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

Qi Gao, Wenbin Zhang, Yan Tang, and Feng Qin
Department of Computer Science and Engineering
The Ohio State University
{*gaoq, zhangwen, tangya, qin*}*@cse.ohio-state.edu*

Abstract

Memory bugs in C/C++ programs severely reduce software availability and security during production runs. This paper presents First-Aid, a lightweight runtime system that survives software failures due to common memory management bugs and prevents future failures caused by the same bugs during production runs. Upon failures, First-Aid diagnoses the occurring bug types and bug-triggering memory objects by rolling back the program to previous checkpoints and leveraging two types of environmental changes that can *prevent* or *expose* memory bug manifestation during re-execution. Based on the accurate diagnosis, First-Aid generates and applies runtime patches to avoid the occurring memory bugs and prevent the recurrence of the same bugs. Furthermore, First-Aid validates the consistent effectiveness of the runtime patches and generates on-site diagnostic reports for developers to fix the bugs.

We implemented First-Aid on Linux and evaluated it with seven applications that contain various types of common 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 failures within 0.084 to 3.978 seconds. Our evaluation shows that the generated runtime patches effectively prevent failures caused by the same bugs. Additionally, First-Aid provides detailed on-site diagnostic information to help programmers understand both the root cause and manifestation of the occurring bugs. Furthermore, First-Aid incurs low overhead, 0.4-11.6% with the average of 3.73% during normal execution for the above seven applications, SPEC INT2000, and four allocation intensive programs.

1 Introduction

1.1 Motivation

Memory management bugs, a major type of common software defects, severely affect system availability and security. During production runs, these bugs such as buffer overflows and dangling pointers can corrupt memory data, leading to program crashes or hangs. Furthermore, malicious users often launch security attacks by exploiting these bugs. According to CERT [35], 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 [34]. Previous studies [33, 2] showed that it takes several weeks on average to diagnose bugs and generate fixing patches. During this long time window, users have to choose either running the software with bugs and tolerating problems such as intermittent crashes and potential attacks, or not running the software and experiencing costly system downtime. Neither option is desirable.

Therefore, it is critical to have a system that can *quickly recover programs from software failures caused by memory bugs, protect program from future failures due to the same bugs, and provide useful on-site failure information for developers to quickly fix the bugs.*

While several recent proposals help programs survive failures caused by memory bugs, they suffer one or more the following limitations: unsafe speculation on programmers' intention [28, 29], inability of preventing failures due to the same bugs [4, 27], little diagnostic information to developers [20], and large runtime and space overhead [23, 4, 20]. For example, failure oblivious computing [28] discards out-of-bound writes and manufactures an arbitrary value for out-of-bound reads. While this approach may survive failures for certain types of applications, the speculation on programmers' intention could easily lead to programs' misbehavior. DieHard [4] and Exterminator [23] probabilistically prevent failures caused by memory bugs via a randomized memory runtime system. However, large time and space overheads restrict them from being adopted for production runs.

Our previous work Rx [27] quickly recovers programs from failures by re-executing program from previous checkpoints with program execution *environmental changes* applied. For example, adding paddings (an environmental change) to *all* memory objects during recovery could avoid the occurring buffer overflow bugs. While being effective and safe to avoid the occurring memory bugs, Rx cannot prevent future failures caused by the same bugs. This is because it will disable the environmental changes after surviving the current failures due to potentially large overhead. Furthermore, the on-site failure recovery information (whether and what environmental change work) is quite limited and may mislead developers because the failure-surviving environmental change may not directly related to the bug.

1.2 Our Contributions

In this paper, 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 occurring memory bugs upon failures by leveraging efficient checkpointing-and-re-execution mechanisms and execution environmental changes. Based on diagnostic results, First-Aid generates and applies *runtime patches* (environmental changes for bug-triggering memory objects) to correct the occurring memory bugs. After recovering the programs, First-Aid further validates the runtime patches and generates detailed bug reports to developers.

To diagnose the occurring bugs, 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* (forcing a certain type of bug to manifest itself) and *preventive changes* (preventing a certain type of bug from manifestation). Based on the execution results such as failure symptoms and exposed bug manifestation, First-Aid can conclude whether one bug type is occurring or not. For example, to determine whether a buffer overflow bug is occurring or not, First-Aid applies the exposing change for buffer overflow, i.e., "padding each memory object with canary", and the preventive changes for all other memory bug types during re-execution. The term "canary" is referred to as certain memory content patterns that unlikely appear during normal program execution. If the canary in some padding is corrupted, First-Aid knows that an occurring buffer overflow bug is exposed. Additionally, exposing changes can help First-Aid identify the bug-triggering memory objects. In the same example, based on the corrupted paddings, First-Aid can identify the overflowed memory objects.

Based on the identified bug type and bug-triggering memory objects, First-Aid generates and applies runtime patches to a small set of memory objects that can potentially trigger the previously-seen bugs. In this way, the patches only incurs very small overhead and thereby are suitable for long term application. As a result, First-Aid keeps these patches persistently so that they not only help programs survive current failures but also prevent future failures caused by the same bugs.

We have implemented First-Aid on Linux and evaluated its functionality and performance with seven applications, including three server applications (Apache, Squid, and CVS) and four client 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 [30] and four allocation intensive benchmarks [3]. Compared with previous approaches, First-Aid has one or more of the following advantages:

- **Prevention on bug recurrence.** First-Aid can apply the runtime patches for the entire program execution and thereby prevent future failures due to the recurrence of the same bug. Furthermore, the patch generated by First-Aid can be stored persistently to prevent bug occurrence on subsequent runs and other process running the same program. This improves the overall reliability.
- **Fast failure recovery.** First-Aid can quickly recover programs from failures due to its lightweight diagnostic algorithm. Our evaluation with seven tested applications shows that the recovery time ranges from 0.084 to 3.978 seconds with an average of 0.887 seconds.
- **Safe recovery.** The runtime patches generated by First-Aid avoid bugs by legitimately changing the program's execution environments with conformance to the standard memory allocator interface specifications.
- **Low normal-run overhead.** First-Aid incurs very low runtime overhead during normal program execution (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 3.73%. This indicates First-Aid is suitable for production runs.
- **Informative bug report.** First-Aid provides programmers with accurate and detailed bug information: the occurring bug type, the bug-triggering memory objects, their allocation/deallocation sites, and relevant illegal memory accesses. This diagnostic information allows programmers to easily understand both root causes and manifestation of the occurring bugs. As a result, programmers can quickly diagnose the problems and fix the bugs in source code.

2 Main Idea of First-Aid

Figure 1 shows the working scenario of First-Aid. During normal execution, First-Aid periodically takes checkpoints for the application process. Upon failures, First-Aid diagnoses the occurring bugs and generates corresponding runtime patches. Then it applies the patches for recovering the programs from the failure and preventing the programs from future failures due to the same bugs. After failure recovery, First-Aid validates the patches and generates detailed bug reports.

Bug diagnosis. First-Aid diagnoses the occurring bugs by re-executing the failed processes from previous checkpoints with multiple iterations. In each iteration, First-Aid rolls back the

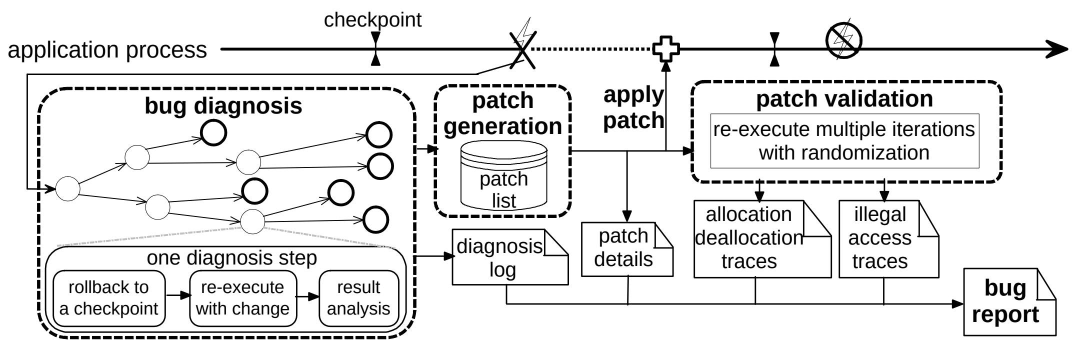


Figure 1: Working scenario of First-Aid

program to a previous checkpoint, tentatively applies one or multiple environmental changes to all or a subset of memory objects, and re-executes the program. Based on the re-execution results, it narrows down the possible causes of the occurring bugs. If First-Aid can identify the bug type and bug-triggering memory objects, the diagnostic process stops. Otherwise, First-Aid will repeat the above process until it identifies the bug or times out.

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

Table 1 describes these two types of changes for each bug type, including buffer overflow, double free, dangling pointer read/write, and uninitialized read. For example, adding paddings to both ends of each newly-allocated memory object can prevent buffer overflow bugs, while adding canary-filled paddings 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 can manifest dangling pointer read bugs as failures and manifest 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 is 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 program continue the normal execution. Second, the failure is caused by deterministic memory management bugs. In this case, First-Aid passes the diagnostic results to the next step for patch generation. The last case is deterministic bugs that First-Aid cannot handle, e.g., semantic bugs. In this case, First-Aid stops the program execution and resorts to other recovery schemes.

Patch generation and application. Based on the bug diagnostic information, First-Aid generates runtime patches for recovering the programs from the failure and preventing future failures caused by the same bugs. A runtime patch is a tuple of a preventive change corresponding to the identified bug type and a patch application point. The patch application point is the allocation or deallocation *call-site* of the bug-triggering memory objects. The *call-site* is referred to in this paper as the multiple-level calling contexts on the stack. It can serve as "signatures" of the bug-triggering memory objects because memory objects with the same call-site of allocation or deallocation often have similar characteristics such as leaking or overflow [26, 23].

First-Aid stores the generated patches and applies them to the application process at run time. More specifically, these patches take effect when a later memory allocation or deallocation call-site matches the patch application points. By applying identified preventive changes at these points, First-Aid can protect programs from future failures caused by the same bugs.

Runtime patches generated by First-Aid usually have a few specific application points to apply and have little performance and resource overhead. Therefore they are applied for the entire program execution after failure recovery. Further more, 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 reliability.

Patch validation and bug report. After a runtime patch being generated and applied for quick recovery, First-Aid performs a further step to validate the patch and collects detailed infor-

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 large padding to objects	pad objects with canary (canary corruption)	allocation
dangling pointer read	1.premature buffer free		fill with canary (failure)	
dangling pointer write	2.forget to set NULL	delay free	fill with canary (canary corruption)	deallocation
double free			check parameters (freed twice)	
uninitialized read	1. assume zeros in buffers	fill objects with zero	fill with canary (failure)	allocation

Table 1: Memory bug types and corresponding environmental changes

mation for the bug report.

To confirm 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., the accesses through dangling pointer, the 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, the corresponding patch will be removed, and the event will be reported.

Instead of hiding bugs from developers, First-Aid assists them in diagnosing and fixing the bugs off-site 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 easily locate the bug-related memory management code. Additionally, the detailed execution traces on memory management operations and illegal accesses can help developers to understand the manifestation process of the occurring bugs.

3 First-Aid System Design

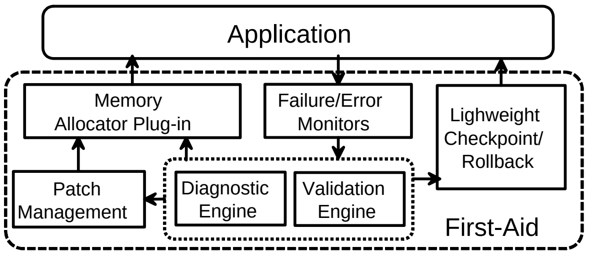


Figure 2: Software architecture of First-Aid

Figure 2 shows the software architecture of First-Aid. It consists of six major user-level and kernel-level components: (1) a lightweight memory allocator plug-in for assisting bug diagnosis, patch validation, and patch application, (2) failure/error monitors 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 patch management module for managing the generated runtime patches and controlling patch application, (5) a diagnostic engine for diagnosing the occurring bugs and generates runtime patches (details in Section 4), and (6) a validation engine for validating the consistent effectiveness of the patches and collecting on-site diagnostic information (details in Section 5).

Memory allocator plug-in. The memory allocator plug-in collects memory object information and applies preventive and/or exposing changes for diagnosing bugs and surviving failures. More specifically, it operates on one of the three modes: *normal* mode during normal execution, *diagnostic* mode during First-Aid's diagnostic procedure upon failures, and *validation* mode during the validation procedure.

In the normal mode, it mainly just performs the standard memory allocation and deallocation requests. Additionally, it checks the availability of patches and applies them if the current call-site matches any patch application points. This ensures that First-Aid incurs low overhead during normal execution (see Section 7.5).

In the diagnostic mode, the memory allocator plug-in performs three functions during re-execution. First, it applies preventive and/or exposing changes, instructed by the diagnostic engine, to all or a subset of memory objects. Second, it collects call-site information for each memory object allocation and deallocation. Such information is used for bug diagnosis and future patch application. Third, it checks the validity of parameters for each deallocation request for detecting invalid free and double free bugs.

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

Failure/Error monitors. The failure/error monitors detect failures or errors at runtime and notify the diagnostic engine upon detection. The cheapest way 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 [39] as long as they incur low overhead. Our current implementation is based on assertion failures and exceptions.

Adaptive checkpoint/re-execution. First-Aid leverages the lightweight checkpointing and re-execution runtime system provided in Rx. 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 [18, 31] by keeping a copy of each accessed file and file pointers at the beginning of each checkpoint and reinstating it for rollback. Additionally, it leverages a network proxy to record network messages during normal execution and replay them during re-execution. More details can be found in Rx [27] and Flashback [31].

Instead of using fixed intervals as in Rx, First-Aid dynamically adjusts the checkpointing intervals for balancing the low normal execution overhead and quick recovery delay. 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 users, i.e., the COW page rate is too high, First-Aid will gradually increase the interval so that the runtime overhead will decrease. On the other hand, the recovery delay becomes longer when the checkpoint interval is larger. Once the checkpoint interval reaches the user-specified maximal interval $T_{checkpoint}$, First-Aid stops increasing the interval.

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 to 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 plug-in queries the patch pool for a certain call-site of allocation or deallocation.

4 Bug Diagnosis

The diagnostic engine uses two phases to diagnose the occurring bugs. The first phase is to search for the best checkpoint to apply the patches for surviving the occurring bugs. The second phase is to perform in-depth diagnosis to identify the memory bug type and the patch application points, i.e., allocation/deallocation call-sites of the bug-triggering memory objects. In subsection 4.3, we compare the First-Aid bug diagnosis with Rx.

4.1 Phase 1: Identify the Checkpoint for Patching

In order to be both effective and efficient, the patch should start taking effect from the latest checkpoint before the bug-triggering point. To identify that checkpoint, First-Aid rolls back the program to previous checkpoints in a reverse chronological order and re-executes the program. It first re-executes the program without any memory management change and if the program succeeds, (the failure is likely caused by non-deterministic bugs), then First-Aid only logs this event and lets the program continue execution. If the program fails, First-Aid will re-execute the program with *all* the preventive changes on *all* memory objects from the same checkpoint. 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 on the checkpoint right before this one. After trying a certain number of previous checkpoints, First-Aid stops searching, reports it as a non-patchable bug, and resorts to other recovery mechanisms. 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 of checkpoint intervals after the failure point.

One key challenge in this scheme is the possible misidentification of the latest checkpoint before the bug-triggering point. In some occasions, 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 manifestation of some memory bugs may rely on the heap layout which could be disturbed by the preventive changes applied later.

Figure 3 shows such an example with the heap containing a few objects. 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 for an object E. At the end, a failure occurs since 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 can avoid the failure. This is because the preventive change for buffer overflow adds paddings to E, denoted as E_{padded}, and makes 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 memory corruption caused by the dangling pointer write.

To address this issue, First-Aid uses a technique called *heap marking* to verify that the bug indeed occurs *after* the checkpoint currently being tried. 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 empty (freed) chunks in the heap by filling their contents with canary. Additionally, it adds a padding filled with canary after the last memory object in the heap. With this change, if the failure originally caused by previously-triggered dangling pointer or buffer overflow bugs is accidentally avoided due to heap layout change, First-Aid can detect corruption to the canary. For dangling pointer reads, the heap marking technique makes the failure still occur during re-execution due to the canary.

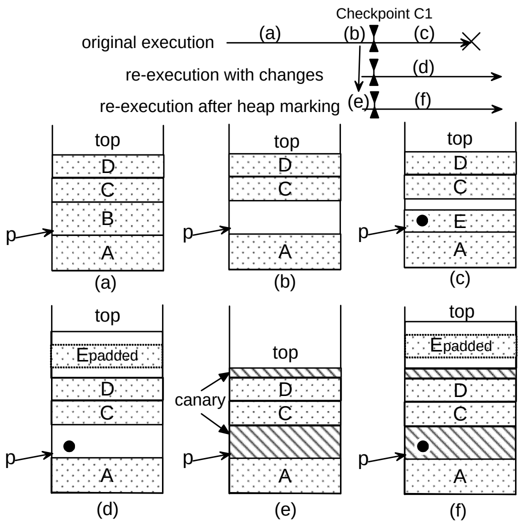


Figure 3: The issue in accurately identifying the checkpoint for patching and the heap marking technique

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 the phase 2 algorithm for more in-depth analysis. It first identifies the types of the occurring bugs and then identifies the allocation/deallocation call-sites of bug-triggering memory objects. Note that First-Aid takes into consideration the case where multiple types of bugs are triggered in the execution and the program will not survive unless all of them are avoided. Therefore, the algorithm needs to carefully separate each type of the occurring bugs.

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, the undecided set S_u contains all the bug types and the identified set S_i is empty. For each bug type b from the undecided set S_u, First-Aid applies the following changes to all memory objects: the exposing change for the bug type b, and the preventive changes for all other bug types in the set of $S_u \cup S_i - \{b\}$. 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 with the type b do exist and 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 simply remove it from the undecided set S_u. At the end, the identified set S_i contains all the types of the occurring bugs.

To minimize the iterations of re-execution in diagnosis, First-Aid checks whether the current identified set S_i covers all occurring bug types after each new bug type is identified. If so, First-Aid stops searching for more bug types. First-Aid performs such check by applying preventive changes for bug types in the identified set along with exposing changes for the undecided set in one iteration of re-execution.

After identifying the bug type, the next step is to identify the call-sites of bug-triggering memory objects. For buffer overflow and dangling pointer write, First-Aid can directly identify the bug-triggering memory objects by looking for canary corruption in paddings and freed memory objects, respectively. For double frees, it identifies the bug-triggering memory objects by checking the parameters passed to free 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 trigger 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 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 one 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 the interferences 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, First-Aid needs to conduct multiple rounds of the above binary 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 takes $O(M * log\ N)$ re-executions in total.

4.3 Bug diagnosis comparison between First-Aid and Rx

Our previous work, Rx [27], can also provide quick recovery for failures caused by memory bugs. However, it does not (and not intend to) perform in-depth diagnosis since it aims for fast recovery. Differently, First-Aid's goals are not only recovering failures quickly, but also preventing recurrence of the same bugs and providing on-site diagnostic information. Therefore, First-Aid performs accurate diagnosis on memory bugs in the following two aspects.

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

Exactness: First-Aid identifies a small set of bug-triggering memory objects, 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 overflow occurs by checking canary in all the paddings.

5 Patch Validation and Bug Report

Although First-Aid's diagnostic algorithm will not misdiagnose one type of memory bug as another, it may diagnose 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 bugs in the source code.

To rule out the possibilities of such misdiagnosis, the random side-effects of a patch must be distinguished from the desired effects. First-Aid does so by checking the consistent effects of a runtime patch under memory layout randomization. During validation, First-Aid re-executes the buggy region of the program three more iterations with randomized allocation algorithm. In each iteration, memory allocator's activities and illegal memory accesses are traced and logged. The allocator plug-in provides its own logging and a dynamic instrumentation tool, Pin [19], is used to trace memory accesses. Specifically, for each memory allocation/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/writes to delay-freed objects, and the uninitialized reads. With these traces, First-Aid checks whether the effects of the patch is consistent among multiple re-executions based on following criteria: a) the patch is triggered by 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 to the corresponding memory object (the memory object address is randomized though). If the consistent check fails, First-Aid will remove the runtime patch and report the problem.

The traces collected in the above validation step will be organized into the bug report for the developers. Specifically, besides the usual bug report package (core dump, event log, etc.), First-Aid provides four pieces of new information in the bug report: a) diagnosis log, which helps developers understanding the diagnostic process; b) runtime patch information, which includes the bug type and accurate call-sites of the relevant allocation/de-allocation operations, pointing developers to the critical source code section related to the bug; c) allocation/deallocation traces in the bug region, which shows clearly when the runtime patch takes effect and what memory objects are affected; and d) illegal memory accesses in the bug region, which shows the instructions that have made illegal memory accesses. With the above information on both the bug root causes and the bug manifestation process, developers can quickly fix the bug.

6 Issues and Discussion

Common assumptions on memory bugs: Even though the First-Aid's diagnosis algorithm is comprehensive, it is based on several assumptions on common characteristics of memory bugs. These key assumptions include: the buffer overflow bugs must corrupt data within a neighboring region of the memory object instead of with an arbitrary offset; the canary must not be coincidentally used in the buggy memory object as normal value; programmers have the intention to initialize the newly allocated buffers to zeros in the case of uninitialized read bugs. These assumptions cover most common cases in real memory bugs and thereby are also used in previous work [27, 23]. However, exceptions can happen theoretically and result in failed attempts to patch.

Customized memory allocator in applications: To avoid some memory bugs or improve performance, many 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 is based on multi-level call-sites. If an application-specific allocator is used, the First-Aid memory allocator plug-in should be ported to the allocator for reaching full capability in handling memory bugs. Porting generally is straightforward.

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 of memory, and a 100Mbps Ethernet connection between them. The operating system kernel was the Linux 2.4.22 kernel modified with Flashback [31] checkpointing support. We ran servers on one machine and clients on the other. We implemented the memory allocator plug-in based on Lea allocator [17], the default memory allocator used in GNU C library.

We evaluated First-Aid with seven applications including three server applications (Apache, Squid, and CVS) and four

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 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.

App.	Ver.	Bug	LOC	App. Desc.
Apache	2.0.51	dangling pointer read	263K	web server
Apache-uir	2.0.51	uninitialized read		
Apache-dpw	2.0.51	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.

client 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 original developers and we injected two bugs into Apache httpd server: Apache-uir contains an uninitialized read and Apache-dpw contains a dangling pointer write.

7.2 Overall Effectiveness

We executed these seven applications with First-Aid. To simulate bug triggering in real scenarios, we mixed the bug-triggering inputs and normal inputs. For each application, we measured the failure recovery time, number of rollbacks for diagnosis, runtime patch, number of call-sites being patched, validation time, and whether the patch can avoid future errors caused by the same bug. Table 3 shows these results.

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

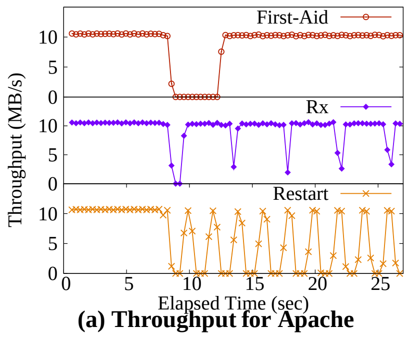
First-Aid provides quick failure recovery and thereby hides program failures from users. As shown in Table 3, the failure recovery time ranges from 0.084 to 3.978 seconds with an average 0.887 seconds. This is because First-Aid's efficient diagnosis algorithm and lightweight checkpointing-and-re-execution mechanism. For example, First-Aid quickly pinpoints the occurring bugs in seven cases after 6-9 iterations of program re-execution, resulting in less than 1 second failure recovery time. The relatively long recovery time for the dangling pointer read case in Apache is because its bug-triggering point is a little far (three checkpoints) from the failure point. Fortunately, previous studies [12] show that the error propagation distance (the distance from the bug-triggering point to the failure point) is very short for most cases and thereby First-Aid can quickly recover programs from failures.

Table 3 shows that First-Aid is effective in avoiding future errors caused by the same bug. In the experiments, we repetitively send the bug-triggering inputs. First-Aid successfully prevents future memory errors due to the same bugs after applying the runtime patches. This is because First-Aid's patch is lightweight, i.e., applying to 1-7 call-sites in the experiments, and thereby can be enabled for preventing future errors during the entire program execution.

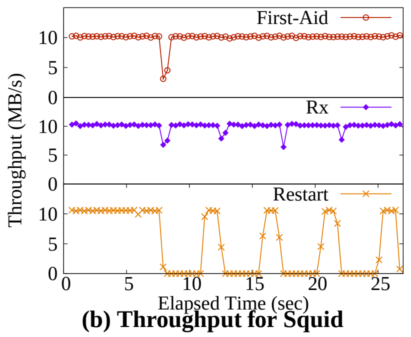
As shown in Table 3, First-Aid successfully validates the generated patches within a small amount of time, i.e., 3-18 seconds, for the tested seven applications. During the validation process, the runtime patches show consistent effects, i.e., patches applied to the same number of memory objects and each illegal access matches across different runs. The small validation time is because First-Aid re-executes the program from the checkpoint identified by the diagnostic engine instead of from the very beginning.

7.3 Future Error Prevention

We evaluated First-Aid's capability of future error prevention and compared it with two alternatives, the restart method and



(a) Throughput for Apache



(b) Throughput for Squid

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

Rx, using two representative server programs, Apache and Squid. In the experiments, after a certain period in normal execution, we periodically triggered the real bug by sending bug-triggering requests mixed with normal inputs.

Figure 4 shows that First-Aid effectively prevents future errors caused by the same bugs while the restart approach and Rx cannot. In the case of Apache (Figure 4 (a)), First-Aid diagnoses the occurring and recovers the programs from the first failure for around 4 seconds and then maintains stable performance when facing the same bug-triggering inputs repetitively. This is because the patch generated by First-Aid is correct and accurate so that it can effectively avoid the same memory bug during future program execution. On the contrary, Rx suffers the same bug repetitively during subsequent program execution even though it can recover the program from the failure for the first time. This is because Rx applies the environmental changes to all the memory objects without accurate bug diagnosis. Consequently, it has to disable the potentially expensive environmental changes after passing the buggy program region. For the restart approach, it suffers the same bug repetitively since the occurring 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 because of its shorter error propagation distance.

Table 4 further illustrates the accuracy of First-Aid's patches as well as the reason why Rx has to disable the environmental change after surviving failures. For the tested seven 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 in 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 all the time during program execution for preventing future errors due to the same bugs.

Name	Call-sites			Objects		
	First -Aid	Rx	Ratio	First -Aid	Rx	Ratio
Apache	7	32	21.8%	315	2567	12.2%
Squid	1	61	1.64%	1	3626	0.028%
CVS	1	44	2.27%	17	306	5.56%
Pine	1	380	0.26%	11	2881	0.381%
Mutt	1	216	0.46%	2	5004	0.040%
M4	2	8	25.0%	3	183	1.64%
BC	3	34	8.82%	5	732	0.683%

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

```
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. 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
5. 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
      ...
```

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

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 bug. It consists of five parts: a failure core-dump, a diagnosis log, runtime patch information, an illegal access trace, and a memory allocation/deallocation trace. The patch information clearly

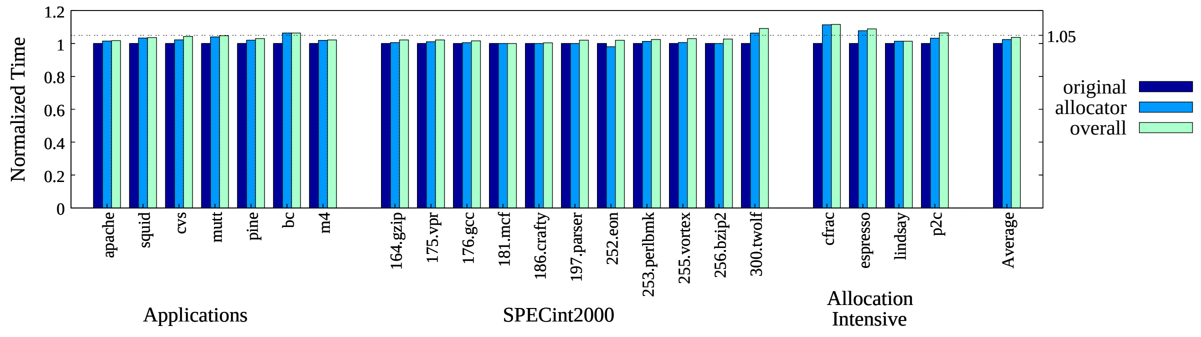


Figure 6: Overhead for First-Aid during normal execution. The 'allocator' bars show the overhead imposed by memory allocator plug-in. The 'overall' bars show the combined overhead for First-Aid including both allocator and checkpointing.

indicates that the bug is the dangling pointer read and related to the LDAP cache. This is because the bug can be avoided by delaying free in the `util_ald_cache_purge` and `util_ldap_search_node_free`, the callers of a wrapper free in Apache (`util_ald_free`). Additionally, the multi-level call-sites show that all the delayed frees are issued indirectly through the `util_ald_cache_purge`. By comparing the allocation/deallocation traces with and without runtime patch applied, we can notice that without patch, the freed memory is reallocated later. Based on all these hints, it is not difficult to pinpoint the bug in `util_ald_cache_purge`: dangling pointers are created in the cache cleanup operation.

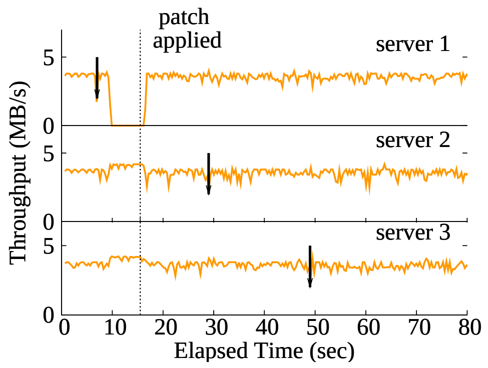


Figure 7: Throughput for three Apache httpd processes under First-Aid with bug triggering inputs sent to each process started at different time

7.5 Normal Execution Overhead

We evaluated the normal execution overhead incurred by First-Aid using three sets of programs: the seven applications in Table 3, SPEC INT2000 benchmarks [30], and four allocation intensive benchmarks [3]. We executed First-Aid with normal user inputs in two configurations: enabling only the memory allocator plug-in, and enabling both the memory allocator plug-in and checkpointing. The default checkpointing interval in the adaptive checkpointing scheme is 200 milliseconds. For SPEC benchmarks, we used the reference data sets as the workload. For other programs, we either chose a large testing program distributed along 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 mail box and reading each email for Pine and Mutt, etc.

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

As shown in Figure 6, First-Aid incurs low overhead (0.4-11.6% with an average of 3.73%) for most of the tested applications. This is because the memory allocator plug-in and the checkpointing mechanism (two major sources for the runtime overhead) are lightweight. Specifically, First-Aid incurs less than 5% for 17 out of the 22 applications. Furthermore, runtime overhead incurred by the allocator plug-in and checkpointing mechanism varies for different programs. For some programs that have large memory working-sets, such as 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 plug-in imposes relatively larger overhead. This is mainly due to the time spent on 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 improving performance.

7.6 Patch Sharing on Multiple Server Processes

We have evaluated First-Aid's ability of protecting multiple server processes that are executing the same program from the same bug. In this experiment, we start three processes of Apache httpd server on one host. The workload drives the throughput of each server process to the maximum 4.1 MB/s, which is limited by the network card maximal bandwidth 11.7 MB/s. We design a longer experiment and start mixing the bug-triggering requests with normal user requests at different time for different server processes. The arrows in Figure 7 indicate the start sending time of bug-triggering requests to the three servers.

Figure 7 shows that First-Aid can effectively prevent future failures caused by the same bugs from occurring at all the processes. For example, the bug-triggering inputs for all the server processes only cause server 1's crash at around the 11th second. This is because First-Aid generates the patch at around the 15th second for server 1 and immediately propagates it to the other two processes. After applying the patch, all the three processes can prevent future failures caused by the same bug. This indicates that First-Aid can alleviate Denial-Of-Service (DOS) attacks made to server farms.

7.7 Effectiveness to Multiple Bugs

To evaluate First-Aid's ability of handling multiple bugs and multiple crashes, we inject five bugs, i.e., one buffer overflow, one dangling pointer read, one dangling pointer write, and two uninitialized reads, to Apache, and construct two variants of Apache, denoted as *Apache-mbug1* and *Apache-mbug2*. In Apache-mbug1 and Apache-mbug2, a single request $R_{bugtrigger}$ triggers all the five bugs. After that, in Apache-mbug1, a single request R_{crash} makes all the five triggered bugs to manifest and crash the program. Differently, in Apache-mbug2, a request R_{crash1} makes the dangling pointer read, the dangling pointer write, and one uninitialized read to manifest, while another request R_{crash2} makes the buffer overflow and the other uninitialized read to manifest.

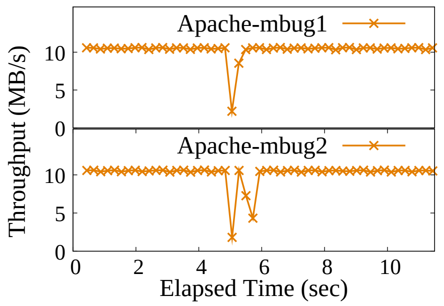


Figure 8: First-Aid with multiple bugs in Apache

Figure 8 shows the throughput when the multiple injected bugs and multiple crashes are triggered. The results indicate that First-Aid can successfully separate multiple bugs and generate patches for each of the bugs, no matter there is one or more crash-triggering requests of these bugs. In the `Apache-mbug1` case, as shown in the top part of Figure 8, First-Aid performs a thorough diagnosis and generates patches for all the five injected bugs. This is because all the five bugs need to be fixed for avoiding the crash triggered by R_{crash}. The recovery time is 0.217 second with 23 rollbacks for diagnosis. In the `Apache-mbug2` case, as shown in the bottom part of Figure 8, the first crash can be avoided by only patching three individual bugs (dangling pointer read/write, an uninitialized read). Therefore, First-Aid only focuses on these three bugs when diagnosing the first crash, even though all the five bugs are triggered at the same time. The recovery time for this crash is 0.18 second. For the second crash, the recovery time is longer (0.198 second) because of the longer distance from the bug triggering point to the crash point.

7.8 Space Overhead in Normal Run

We have evaluated the space overhead incurred by First-Aid, which mainly comes from the memory allocator plug-in and the checkpointing module. Table 5 shows the detailed results on the space overhead from the memory allocator plug-in. In most cases, the memory allocator plug-in incurs very low overhead, less than 5%. This is because the plug-in only adds 16 bytes meta data for each memory objects. However, there are several cases where the relative heap overhead is large, e.g., 93.17% for cfrac. The reason is that these applications have a large number of small objects, which makes the relatively space overhead large. We expect optimizations can reduce the meta data size from 16 bytes to 8 bytes, which further reduces space overhead incurred by the memory allocator plug-in.

Table 6 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 6.8MB and 21.1MB, respectively. However, for several SPEC INT2000 benchmarks, such as vortex and bzip2, the checkpointing overhead is large due to their large working set. In these cases, First-Aid leverages the adaptive checkpointing scheme to alleviate the overhead by increasing the checkpoint intervals. As a result, the space overhead per second is kept low. When the checkpoint interval is increased and old checkpoints get discarded, First-Aid maintains the same length of history while keeping less data in memory. The downside is that the recovery time would be longer when the checkpoint interval gets larger.

7.9 Trade-off between Recovery Time and Normal Run Overhead

We have evaluated the trade-off between recovery time and normal run overhead in First-Aid with different checkpoint intervals. Figure 9 shows the throughput and average recov-

	Apache	Squid	CVS	Mutt	Pine	M4	BC	cfrac	espresso	lindsay	p2c
Original heap (MB)	0.806	2.283	0.285	0.345	0.636	15.96	0.059	0.205	0.272	1.822	0.461
First-Aid heap (MB)	0.810	2.357	0.285	0.392	0.980	16.00	0.063	0.396	0.354	1.826	0.715
Overhead	0.49%	3.24%	0.0%	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)	180.4	20.11	83.68	94.91	0.856	30.11	0.346	56.92	108.6	184.9	3.224
First-Aid heap (MB)	180.4	20.66	83.75	94.91	0.856	30.11	0.352	63.05	109.4	184.9	5.251
Overhead	0.0%	2.75%	0.08%	0.0%	0.0%	0.0%	1.89%	10.76%	0.65%	0.0%	62.88%

Table 5: Space overhead incurred by memory allocator plug-in

	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	0.04	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	10.87	0.056	4.566	33.39	16.08	1.585
MB/second	6.852	6.765	7.074	7.035	4.657	6.836	0.28	6.732	7.120	6.945	6.305

Table 6: Space overhead incurred by checkpointing

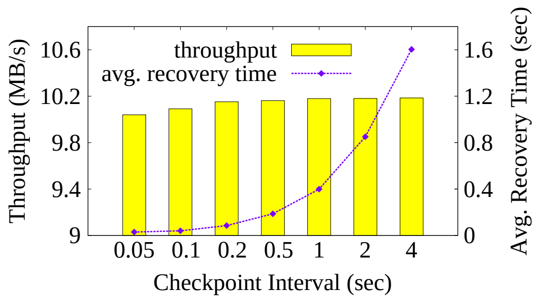


Figure 9: Trade-off between checkpoint frequency and recovery delay for Squid

ery time for Squid with different checkpoint intervals. We can observe that as the checkpoint interval decreases to a certain level, it starts to show noticeable performance drop. In the meantime, as the checkpoint interval decreases, the recovery time becomes smaller. This is because the recovery time largely depends on the time for re-execution. In many cases including Squid, the re-execution time depends on the distance between the failure point and the last checkpoint. Therefore, increasing checkpoint frequency will reduce the average length to replay and thus effectively reduce the average recovery time. For users who consider failures as infrequent events and are not willing to sacrifice normal execution performance, a large checkpoint interval is recommended, especially for memory intensive programs.

8 Related Work

Due to the space limit, this section briefly discusses the related work that is not discussed in previous sections.

Fault tolerance. First-Aid is related to a large body of work on fault tolerance, including restart [11, 32, 6, 7], software rejuvenation [5, 9, 14] and checkpointing-based fault tolerance [25, 24]. Recently, based on virtual machine replication, Remus [8] 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 diagnose and correcting memory errors, which are often deterministically manifested. Furthermore, complementary to Dimmunix [15], 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.

Failure diagnosis. Many studies on failure diagnosis focus on off-site tools, either incurring heavy overhead or relying on extensive human effort. Examples of such tools are delta debugging [21, 37], program slicing [1, 36, 38], interactive debugging [10], and deterministic replay [31, 16]. Unlike these approaches, First-Aid diagnoses memory bugs based on end-user site information, which is unavailable at developers' sites. A recent work, Triage [34], focuses on on-site failure diagnosis. Unlike Triage, First-Aid's diagnostic algorithm aims for quick recovery and thus is more lightweight. In addition to diagnosis, First-Aid generates online patches for avoiding the same memory bug occurrences during subsequent production runs.

Dynamic memory bug detection. Many dynamic memory bug detection tools such as Purify [13] and Valgrind [22] are mainly for in-house testing due to their heavy instrumentation on every memory accesses. First-Aid, as an online tool for different purposes, focus on diagnosing and patching occurring bugs instead of bug detection. Therefore, it uses a lightweight diagnosis algorithm on demand. Furthermore, First-Aid can benefit from other lightweight memory error detection tools, such as AccMon [39], SafeMem [26], etc., by deploying them as failure/error monitors.

Checkpointing and re-execution. Previous work relies on checkpointing and re-execution for many different purposes, including interactive debugging [31, 16], failure recovery [27, 24, 25], failure diagnosis [34]. 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 to common memory bugs, including buffer overflow, uninitialized read, dangling pointer read/write, and double free. By leveraging 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 minimal set of memory objects to tolerate the occurring bugs and prevent future failures caused by the same bugs.

Our evaluation with seven applications shows that First-Aid can successfully diagnose and generate runtime patches for common memory bugs. It provides fast recovery, i.e., 0.084-3.978 seconds recovery time. The results also show that First-Aid is effective in preventing future bug occurrences. Additionally, First-Aid provides detailed on-site bug report that helps developers understand the root cause and manifestation of the occurring bugs. Furthermore, our evaluation with the seven applications, SPEC INT2000, and four allocation intensive benchmarks shows that First-Aid incurs low overhead (0.4 to 11.6% with an average of 3.73%) during normal program execution.

References

[1] H. Agrawal, R. A. DeMillo, and E. H. Spafford. An execution-backtracking approach to debugging. *IEEE Softw.*, 1991.

[2] W. A. Arbaugh, W. L. Fithen, and J. McHugh. Windows of vulnerability: A case study analysis. *Computer*, 2000.

[3] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. In *ASPLOS '00*.

[4] E. D. Berger and B. G. Zorn. Diehard: probabilistic memory safety for unsafe languages. In *PLDI '06*.

[5] A. Bobbio and M. Sereno. Fine grained software rejuvenation models. In *ICPDS '98*.

[6] G. Candea, J. Cutler, A. Fox, R. Doshi, P. Garg, and R. Gowda. Reducing recovery time in a small recursively restartable system. In *DSN '02*.

[7] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot – A technique for cheap recovery. In *OSDI '04*.

[8] 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.

[9] S. Garg, A. Puliafito, M. Telek, and K. S. Trivedi. On the analysis of software rejuvenation policies. In *CA '97*.

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

[11] J. Gray. Why do computers stop and what can be done about it? In *RDS '86*.

[12] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z. Yang. Characterization of linux kernel behavior under errors.

[13] R. Hasting and B. Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *USENIX Winter '92*.

[14] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton. Software rejuvenation: Analysis, module and applications. In *FTC '95*.

[15] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea. Microreboot – A technique for cheap recovery.

[16] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX '05*.

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

[18] 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.

[19] 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.

[20] V. B. Lvin, G. Novark, E. D. Berger, and B. G. Zorn. Archipelago: trading address space for reliability and security. In *ASPLOS '08*.

[21] G. Misherghi and Z. Su. Hdd: hierarchical delta debugging. In *ICSE '06*.

[22] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07*.

[23] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: automatically correcting memory errors with high probability. In *PLDI '07*.

[24] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of zap: a system for migrating computing environments. In *OSDI '02*.

[25] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under Unix.

[26] F. Qin, S. Lu, and Y. Zhou. Safemem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. In *HPCA '05*.

[27] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating Bugs as Allergies – A Safe Method to Survive Software Failure. In *SOSP '05*.

[28] 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 *OSDI '04*.

[29] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building a reactive immune system for software services.

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

[31] S. Srinivasan, C. Andrews, S. Kandula, and Y. Zhou. Flashback: A light-weight extension for rollback and deterministic replay for software debugging. In *USENIX '04*.

[32] M. Sullivan and R. Chillarege. Software defects and their impact on system availability – A study of field failures in operating systems. In *FTC '91*.

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

[34] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: diagnosing production run failures at the user's site. In *SOSP '07*.

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

[36] M. Weiser. Programmers use slices when debugging. *Commun. ACM*, 1982.

[37] A. Zeller. Isolating cause-effect chains from computer programs. In *FSE '02*.

[38] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *ICSE '03*.

[39] 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 *MICRO '04*.