Treating Bugs As Allergies: A Safe Method for Surviving Software Failures

Feng Qin, Joseph Tucek and Yuanyuan Zhou
Department of Computer Science, University of Illinois at Urbana-Champaign
{fengqin, tucek, yyzhou}@cs.uiuc.edu

ABSTRACT

Many applications demand availability. Unfortunately, software failures greatly reduce system availability. Previous approaches for surviving software failures suffer from several limitations, including requiring application restructuring, failing to address deterministic software bugs, unsafely speculating on program execution, and requiring a long recovery time.

This paper proposes an innovative, *safe* technique, called Rx, that can quickly recover programs from many types of common software bugs, both deterministic and non-deterministic. Our idea, inspired by allergy treatment in real life, is to rollback the program to a recent checkpoint upon a software failure, and then to reexecute the program in a *modified* environment. We base this idea on the observation that many bugs are correlated with the execution environment, and therefore can be avoided by removing the "allergen" from the environment. Rx requires few to no modifications to applications and provides programmers with additional feedback for bug diagnosis.

1 Introduction

1.1 Motivation

Many applications, especially critical ones such as process control or on-line transaction monitoring, require high availability [14]. For server applications, downtime leads to lost productivity and lost business. According to a report by Gartner Group [22], the average cost of an hour of downtime for a financial company exceeds six million US dollars. With the tremendous growth of e-commerce, almost every kind of organization increasingly depends on highly available systems.

Unfortunately, software failures greatly reduce system availability. A recent study showed that software failures account for up to 40% of system failures [16]. Among them, memory-related bugs and concurrency bugs are common and severe software defects, causing more than 60% of system vulnerabilities [9]. For this reason, software companies invest enormous effort and resources on software testing and bug detection prior to releasing software. However, software failures still occur during production runs since some bugs will inevitably slip through even the strictest testing. Therefore, to achieve higher system availability, mechanisms must be devised to allow systems to survive the effects of uneliminated software bugs to the largest extent possible.

Previous work on surviving software failures can be classified into four categories. The first category encompasses various flavors of rebooting (restarting) techniques, including whole program rebooting [14, 25], micro-rebooting of partial system components [7, 6, 8], and software rejuvenation [15, 13, 4]. Since many of these techniques were originally designed to handle *hardware* failures, most of them are ill-suited for surviving software failures. For example, they cannot deal with deterministic software bugs, a major cause of software failures [10], because these bugs will still occur even after rebooting. Another important limitation of these methods is service unavailability while restarting, which can take up to several seconds [26]. For servers that buffer significant amounts of state in main memory (e.g. data caches), it requires a long period to warm up to full service capacity [5, 27]. Micro-rebooting [8] addresses this problem to some extent by only rebooting the failed components. However, it requires legacy software to be reconstructed in a loosely-coupled fashion.

The second category includes general checkpointing and recovery. The most straightforward method in this category is to checkpoint, rollback upon failures, and then reexecute either on the same machine [12, 19] or on a different machine designated as the "backup server" (either active or passive) [14, 3, 5, 27]. Similar to whole program rebooting, these techniques were also proposed to deal with hardware failures, and therefore suffer from the same limitations in addressing software failures. Progressive retry [28] is an interesting improvement over these works. It reorders messages to increase the degree of non-determinism. While this work proposes a promising direction, it limits the technique to message reordering. As a result, it cannot handle bugs unrelated to message order. For example, if a server receives a malicious request that exploits a buffer overflow bug, simply reordering messages will not solve the problem. The most aggressive approaches in this category include recovery blocks [18] and n-version programming [2, 1], both of which rely on different implementation versions upon failure. These approaches may survive deterministic bugs under the assumption that different versions fail independently. But they are too expensive to be adopted

by software companies because they double the software development costs and efforts.

The third category comprises application-specific recovery mechanisms, such as the multi-process model (MPM), exception handling, etc. Some multi-processed applications, such as the multi-processed version of the Apache HTTP server and the CVS server, can simply kill a failed process and start a new one to handle a failed request. While simple and capable of surviving certain software failures, this technique has several limitations. First, if the bug is deterministic, the new process will most likely fail again at the same place given the same request (e.g. a malicious request). Second, if a shared data structure is corrupted, simply killing the failed process and restarting a new one will not restore the shared data to a consistent state, therefore potentially causing subsequent failures in other processes. Other application-specific recovery mechanisms require software to be failure-aware, which adversely affects programming difficulty and code readability.

The fourth category includes several recent non-conventional proposals such as failure-oblivious computing [20, 21] and the reactive immune system [23]. Failure-oblivious computing proposes to deal with buffer overflows by providing *artificial* values for out-of-bound reads, while the reactive immune system returns a *speculative* error code for functions that suffer software failures (e.g. crashes). While these approaches are inspiring and may work for certain types of applications or certain types of bugs, they are *unsafe* to use for correctness-critical applications (e.g. on-line banking systems) because they "speculate" on programmers' intentions, which can lead to program misbehavior. The problem becomes even more severe and harder to detect if the speculative "fix" introduces a silent error that does not manifest itself immediately. Such problems, if they occur, are very hard for programmers to diagnose since the application's execution has been forcefully and silently perturbed by those speculative "fixes".

Besides the above individual limitations, existing work provides insufficient feedback to developers for debugging. For example, the information provided to developers may include only a core dump, several checkpoints, and an event log for the deterministic replay of a few seconds of recent execution. To save debugging effort, it is desirable if the run-time system can provide information regarding the bug type, under what conditions the bug is triggered, and how it can be avoided. Such diagnostic information can guide programmers during their debugging process and thereby enhance efficiency.

1.2 Our Contributions

In this paper, we propose a *safe* technique, called *Rx*, to quickly recover from many types of software failures caused by common software defects, both deterministic and non-deterministic. It requires few to no changes to applications' source code, and provides diagnostic information for postmortem bug analysis. Our idea is to rollback the program to a recent checkpoint when a bug is detected, *dynamically change the execution environment based on the failure symptoms*, and then reexecute the buggy code region in the new environment. If the reexecution successfully passes through the problematic region, the environmental changes are disabled to avoid imposing time and space overheads.

Our idea is inspired from real life. When a person suffers from an allergy, the most common treatment is to remove allergens from their *living environment*. For example, if patients are allergic to milk, they should remove diary products from the diet. If patients are allergic to pollen, they may install air filters to remove pollen from the air. Additionally, when removing a candidate allergen from the environment successfully treats the symptoms, it allows diagnosis of the cause of the symptoms. Obviously, such treatment cannot and also should not start before patient shows allergic symptoms since changing living environment requires special effort and may also be unhealthy.

In software, many bugs resemble allergies. That is, their manifestation can be avoided by *changing the execution environment*. According to a previous study by Chandra and Chen [10], around 56% of faults in Apache depend on execution environment[1]. Therefore, by removing the "allergen" from the execution environment, it is possible to avoid such bugs. For example, a memory corruption bug may disappear if the memory allocator delays the recycling of recently freed buffers or allocates buffers non-consecutively in isolated locations. A buffer overrun may not manifest itself if the memory allocator pads the ends of every buffer with extra space. Data races can be avoided by changing timing events such as thread-scheduling, asynchronous events, etc. Bugs that are exploited by malicious users can be avoided by dropping such requests during program reexecution. Even though dropping requests may make a few users (hopefully the malicious ones) unhappy, they do not introduce incorrect behavior to program execution like the failure-oblivious approaches do. Furthermore, given a spectrum of possible environment changes, the least intrusive changes can be tried first, reserving the most extreme one as a last resort for when all other changes have failed. Finally, the specific environment change which cures the problem gives diagnostic information as to what the bug might be.

Similar to an allergy, it is difficult and expensive to apply these execution environmental changes from the very beginning of the program execution because we do not

[1] Note that our definition of execution environment is different from theirs. In our work, the standard library calls, such as *malloc*, and system calls are also part of execution environment.

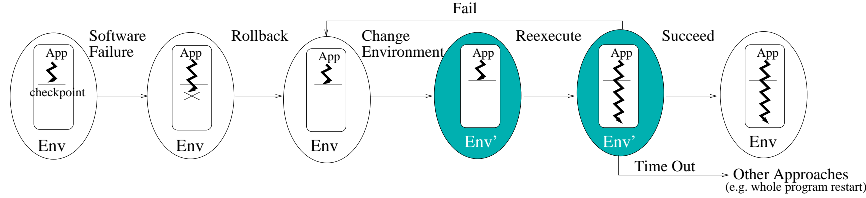

Figure 1: Rx main idea

know what bugs might occur later. For example, zero-filling newly allocated buffers imposes time overhead. Therefore, we should lazily apply environmental changes only when needed.

Compared to previous solutions, Rx has the following unique advantages:

(1) Comprehensive: Besides non-deterministic bugs, Rx can also survive deterministic bugs. We have evaluated our idea using several server applications with common software bugs and our preliminary results show that Rx can successfully survive these software bugs.

(2) Safe: Rx does not speculatively "fix" bugs at run time. Instead, it prevents bugs from manifesting themselves by changing only the program's execution environment. Therefore, it does not introduce uncertainty or misbehavior into a program's execution, which is difficult for programmers to diagnose.

(3) Noninvasive: Rx requires few to no modifications to applications' source code. Therefore, it can be easily applied to legacy software.

(4) Efficient: Because Rx requires no rebooting or warm-up, it significantly reduces system down time and provides reasonably good performance during recovery. Additionally, Rx imposes only the minimal overhead of lightweight checkpointing during normal execution.

(5) Informative: Rx does not hide software bugs. Instead, bugs are still exposed. Furthermore, besides the usual bug report package (e.g. core dumps, checkpoints and event logs), Rx provides programmers with additional diagnostic information for postmortem analysis, including what conditions triggered the bug and which environmental changes can and cannot avoid the bug. Based on such information, programmers can more efficiently find the root cause of the bug. For example, if Rx successfully avoids a bug by padding newly allocated buffers, the bug is likely to be a buffer overflow. Similarly, if Rx avoids a bug by delaying the recycling of freed buffers, the bug is likely to be caused by double free or dangling pointers.

2 Main Idea of Rx

The main idea of Rx is to reexecute a failed code region in a new environment that has been modified based on the failure symptoms. If the bug's "allergen" is removed from the new environment, the bug will not occur during reexecution, and so the program will survive this software failure without rebooting the whole program. After the reexecution safely passes through the problematic code region, the environmental changes are disabled to reduce time and space overhead.

Figure 1 shows the process by which Rx survives software failures. Rx periodically takes light-weight checkpoints that are specially designed to survive software failures instead of hardware failures or OS crashes [24]. When a bug is detected, either by an exception or by the integrated dynamic defect detection tools called Rx sensors, the program is rolled back to a recent checkpoint. Rx then analyzes the occurring failure based on the failure symptoms and "experiences" accumulated from previous failures, and determines how to apply environmental changes to avoid this failure. Finally, the program reexecutes from the checkpoint in the modified environment. This process may repeat several times, each time with a different environmental change or from a different checkpoint, until either the failure disappears or a timeout occurs. If the failure does not recur in a reexecution attempt, the execution environment is reset to normal to avoid the time and space overhead imposed by some of the environmental changes.

In our idea, the execution environment can include almost everything that is external to the target application but can affect the execution of the target application. At the lowest level, it includes the hardware such as process architectures, devices, etc. At the middle level, it includes the OS kernel such as scheduling, virtual memory management, device drivers, file systems, network protocols, etc. At the highest level, it includes standard libraries, third-party libraries, etc. Such definition of the execution environment is much broader than the one used in previous work [10].

Obviously, the execution environment cannot be arbitrarily modified for reexecution. A useful reexecution environmental change should satisfy two properties. First, it should be *correctness-preserving*, i.e., executing the original program and every step (e.g., instruction, library call and system call) of the program is executed according to the APIs. For example, in the *malloc* library call, we have the flexibility to decide where buffers should be allocated, but we cannot allocate a smaller buffer than requested. Second, a useful environmental change should be able to potentially avoid some bugs. For example, padding every allocated buffer can avoid some buffer overflows from manifesting during reexecution.

Category	Environmental Changes	Potentially-Avoided Bugs	Deterministic?
Memory Management	delayed recycling of freed buffer	double free, dangling pointer	YES
	padding allocated memory blocks	buffer overflow	YES
	allocating memory in an isolated location	memory corruption	YES
	zero-filling newly allocated memory buffers	uninitialized read	YES
Timing-related	scheduling	concurrency bugs	NO
	signal delivery	concurrency bugs	NO
	message reordering	concurrency bugs	NO
User Request Related	dropping user requests	bugs related to the dropped request	Depends

Table 1: Possible environmental changes and their potentially-avoided bugs

Examples of useful execution environmental changes include, but are not limited to, the following categories:

(1)Memory management based: Many software bugs are memory related, such as buffer overflows, dangling pointers, etc. These bugs may not manifest themselves if memory management is performed slightly differently. For example, each buffer allocated during reexecution can have padding added to both ends to prevent some buffer overflows. Delaying the recycling of freed buffers can reduce the probability for a dangling pointer to cause memory corruption. In addition, buffers allocated during reexecution can be placed in isolated locations far away from existing memory buffers to avoid some memory corruption. Furthermore, zero-filling new buffers can avoid some uninitialized read bugs. *Since none of the above changes violate memory allocation or deallocation interface specifications, they are safe to apply*.

(2)Timing based: Most non-deterministic software bugs, such as data races, are related to the timing of asynchronous events. These bugs will likely disappear under different timing conditions. Therefore, Rx can forcefully change the timing of these events to avoid these bugs during reexecution. For example, increasing the length of a scheduling time slice will likely avoid context switches during buggy critical sections.

(3)User request based: Since it is infeasible to test every possible user request before releasing software, many bugs occur due to unexpected user requests. For example, malicious users issue malformed requests to exploit buffer overflow bugs during stack smashing attacks [11]. These bugs can be avoided by dropping some users' requests during reexecution. Of course, since the user may not be malicious, this method should be used as a last resort after all other environmental changes fail.

Table 1 lists some environmental changes and the types of bugs that can be potentially avoided by them. Although there are many such changes, due to space limitations, we only list a few examples for demonstration.

After a reexecution attempt successfully passes the problematic program region for a threshold amount of time, the environmental changes applied during the successful reexecution are disabled to reduce space and time overhead. Furthermore, the failure symptoms and the effects of the environmental changes applied are recorded. This speeds up the process of dealing with future failures with similar symptoms and code locations. Additionally, Rx provides all such diagnostic information to programmers together with core dumps and other basic postmortem bug analysis information.

If the failure still occurs during a reexecution attempt, Rx will rollback and reexecute the program again, either with a different environmental change or from an older checkpoint. For example, if one change (e.g. padding buffers) cannot avoid the bug during the reexecution, Rx will rollback the program again and try another change (e.g. zero-filling new buffers) during the next reexecution. If none of the environmental changes work, Rx will rollback further and repeat the same process. If the failure still remains after a threshold number of iterations of rollback-reexecute, Rx will resort to previous solutions, such as whole program rebooting [14, 25] or micro-rebooting [7, 6, 8], as supported by applications.

Upon a failure, Rx follows several rules to determine the order in which environmental changes should be applied during the recovery process. First, if a similar failure has been successfully avoided by Rx before, the environmental change that worked previously will be tried first. If this does not work, or if no information from previous failures exists, changes with small overheads (e.g. padding buffers) are tried before those with large overheads (e.g. zero-filling new buffers). Changes with negative side effects (e.g. dropping requests) are tried last. Changes that do not conflict, such as padding buffers and changing event timing, can be applied simultaneously.

There is a rare possibility that a bug still occurs during reexecution but is not detected in time by Rx's sensors. In this case, Rx will claim a recovery success while it is not. Addressing this problem requires using more rigorous on-the-fly software defect checkers as sensors. This is currently a hot research area that has attracted much attention. In addition, it is also important to note that, *unlike in failure oblivious computing, this problem is caused by the application's bug instead of Rx's environmental changes*. Environmental changes just make the bug manifest itself in a different way. Furthermore, since Rx logs its every action including what environmental changes are applied and what the results are, programmers can use this information to analyze the bug.

3 Rx Design Overview

While the Rx implementation borrows ideas from previous work, many design issues need to be addressed differently due to differing goals. First, Rx targets software failures instead of hardware failures or OS crashes. Therefore, the checkpointing component does not need to be heavy-weight. Second, Rx does not require deterministic replay. Instead, Rx needs the exact opposite: non-determinism. Therefore, issues such as checkpoint management and the output commit problem [12] need to be addressed differently.

As shown in Figure 2, Rx consists of five components: (1) sensors for detecting failures and bugs, (2) a Checkpoint-and-Rollback (CR) component, (3) a proxy for making server recovery process transparent to clients, (4) environmental wrappers, and (5) a control unit that determines the recovery plan for an occurring failure.

Sensors detect software bugs and failures by dynamically monitoring applications' execution. There are two types of sensors. The first type detects software errors such as assertion failures, access violations, divide-by-zero exceptions, etc. This type of sensor is relatively easy to implement by simply taking over OS-raised exceptions. The second type of sensor detects software bugs such as buffer overflows, accesses to freed memory etc., before they cause the program to crash. This type of sensor leverages existing dynamic bug detection tools, such as our previous work, SafeMem [17], that have low run-time overhead (only 1.6-14%) for detecting memory-related bugs in server programs.

The CR (Checkpoint-and-Rollback) component takes checkpoints of the target application and rolls back the application to a previous checkpoint upon failure. Rx uses a light-weight checkpointing solution that is designed for surviving software failures. At a checkpoint, Rx stores a snapshot of the application into memory. Similar to the fork operation, Rx copies application memory in a copy-on-write fashion to minimize overhead. The details were discussed in our previous work [24]. Performing rollback is straightforward: simply reinstate the program from the shadow process associated with the specified checkpoint. The CR also supports multiple checkpoints and rollback to any of them.

The environment wrapper performs environmental changes during reexecution. We implement different environmental changes in different components. For example, we implement memory management based changes by wrapping the memory allocation library calls. The kernel deals with timing based changes, such as thread scheduling, signal delay, and other asynchronous timing events. The proxy process, which will be described next, manipulates user requests.

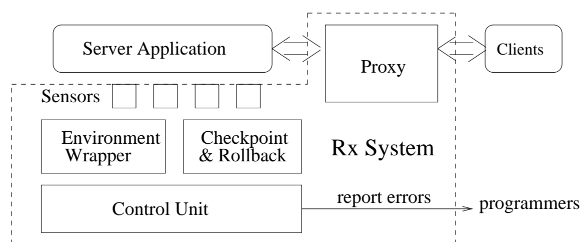

Figure 2: System architecture

To provide the reexecution functionality, Rx uses a proxy to buffer messages between the server and its remote clients. The proxy runs as a separate process to avoid corruption by the server. During normal operation, the proxy simply bridges between the server and its clients, and buffers user requests that are made since the oldest undeleted checkpoint. During a reexecution attempt from a checkpoint, the proxy replays all the user requests received since the checkpoint.

To address the output commit problem, the proxy ensures that every user request is replied to once and only once. For each request, the proxy records whether this request has been answered. If so, a reply made during reexecution is dropped silently. Otherwise, the reply is sent to the corresponding client. In other words, only the first reply goes to the client, no matter whether this first reply is made during the original execution or a successful reexecution attempt.

For applications such as on-line shopping or the SSL hand-shake protocol that require strict session consistency (i.e. later requests in the same session depend on previous replies), Rx can record the signatures (hash values) of all committed replies for each outstanding session, and perform MD5 hash-based consistency checks during reexecution. If a reexecution attempt generates a reply that does not match with the associated committed reply, the session can be aborted abnormally to avoid confusing users.

The control unit analyzes occurring failures and determines which checkpoint to roll back to and which environmental changes to apply during reexecution. After each reexecution, it records the effects (success or failure) into its failure table. This table is used as a reference for future failures and is also provided to programmers for postmortem bug analysis. The control unit also monitors the recovery time and when it exceeds some threshold, it resorts to program restart solutions.

4 Preliminary Results

We have investigated some real, buggy server programs, listed in Table 2. Our analysis shows that these software failures can be dynamically survived by our methods.

In the evaluation, we design four sets of experiments to evaluate different key aspects of Rx: (1) the functionality of Rx in surviving software failures caused by common software defects; (2) the performance overhead of Rx in both server throughput and average response time; (3) how Rx would behave while under malicious attacks

Bugs	Applications	Environment Modification
data race	mysql-4.1.1	change process's priority or make CPU scheduling timeslot longer
buffer overflow	squid-2.3	allocate memory blocks in an isolated address space, or drop request
	apache-2.0.47	
double free	cvs-1.11.4	delay the recycling of recently freed buffers

Table 2: Examples of applications that can benefit from Rx

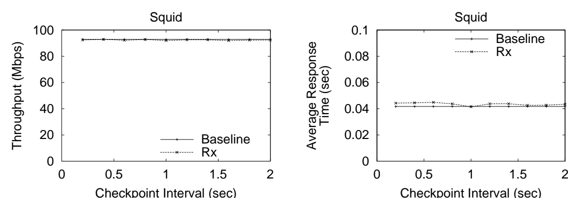

Figure 3: Rx overhead in terms of throughput and average response time for Squid. In these experiments, we do not send the bug-exposing request since we want to compare the pure overhead of Rx with the baseline in normal cases.

that continuously send bug-exposing requests triggering software defects; (4) the benefits of Rx's mechanism of learning from previous failures to speed up recovery.

In particular, Figures 3 shows the overhead of Rx for Squid compared to the baseline (without Rx) for various frequencies of checkpointing. We can see that both throughput and response time are very close to baseline for all tested checkpoint rates. Results for other server applications are similar. In this experiment, we use a workload similar to the one used in [24].

5 Conclusions

In summary, Rx is a non-invasive, informative and safe method for quickly surviving software failures to provide highly available service. It does so by reexecuting the buggy program region in a modified execution environment. It can deal with both deterministic and non-deterministic bugs, and requires little to no modification to applications' source code. Because Rx does not forcefully change programs' execution by returning speculative values, it introduces no uncertainty or misbehavior into programs' execution. Moreover, it also provides additional feedback to programmers for their bug diagnosis. Our preliminary results show that Rx is a viable solution and many server programs should be able to benefit from our approach.

6 Acknowledgments

The authors would like to thank the anonymous reviewers for their invaluable feedback. We appreciate useful discussion with the OPERA group members. This research is supported by IBM Faculty Award, NSF CNS-0347854 (career award), NSF CCR-0305854 grant and NSF CCR-0325603 grant.

REFERENCES

[1] A. Avizienis. The N-version approach to fault-tolerant software. *IEEE TSE*, SE-11(12), 1985.

[2] A. Avizienis and L. Chen. On the implementation of N-version programming for software fault tolerance during execution. In *COMPSAC*, 1977.

[3] J. F. Bartlett. A NonStop kernel. In *SOSP*, 1981.

[4] A. Bobbio and M. Sereno. Fine grained software rejuvenation models. In *IPDS*, 1998.

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

[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*, 2002.

[7] G. Candea and A. Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *HotOS*, 2001.

[8] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. A mirorebootable system – Design, implementation, and evaluation. In *OSDI*, 2004.

[9] CERT/CC. Advisories. http://www.cert.org/advisories/.

[10] S. Chandra and P. M. Chen. Whither generic recovery from application faults? A fault study using open-source software. In *DSN/FTCS*, 2000.

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

[12] E. N. Elnozahy, D. B. Johnson, and Y. M. Wang. A survey of rollback-recovery protocols in message-passing system. Technical report, TR CMU-CS-96-181, Carnegie Mellon Univ., 1996.

[13] S. Garg, A. Puliafito, M. Telek, and K. S. Trivedi. On the analysis of software rejuvenation policies. In *COMPASS*, 1997.

[14] J. Gray. Why do computers stop and what can be done about it? In *SRDS*, 1986.

[15] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton. Software rejuvenation: Analysis, module and applications. In *FTCS*, 1995.

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

[17] F. Qin, S. Lu, and Y. Zhou. Safemem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. In *HPCA*, 2005.

[18] B. Randell. System structure for software fault tolerance. *IEEE TSE*, 1(2), Jun 1975.

[19] B. Randell, P. A. Lee, and P. C. Treleaven. Reliability issues in computing system design. *ACM Comput. Surv.*, 10(2), Jun 1978.

[20] 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*, 2004.

[21] M. Rinard, C. Cadar, D. Roy, and D. Dumitran. A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors). In *ACSAC*, 2004.

[22] D. Scott. Assessing the costs of application downtime. Gartner Group, May 1998.

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

[24] S. Srinivasan, C. Andrews, S. Kandula, and Y. Zhou. Flashback: A light-weight extension for rollback and deterministic replay for software debugging. In *USENIX ATC*, 2004.

[25] M. Sullivan and R. Chillarege. Software defects and their impact on system availability – A study of field failures in operating systems. In *FTCS*, 1991.

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

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

[28] Y.-M. Wang, Y. Huang, and W. K. Fuchs. Progressive retry for software error recovery in distributed systems. In *FTCS*, 1993.