive or prevent memory leak bugs is to deploy memory leak detectors and swap the leaked objects onto disk [Bond 2008, Tang 2008]. Additionally, First-Aid cannot handle memory bugs that slip through the deployed error monitors. One way to address this issue is to deploy more rigorous dynamic integrity and correctness checkers as First-Aid's error monitors. This is currently an active research area. Moreover, First-Aid cannot deal with latent bugs – bugs whose root causes are far away from the error symptoms. Fortunately, this is a rare case, as a previous study [Gu 2003] shows that most bugs tend to cause quick crashes.

Customized memory allocator in applications: To avoid certain memory bugs or improve performance, some applications use customized memory allocation wrappers or even their own memory allocators. Memory allocation wrappers have little impact on First-Aid because First-Aid's diagnosis and patching are based on multi-level call-sites. If an application-specific allocator is used, First-Aid's memory allocator extension should be ported to the application-specific allocator. We do not see any particular challenge for porting.

7. Evaluation and Experimental Results

7.1 Experimental setup

Our experimental platform consists of two machines with Intel Xeon 3.00 GHz processors, 2MB L2 cache, 2GB memory, and a 100Mbps Ethernet connection between them. The operating system kernel is the Linux 2.4.22 modified with Flashback [Srinivasan 2004] checkpointing support. We ran servers on one machine and clients on the other. We implemented the memory allocator extension by modifying the Lea allocator [Lea 1996], the default memory allocator used in the GNU C library.

Application	Ver.	Bug	LOC	App. Desc.
Apache	2.0.51	dangling pointer read	263K	web server
Apache-uir		uninitialized read		
Apache-dpw		dangling pointer write		
Squid	2.3	buffer overflow	93K	proxy cache
CVS	1.11.4	double free	114K	version control
Pine	4.44	buffer overflow	330K	email client
Mutt	1.3.99i	buffer overflow	86K	email client
M4	1.4.4	dangling pointer read	17K	macro processor
BC	1.06	buffer overflow	14K	calculator

Table 2. Applications and bugs used in evaluation.

We evaluated First-Aid with a range of applications including three server applications (Apache, Squid, and CVS) and four desktop applications (Pine, Mutt, M4, and BC), as shown in Table 2. The applications contain various types of memory bugs, including buffer overflow, dangling pointer read/write, double free, and uninitialized read. Seven of these bugs were introduced by the original developers. We have not been able to locate applications that contain dangling pointer write or uninitialized read bugs. To evaluate First-Aid's functionality of handling these two types of bugs, we injected them into Apache httpd server separately – Apache-uir contains an uninitialized read bug and Apache-dpw contains a dangling pointer write bug.

7.2 Overall effectiveness

We executed these seven applications with First-Aid. The checkpoint intervals are 200 milliseconds. To simulate bug occurrences in real scenarios, we mixed the bug-triggering inputs and normal inputs. For each bug case, we measured the failure recovery time, the number of rollbacks for diagnosis, the number of the patched call-sites, and the validation time. We also recorded the runtime patches generated by First-Aid and whether they can avoid future errors caused by the same bug. Table 3 shows the results.

First-Aid is effective in diagnosing memory bugs. As shown in Table 3, for all tested cases, First-Aid correctly identifies the bug type and the call-sites of bug-triggering memory objects. The diagnosis is accurate because First-Aid leverages both preventive and exposing changes for separating the interferences among different bugs.

First-Aid provides quick failure recovery and thereby hides program failures from users. As shown in Table 3, the failure recovery time ranges from 0.084 to 3.978 seconds with an average 0.887 seconds. This is because of First-Aid's lightweight diagnosis algorithm and checkpointing-and-re-execution mechanism. For example, First-Aid quickly pinpoints the bug in seven cases after 6-9 iterations of program re-execution, resulting in a failure recovery time less than 1

Application	Diagnosed bugs	Runtime patch (No. of call-sites applied)	Recovery time (s)	Avoid future errors?	No. of rollbacks for diagnosis	Validation time (s)
Apache	dangling pointer read	delay free(7)	3.978	Yes	28	9.620
Squid	buffer overflow	add padding(1)	0.386	Yes	7	14.198
CVS	double free	delay free(1)	0.121	Yes	6	3.887
Pine	buffer overflow	add padding(1)	0.722	Yes	7	18.276
Mutt	buffer overflow	add padding(1)	0.617	Yes	7	10.610
M4	dangling pointer reads	delay free(2)	1.396	Yes	18	3.407
BC	two buffer overflows	add padding(3)	0.573	Yes	6	2.625
Apache-uir	uninitialized read	fill with zero(1)	0.102	Yes	9	5.750
Apache-dpw	dangling pointer write	delay free(1)	0.084	Yes	7	5.718

Table 3. Overall results for First-Aid in surviving and preventing memory bugs. The recovery time is from when the failure is first caught to when the program changes back to normal mode with the applied runtime patches. The validation time is the extra time taken when enabling a three-iteration validation. `Apache-uir` and `Apache-dpw` correspond to the cases with injected uninitialized read and dangling pointer write, respectively.

second. The recovery time for the dangling pointer read case in Apache is relatively long because its bug-triggering point is a little farther (3 checkpoints) from the failure point.

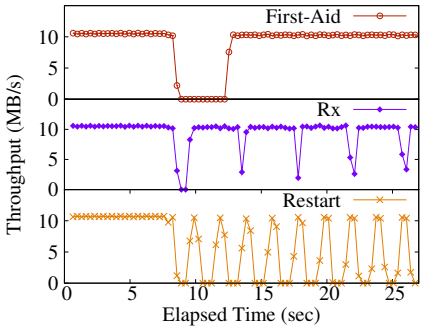
The number of rollbacks First-Aid performs for diagnosis depends on the bug type. For buffer overflow, dangling pointer write, and double-free bugs, the number of rollbacks is small (6-7 times in our evaluation). This is because the bug-triggering objects can be directly identified by checking the integrity of the canary or parameters of deallocation requests. For uninitialized read and dangling pointer read bugs, more rollbacks may be needed because the patch application points need to be identified via binary search.

Table 3 shows that First-Aid is effective in avoiding future errors caused by the same bug. In the experiments, we repeatedly sent the bug-triggering inputs to the applications. First-Aid successfully prevents future memory errors due to the same bug after applying the runtime patches. This is because First-Aid's patch is accurate (Section 7.3) and thereby can be enabled for preventing future errors during the entire program execution.

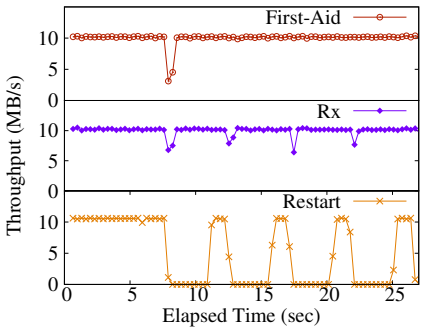
As shown in Table 3, First-Aid successfully validates the generated patches within a small amount of time, i.e., 3-18 seconds, for the tested cases. The validation is done in parallel with a snapshot of the application and does not delay the recovery. During the validation process, the runtime patches show consistent effects, i.e., patches are applied to the same number of memory objects and each illegal access matches across different runs. The validation time is small because First-Aid re-executes the program from the checkpoint identified by the diagnostic engine instead of from the beginning.

7.3 Future error prevention

We evaluated First-Aid's capability of future error prevention and compared it with two alternatives: Rx [Qin 2005b] and the restart method [Gray 1986, Sullivan 1991], using two representative server programs, Apache (the dangling pointer read bug) and Squid (the buffer overflow bug). In the experiments, after a certain period of normal execution, we periodically triggered the real bugs by sending bug-triggering requests mixed with normal inputs.



(a) Throughput for Apache



(b) Throughput for Squid

Figure 4. Comparison among First-Aid, Rx, and restart

Figure 4 shows that First-Aid effectively prevents future errors caused by the same bug while Rx and the restart approach cannot. In the case of Apache (Figure 4 (a)), First-Aid diagnoses the bug and recovers the programs from the first failure in around 4 seconds and then maintains stable performance when facing the same bug-triggering inputs repeatedly. This is because the patch generated by First-Aid is correct and accurate so that it can avoid the same memory bug during future program execution. While Rx recovers the program from failures more quickly than First-Aid

does, it suffers the same bug repetitively during subsequent program execution. This is because Rx disables the environmental changes after passing the buggy program region. For the restart approach, it suffers the same bug repetitively since the bug is deterministically triggered during subsequent program execution. Figure 4 (b) shows similar results for Squid. One difference is that First-Aid recovers the program from the first failure faster for Squid than for Apache because of Squid's shorter error propagation distance.

Table 4 further illustrates the accuracy of First-Aid's patches as well as the reason why Rx disables the environmental changes after surviving failures. For the seven tested real bugs, we compare patch application in First-Aid and environmental change application in Rx for the buggy region in terms of the number of call-sites and objects being applied with the changes. As shown in Table 4, the patch generated by First-Aid is of much lighter weight. For example, First-Aid only affects 1 to 7 call-sites and 1 to 315 objects, while Rx affects 8 to 380 call-sites and 183 to 5004 objects. Furthermore, after passing the buggy region, First-Aid's patches are less likely to be triggered by normal user inputs. Therefore, First-Aid can enable the lightweight patches during the entire program execution for preventing future errors due to the same bug.

Name	Call-sites			Objects		
	First -Aid	Rx	Ratio	First -Aid	Rx	Ratio
Apache	7	32	21.88%	315	2567	12.23%
Squid	1	61	1.64%	1	3626	0.03%
CVS	1	44	2.27%	17	306	5.56%
Pine	1	380	0.26%	11	2881	0.38%
Mutt	1	216	0.46%	2	5004	0.04%
M4	2	8	25.00%	3	183	1.64%
BC	3	34	8.82%	5	732	0.68%

Table 4. The call-sites and memory objects affected by the runtime patch in the buggy region

7.4 Bug report

Our manual inspection of the bug reports shows that the on-site diagnostic information is helpful in fixing the bug. Figure 5 shows the report generated by First-Aid for the Apache dangling pointer read bug. It consists of five parts: a failure core-dump, a diagnosis log, runtime patch information, memory allocation and deallocation traces, and an illegal access trace. Note that some details of the report are omitted in Figure 5.

The patch information (item 3 in Figure 5) states that the bug is a dangling pointer read and can be avoided by delaying frees in the functions `util_ald_cache_purge` and `util_ldap_search_ node_free`, which are the callers of a wrapper (`util_ald_free`) for `free` in Apache. Additionally, the multi-level call-sites show that all the delayed frees are issued indirectly through `util_ald_ cache_purge`. This information indicates that the bug is

```
Bug report:
1. Failure coredump: failure.core
2. Diagnosis summary: recovery: 3.978(s); validation: 9.620 (s),
   Diagnosis log: diag.log
3. Patch applied: delay free x 7 for dangling pointer read
   Patch 1: delay free on callsite:  0x4022f971@util_ald_free
            (triggered 44 times)   0x806437b@util_ald_cache_purge
                                   0x80646dc@util_ald_cache_insert
   Patch 2: delay free on callsite:  0x4022f971@util_ald_free
            (triggered 44 times)   0x8063eac@util_ldap_search_node_free
                                   0x8064372@util_ald_cache_purge
   Patch 3: ...
4. Memory allocations/deallocations in buggy region:
   Without patch: mm_trace_orig.log,  with patch: mm_trace_patched.log, diff:
      free(0x81865c0)              | free(0x81865c0)  (delayed, patch 2)
      free(0x8186360)              | free(0x8186360)  (delayed, patch 3)
      ...
      malloc(1000): 0x8186250      |  malloc(1000): 0x818e4f8
      ...
5. Illegal access trace in buggy region:
   Summary: patch 1: 68 accesses (68 read, 0 write):
                     from 2 instructions in util_ald_cache_fetch
                     from 4 instructions in util_ald_cache_purge
            patch 2: 90 accesses (90 read, 0 write):
                     from 6 instructions in  util_ldap_search_node_free
            patch 3: ...
   Detailed access log: illegal_access.log
```

Figure 5. The bug report generated by First-Aid for the Apache dangling pointer read bug

related to the LDAP (Lightweight Directory Access Protocol) cache. By comparing the allocation/deallocation traces (item 4 in Figure 5) with and without the runtime patch applied, we notice that without the patch, the freed memory is reallocated later (as indicated by the italic font at item 4 in Figure 5). Furthermore, the illegal access trace (item 5 in Figure 5) in the buggy region clearly shows that in a few LDAP cache related functions, the application accesses the memory objects that have been freed by the application (but not deallocated due to First-Aid's runtime patches). All of these suggest that the bug is in `util_ald_cache_purge`: dangling pointers are created in the cache cleanup operation.

7.5 Runtime overhead during normal execution

We evaluated the normal execution overhead incurred by First-Aid using three sets of programs: the seven applications in Table 3, the SPEC INT2000 benchmarks [SPEC], and four allocation intensive benchmarks [Berger 2000]. We executed First-Aid with normal user inputs in two configurations: enabling only the memory allocator extension, and enabling both the memory allocator extension and checkpointing. The default checkpointing interval in the adaptive checkpointing scheme is 200 milliseconds. For the SPEC benchmarks, we used the reference data sets as the workload. For the other programs, we either chose a large testing program distributed along with the package or constructed synthesized workloads based on the commonly exercised operations, e.g., fetching various sizes of html pages for Apache and Squid servers, exporting a directory with files for CVS server, going through a mail box and reading each email for Pine and Mutt, etc.

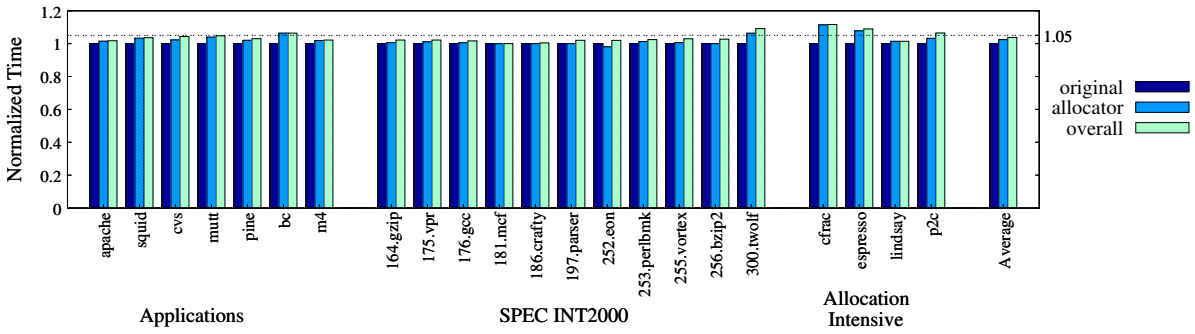


Figure 6. Overhead for First-Aid during normal execution. The 'allocator' bars show the overhead imposed by the memory allocator extension. The 'overall' bars show the combined overhead for First-Aid including both the allocator extension and checkpointing.

Figure 6 shows the overhead of First-Aid during normal execution. For desktop programs, we compare the execution time, while for server programs, we compare the average response time. We show the overhead imposed by the memory allocator extension (the second bar) and the overall overhead incurred by both the memory allocator extension and checkpointing (the third bar).

As shown in Figure 6, First-Aid incurs low overhead (0.4-11.6% with an average of 3.7%) for the tested applications. This is because both the memory allocator extension and the checkpointing mechanism (the two main sources of runtime overhead) are lightweight. Specifically, First-Aid incurs less than 5% overhead for 17 out of the 22 applications. We observe that the runtime overhead incurred by the allocator extension and the checkpointing mechanism varies for different programs. For some programs that have large memory working sets, such as the SPEC benchmarks, the checkpointing overhead is generally higher due to the frequent copy-on-write page replication. For some programs that perform intensive allocation and deallocation, such as BC, cfrac, the memory allocator extension imposes a relatively larger overhead. This is mainly due to the time spent checking for available patches and maintaining additional meta data on memory management. Since we did not spend much effort on optimization, there is room for improvement.

7.6 Space overhead

We evaluated First-Aid's space overhead, which mainly comes from the runtime patches, the memory allocator extension, and the checkpointing module.

7.6.1 Space overhead of runtime patches

We evaluated the space overhead incurred by the patches for bugs in the seven applications. We mixed the bug-triggering inputs and normal inputs to trigger the bugs. After patches were applied, we monitored all the memory object allocations and deallocations to measure the space overhead. For the padding patch, we measured the maximal memory space occupied by the padding. For the delay free patch, we measured the accumulated memory space occupied by the delay-freed objects.

Name	Heap size (Kbytes)	Patch type	Space overhead (Bytes)	Ratio
Squid	2338	padding	1016	0.04%
Pine	651	padding	1016	0.15%
Mutt	353	padding	1016	0.28%
BC	61	padding	3048	4.96%
Apache	825	delay free	14512	1.72%
CVS	292	delay free	1496	0.50%
M4	16343	delay free	128	0.0008%

Table 5. The space overhead for patches

Table 5 shows that the space overhead incurred by patches for each tested application is small, ranging from 128 bytes to 14512 bytes. This is mainly because patches are only applied to a small number of memory objects. For the padding patch, since the padded objects are usually freed very soon, the extra memory space for padding only appears for a very short time. Specifically, for Squid, Pine, and Mutt, there is only one padded object in the heap at a time; for BC, there are at most three padded objects at the same time. Even though the padding used in First-Aid is relatively large (almost 1 KB), the space overhead is still small. For the delay free patch, the space overhead is accumulated slowly. For Apache, where we triggered the bug aggressively, delay-freeing 315 objects only causes a space overhead of less than 15 KB. Note that First-Aid keeps track of all the delay-freed objects so that they can be freed once the memory consumption incurred by them reaches a customizable threshold (1 MB in our experiments).

7.6.2 Space overhead of memory allocator extension

We also evaluated the space overhead incurred by the First-Aid memory allocator extension. Detailed results are shown in Table 6. In most cases, the memory allocator extension in-

	Apache	Squid	CVS	Mutt	Pine	M4	BC	cfrac	espresso	lindsay	p2c
Original heap (MB)	0.806	2.283	0.285	0.345	0.636	15.958	0.059	0.205	0.272	1.822	0.461
First-Aid heap (MB)	0.810	2.357	0.285	0.392	0.980	16.001	0.063	0.396	0.354	1.826	0.715
Overhead	0.49%	3.24%	0.00%	13.62%	54.09%	0.25%	6.78%	93.17 %	30.15%	0.22%	55.10%
	gzip	vpr	gcc	mcf	crafty	parser	eon	perlbmk	vortex	bzip2	twolf
Original heap (MB)	180.371	20.108	83.686	94.914	0.856	30.111	0.346	56.922	108.644	184.923	3.224
First-Aid heap (MB)	180.374	20.661	83.755	94.914	0.856	30.111	0.352	63.047	109.353	184.923	5.251
Overhead	0.00%	2.76%	0.08%	0.00%	0.00%	0.00%	1.89%	10.76%	0.65%	0.00%	62.88%

Table 6. Space overhead incurred by the memory allocator extension

	Apache	Squid	CVS	Mutt	Pine	M4	BC	cfrac	espresso	lindsay	p2c
MB/checkpoint	0.068	0.211	1.068	0.286	0.345	0.222	0.040	0.210	0.185	0.297	0.055
MB/second	0.341	1.056	4.942	1.429	1.728	1.113	0.200	1.049	0.923	1.484	0.273
	gzip	vpr	gcc	mcf	crafty	parser	eon	perlbmk	vortex	bzip2	twolf
MB/checkpoint	4.574	1.355	4.488	9.691	0.941	10.870	0.056	4.566	33.390	16.080	1.585
MB/second	6.852	6.765	7.074	7.035	4.657	6.836	0.280	6.732	7.120	6.945	6.305

Table 7. Space overhead incurred by checkpointing

curs low space overhead, i.e., less than 5%. This is because it only adds 16 bytes of meta data for each memory object. However, the relative heap overhead is large in some cases, e.g., 93.17% for cfrac. These applications have a large number of small objects, which makes the relatively space overhead large. We expect that optimizations can reduce the meta data from 16 bytes to 8 bytes, which would further reduce space overhead incurred by the memory allocator extension.

7.6.3 Space overhead of checkpointing

Table 7 shows the space overhead for keeping the checkpoints in memory. Our checkpointing tool uses copy-on-write (COW) to save the dirty pages, so the space overhead is directly affected by the working set for each application. For many applications we tested, the checkpointing space overhead is low, less than 1 MB for each checkpoint. For example, keeping 100 checkpoints for Apache and Squid only takes 6.8 MB and 21.1 MB, respectively. However, for several SPEC INT2000 benchmarks, such as vortex and bzip2, the checkpointing overhead is large due to their large working sets. In these cases, First-Aid leverages an adaptive checkpointing scheme to alleviate the space overhead. As a result, the space overhead per second is kept low. When the checkpoint interval is increased and old checkpoints are discarded, First-Aid maintains the same length of history while keeping less data in memory. The downside is that the recovery time is longer when the checkpoint interval is larger.

8. Related Work

First-Aid is related to previous studies on fault tolerance, failure diagnosis, dynamic memory bug detection, defense against security attacks, and checkpointing mechanisms.

Fault tolerance. First-Aid is related to a large body of work on fault tolerance, including restart [Candea 2002; 2004, Gray 1986, Sullivan 1991], software rejuvenation [Bobbio 1998, Garg 1997, Huang 1995], and checkpointing-based fault tolerance [Osman 2002, Plank 1995]. Based on Virtual Machine (VM) replication, recently-proposed Remus [Cully 2008] achieves seamless recovery of the whole VM. These methods are effective in addressing hardware errors and non-deterministic software bugs. First-Aid complements these approaches in that it diagnoses and corrects memory errors, which are often deterministic. Furthermore, complementary to Dimmunix [Jula 2008], which is for preventing programs from re-encountering previously-seen deadlocks, First-Aid prevents programs from future memory errors caused by the same memory bugs.

Exterminator [Novark 2007], is closely related to our work. Based on a randomized memory allocator, Exterminator probabilistically pinpoints errors caused by memory bugs and generates patches for fixing the errors. To do this, Exterminiator either requires multiple runs of the same program from the entry of the program to the failure point, or relies on multiple replicas of the same program. On the contrary, First-Aid only requires short re-execution from previous checkpoints to the failure point with lightweight checkpointing support. Furthermore, the randomized memory allocator in Exterminator sizes its heap to be multiple times larger than the maximum needed by the application, which may not be acceptable to many server programs.

Failure diagnosis. Many studies on failure diagnosis focus on off-site tools, either incurring large overhead or relying on extensive human effort. Examples of such tools are delta debugging [Misherghi 2006, Zeller 2002], program slicing [Agrawal 1991, Weiser 1982, Zhang 2003], interactive debugging [GNU], and deterministic replay [King 2005, Srinivasan 2004]. Unlike these approaches, First-Aid diagnoses memory bugs based on end-user site information, which is unavailable at the developer site. Triage [Tucek 2007a] focuses on detailed on-site failure diagnosis. Unlike

Triage, First-Aid has different design trade-offs – its diagnosis algorithm aims for quick recovery and thus is more lightweight (it does not rely on any heavy-weight techniques such as program slicing or memory access instrumentation). In addition to diagnosis, First-Aid generates online patches for avoiding reoccurrence of the same memory bugs during subsequent program execution.

Dynamic memory bug detection. Many dynamic memory bug detection tools such as Purify [Hasting 1992] and Valgrind [Nethercote 2007] are mainly for in-house testing due to their heavy instrumentation of every memory access. First-Aid, as an online tool, focuses on diagnosing and patching bugs instead of detecting bugs. Furthermore, First-Aid can benefit from other lightweight memory error detection tools, such as AccMon [Zhou 2004], SafeMem [Qin 2005a], etc., by deploying them as error monitors.

Defense against security attacks. There are several studies [Brumley 2007, Costa 2007, Tucek 2007b] on helping server programs defend against security attacks that exploit memory bugs. After detecting and analyzing the attacks, these methods block "bad" inputs based on attack signatures. Complementarily, First-Aid's runtime patches enable programs to serve legitimate requests that accidentally trigger memory bugs. First-Aid does this by neutralizing the potential negative effects of such requests instead of totally discarding them.

Checkpointing and re-execution. Previous work relies on checkpointing and re-execution for many different purposes, including interactive debugging [King 2005, Srinivasan 2004], failure recovery [Osman 2002, Plank 1995, Qin 2005b], failure diagnosis [Tucek 2007a]. First-Aid exploits lightweight checkpointing and re-execution mechanisms for diagnosing and preventing failures caused by memory bugs.

9. Conclusions

In summary, First-Aid is a lightweight runtime system that provides accurate bug diagnosis, failure recovery, and future failure prevention for common memory bugs, including buffer overflow, uninitialized read, dangling pointer read/write, and double free. Using exposing and preventive environmental changes, First-Aid can accurately identify the bug types and bug-triggering memory objects. Based on such diagnostic information, First-Aid generates runtime patches and applies them to a small set of memory objects to tolerate the bug and prevent future failures caused by the same bug.

Our evaluation based on seven applications shows that First-Aid can successfully diagnose and generate runtime patches for common memory bugs. It provides fast failure recovery, i.e., 0.084-3.978 seconds recovery time with an average of 0.887 seconds. The results also show that First-Aid is effective in preventing future bug reoccurrences. Additionally, First-Aid generates a detailed on-site bug report that helps developers understand the root cause and manifestation of the bug. Furthermore, our evaluation with the seven applications, SPEC INT2000, and four allocation intensive benchmarks shows that First-Aid incurs a low runtime overhead (0.4 to 11.6% with an average of 3.7%) during normal program execution.

Acknowledgments

We would like to thank Julia Lawall, our shepherd, and anonymous reviewers for their insightful feedbacks. We would like to thank Yuanyuan Zhou, for helpful discussion on the idea in its early stages. We are also grateful to Matthew Koop, Guoqing Xu, Wei Huang for their helpful comments on improving the presentation.

References

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

[Arbaugh 2000] W. A. Arbaugh, W. L. Fithen, and J. McHugh. Windows of vulnerability: A case study analysis. *Computer*, 33 (12):52–59, 2000.

[Berger 2000] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of Intl. Conf. on Architectural support for programming languages and operating systems (ASPLOS'00)*, pages 117–128, 2000.

[Berger 2006] E. D. Berger and B. G. Zorn. DieHard: Probabilistic memory safety for unsafe languages. In *Proceedings of ACM SIGPLAN conference on Programming language design and implementation (PLDI'06)*, pages 158–168, 2006.

[Bobbio 1998] A. Bobbio and M. Sereno. Fine grained software rejuvenation models. In *Intl. Computer Performance and Dependability Symposium (ICPDS '98)*, pages 4–12, 1998.

[Bond 2008] M. D. Bond and K. S. McKinley. Tolerating memory leaks. In *Proceedings of Intl. Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '08)*, pages 109–126, Oct. 2008.

[Brumley 2007] D. Brumley, H. Wang, S. Jha, and D. Song. Creating vulnerability signatures using weakest pre-conditions. In *Proceedings of Computer Security Foundations Symposium (CSF'07)*, pages 311–325, Venice, Italy, 2007.

[Candea 2002] G. Candea, J. Cutler, A. Fox, R. Doshi, P. Garg, and R. Gowda. Reducing recovery time in a small recursively restartable system. In *Proceedings of Intl. Conf. on Dependable Systems and Networks (DSN'02)*, pages 605–614, 2002.

[Candea 2004] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot – A technique for cheap recovery. In *Proceedings of Symposium on Operating System Design and Implementation (OSDI'04)*, pages 31–44, 2004.

[Costa 2007] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: Securing software by blocking bad input. In *Proceedings of ACM SIGOPS symposium on Operating systems principles (SOSP'07)*, pages 117–130, 2007.

[Cully 2008] B. Cully, G. Lefebvre, D. T. Meyer, A. Karollil, M. J. Feeley, N. C. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of Symposium on Networked Systems Design and Implementation (NSDI'08)*, pages 161–174, Apr 2008.

[Garg 1997] S. Garg, A. Puliafito, M. Telek, and K. S. Trivedi. On the analysis of software rejuvenation policies. In *Proceedings of the Annual Conference on Computer Assurance (CA'97)*, pages 88–96, 1997.

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

[Gray 1986] J. Gray. Why do computers stop and what can be done about it? In *Proceedings of Symposium on Reliable Distributed Systems (RDS' 86)*, pages 3–12, 1986.

[Gu 2003] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z. Yang. Characterization of Linux kernel behavior under errors. In *Proceedings of Intl. Conf. on Dependable Systems and Networks (DSN'03)*, pages 459–468, Jun 2003.

[Hasting 1992] R. Hasting and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 125–136, Dec 1992.

[Huang 1995] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton. Software rejuvenation: Analysis, module and applications. In *Proceedings of International Symposium on Fault-Tolerant Computing (FTC'95)*, pages 381–390, Jun 1995.

[Jula 2008] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *Proceedings of Symposium on Operating System Design and Implementation (OSDI'08)*, pages 295–308, Dec 2008.

[King 2005] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of USENIX Annual Technical Conference (USENIX'05)*, pages 1–15, 2005.

[Lea 1996] D. Lea. A Memory Allocator, 1996.

[Lowell 1998] D. E. Lowell and P. M. Chen. Discount checking: Transparent, low-overhead recovery for general applications. Technical report, CSE-TR-410-99, University of Michigan, 1998.

[Luk 2005] C. 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 Programming language design and implementation (PLDI'05)*, pages 190–200, 2005.

[Lvin 2008] V. B. Lvin, G. Novark, E. D. Berger, and B. G. Zorn. Archipelago: Trading address space for reliability and security. In *Proceedings of Intl. Conf. on Architectural support for programming languages and operating systems (ASPLOS'08)*, pages 115–124, 2008.

[Misherghi 2006] G. Misherghi and Z. Su. HDD: Hierarchical delta debugging. In *Proceedings of the Intl. Conf. on Software engineering (ICSE'06)*, pages 142–151, 2006.

[Nethercote 2007] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of ACM conference on Programming language design and implementation (PLDI'07)*, pages 89–100, 2007.

[Novark 2007] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: Automatically correcting memory errors with high probability. In *Proceedings of ACM SIGPLAN conference on Programming language design and implementation (PLDI'07)*, pages 1–11, 2007.

[Osman 2002] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of Zap: A system for migrating computing environments. In *Symposium on Operating systems design and implementation (OSDI'02)*, pages 361–376, 2002.

[Plank 1995] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under Unix. In *Usenix Winter Technical Conference*, pages 213–224, January 1995.

[Qin 2005a] F. Qin, S. Lu, and Y. Zhou. SafeMem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. In *Proceedings of Intl. Symposium on High-Performance Computer Architecture (HPCA'05)*, pages 291–302, Feb 2005.

[Qin 2005b] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies – a safe method to survive software failure. In *Proceedings of ACM Symposium on Operating System Principles (SOSP'05)*, pages 235–248, Oct 2005.

[Rinard 2004] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of Symposium on Operating System Design and Implementation (OSDI '04)*, pages 21–21, Dec 2004.

[Sidiroglou 2005] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building a reactive immune system for software services. In *Proceedings of USENIX Annual Technical Conference (USENIX'05)*, pages 149–161, 2005.

[SPEC] SPEC. http://www.spec.org/cpu2000.

[Srinivasan 2004] S. Srinivasan, C. Andrews, S. Kandula, and Y. Zhou. Flashback: A light-weight extension for rollback and deterministic replay for software debugging. In *Proceedings of the USENIX 2004 Annual Technical Conference (USENIX'04)*, pages 29–44, Jun 2004.

[Sullivan 1991] M. Sullivan and R. Chillarege. Software defects and their impact on system availability – A study of field failures in operating systems. In *Proceedings of the Annual Intl. Symposium on Fault-Tolerant Computing (FTC'91)*, pages 2–9, Jun 1991.

[Symantec 2006] Symantec. Internet security threat report. http://www.symantec.com/enterprise/threatreport/index.jsp, Sept 2006.

[Tang 2008] Y. Tang, Q. Gao, and F. Qin. LeakSurvivor: Towards safely tolerating memory leaks for garbage-collected languages. In *Proceedings of USENIX Annual Technical Conference (USENIX'08)*, pages 307–320, Jun. 2008.

[Tucek 2007a] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: Diagnosing production run failures at the user's site. In *Proceedings of Symposium on Operating systems principles (SOSP'07)*, pages 131–144, 2007.

[Tucek 2007b] 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 2007 EuroSys Conference*, pages 115–128, 2007.

[US-CERT] US-CERT. US-CERT vulnerability notes database. http://www.kb.cert.org/vuls.

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

[Zeller 2002] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of ACM symposium on Foundations of software engineering (FSE'02)*, pages 1–10, 2002.

[Zhang 2003] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *Proceedings of International Conference on Software Engineering (ICSE'03)*, pages 319–329, 2003.

[Zhou 2004] P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas. AccMon: Automatically detecting memory-related bugs via program counter-based invariants. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO'04)*, pages 269–280, 2004.