Self-Healing Multitier Architectures Using Cascading Rescue Points

Angeliki Zavou, Georgios Portokalidis, Angelos D. Keromytis
Department of Computer Science
Columbia University, New York, NY, USA
{azavou, porto, angelos}@cs.columbia.edu

ABSTRACT

Software bugs and vulnerabilities cause serious problems to both home users and the Internet infrastructure, limiting the availability of Internet services, causing loss of data, and reducing system integrity. Software self-healing using rescue points (RPs) is a known mechanism for recovering from unforeseen errors. However, applying it on multitier architectures can be problematic because certain actions, like transmitting data over the network, cannot be undone. We propose *cascading rescue points* (CRPs) to address the state inconsistency issues that can arise when using traditional RPs to recover from errors in interconnected applications. With CRPs, when an application executing within a RP transmits data, the remote peer is notified to also perform a checkpoint, so the communicating entities checkpoint in a coordinated, but loosely coupled way. Notifications are also sent when RPs successfully complete execution, and when recovery is initiated, so that the appropriate action is performed by remote parties. We developed a tool that implements CRPs by dynamically instrumenting binaries and transparently injecting notifications in the already established TCP channels between applications. We tested our tool with various applications, including the MySQL and Apache servers, and show that it allows them to successfully recover from errors, while incurring moderate overhead between 4.54% and 71.56%.

Categories and Subject Descriptors

D.4.5 [**Software**]: Operating Systems—*Reliability*

General Terms

Reliability, Security

Keywords

Software self-healing, error recovery, reliable software, multitier applications

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA
Copyright 2012 ACM 978-1-4503-1312-4/12/12 ...$15.00.

1. INTRODUCTION

Software bugs and vulnerabilities cause serious problems to both home users and the Internet infrastructure. Such problems include broad outages [23], integrity violations [26], and data loss [14]. Despite the great combined efforts of both industry [12] and researchers [4, 8] the continuously increasing size and complexity of software makes it extremely difficult to produce error-free software. To mitigate the effects of bugs that can reduce the integrity of systems, a plethora of runtime protection mechanisms have been devised, like stack smashing protection [10], write integrity testing [2], address space layout and code randomization [22, 21, 27]. Nevertheless, while protection mechanisms render certain types of vulnerabilities infeasible or impractical, they do not also offer high availability and reliability, as they frequently resort to terminating applications that behave abnormally to prevent attackers from performing any useful action.

To increase software availability, many mechanisms that aim to recover execution when unhandled errors occur have been proposed [16]. One of these mechanisms is software self-healing based on rescue points [30]. It operates based on the observation that applications already contain code for handling anticipated errors and proposes reusing this code to also handle unexpected errors. Rescue points (RPs) are essentially functions that contain error handling code, which can be exploited to recover from errors occurring within the RP, including the RP routine itself and all called routines. A checkpoint is taken upon entering a RP, and execution is rolled back to that checkpoint when an unhandled error occurs, while concurrently a valid error code is returned by the RP to the application (*i.e.,* through the routine's return value), so that it can gracefully handle the failure.

Applying RP-based self-healing on self-contained functions is straightforward, however there are many functions that have side effects, such as transmitting data to other entities on the network. Applications that are part of multitier architectures, like client-server or three-tier architectures (comprised by presentation, logic, and data tiers), contain many such functions. Introducing RPs in such architectures can be problematic because it can result in inconsistent states between the tiers when a roll back occurs. For example, consider the following. The first tier communicates certain information to the second tier, which then communicates with the third tier, and so on. If an error occurs in the first tier, triggering a rescue point, the application will think that an error, like a communication failure has occurred, while in fact the effects of the transmission have already propagated to other tiers.

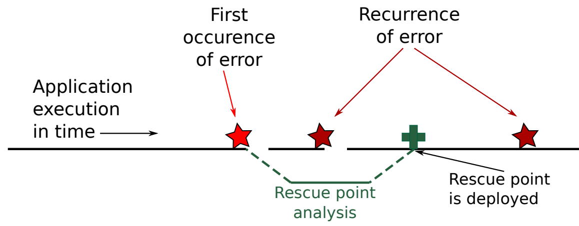


Figure 1: Software self-healing transforms unanticipated errors from fail-stop to fail-once. After an unexpected error first occurs, causing the application to terminate, the data produced during the fault (*e.g.,* a core dump) are used to analyze the fault and produce a remedy in the form of a rescue point. While the application is still "vulnerable" until the (either automatic or manual) offline analysis is completed, after the rescue point is deployed, a recurrence of the fault will be gracefully handled.

We propose *cascading rescue points* (CRPs) for self-healing applications in multitier architectures to address the inconsistency issues introduced by traditional RPs. In our approach, when an application executing within a RP communicates with an application on the next tier, we notify the remote peer to also perform a checkpoint, cascading, in this way, the checkpoint and RP to the lower tiers of the architecture. If a RP successfully completes execution or if it triggers a roll back due to an error occurring, a notification is also send to all the peers that were instructed to checkpoint, so that they also perform the appropriate action.

We have implemented CRPs using the Pin [18] dynamic binary instrumentation framework for x86 Linux, extending our previous work [28] on deploying traditional RPs using Pin. We improve the checkpointing mechanism used by utilizing the *fork()* system call to quickly create copy-on-write copies of an application's image and use filters to mark the individual bytes modified by threads for efficient thread-wide checkpointing. We also intercept system calls to restore the contents of overwritten memory and to transparently inject information in the communication channels between applications of different tiers that run on top of our tool. We use the injected data to implement a protocol for conveying notifications between the various parties. Additionally, we utilize TCP out-of-band data to asynchronously notify remote peers of a successful exit from a RP.

In practice, we envision RPs being employed as a temporary solution for running critical software until a concrete solution, in the form of a dynamic patch or update, is available. Using a dynamic framework like Pin enables us to attach and detach our tool on already running applications without interrupting its operation, applying RPs only for as long as they are required. Combined with a dynamic patching mechanism [7, 11, 19], applications can be run and eventually patched without any interruption.

Distributed checkpointing and recovery has been a popular subject of research [5, 32]. However our work is driven by other goals and differs from previous work in the following ways:

- Our approach is *transparent* and *self-contained*. It does not require that applications are designed with self-healing in mind, nor does it require support from the operating system, and it is applicable on binary-only software
- We do not checkpoint at arbitrary points of execution, but instead checkpointing is driven by rescue points
- We can dynamically engage/disengage software self-healing to apply it only when needed
- Our tool piggybacks the checkpointing protocol on existing communication channels

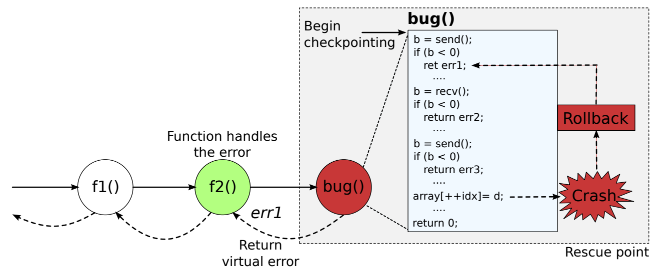


Figure 2: Software self-healing using rescue points. Function *bug()* contains an error which can cause an application crash. If it, or a caller function (*e.g., f1()* and *f2()*), contains error handling code for expected faults, it can be used to handle unexpected errors, *i.e.,* it is a possible *rescue point*. A checkpoint is made upon entering the rescue point, and execution is rolled back when an error occurs. We return a valid error code to allow the application to continue executing (dashed arrows).

We evaluate our approach using popular servers applications, like Apache and MySQL, that suffer from well known vulnerabilities and show that our CRP protocol does not introduce prohibitive overheads. The performance overhead imposed by our approach varies between 4.54% and 71.96% depending on the application. Note that our approach can be ported with moderate effort to operate on other platforms supported by Pin, including Windows and BSD operating systems, and the x86-64 architecture.

This paper is organized as follows: Section 2 contains some background information on the tool we use for developing CRPs, and discusses the limitations of traditional RPs. An overview of cascading rescue points is given in Sec. 3. We describe the implementation of a prototype in Sec. 4, and evaluate its effectiveness and performance in Sec. 5. Related work is discussed in Sec. 6. We conclude in Sec. 7.

2. BACKGROUND

2.1 Software Self-healing Using Rescue Points

The goal of software self-healing is to allow applications to operate normally by healing themselves when unanticipated errors occur. ASSURE [30] was one of the first works to present a practical and automatic approach to software self-healing. Fig. 1 depicts a high level overview of the concept. When an error first occurs, it is analyzed offline to determine its location and the appropriate remedy to be applied that will allow the application to self-heal when it reoccurs.

One of the key ideas of software self-healing is rescue points (RPs). RPs are essentially routines that contain

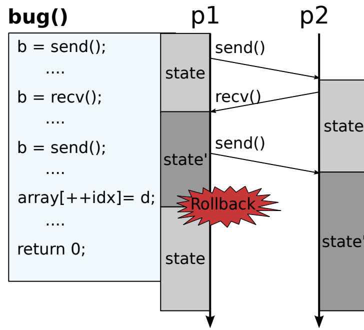


Figure 3: A rescue point deployed on function *bug()* of process $p1$ needs to both send and receive data to and from $p2$. When an error triggers a roll back, $p1$ can end up in an inconsistent state with $p2$. Deploying rescue points in routines that communicate with other parties over the network can be problematic because their effects cannot be reversed.

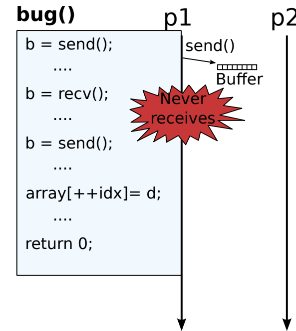


Figure 4: Adopting a naive approach to address the issue in Fig. 3 will not work. For example, buffering the data being send from a rescue point, and only transmitting them after determining that an error did not occur, can break applications. In this case, $p2$ never receives the data that will cause it to respond to $p1$.

error handling code written by the programmer to handle expected error conditions, and directly or indirectly (*e.g.,* through a function call) engulf code containing an unexpected fault. ASSURE proposed the use of existing error handling code to gracefully handle unanticipated faults, virtualizing in this manner error handling, by mapping the larger set of unknown errors that can occur during execution (*e.g.,* invalid memory accesses and attacks) to the smaller set of handled errors (*e.g.,* a system call failing).

2.1.1 Discovering Rescue Points

ASSURE proposed a mechanism for automatically discovering possible RPs and selecting the one that is more likely to patch an observed error. The goal was to identify program functions and returned error codes through fault injection. Consider the function *bug()* in Fig. 2; *bug()* may return *err1* or *err2*, if *send()* or *recv()* fail respectively. This designates the function as a potential RP for errors occurring within the function, or a function that it calls, because it can return a valid error code to *f2()*, allowing it to handle an unanticipated error, such as an out of bounds access of *array* that could cause the application to crash.

The simplest way to detect unknown errors and initiate the rescue point analysis is to intercept the signals (or exceptions in Windows OSs) that are raised when a serious error such as an invalid memory reference occurs. In Linux, such signals include *SIGSEGV* for memory faults, *SIGFPE* for floating point errors like division by zero, *etc.* Software self-healing can be also employed in conjunction with protection mechanisms already incorporated in the application [10, 2, 22], or retrofitted on the binary after it was deployed [27, 15]. For example, ProPolice [10] uses the *abort()* system call, which raises signal *SIGABRT*, when a stack smashing attack is detected.

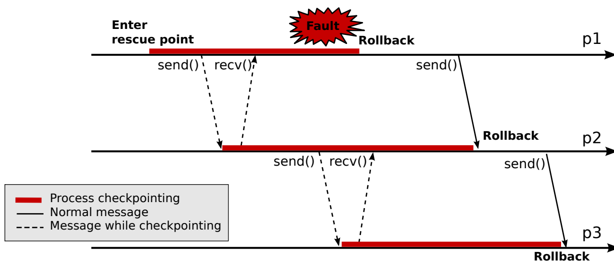
The primary goal of ASSURE was to automate the process of discovering, selecting and deploying an RP, however RPs can be also discovered manually. For example, the operating system can be configured to produce a dump of the memory image of processes crashing due to a memory violation error. This core dump can be manually analyzed by a developer or administrator to determine the location of the error [1] and look for an appropriate RP. While this process is time consuming and requires user intervention, producing and distributing an actual software patch that corrects the error at its source, frequently requires even more time and resources. Security related patches can take as much as two weeks from the date they have been disclosed [3], while less critical faults that only affect the availability of software may take even longer [37].

2.1.2 Rescue Point Deployment

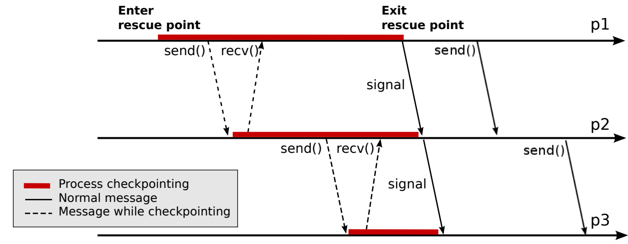
ASSURE relies on process-wide checkpoint/rollback based on Zap [20] to create checkpoints, as well as to rollback to a checkpoint when an error occurs. Because of Zap, the overhead is little, but it is not extremely practical as it requires modifications to the Linux kernel, and cannot be dynamically installed and removed. In previous work [28], we designed and implemented REASSURE a tool that simplified the deployment of RPs. We built upon Intel's Pin dynamic instrumentation framework [18], to create a self-contained mechanism that can dynamically deploy RPs on binary-only software, for as as long as it is required. The use of Pin enables us to attach to an already running application to deploy a RP on demand, while we can also detach from the application to apply a patch at runtime [19]. In this paper, we build on our previous work to enable the application of rescue points on multiparty software systems.

2.2 The Problem: Irreversible Side-effects within Rescue Points

Previous software self-healing approaches cannot apply rescue points on functions that have side effects, such as transmitting data to other entities on the network. Doing so can result in inconsistent states between the communicating parties, as shown in Fig. 3, because the effects of process $p1$ sending a message to $p2$ cannot be undone. The problem with this scenario is that the client's state has been rolled back and the client believes that an error, such as being unable to communicate with the server, occurred. However, data has been exchanged with the server, which is oblivious of the error that occurred in the client. Depending on the nature of the communicating applications this can lead



(a) Fault occurs. The next transmission from *p1* to *p2* will notify the latter to also roll back. *p2* will eventually notify *p3*.



(b) No fault. When *p1* exits the rescue point, it immediately notifies *p2*, which also exits checkpointing and notifies *p3*, and so forth.

Figure 5: Cascading rescue points overview. When process *p2* receives a message from process *p1*, which executes within a rescue point, it also begins checkpointing. Other processes, like *p3*, that receive messages from a checkpointing process also begin checkpointing. This way the original rescue point *cascades* to the communicating processes.

to various problems and can require additional mechanisms, like transactions employed by database (DB) servers, for restoring them to a consistent state. By consistent state, we refer to every party having a correct view of what is the state of their peer. For example, if $p1$ tries to issue a command to $p2$ to switch it to state $state'$ and it fails, $p1$ can still think that $p2$ is in state $state$.

Previous designs simply ignore data exchanges and rely on the protocols implemented by applications to discover and correct such inconsistencies (*e.g.,* they use transactions). Moreover, the problem cannot be trivially addressed by simply delaying the transmission of messages. Fig. 4 depicts an example where the application expects to receive a response to a message send from within a RP. Buffering the transmitted message would break function *bug()*, causing it to fail or wait forever because no data is sent as a response from $p2$.

3. CASCADING RESCUE POINTS

3.1 Overview

Self-healing using cascading rescue points aims to enable applications participating in multitier architectures to self-heal without facing the problems presented in Sec. 2.2. To achieve this, we introduce a protocol, which is transparently implemented over the application's TCP connections. The protocol encapsulates application data, and serves the sole purpose of allowing us to convey signals between applications of the architecture.

Consider process *p1* shown in Fig. 5. All of its communications with other processes in the architecture are modified to implement our CRP protocol. When *p1* executes within a RP, it is essentially checkpointing, indicated by the highlighted areas in Fig. 5. This means that a fault will cause all the changes performed within the RP to be undone, and we will simulate the return of an error code from the RP routine. When *p1* transmits data to another process (while in a RP), we use our protocol to instruct the remote peer to also begin checkpointing. Later on, if an errors occurs in *p1*, the RP will recover the process. Since we piggyback our protocol on existing communications, *p1* does not immediately notify *p2* that it discarded the state generated in the RP, and *p2* will continue checkpointing until the next message is received by *p1*. Figure 5(a) depicts this process, which is propagating in time to the other processes. If *p2* sends any data to another process (*e.g., p3*), that process also begins checkpointing, and so forth (see Sec 3.2 for limitations).

If no fault occurs, the process is almost entirely the same, and it is shown in Fig. 5(b). Like before, the RP of *p1* causes the checkpointing to cascade to *p2* and *p3*. However, in this case no error occurs and *p1* successfully exits the RP. When this happens, we immediately notify the other processes by utilizing TCP's out-of-band (OOB) data [33]. OOB data are not part of the regular data stream, so we can signal *p2* and *p3* without corrupting the application data stream and without requiring data to be read by the processes. Instead, we can rely on the OS to notify the process when such a packet is received (*e.g.,* by raising a signal or exception). TCP does not support multiple OOB signals on a particular stream (*i.e.,* a second OOB would overwrite the first and would be the only one to raise a signal on the receiver). For this reason, we can only use it to signal successful exits from RPs. Our approach is an optimistic one, assuming that errors will be rare.

We are planning to explore scenarios with more frequent errors, *e.g.,* when the application is under attack. One approach we are looking into is to be able to allow our protocol's notification system to adapt depending on the conditions of the involved applications. For example, for processes where errors are too frequent, the CRP protocol could switch the notification methods used for notifying the communicating processes for the events of successful or not exit from an RP. The OOB signal used in the current CRP protocol as to notify the other processes for a successful exit from a RP, could be used for the opposite purpose, *i.e.,* to signal the rest of the processes to rollback to a previous state. Consequently, the next transmission of data would have to piggyback the signal for successful exit from a RP. This new approach requires exploring other important factors first, such as the *necessary* conditions under which the switch of the notification methods would make the CRP protocol more effective.

3.2 Limitations: Overlapping Checkpoints

Our approach enables multiple processes in a multiparty architecture to checkpoint in a loosely coordinated way. However, our goal is *not* to provide another algorithm for coordinated checkpoint/restart for an unstructured distributed or peer-to-peer system. *Our approach is a good fit for archi-*

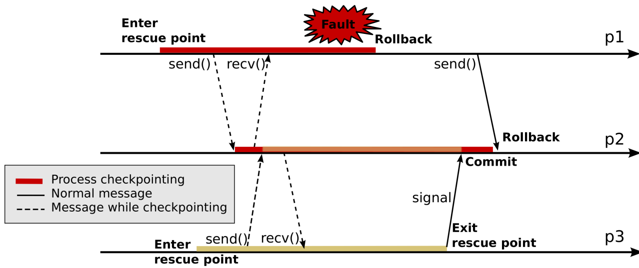


Figure 6: Overlapping checkpoints. Processes *p1* and *p3*, while both in RPs, try to *cascade* their RP to the receiving process *p2*. *p1* requests a rollback, while *p3* commits. The changes made by the latter are lost.

tectures that have some innate hierarchy, like a three-tier or client-server architecture. As it should be already obvious from Fig. 5, receiving a checkpoint request from a process, while already checkpointing due to the request of another process or a RP, has no effect. In this sense, our approach is best-effort and does not offer strong guarantees on establishing a globally consistent state between all processes.

Figure 6 depicts such a case of *overlapping* checkpoints. Two processes *p1* and *p3* enter RPs independently and both transmit data to process *p2*. According to the CRP protocol, *p2* will start the checkpointing, the moment it receives the data from a process in a RP, in this case *p1*. When a process is already checkpointing, it will ignore signals to start checkpointing (*e.g.,* the signal sent from *p3*). We should emphasize that even though we do not handle overlapping checkpoints, their occurrence is not catastrophic to CRPs. However, we do have to rely on application logic to identify and correct errors, which was the case before introducing CRPs.

4. IMPLEMENTATION

4.1 Self-contained Rescue Point Deployment

In previous work [28], we implemented REASSURE a tool for deploying rescue points on binaries in an on-demand fashion and without the need for source code. We built our tool using Pin [18], a framework that enables the development of tools that can at runtime augment or modify a binary's execution at the instruction level through an extensive API. The target binary executes on top of Pin's virtual machine (VM), which essentially consists of a just-in-time (JIT) compiler that combines the binary's original code with the code inserted by the tool and places the produced code blocks into a code cache, where the application executes from. Pin facilitates the instrumentation of binaries by enabling developers to inspect and modify a program's instructions, as well as intercept system calls and signals. It is actively developed and supports multiple hardware architectures and OSs. Pintools can be applied on binaries by either launching them through Pin or by attaching on already running binaries. The latter behavior is highly desirable, as it allows us to deploy RPs without interrupting an already executing application. Our tool currently runs on x86 Linux systems, however there are no significant challenges in porting it to other OSs and architectures supported by Pin.

Table 1: Example of rescue points covering known bugs on popular applications (also see Tab. 2).

Application	Function name/Return value
MySQL v5.0.67	Item_func_set_user_var::update()/1
Apache v1.3.24	ap_bread()/-1

RPs can be installed on any callable application function. Table 1 lists RPs for a set of popular applications, including the error codes that should be returned when an unexpected error occurs. When a RP function is first entered by an application thread, we use Pin's API to insert code that will switch the thread into *checkpointing mode* and save CPU state. Instructions that can be used to exit the RP, such as the function return *RET* instruction on the x86 instruction-set, are also instrumented to return the thread exiting the rescue point to *normal mode* and to discard the previously saved state.

When executing in checkpointing mode, our tool instruments all memory write instructions and logs the overwritten memory contents into a dynamically expanding array, the *write log*. Pin provides us with facilities so that only the instructions being reached from within a RP are actually instrumented in this fashion. This way individual threads of an application can enter RPs and checkpoint individually (assuming that no shared state is updated).

To identify errors our tool relies on signals. In particular, we intercept the following signals on Linux: *SIGSEGV*, *SIGILL*, *SIGABRT*, *SIGFPE*, *SIGPIPE*.[1] When a thread executing within a RP receives one of these signals, we initiate the recovery process. The recovery process restores memory contents and execution returns to the callee, also returning the error code specified by the RP. In x86 architectures, function return values are usually returned in the *EAX* register.

Concurrency.

Checkpointing is *thread-specific*, that is multiple threads can enter a RP at the same time and each thread can roll back independently. However, if a RP is processing data shared by multiple threads, or if the error that causes the application crash corrupts data used by other threads, since they are all executing in the same address space, this type of checkpointing can leave the process in an inconsistent state after a roll back. We address such issues by introducing *blocking* RPs that block other threads for their duration. We can block threads by conditionally instrumenting every block of instructions, so that when a certain flag is asserted execution is blocked. This *always-on blocking* approach is appropriate for applications that expect a very high rate of faults. Alternatively, we utilize signals to asynchronously interrupt the remaining threads of a process and block them until execution has left the RP. This *on-demand blocking* mode generally incurs less overhead, since no additional instrumentation needs to be added for non-RP code.

[1]Note that other OSs have similar mechanisms to synchronously notify applications of such errors. For example, Windows OSs use exceptions.

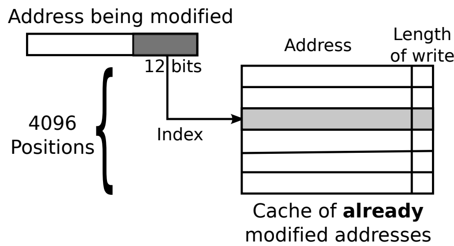


Figure 7: A small associative cache is used to quickly check if a memory location has been already modified. The cache is indexed using the lower 12-bits of an address. Each slot stores the address and number of bytes already modified.The original and cached addresses, as well as the length of the write, are compared to determine a cache hit. A cache miss updates the appropriate slot.

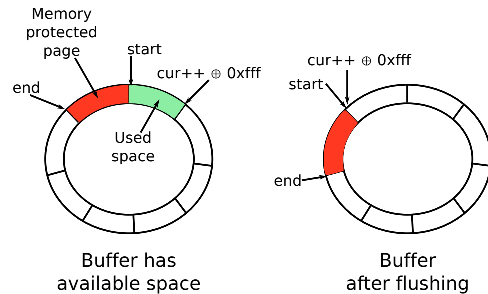


Figure 8: A circular buffer can be used to store the memory locations modified by a thread during checkpointing. When full, a page protection fault is generated. We capture the fault to flush the buffer contents to disk, and setup a new protected page. The failed insertion can then simply resume to be completed.

4.2 Efficient Thread Checkpointing

Checkpointing threads individually is beneficial to software self-healing. If an error occurs while a thread is executing within a RP, we only need to roll back the state of the thread that suffered the fault, while the remaining threads can execute unhindered. However, using a *writes log* to store the overwritten memory values does not scale for certain applications and can lead to excessive memory overheads. *We address these issues by extending our tool in three ways*:

1. We introduce a small associative cache (shown in Fig. 7) to quickly check if a memory location has been already modified
2. We use the *fork()* system call to create a copy-on-write image of the process's address space and employ a *filter* to mark the memory locations updated by the checkpointing thread. Two types filters are currently supported: a statically allocated *bitmap* where each bit corresponds to a four-byte memory area, and a *circular buffer* that stores the modified addresses of memory
3. For the circular buffer, we utilize page protection and intercept OS page-faults to identify when the buffer is full, and write its contents to disk to avoid excessive memory usage

The cache allows us to minimize the number of updates performed in the writes log and the filter. This has the effect of reducing the amount of memory required for checkpointing, as addresses repeatedly written (*e.g.,* stack variables) are only updated once. We use the lower address bits to index the cache to exploit locality in memory accesses.

Checkpointing using *fork()* is not a novel concept [25]. *fork()* is used to cheaply obtain a copy of the memory contents of the entire process. Memory pages are initially shared between the processes, while the OS creates copies of the pages when they are modified. When a RP is entered, a *checkpointing process* is forked. This newly forked process uses a shared memory segment to communicate with the original one and utilizes a semaphore for proper synchronization. It initially sleeps waiting for events from its parent. If the RP successfully exits (no error occurs), the checkpointing process is signaled to exit. Otherwise, it accesses the filter to obtain the addresses that were modified in the original, and uses a pipe (used in UNIX systems for unidirectional inter-process communication) to return the original memory contents to the main process.

When using a bitmap filter, the overhead is low as the bitmap is allocated once on RP entry and updating it is efficient. Using one bit per four memory bytes means that we could erroneously restore a byte that was not overwritten by the current thread. This could lead in memory corruption, if the particular byte was concurrently updated by another thread. Most compilers tend to use four or even eight byte alignment for variables, so in practice such cases should rarely (if ever) occur. Note that we do not address cases that the application itself erroneously implements synchronization primitives, leading to inconsistent updates of shared state.

Using the circular buffer for storing modified addresses does not suffer from such limitations. Moreover, it uses less memory, making it a good fit for highly parallel processes that may have many active RPs concurrently, and it supports 64-bit systems (64-bit address spaces are too large to be covered by a statically allocated bitmap).

Our circular buffer implementation focuses on very fast updates. This is achieved by first aligning its size to a power of two. This allows us to simply increase the cursor (*i.e.,* the index pointing to the first available slot) after inserting data in the buffer, and use a cheap bit masking operation to down cast it to the size of the buffer. Second, we use page protection to signal buffer fullness instead of checking the number of available bytes on each update. This is accomplished by memory protecting the last page (usually 4KB) of the buffer. When an update spills into the protected region, we flush the buffer to disk, remove the page's protection, update the buffer header, and protect the page that is currently last, as depicted in Fig. 8.

Reverting System Call Effects.

Process memory is not only modified by write instructions, but it can be also written by the kernel during a system call. We extended our tool to intercept system calls and mark the memory locations modified by them in the filter holding the altered memory locations. For this purpose, we define a static array for storing system call related metadata, spec-

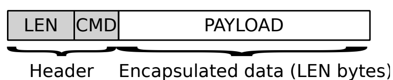


Figure 9: The cascading rescue points protocol encapsulates user data using a small header prepended to every data write made by the user.

ifying the size of their arguments and whether they return data. For example, the *read(int fd, void *buf, size_t count)* system call writes data in the pointer specified by its second argument. The amount of written data depends on its return value. More elaborate calls like *socketcall()* are handled by defining a call back that de-multiplexes the various network calls implemented by it.

4.3 Cascading Rescue Point Protocol

4.3.1 I/O Interception

The cascading rescue point protocol is used to communicate events between peers exchanging data over TCP sockets. The protocol is implemented transparently over the sockets used by the application. This is done by intercepting system calls used with TCP sockets. We can classify these system calls into two groups. The first group consists of calls handling socket creation and termination, and the second group is dealing with data transmission and reception. We intercept system calls *socket()*, *close()*, *shutdown()*, *connect()*, *accept()*, *socketpair()*, and the *dup()* family of calls to track the state of descriptors used by the application (*i.e.,* distinguish TCP socket descriptors from others, like files). For this reason, we maintain a global array to store information on active descriptors, like their type and protocol state data. We also intercept the *read()*, *write()*, *recv()*, and *send()* family of system calls that are used to transmit and receive data from sockets to implement our protocol.

The protocol consists of variable length messages that encapsulate user data as shown in Fig. 9. In particular, we use a small header that comprises of a 4-byte field specifying the length of user data, and a single-byte *CMD* field used to communicate events to remote peers.

The header is inserted into existing TCP streams using Pin to replace system calls used to write data, like *write()* and *send()*, with *writev()* which allows us to transmit data from multiple buffers by performing a single call. This minimizes the number of operations (data copies and system calls) required to transparently inject the header into the stream. If the message cannot be written in its entirety, for instance because non-blocking I/O is performed and the kernel buffers are full, we keep trying until we are successful.

To extract the header from the stream, the reverse procedure is followed. Initially, we replace calls used to receive data with *readv()* to read into multiple buffers. If necessary, we repeat the process until the whole header is received. User data is placed directly in the buffer supplied by the application. However, we can read into the next message, which will be placed into the application's buffer. When this happens, we move the data belonging to subsequent messages into a buffer associated with the socket descriptor. Consequent reads will read data from this buffer instead of performing a system call. Reading one message at a time may be suboptimal performance-wise, but allows us to pair read system calls with particular events received on a socket (*e.g.,* a request to begin checkpointing), which is necessary for rolling back.

Table 2: Applications and benchmarks used for the evaluation of CRP. All of applications contain exploitable bugs as described by their *common vulnerability and exposure* (CVE) id.

Application	Bug type	Benchmark
MySQL v5.0.67	Input validation (CVE-2009-4019)	MySQL's *test-select*
Apache v1.3.24	Memory corruption (CVE-2002-0392)	Apache's *ab* utility

4.3.2 Protocol Commands

The *CMD* field in the protocol is used to inform remote peers of changes in the state of the running thread. For instance, when data are written to a socket while in a rescue point, *CMD* changes to indicate that the destination should also begin checkpointing, the socket is marked as having been signaled to checkpoint, and is placed in a list containing other similar sockets (*fd_checkpointed*). If an error occurs and the thread rolls back, sockets in *fd_checkpointed* are marked accordingly, so that the next write will convey the status change. *If the next write occurs within a RP, the fact is also passed to the remote process, so that it first rolls back memory changes and then enters a new checkpoint.*

On the receiving end, if a thread receives a command to checkpoint, it begins checkpointing similarly to entering a RP. The socket descriptor number where the command was received is saved, so that a consequent request to roll back is only honored, if it was received on the same socket. On rollback, execution resumes right before the system call that caused the thread to checkpoint. Note that receiving requests to begin checkpointing from other sockets, while already checkpointing or executing in a RP are ignored (discussed in Sec. 3.2).

Checkpoint Commits Through Out-of-band Signaling.

To notify remote peers of a successful exit from a RP, we utilize out-of-band (OOB) signaling, as provided by the TCP protocol and the OS. In particular, we make use of TCP's OOB data to notify a remote application that it should also commit changes performed within a checkpoint. We send OOB data by using the *send()* system call and supplying the *MSG_OOB* flag for every descriptor in *fd_checkpointed*.

On the receiver, the reception of an OOB signal by the OS, causes the signal *SIGURG* to be delivered to the thread, which previously took ownership of the socket descriptor that triggered the checkpointing by calling *fcntl()*.[2] The signal is intercepted, and execution is switched from checkpointing to normal execution. If a RP is entered very frequently, multiple OOB signals can be transmitted in succession. On account of TCP's limitations, only a single OOB byte can be pending at any time, so previous OOB signals are essentially overwritten. This does not affect the correct operation of our system, but unfortunately implies that we

[2]In Linux a thread can take ownership of a descriptor, causing the OS to deliver all descriptor related asynchronous events to the specific thread, instead of a randomly selected thread of the process.

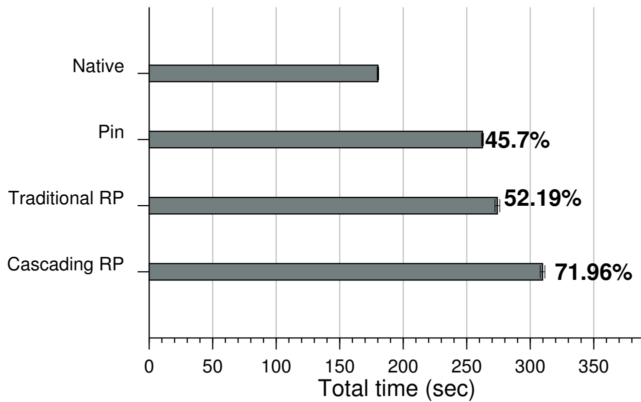


Figure 10: MySQL performance

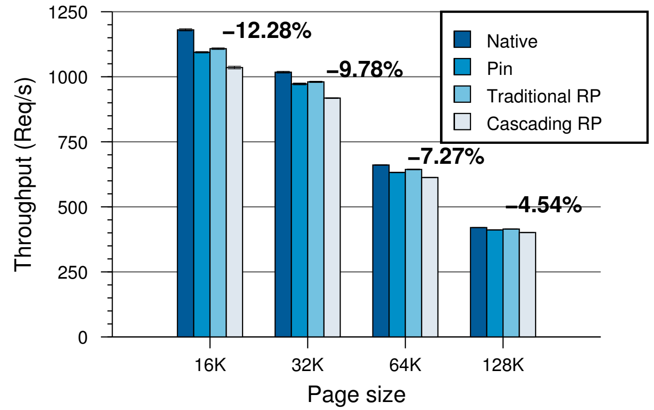


Figure 11: Apache performance

cannot also use OOB signaling to notify remote peers of roll backs.

5. EVALUATION

We evaluated our implementation to establish its effectiveness and performance. First, we validated the effectiveness of CRPs in addressing state inconsistency issues that arise when using RPs to recover from errors in interconnected client-server applications. Second, we evaluated the performance overhead imposed by CRP with real server applications. In both cases, we employed existing benchmarks and tools to generate workloads. Table 2 lists the applications and benchmarks used during the evaluation. We conducted the experiments presented in this section on two DELL Precision T5500 workstations with dual 4-core Xeon CPUs and 24GB of RAM, one running Linux 2.6 and acting as a server and the other one running Linux 3.0 and acting as the client.

5.1 Effectiveness

We used our tool to deploy RPs on known bugs in the applications listed in Table 2, while concurrently running the corresponding benchmarks from the client-side. When RPs are not employed, the applications terminate and the benchmarks are interrupted in all cases. In contrast, when using RPs the applications recover from the error and the benchmarks concluded successfully.

We also used our own artificial client-server applications, that employed our mechanism to *cascade* a RP, which engulfed the exchange of messages between the peers. We manually injected faults in the client and observed that both peers did not crash, but instead they managed to revert to consistent states (*i.e.,* the state they had before entering the RP).

5.2 Performance

For each application in Table 2, we performed the corresponding benchmark, first with the application executing natively, second running under the Pin DBI framework, then employing traditional RPs, and finally with CRPs. This allows us to quantify the overhead imposed by CRP compared with native execution and execution under Pin, as well as the relative overhead compared with the baseline of a self-healing tool with traditional RPs. In the tests described in this section, we did not inject any requests that would trigger the bugs each application suffers from, nevertheless the relevant RPs to the known bugs were installed and in the last case the CRP mechanism was employed.

Figure 10 shows the results obtained after running 10 iterations of MySQL's *test-select* benchmark test over an 1Gb/s network link. The y-axis lists the four different server configurations tested, which from top to bottom are: native execution, execution over Pin, execution with our self-healing tool and traditional RPs, and finally execution with CRPs. The x-axis shows the average time (in seconds) needed to complete the benchmark, while the errors bars represent standard deviation. Note that the standard deviation for is insignificant and thus not visible. We observe that the benchmark takes on average 71.96% more time to complete when running the server with a CRP, while a significant part of the overhead is because of Pin (under Pin the test takes 45.7% more time). In particular, the high overhead is attributed to the significantly larger size of MySQL's code consisting of many indirect control transfers (*e.g.,* callback functions and frequent function return), which exerts pressure on Pin's JIT compiler and code cache.

Figure 11 depicts the results obtained after performing 10 iterations of Apache's *ab* benchmark utility over an 1Gb/s network link. The y-axis displays the average throughput in requests per second as reported by *ab*, and the error bars represent the standard deviation. We performed the experiments requesting files of size 16K, 32K, 64K and 128K from the web server (as listed in the x-axis), and we repeated each test with the corresponding server running: natively, over Pin, with the traditional RPs, and with CRPs. Apache performs on average 8.46% slower when using CRPs, and Pin seems to be responsible for the biggest part of this overhead. Note that this difference drops as the size of the requested file increases. This is due to the workload becoming more I/O intensive (*i.e.,* more data need to be transferred per request) and the number of requests arriving at the server shrinks.

6. RELATED WORK

6.1 Software Self-healing

Software self-healing using RPs was first proposed in ASSURE [30]. RPs are automatically identified and selected by using kernel-level checkpoint-restart, powered by Zap [20] and input fuzzing. RPs were deployed using a modified OS

featuring the Zap virtual execution environment. In contrast to our approach, they require modifications to the OS for deploying RPs and system calls are ignored. Nonetheless, the RP identification component of ASSURE can be used in combination with our work.

Selective transactional emulation (STEM) [31] is a speculative recovery technique that also identifies the function where an error occurs, and it could also be used to assist in identifying RPs. STEM requires source code to analyze errors, and does not support multithreaded applications. Similarly, failure-oblivious computing [29] is another technique that uses a modified compiler to inject code to detect invalid memory writes and correct them by virtually extending the target buffer. This approach is more robust against buffer overflow errors, but comes at significant performance overhead, ranging from 80% up to 500% for a variety of different applications. Moreover, it requires recompilation of the target applications, and it does not handle failures due to other bugs, such as null pointer dereferences.

Instead of attempting recovery, rebooting techniques [34, 13, 9] focus on restoring a system to a clean state. Program restart is a significantly lengthier process than recovery, resulting in substantial application down-time, while data loss is also more frequent. Micro-rebooting aims to accelerate rebooting by only restarting parts of the system, but requires a complete rewrite of applications to compartmentalize failures. These techniques cannot recover from deterministic bugs, and restart all execution threads of a given application. Checkpoint-restart techniques [6, 17] are used in a similar way to rebooting, but restart from a checkpoint. While down time is reduced, they still do not handle deterministic bugs, or bugs maliciously triggered by an attacker (*e.g.,* a DoS attack).

Checkpoint-restart has been also combined with running multiple versions of programs [6]. This approach is based on the assumption that not all versions will be prone to the same error, and it introduces prohibitive costs for most applications, as multiple versions need to be maintained and run concurrently.

Other works have focused on reducing the time from bug discovery to patch generation by automatically generating and applying patches [24, 19, 35]. Unfortunately, automatically applying patches is not very practical, due to the possibility that additional errors are introduced during the patching, or that the patch alters program behavior.

6.2 Coordinated Checkpointing

Our work is also loosely related with work in the area of coordinated checkpointing for distributed systems. Bhargave *et al.* [5] present a checkpoint algorithm for distributed systems, where each process takes checkpoints independently. To recover, a two-phase rollback algorithm is invoked to determine the processes that need to rollback, and the checkpoint they need to rollback to. This is an optimistic algorithm in the sense that it performs well, when errors are infrequent. Independent checkpoint algorithms have the benefit that no coordination between the members of a distributed system is required, but may suffer from the "*domino effect*' [36]'. The "domino effect" occurs when two or more members of the system keep rolling back to previously taken checkpoints in an attempt to reach a globally consistent state, leading to unnecessary delays in the completion of an action.

Sistla *et al.* [32] propose various algorithms based on the asynchronous message logging of incoming messages from individual members of a distributed system. Similarly to our approach, they piggyback tags in the exchanged messages, which are later used to determine the point to roll back and the messages that need to be replayed. In our proposal, we only interleave signaling data in the communications, while we also utilize existing OOB signaling mechanisms provided by TCP and the OS.

7. CONCLUSION

We introduced cascading rescue points, a new mechanism for performing software self-healing on multitier architectures. Our approach enables communicating applications to checkpoint in a loosely coordinated way, so that recovery does not lead to inconsistent states between applications. We intercept existing connections and encapsulate application data using our CRP protocol, which we use to notify remote peers to rollback. We also exploit TCP's OOB signaling to quickly signal peers to stop checkpointing when no faults occur. We implemented a prototype tool that can apply CRPs on binary-only software and evaluate it using the Apache and MySQL servers. We show that it successfully allows them to recover from otherwise fatal errors. In the applications tested, the performance overhead introduced by our approach ranges between 4.54% and 71.96%.

8. REFERENCES

[1] H. Agrawal, R. A. Demillo, and E. H. Spafford. Debugging with dynamic slicing and backtracking. *Software Practice and Experience*, 23:589–616, 1993.

[2] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *Proc. of the Symposium on Security and Privacy*, pages 263–277, May 2008.

[3] A. Arora, R. Krishnan, R. Telang, and Y. Yang. An empirical analysis of software vendors' patch release behavior: Impact of vulnerability disclosure. *Information Systems Research*, 21(1):115–132, 2010.

[4] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53:66–75, February 2010.

[5] B. Bhargava and S. Lian. Independent checkpointing and concurrent rollback for recovery in distributed systems-an optimistic approach. In *Proc. of the 7th Symposium on Reliable Distributed Systems*, pages 3–12, October 1998.

[6] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. In *Proc. of the* 15th *ACM symposium on Operating systems principles (SOSP)*, pages 1–11, 1995.

[7] B. Buck and J. K. Hollingsworth. An api for runtime code patching. *Int. J. High Perform. Comput. Appl.*, 14:317–329, November 2000.

[8] C. Cadar, D. Dunbar, and D. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. of the 8th OSDI*, pages 209–224, 2008.

[9] G. Candea and A. Fox. Crash-only software. In *Proc. of the* 9th *Workshop on Hot Topics in Operating*

Systems (HotOS IX), May 2003.

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

[11] M. Hicks and S. Nettles. Dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 27:1049–1096, November 2005.

[12] M. Howard. A look inside the security development lifecycle at microsoft. MSDN Magazine – http://msdn.microsoft.com/en-us/magazine/cc163705.aspx, November 2005.

[13] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton. Software rejuvenation: Analysis, module and applications. In *Proc. of the* 25th *International Symposium on Fault-Tolerant Computing (FTCS)*, page 381, 1995.

[14] InformationWeek. Windows home server bug could lead to data loss. http://informationweek.com/news/205205974, December 2007.

[15] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. In *Proc. of the* 8th *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, March 2012.

[16] A. D. Keromytis. Characterizing self-healing software systems. In *Proc. of the* 4th *MMM-ACNS*, September 2007.

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

[18] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. of the 2005 PLDI*, pages 190–200, June 2005.

[19] K. Makris and K. D. Ryu. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *Proc. of the* 2nd *EuroSys*, pages 327–340, March 2007.

[20] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of Zap: a system for migrating computing environments. In *Proc. of the* 5th *OSDI*, pages 361–376, December 2002.

[21] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the* 33rd *IEEE Symposium on Security & Privacy (S&P)*, 2012.

[22] PaX Project. Address space layout randomization, Mar 2003. http://pageexec.virtualave.net/docs/aslr.txt.

[23] PCWorld. Amazon EC2 outage shows risks of cloud. http://www.pcworld.com/businesscenter/article/226199/amazon_ec2_outage_shows_risks_of_cloud.html, April 2011.

[24] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *Proc. of the ACM SIGOPS* 22nd *symposium on Operating systems principles*, pages 87–102, 2009.

[25] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: transparent checkpointing under unix. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, pages 18–18, Berkeley, CA, USA, 1995. USENIX Association.

[26] P. Porras, H. Saidi, and V. Yegneswaran. Conficker C analysis. Technical report, SRI International, 2009.

[27] G. Portokalidis and A. D. Keromytis. Fast and practical instruction-set randomization for commodity systems. In *Proc. of the 2010 Annual Computer Security Applications Conference (ACSAC)*, December 2010.

[28] G. Portokalidis and A. D. Keromytis. REASSURE: A self-contained mechanism for healing software using rescue points. In *Proc. of the* 6th *International Workshop in Security (IWSEC)*, pages 16–32, November 2011.

[29] M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu, and J. W Beebee. Enhancing server availability and security through failure-oblivious computing. In *Proc. of the* 6th *OSDI*, December 2004.

[30] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis. ASSURE: automatic software self-healing using rescue points. In *Proc. of the* 14th *ASPLOS*, pages 37–48, 2009.

[31] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building a reactive immune system for software services. In *Proc. of the 2005 USENIX ATC*, April 2005.

[32] A. P. Sistla and J. L. Welch. Efficient distributed recovery using message logging. In *Proc. of the* 8th *annual ACM Symposium on Principles of distributed computing (PODC)*, pages 223–238, 1989.

[33] W. R. Stevens, B. Fenner, and A. M. Rudoff. Chapter 24. Out-of-Band Data. In *UNIX Network Programming Volume 1, Third Edition: The Sockets Networking API*. Addison Wesley, 2003.

[34] M. Sullivan and R. Chillarege. Software defects and their impact on system availability - A study of field failures in operating systems. In *Digest of Papers.,* 21st *International Symposium on Fault Tolerant Computing (FTCS-21)*, pages 2–9, 1991.

[35] M. Susskraut and C. Fetzer. Automatically finding and patching bad error handling. In *Proc. of the Sixth European Dependable Computing Conference*, pages 13–22, 2006.

[36] K. Venkatesh, T. Radhakrishnan, and H. Li. Optimal checkpointing and local recording for domino-free rollback recovery. *Inf. Process. Lett.*, 25:295–304, July 1987.

[37] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller. How long will it take to fix this bug? In *Proc. of the* 4th *International Workshop on Mining Software Repositories (MSR)*, 2007.