Using Rescue Points to Navigate Software Recovery (Short Paper)

Stelios Sidiroglou, Oren Laadan, Angelos D. Keromytis, and Jason Nieh
Department of Computer Science, Columbia University
{stelios, orenl, angelos, nieh}@cs.columbia.edu

Abstract

We present a new technique that enables software recovery in legacy applications by retrofitting exception-handling capabilities, *error virtualization using rescue points*. We introduce the idea of "rescue points" as program locations to which an application can recover its execution in the presence of failures. The use of rescue points reduces the chance of unanticipated execution paths thereby making recovery more robust by mimicking system behavior under controlled error conditions. These controlled error conditions can be thought of as a set erroneous inputs, like the ones used by most quality-assurance teams during software development, designed to stress-test an application. To discover rescue points applications are profiled and monitored during tests that bombard the program with bad/random inputs. The intuition is that by monitoring application behavior during these runs, we gain insight into how programmer-tested program points are used to propagate faults gracefully.

1 Introduction

In the absence of perfect software, error toleration and recovery techniques become a necessary complement to proactive approaches. The pressing need for techniques that address the issue of recovering execution in the presence of faults is reflected by recent emergence of a few novel research ideas [19, 22]. For example, *error virtualization* operates under the assumption that there exists a mapping between the set of errors that *could* occur during a program's execution (*e.g.,* a caught buffer overflow attack, or an illegal memory reference exception) and the limited set of errors that are explicitly handled by the program's code. Thus, a failure that would cause the program to crash is translated into a return with an error code from the function in which the fault occurred (or from one of its ancestors in the stack). These techniques, despite their novelty in dealing with this pressing issue, have met much controversy, primarily due to the lack of guarantees, in terms of altering program semantics, that can be provided. Masking the occurrence of faults will always carry this stigma since it forces programs down unexpected execution paths. However, we believe that the basic premise of masking failures to permit continued program execution is promising, and our goal is to minimize the likelihood of undesirable side-effects.

We outline a new technique for retrofitting legacy applications with exception-handling capabilities. Our approach consists of a general software fault-recovery mechanism that uses operating system virtualization techniques to provide "rescue points" to which an application can recover execution in the presence of faults [14]. When a fault occurs at an arbitrary location in the program, we restore program execution to a "rescue point" and imitate its observed behavior to propagate errors and recover execution. The use of rescue points reduces the chance of unanticipated execution paths, thereby making recovery more robust, by mimicking system behavior under controlled error conditions. These controlled error conditions can be thought of as a set of erroneous inputs, like the ones used by most quality assurance teams during software development, designed to stress test an application. To discover "rescue points", applications are profiled and monitored during tests that bombard the program with "bad input". The intuition is that by monitoring application behavior during these runs, we gain insight into how programmer-tested program points propagate faults gracefully.

The key difference between this work and previous techniques that try to be oblivious to occurrence of faults [19, 21, 22] is the type of impact on program semantics. Rescue points do not try to mask errors in a demonstration of blind faith. In fact, rescue points force the exact opposite behavior: they induce faults at locations that are *known* (or strongly suspected) to handle faults correctly. We achieve this through an offline analysis phase, in which we profile programs during erroneous test runs in order to build a behavioral model for the application. Using this model, we discover candidate rescue points. We then use a set of software probes that monitor the application for specific types of faults. Upon detection of a fault, an operating system recovery mechanism is invoked that allows the application to rollback application state to a rescue point and replay execution pretending that an error has occurred. Using continuous hypothesis testing, we confirm that our action has repaired the fault by re-running the application against the event sequence that apparently caused the failure. We focus on automatic healing of services against newly detected faults, including (but not limited to) software attacks.

This work focuses on server-type applications for two reasons: they typically have higher availability requirements than user-oriented applications, and they tend to have short error-propagation distances [19], *i.e.,* an error that might occur during the processing of a request has little or no impact on the service of future requests. To provide an analogy, if one considers an organization like NASA, history has shown that it would not make sense to use a mechanism like error virtualization on a software system that calculates space-travel trajectories since the correctness of the results cannot be guaranteed. However, the software on the Mars Rover would benefit greatly from a technique that allows for continued execution in the presence of faults [20]. Our plans for future work include investigating the applicability of this technique to client applications.

2 Related Work

The acceptability envelope, a region of imperfect but acceptable software systems that surround a perfect system, as introduced by Rinard [17] promotes the idea that current software development efforts might be misdirected, based on the observation that certain regions of a program can be neglected without adversely affecting the overall availability of the system. To support these claims, a number of case studies are presented where introducing faults such as an off-by-one error does not produce unacceptable behavior. This work supports our claim that most complex systems contain the necessary framework to propagate faults gracefully and the error toleration allowed by our system expands the acceptability envelope of a given application.

In the same motif, Rinard *et al.* [18, 19] developed *failure-oblivious computing*, an instantiation of which is a compiler that inserts code to deal with writes to unallocated memory by virtually expanding the target buffer. Such a capability aims toward the same goal our system does: provide a more robust fault response rather than simply crashing. Because the program code is extensively re-written to include the necessary checks for *every* memory access, their system incurs overheads ranging from 80% up to 500% for a variety of different applications. Similar to our previous work [21, 22], there is only limited empirical examination of the side effects on the program execution. Finally, this technique is only applicable to memory errors, whereas our technique can be applied to a variety of faults.

One of the most critical concerns with recovering from software faults and vulnerability exploits is ensuring the consistency and correctness of program data and state. An important contribution in this area is Automatic Data-structure Repair [8], which discusses mechanisms for detecting corrupted data structures and fixing them to match some pre-specified constraints. While the precision of the fixes with respect to the semantics of the program is not guaranteed, their test cases continued to operate when faults were randomly injected. Similar results are shown in Y-Branches [23]: when program execution is forced to take the "wrong" path at a branch instruction, program behavior remains the same in over half the times.

In the Rx system [16] applications are periodically checkpointed and continuously monitored for faults. When an error occurs, the process state is rolled back and replayed in a new "environment". If the changes in the environment do not cause the bug to manifest, the program will have survived that specific software failure. However, previous work [6, 7] found that over 86% of application faults are independent of the operating environment and entirely deterministic and repeatable, and that recovery is likely to be successful only through application-specific (or application-aware) techniques. Thus, it seems likely that Rx is only applicable in a small number of cases.

3 Approach

From a bird's eye view, our system provides a general mechanism that applications can use to recover execution in the presence of faults. Our approach can be summarized by the following set of off-line and on-line actions. Off-line, applications are profiled during "bad runs" in order to build an application behavioral model. These bad runs are generated by regression tests, if available, or through input fuzzing techniques [3, 12, 10], that stress the error handling capabilities

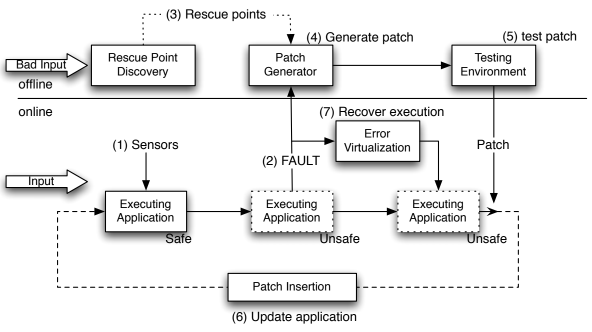

Figure 1: **System overview:** (1) **Sensors monitor the application for faults** (2) **when a fault is detected, the function where it manifested is protected using one of fault detection techniques** (3) **a rescue point is determined and inserted into the application** (4) **a patch containing the protection and rescue point is inserted into the application** (5) **the generated patch is tested with the input that caused fault and general application behavior is monitored** (6) **the production version of the server is updated with the patch** (7) **application is now able to detect and recover from the fault using error virtualization.**

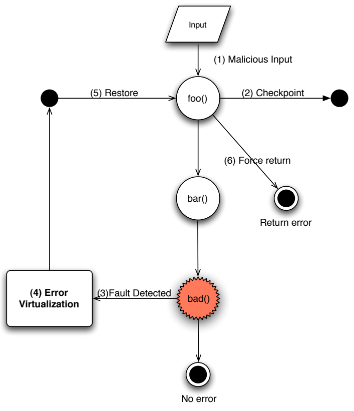

Figure 2: **Error virtualization (EV) using rescue points:** (1) **Application checkpoints state at "rescue points"** (2) **fault detection component monitors protected function for faults** (3) **when a fault occurs in a protected function, the EV component is invoked** (4) **the EV component decides on a recovery strategy and** (5) **restores application state to the rescue point** (6) **once application state is restored, the rescue point forces an error return using the value determined by the EV component.**

of applications. The intuition is that there exists a set of programmer-tested application points that are routinely used to propagate "expected" errors. In turn, these application points can be harnessed to recover from failures and thus maintain system availability. Using this model, we isolate program locations that can be used as candidate rescue points. On-line, our architecture allows for the use of a variety of fault monitors. Upon detection of a fault, application state is rolled back to a predetermined program location, a rescue point, where the program is forced to return an error, imitating the behavior observed during the erroneous runs.

The different components are illustrated in Figure 1: A set of *sensors* that continuously monitor the application for faults and takes control whenever one is detected; an *Error Virtualization* component that is responsible for determining values to inject in case of a fault; a *rescue-point discovery* component used to identify candidate rescue points through static and analysis; a *recovery* enabler that employs a checkpoint-restart mechanism to capture the state of the application and rollback back to a saved state; a *patch generator* that produces rescue-point-enabled patches for vulnerable applications; a *testing environment* in which the proposed patches are evaluated subsequently evaluated; and a *patch insertion* tool that facilitates the insertion of approved patches into running binaries.

All of the components are designed to operate without human intervention in order to minimize reaction time. In the remaining of this section we elaborate on each of these components.

3.1 Rescue-point Discovery

Determining an acceptable recovery point is of pivotal importance to error virtualization. It determines, to a large extent, the likelihood that the application will survive a fault. Two mechanisms are used for the discovery of rescue points: one static and one dynamic with more prevalence given to the latter.

Dynamic Analysis: Dynamic analysis is the preferred mechanism for discovering suitable rescue points because it grants unambiguous insight into application behavior. In particular, our goal is to learn how an application responds to "bad input", under controlled conditions, and use this knowledge to map previously unseen faults to a set of observed fault behavior. The intuition is that there exists a set of programmer-tested application points that are routinely used to propagate "expected" errors. For example, we would like to see how a program normally propagates faults when stress-tested by quality assurance tests. Learning from "bad behavior" has been used in machine learning and subsequently intrusion detection systems but to the best of our knowledge has not being used to recover from software faults.

Specifically, we instrument applications by inserting monitoring code at every function's entry and exit points using the run-time injection capabilities of dyninst [2], a runtime binary injection tool. The instrumentation records both function parameters and return types and values while the application is bombarded with faults. From these traces, we extract function call-graphs along with the return type information for each point in the graph. We call these graphs the *rescue graphs*. The rescue-graphs are used as program slices at function-level granularity that can, in turn, be used to isolate the control-flow of a fault and define possible rescue points.

Static Analysis: Static analysis is used to augment the findings of the dynamic analysis techniques described above. Specifically, we use static analysis in order to facilitate with error virtualization and rescue point discovery. For error virtualization, static analysis can help determine appropriate error return values through code inspection and backward program slicing. In detail, we explore the backward path originating at the manifestation of a fault to determine where the vulnerable function resides on the call tree. At that point, we examine how the return value of the function gets used in the function handling code. This provides insight into what kind of values we can use during error virtualization. For example, when a function's return values are used in control statements followed by exit statements this provides fertile ground for using this value as an appropriate return value during error virtualization. During this process we also pay close attention to the offending program slice for any problematic error virtualization cases, including I/O that is performed along with the use of global variable and the existence of signal-handling code.

3.2 Error Virtualization Using Rescue Points

Here, we describe the basic concept of error virtualization using rescue points and examine the basic building blocks of the system. As illustrated in Figure 2, error virtualization can be summarized by the following steps: checkpoint application state at rescue point; monitor application for fault; when a fault occurs, undo state changes made by the function, all the way back to the rescue point; after restoring execution to a rescue point, force error return using observed value

Error Virtualization When a fault is detected, using one of the available fault detection techniques, the call-stack is examined to derive the sequence of functions that led to the fault. At that point, we compare the call-stack with the rescue-graph to derive common nodes. The common nodes form a set of candidate rescue points. Candidate rescue points are then filtered according to their return type. For this particular implementation, candidate rescue points are functions with non-pointer return types or function that return pointers but the observed return value is `NULL`. Functions that return pointers require a deeper inspection of the data-structures to ascertain the values of their return types beyond the simple case of returning a `NULL`. Preliminary empirical examination shows that the examined *C* programs tend to favor the use of integer return types as failure indicators. After the function filtering, we have the final set of candidate rescue points. At that point, the candidate rescue-point graph is used to determine potential rescue points.

The function where the fault occurred forms the root of the tree. For each node in the rescue-graph, we replay the input that caused the failure and successively try error virtualization at each of the nodes. With the protection mechanism in place, faults are "caught" by the application monitor and program state is rolled back to the rescue point. The rescue point, examines the rescue-graph to determine which value to force as a return. This value is derived by analyzing the return values of the rescue-point function.

Using the example in Figure 2, when an error is detected in function `bad()`, we first extract the call-stack, which includes functions `bar()` and `foo()`. Assuming that the rescue-graph contains every function found in the call stack, we initiate error virtualization with the root node, `bad()`. We iterate the rescue-graph attempting error virtualization on nodes `bad()` and `bar()` until we reach function `foo()` that recovers program execution without side effects.

In the case where there is no overlap between the rescue and function call-graphs there are alternative recovery mechanisms that can be used. First, we can recover to a programmer-annotated point or we iterate through the call-stack of the vulnerable function (potentially all the way to main) until we find a suitable rescue point, *i.e.,* one that doesn't crash the application by applying the heuristics described in previous work [22].

3.3 Fault Detection Monitors

We treat the fault-detection component as a black box, which need only be able to notify the fault monitor of the occurrence of a fault. In addition to standard operating system error handling (*e.g.* illegal memory dereferences, etc.), we use additional mechanisms for detecting memory errors. There are a number of available fault detection components that can detect memory errors (such as ProPolice [9] and TaintCheck [13]) and some that detect violations to underlying security policies [1, 11, 15]. For the purposes of our system, we use two fault detection components that have been previously developed [21, 22] that offer tradeoffs between performance overhead and the range of faults that they can detect. For this implementation, we assume source-code availability but we plan to address applicability to commercial off-the-shelf (COTS) software in future work.

- Selective Transactional EMulation (STEM) [22] is an instruction-level emulator that can be selectively invoked for arbitrary segments of code. This tool permits the execution of emulated and non-emulated code inside the same process. The emulator is implemented as a *C* library that defines special tags (a combination of macros and function calls) that mark the beginning and end of emulation. To use the emulator in our system, we use source-to-source transformations to insert the emulator tags into every function of the application we are trying to protect and use a binary insertion tool to swap non-protected functions with emulated functions. The use of an emulator allows to catch a variety of faults ranging from memory violations to algorithmic denial-of-service attacks.
- DYnamic Buffer Overflow Containement (DYBOC) [21] uses source-to-source transformations to protect legacy applications against memory violations, such as buffer overflows, by effectively creating a moat around protected buffers. When a buffer overflow (or underflow) occurs, a SIGSEGV is raised. We use DYBOC as a more light-weight protection mechanism than STEM in cases where the manifested fault is a memory violation.

3.4 Decision: Hypothesis Testing

Once a candidate rescue point has been elected, the system proceeds to the patch testing and analysis phase. At this stage, to verify the efficacy of the proposed fix, the rescue-enabled version of the application is restarted and supplied with the input that caused the fault to manifest (or the N most recent inputs if the offending one cannot be easily identified, where N is a configurable parameter). If the application crashes, a new fix is created using the next available candidate "rescue point" and the testing and analysis phase is repeated. For our initial approach, we are primarily concerned with failures where there is a one-to-one correspondence between inputs and failures, and not with those that are caused by a combination of inputs. Note, however, that many of the latter type of failures are in fact addressed by our system, because the last input (and code leading to a failure) will be recognized as "problematic" and handled as we have discussed.

If the fix does not introduce any faults that cause the application to crash, the application is examined for semantic bugs using a set of user-supplied tests. The purpose of these tests are to provide some level of confidence on the semantic correctness of the generated patch. For example, an on-line vendor could run tests that make sure that client orders can be submitted and processed by the system.

4 Discussion

One of the most critical concerns with recovering from software faults and vulnerability exploits is ensuring the consistency and correctness of program data and state. This is an issue that is present in the majority of recovery efforts. The presence of rescue points, whether derived automatically or with programmer assistance, can alleviate most of the concerns with unpredictable execution paths but alas not completely dismiss them. In this section, we examine, in more detail, the advantages and disadvantages of fault recovery in general and error virtualization in particular.

Fault Response: A fault recovery mechanism must evaluate and choose a response from a wide array of choices. Currently, when encountering a fault, a system can pick from the following options: crash [9], crash and be restarted by a monitor [4, 5], return arbitrary values [18, 19], change environment and replay [16], slice off the functionality [21, 22], or jump to a safe (rescue) point and force error.

Previous approaches focused on crash-based methods, working under the assumption that there is no acceptable alternative. More recent work [19, 22] showed that there exists a set of alternative reactive techniques that seem to work well in practice. We elect to take the last approach of recovering execution to safe points and forcing errors in a program's execution. Early experimentation has shown that this choice seems to work extremely well. This phenomenon also appears at the machine instruction level [23]. However, there is a fundamental problem in choosing a particular response. Since the high-level behavior of any system cannot be algorithmically determined, the system must be careful to avoid cases where the response would take execution down a semantically (from the viewpoint of the programmer's intent) incorrect path. An example of this type of problem is skipping a check in *sshd* which would allow an otherwise unauthenticated user to gain access to the system. We posit that through the use of rescue points, we are able to minimize (unfortunately not eliminate) the uncertainty that a program will go down an unexpected execution path. The reason we are able to make such claims stems from the fact we opt to use as recovery points, positions in the program that are *known* to propagate errors. If higher level of assurance is required, one can rely on the programmer to provide annotations as to which parts of the code should be used for recovery and which should not be circumvented.

Programming with Error Virtualization: In this paper, we focus on fully automated techniques for every aspect of our system. However, it would not be prudent to dismiss the use of programmer assistance in program recovery. In particular, programmers can design software with error virtualization in mind, where specific locations in the code can be assigned, a priori, as rescue points that propagate faults gracefully. Programmer insight is difficult to replicate with automated techniques, especially when dealing with code cleanup and efficiency. We envision that programming with error virtualization will prove easier than dealing with language specific constructs, such as exception handling, since attention can be focused on a few select program points.

Applicability to safe languages: A pressing question that always comes up when discussing techniques that aim to protect legacy applications written in unsafe languages is: can we avoid the problems we are trying to solve by using a safe dialect? Unfortunately, it seems that having the appropriate language constructs for handling errors (exceptions) solves some of the issues but it is far from a panacea. This is especially true for large evolving systems where the complexity of the system makes it very difficult to cover all corner cases. Our approach can be applied to such systems by creating a map between the finite set of existing error handling capabilities and the infinite set of future add-on capabilities of a system. We often see systems that begin with the best of intentions, trying to cover error cases but as features creep in to the product the complexity of examining all cases becomes prohibitively large.

Availability: One of the principal goals of our work is to help reduce system down-time and consequently increase service availability in the face of failures and attacks. We anticipate that the reduction of down-time will be non-negligible; error virtualization relies on the fault detection monitors detecting faults, finding appropriate rescue points, create a patch and inserting the patch to the running application. This process requires some down-time but the cost is amortized since this cost is incurred once per detected vulnerability. Combining our approach with techniques such as micro-rebooting [4, 5] is a topic of future research.

Dealing with non-server type applications: The success of our system in recovering program execution can be explained partly by the basic characteristics of the types of applications that we examine. As articulated in [19], server type applications tend to have short error propagation distances and forcing errors in one request has little if any impact on future requests. While server application might have an inherent advantage in propagating errors, we believe that most application are written with some error handling capabilities. Correctly identifying these rescue points should translate our approach to a wide range of applications, although, as mentioned previously, applications that rely on the integrity of their computation might be better off using an alternative strategy.

5 Conclusions

We have outlined *error virtualization using rescue points,* a new software self-healing technique for detecting, tolerating and recovering from software faults in server applications. Our approach leverages existing quality assurance testing to generate known bad inputs to an application to create a call graph of functions and their return values as potential rescue points. We then use target systems to detect software faults in the application caused by attacks to exploit software vulnerabilities, and obtain a resulting call stack. This is matched with the potential set of rescue points by rolling back and repeating execution with the fault to determine which rescue point can be used for recovering from the fault. Our system dynamically patches the running production application to self-checkpoint at the rescue point and, if a fault occurs, roll back to the checkpoint and return a known return value used to respond to bad input, which is used by the applications own built-in error handling mechanisms to recover from the fault. Our plans for future work include demonstrating the effectiveness of our technique using a battery of real and synthetic attacks and failures, and evaluating its performance impact on applications.

6 Acknowledgements

We thank the anonymous reviewers and our shepherd Diego Zamboni for their insightful comments and suggestions on the paper. This material is based on research sponsored by Air Force Research Laboratory under agreement number FA8750-06-2-0221 and by NSF Grant 06-27473. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow Integrity. In *Proceedings of ACM CCS*, pages 340–353, November 2005.

[2] B. Buck and J. K. Hollingsworth. An API for Runtime Code Patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[3] C. Cadar and D. R. Engler. Execution generated test cases: How to make systems code crash itself. In P. Godefroid, editor, *SPIN*, volume 3639 of *Lecture Notes in Computer Science*, pages 2–23. Springer, 2005.

[4] G. Candea and A. Fox. Recursive Restartability: Turning the Reboot Sledgehammer into a Scalpel. In *Proceedings of the HotOS Workshop*, pages 125–132, May 2001.

[5] G. Candea and A. Fox. Crash-Only Software. In *Proceedings of the HotOS Workshop*, May 2003.

[6] S. Chandra. *An Evaluation of the Recovery-related Properties of Software Faults*. PhD thesis, University of Michigan, 2000.

[7] S. Chandra and P. M. Chen. Wither Generic Recovery from Application Faults? A Fault Study using Open-Source Software. In *Proceedings of DSN/FTCS*, June 2000.

[8] B. Demsky and M. C. Rinard. Automatic Detection and Repair of Errors in Data Structures. In *Proceedings of ACM OOPSLA*, October 2003.

[9] J. Etoh. GCC extension for protecting applications from stack-smashing attacks. http://www.trl.ibm.com/projects/security/ssp/, June 2000.

[10] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005.

[11] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution Via Program Shepherding. In *Proceedings of the* 11^{th} *USENIX Security Symposium*, pages 191–205, August 2002.

[12] B. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Technical report, 1995.

[13] J. Newsome, D. Brumley, and D. Song. Vulnerability-Specific Execution Filtering for Exploit Prevention on Commodity Software. In *Proceedings of the* 13^{th} *NDSS Symposium*, 2006.

[14] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of zap: A system for migrating computing environments, 2002.

[15] N. Provos. Improving Host Security with System Call Policies. In *Proceedings of the* 12^{th} *USENIX Security Symposium*, pages 257–272, August 2003.

[16] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: treating bugs as allergies - a safe method to survive software failures. In A. Herbert and K. P. Birman, editors, *Proceedings of ACM SOSP)*, pages 235–248, 2005.

[17] M. Rinard. Acceptability-oriented Computing. In *Proceedings of ACM OOPSLA*, October 2003.

[18] M. Rinard, C. Cadar, D. Dumitran, D. Roy, and T. Leu. A Dynamic Technique for Eliminating Buffer Overflow Vulnerabilities (and Other Memory Errors). In *Proceedings of ACSAC*, December 2004.

[19] M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu, and J. W Beebee. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *Proceedings of OSDI*, December 2004.

[20] R. Sengupta, O. J. D., F. D. J., K. D. S., Springer, S. P. L., N.-S. H. S., H. M. A., M. R. J., and J. C. Software Fault Tolerance for Low-to-Moderate Radiation Environments. In *ASP Conf. Ser., Vol. 238, Astronomical Data Analysis Software and Systems X*, 2001.

[21] S. Sidiroglou, Y. Giovanidis, and A. Keromytis. A Dynamic Mechanism for Recovery from Buffer Overflow attacks. In *Proceedings of the 8th Information Security Conference (ISC)*, September 2005.

[22] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building a Reactive Immune System for Software Services. In *Proceedings of the USENIX Technical Conference*, April 2005.

[23] N. Wang, M. Fertig, and S. Patel. Y-Branches: When You Come to a Fork in the Road, Take It. In *Proceedings of the* 12^{th} *International Conference on Parallel Architectures and Compilation Techniques*, September 2003.