Safe Software Updates via Multi-version Execution

Petr Hosek Cristian Cadar
Imperial College London
{*p.hosek,c.cadar*}*@imperial.ac.uk*

Abstract—**Software systems are constantly evolving, with new versions and patches being released on a continuous basis. Unfortunately, software updates present a high risk, with many releases introducing new bugs and security vulnerabilities.**

We tackle this problem using a simple but effective multi-version based approach. Whenever a new update becomes available, instead of upgrading the software to the new version, we *run the new version in parallel with the old*; by carefully coordinating their executions and selecting the behavior of the more reliable version when they diverge, we create a more secure and dependable multi-version application.

We have implemented this technique in a prototype system targeting multicore processors, and show that it can be applied successfully to several security-critical applications, such as `lighttpd` and `redis`.

Keywords-**software updates, multi-version execution, security vulnerabilities**

I. Introduction

The last decade has seen the emergence of new computing platforms, ranging from multicore processors to large-scale data centers, which provide an abundance of computational resources and a high degree of parallelism. These platforms are already being successfully used to increase the performance of certain classes of applications, through data-processing systems such as MapReduce [12], Hadoop [30] or Dryad [14]. However, relatively little attention has been payed to exploiting this abundance of resources to improve the safety, reliability and security of software systems, especially in the case of code with limited or no inherent parallelism.

In this paper, we propose a novel technique that takes advantage of the resources made available by these platforms (e.g., idle processor time) to increase the reliability and security of software systems. Our approach targets the software update process. Software updates are an integral part of the software life-cycle, but present a high failure rate, with many users and administrators refusing to upgrade their software and relying instead on outdated versions, which often leaves them exposed to critical bugs and security vulnerabilities. For example, a recent survey of 50 system administrators has reported that 70% of respondents refrain from installing a software upgrade, regardless of their experience level [11].

One of the main reasons for which users hesitate to install updates is that a significant number of them result in failures. It is only too easy to find examples of updates that fix a bug or a security vulnerability only to introduce another problem in a different part of the code. Our goal is to improve the software update process in such a way as to encourage users to upgrade to the latest software version, without sacrificing the stability of the older version.

Our proposed solution is simple but effective: whenever a new update becomes available, instead of upgrading the software to the newest version, we *run the new version in parallel with the old*. Then, by selecting the output of the more reliable version when their executions diverge, we can increase the overall reliability of the software; in effect, our goal is to have the multi-version software system be at least as reliable as each individual version by itself. As new versions arrive, we execute them in parallel with the existing ones, until all available resources have been exhausted, or a user-specified threshold has been reached. At that point, we can either discard the oldest versions, or we can use more sophisticated replacement strategies.

In this paper, we present a prototype targeting multicore processors, and a relatively small number of versions. However, our approach can be extended to work with several different platforms, handle a large number of versions, and balance conflicting requirements such as performance, reliability and energy consumption.

The rest of this paper is organized as follows. Section II gives an overview of our approach, by walking the reader through an example usage scenario (§II-A), discussing the main goals and challenges of our approach (§II-B), and defining its scope (§II-C). Then, Section III presents a protype implementing our safe updates approach in the context of multicore processors and Section IV presents our experience applying it to several real applications. Finally, Section V discusses related work and Section VI concludes.

II. Overview

A. Example Scenario

To motivate our approach, we present a real scenario targeting `lighttpd`, which is representative of one type of applications which could benefit from our approach, namely server applications with stringent security and availability requirements.

`lighttpd`[1] is a popular open-source web-server that achieves high-scalability, without sacrificing standards-compliance and security. As a result, `lighttpd` is used by

[1]http://www.lighttpd.net/

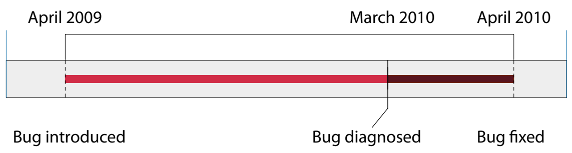

Figure 1. Crash bug #2169 from `lighttpd`.

several high-traffic websites such as YouTube, Wikipedia, and Meebo.

Despite its popularity, crash bugs are still a common occurrence in `lighttpd`, as evident from its bug tracking database[2]. Below we discuss one such bug, which our approach could successfully eliminate.

In April 2009, a patch was applied[3] to `lighttpd`'s code related to the HTTP ETag functionality. An ETag is a unique string assigned by a web server to a specific version of a web resource, which can be used to quickly determine if the resource has changed. The patch was a one-line change, which discarded the terminating zero when computing a hash representing the ETag. More exactly, line 47 in `etag.c`:

```
for (h=0, i=0; i < etag->used; ++i) h = (h<<5)^(h>>27)
^(etag->ptr[i]);
```

was changed to:

```
for (h=0, i=0; i < etag->used-1; ++i) h = (h<<5)^(h>>27)
^(etag->ptr[i]);
```

This correctly changed the way ETags are computed, but unfortunately, it broke the support for compression, whose implementation depended on the previous computation. More exactly, `lighttpd`'s support for HTTP compression uses caching to avoid re-compressing files which have not changed since the last compression. To determine whether the cached compressed file is still valid, `lighttpd` uses ETags. Unfortunately, the code implementing HTTP compression did not consider the case when ETags are disabled. In this case, `etags->used` is `0`, and when the line above is executed, `etag->used-1` underflows to a very large value, and the code crashes while accessing `etag->ptr[i]`. Interestingly enough, the original code was still buggy (it always returns zero as the hash value, and thus it would never re-compress the files), but it was not vulnerable to a crash.

The segfault was diagnosed and reported in March 2010[4] and fixed at the end of April 2010[5], more than one year after it was introduced. The history is depicted graphically in Figure 1. The bottom line is that for about one year, users affected by this bug essentially had to decide between (1) incorporating the new features and bug fixes added to the code, but being vulnerable to this crash bug, and (2) giving up on these new features and bug fixes and using an old version of lighttpd, which is not vulnerable to this bug. Note that this is particularly true for the eleven-month period between the time when the bug was introduced and the time it was diagnosed, since during this time most users would not know how to change the server's configuration to avoid the crash.

[2]http://redmine.lighttpd.net/issues/
[3]http://redmine.lighttpd.net/projects/lighttpd/repository/revisions/2438
[4]http://redmine.lighttpd.net/issues/2169
[5]http://redmine.lighttpd.net/projects/lighttpd/repository/revisions/2723

In our proposed approach, when a new version arrives, instead of replacing the old version, we run both versions in parallel. As more versions arrive, we execute them in parallel with the existing ones, until all available resources have been exhausted, at which point we discard some of the versions according to some strategy.

In our example, consider a system that is running a version of `lighttpd` from March 2009. When the buggy April 2009 version is released, our system runs it in parallel with the old one. As the two versions run, the system checks that their external behavior is identical (e.g., they write the same values into the same files, or send the same data over the network). When the two versions diverge, the divergence is resolved in the favor of the more reliable version. In particular, if one of the two versions crashes, the behavior of the non-crashing version is used, and the other version is transparently modified to survive the crash. If the system cannot determine which behavior is correct, a simple heuristic can be used, such as always preferring the behavior of the newer version. In our example, this effectively eliminates the bug in Figure 1, while still allowing users to use the latest features and bug fixes of the recent versions.

B. Goals and Challenges

There are a number of challenges that need to be addressed to make the proposed approach work in practice, which we group into four main categories.

Multi-execution environment: To be able to run multiple versions of a single application in parallel, a specialized execution environment is needed. The main goal of this execution environment is to allow multiple software versions to act as one to external users. To achieve this goal, the execution environment has to *mediate* any interactions with the outside world (e.g., so that a web-server does not send duplicate responses to clients), *synchronize* the execution of different versions (so that they perform similar actions at the same time), resolve any *divergences* in their behavior in favor of the correctly-executing version (so that the overall application exhibits greater security and reliability), and allow the other versions to *survive* the divergence. Note that the last point is of key importance, as the success of our technique depends on having all versions running at all times.

Reasonable performance overhead: To be practical, this mechanism has to incur minimal overhead on top of native execution. In addition, we need to ensure that the overall system is able to scale up and down the number of software versions run in parallel in order to balance conflicting requirements such as performance, reliability, security and energy consumption.

Support for native applications: While not an absolute requirement, we believe that to be usable in practice, our system should operate directly on application binaries without access to the target application source code. While such a requirement poses many implementation challenges, it makes it easy to integrate our approach with existing software package managers (e.g., `apt`, `yum`) and has the additional advantage that it doesn't require separate support for different programming languages and runtime environments.

Deployment strategy: While our approach eases the decision of applying a software update— as incorporating a new version would never decrease the security and reliability of the overall multi-version application—the number of versions that can be run in parallel is limited, being dictated by the number of available resources (e.g., the number of available CPU cores). As a result, we need a deployment strategy to decide what versions are run in parallel. For example, we could always run the last n released versions (where n is the number of available resources), or we could always keep a one-year old version, etc. This paper focuses on techniques for allowing multiple versions to successfully coordinate their parallel execution, but in future work we plan to explore deployment strategies in more detail.

C. Scope

The success of our approach depends on two important assumptions: the change in external behavior between the versions run in parallel has to be relatively small, and there must be a way to decide on the correct behavior when the versions diverge. We discuss each of them below.

Small changes in external behavior: First, the behavior of the versions that are run in parallel has to be *similar enough* to allow us to synchronize their execution. Moreover, we expect versions to re-converge to the same behavior after any given divergence. To this end, our approach is particularly suitable for consecutive software versions, which have relatively small differences in behavior. Our empirical study in Section IV-A shows that changes to externally observable behavior of an application are often minimal. Note that the key insight here is that we are only concerned with *externally observable behavior*, and are oblivious to the way the external behavior is generated. As a trivial example, given two versions of a routine that outputs the smallest element of an array, our approach considers them equivalent even if the first version scans the array from the first to the last element, while the other scans it in reverse order.

Types of applications and code changes: As mentioned above, our system relies on the assumption that versions re-converge to the same behavior after any given divergence. This places certain restrictions on both the type of applications and the type of code changes. For example, we believe our approach is very suitable to the kind of applications that perform a series of mostly independent requests, such as network servers. These applications are usually structured around a main dispatch loop, which provides a useful re-convergence point. Similarly, our approach is most suitable to local code changes, which have small propagation distances, which ensures that the different versions will eventually re-converge to the same behavior.

Resolving divergences: While detecting divergences between different versions is relatively easy, deciding which behavior to use when multiple ones are available is much more difficult. In fact, we recognize that in the general case it is impossible to determine which software version is the "correct" one, without having access to a higher-level specification. Instead, in this paper we focus on surviving generic bugs, such as those that result in a segmentation fault. For all other divergences, our approach is to favor the latest software version. Thus, if we run a version v_1 in parallel with a more recent version v_2, we always choose v_2's behavior when the two versions disagree, unless v_2's behavior results in a crash, in which case we use v_1 to survive the crash, and restart v_2 after the crash point using v_1's state (see §III for details).

As illustrated by our example in Section II-A, our approach targets the common situation in which the newly released version fixes an existing bug or security vulnerability, or adds some new desired functionality, but at the same time introduces a new failure. In such cases, our approach often allows the user to benefit from the positive changes in the new version, without sacrificing the stability of the old one.

III. Prototype System

We have implemented our approach in a prototype system called Mx, targeted at multicore processors running Linux. Currently, our prototype fully works with only two application versions, but we are working on adding support for a larger number of versions.

Figure 2 shows a platform running Mx, on which conventional (i.e., unmodified) applications and multi-version (MV) applications run side by side. The key property that must hold on such a platform is that without purposely trying to do so, applications should not be able to distinguish between conventional and MV applications running on the platform. case. In particular, the multiple versions of an MV application should appear as one to any other entity interacting with them (e.g., user, operating system, other machines). Furthermore, MV applications should be more reliable and secure than their component versions, and their performance should not be significantly degraded.

To achieve these goals, our prototype Mx employs several different components, as shown in the architectural overview

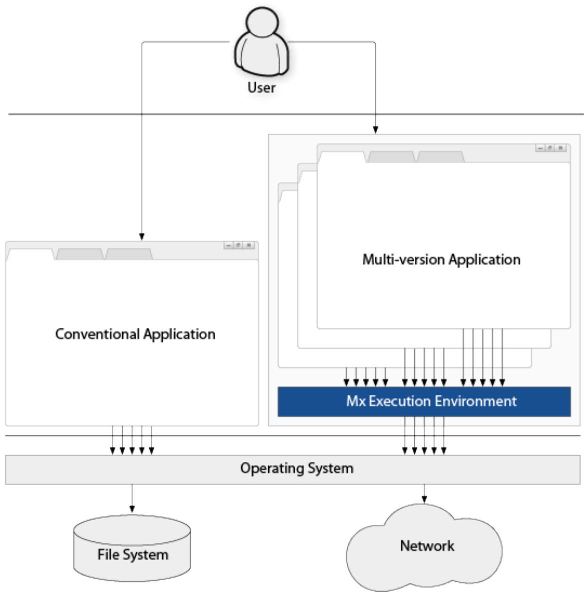

Figure 2. A platform running both conventional and multi-version applications.

of Figure 3. The input to Mx consists of the binaries of two versions of an application, which we'll refer to as the *old version*—the one already running on the system), and the *new version*—the one newly released.

These two binaries are first statically analyzed by the SEA (Static Executable Analyzer) component, which constructs a mapping from the control flow graph (CFG) of the old version to the CFG of the new version (§III-C). The two versions are then passed to MXM (Multi-eXecution Monitor), whose job is to run the two versions in parallel, synchronize their execution, virtualize their interaction with the outside environment, and detect any divergences in their external behavior (§III-A). Once a divergence is detected, it is resolved by REM (Runtime Execution Manipulator), which selects between the available behaviors, and resynchronizes the two versions after the divergence (§III-B).

The system prototype has been implemented in C with a small amount of assembly, and the current version has approximately 28,200 source lines of code. The implementation currently supports Linux 2.6.39 and above, running x86 and x86-64 architectures.

The rest of this section describes the main Mx system components and their implementation in more detail, and discusses how they work together to support safe software updates.

A. MXM: *Multi-eXecution Monitor*

One of the main components of our multi-version execution environment is the MXM monitor. MXM's main jobs are to run the two versions concurrently, mediate their interaction with the outside world, synchronize their executions based on their external behavior, and detect any divergences in their external behavior.

We define the *external behavior* of an application as its sequence of system calls, which are the only mechanism for an application to change the state of its environment. Therefore, MXM works by intercepting all system calls issued by each application version, and manipulating them to ensure that the two versions are executed in a synchronized fashion and that they act as one to the outside world.

MXM is implemented using the `ptrace` interface provided by Linux kernels. This interface, often used for application debugging, allows simple deployment (without any need for compile-time instrumentation) and makes the monitor itself lightweight since it is running as a regular unprivileged process. MXM is similar in operation to previous monitors whose goal is to synchronize applications at the level of system calls, in particular that implemented by the Orchestra project [25].

MXM runs each version in a separate child process, intercepting all their system calls. When a system call is intercepted in one version, MXM waits until the other version also performs a system call. With a pair of system calls in hand (one executed by the old version, and one by the new version), MXM compares their types and arguments. If they differ, MXM has detected a divergence and invokes the REM component to resolve it (§III-B).

Otherwise, if the two versions perform the same system call with the same arguments, MXM mediates their interaction with the environment. If the operation performed by the system call is idempotent (e.g., `sysinfo`), MXM allows both processes to execute it independently. Otherwise, MXM executes the system call on their behalf and copies its results into the address spaces of both versions.

There are several challenges that we encountered while implementing MXM. First, MXM must partly understand the semantics of system calls. For example, many system call parameters use complex (often nested) structures with

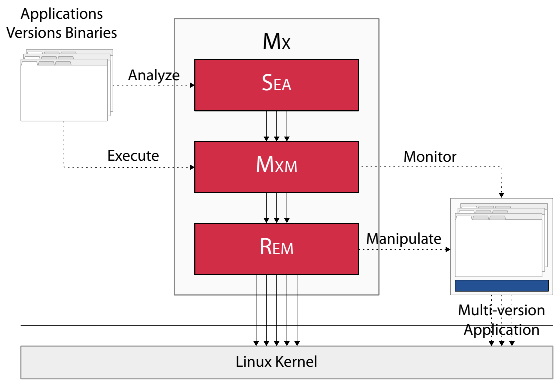

Figure 3. System architecture for multicore processors.

complicated semantics to pass values to the operating system kernel, as in the case of `ioctl` or `futex`. In order to be able to compare the parameters of these system calls and to copy back their results, MXM needs to understand the semantics of these structures. However, there are only a small number of system calls in Linux, and once they are implemented they can be reused by any application that uses them. MXM currently implements 102 system calls (out of the 306 provided by Linux-x64 2.6.39), which was enough to allow us to run Mx on our benchmarks (§IV).

Second, the arguments of a system call are often passed through pointers, which are only valid in the application address space, which is not directly available to MXM. Therefore, MXM needs to copy the content pointed to by these structures to its own address space in order to perform their comparison.

Third, because the structures passed as arguments to system calls often have variable-size, MXM also needs a fast way to allocate and deallocate memory for them in order to minimize the overall overhead imposed by our system. For this purpose, MXM uses a region-based memory allocator [24], namely the `obstack` library[6].

Finally, a particular challenge arises in the context of multi-process and multi-threaded applications. Using a single monitor instance to intercept both versions and their child processes (or threads) can cause significant delays in handling their system calls, which eliminates any advantage that these applications derive from using concurrency. Therefore, to avoid the bottleneck of a single monitor instance, MXM uses a new monitor thread for each set of child processes (or threads) spawned by the monitored versions. For instance, if the old and new versions each have a parent and a child process, then MXM will use two threads: one to monitor the parent processes, and one to monitor the child processes in each version.

Due to limitations of the `ptrace` interface (which was not designed to be used in a multi-process/multi-threaded environment), handing the control of any child processes being spawned by the application over to a new monitoring thread is somewhat complicated. In MXM we adopt the solution described in [25]. When new child processes are spawned, we let the parent monitoring thread to supervise their execution until the first system call. Then, we replace this system call with a `pause` system call, disconnect the parent monitor (which causes a `continue` signal to be sent to all new child processes), and spawn a new monitoring thread which immediately reconnects to the new child processes, restores their original system calls, and resumes their execution.

Limitations and future work: The main limitation of MXM is related to its performance (see §IV-E). In future work, we plan to improve performance in several ways. First, we plan to replace the `ptrace` interface with the faster `pread` and `pwrite` interfaces, which allow direct access to the address space of child processes. Second, we plan to synchronize versions at a coarser granularity, by using an window/epoch approach [29], and by performing certain synchronizations at the level of shared library calls. Finally, we could explore the possibility of not intercepting system calls in certain pieces of code that were previously shown to be safe and do not need to be replicated across multiple versions, using a binary translation approach.

[6]http://www.gnu.org/software/hello/manual/libc/Obstacks.html

We also plan to improve the precision with which we detect divergences. The current implementation considers two versions to be equivalent as long as they perform the same system calls with the same arguments. While this covers a large class of software updates, there are certain refactorings which may affect the order of system calls, without changing the overall application behavior. We plan to explore approaches similar to some compiler optimizations, such as *peephole optimization* [1], and adapt them to work on the level of kernel and library calls.

B. REM*: Runtime Execution Manipulator*

At the core of our approach lies the REM component of Mx, which is invoked by MXM whenever a divergence is detected. REM has two main jobs: (1) to decide whether to resolve the divergence in favor or the old or the new version; and (2) to allow the other version to execute through the divergence and resynchronize the execution of the two versions after the divergence. As discussed in Section II-C, in this paper we restrict our attention to surviving crash errors, so the key challenge is to allow the crashing version to survive the crash. This is essential to the success of our approach, which relies on having both versions be alive at all times, so that the overall application can survive any crash bugs that happen in either the old or the new version (although of course, not in both).

Suppose that one of the versions has crashed between the execution of system call s_1 and the execution of system call s_2. Then, in many common scenarios, the code executed between the two system calls is responsible for the crash (e.g., the old version crashes because it doesn't incorporate a bug fix present in the new version, or the new version crashes because its code was patched incorrectly). Therefore, our strategy is to *use the code of the non-crashing version to execute over this critical point in the crashing version.*

Our exact recovery mechanism is illustrated in Figure 4. At each system call, Mx creates a lightweight checkpoint of each version. This is implemented using the `clone` system call in Linux, which internally uses a copy-on-write strategy.

As shown in Figure 4, suppose that the crash happens in version v_2, between system calls s_1 and s_2. Then, REM first restores v_2 at point s_1, copies v_1's code into v_2's code segment, executes over the critical point using v_1's code

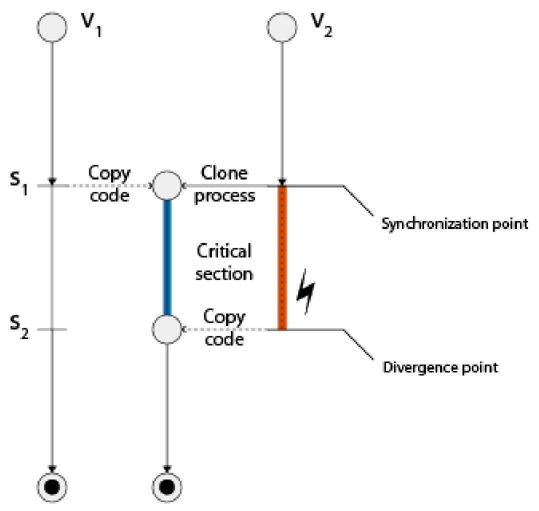

Figure 4. REM's recovery mechanism uses the code of the non-crashing version to run through the critical section.

(but note that we are still using v_2's memory state), and then restore v_2's code at point s_2.

There are several challenges in implementing this functionality. First, REM needs the ability to read and write the application code segment. In the current implementation, we bypass this by linking together the two application versions after renaming all the symbols in one of the versions using a modified version of the `objcopy` tool[7]. However, in the future we plan to implement this transparently by using the `pread` and `pwrite` interface to directly read and write the process memory via the *proc* file system.

Second, REM needs to modify the contents of the stack in v_2. This is necessary because the return addresses on the stack frames of v_2 still point to v_2's original code, which was now replaced by v_1's code. Without also modifying v_2's stack, any function `return` instruction executed between s_1 and s_2 would most likely veer execution to an incorrect location, since function addresses are likely to be different across different versions. Thus, after REM replaces v_2's code, it also updates the return addresses on v_2's stack with the corresponding return addresses in v_1, which are obtained via static analysis (§III-C). Because system calls are invoked via wrapper functions in `libc`, this ensures that when v_2 resumes execution, it will immediately return to the code in v_1.

To implement this functionality, REM makes use of the `libunwind` library[8], which provides a portable interface for accessing the program stack, for both x86 and x86-64 architectures. To actually modify the execution stack of v_2, REM uses again the `ptrace` interface.

Unfortunately, updating the stack return addresses is not sufficient to ensure that v_2 uses v_1's code between s_1 and s_2, as v_2 may also use function pointers to make function calls. To handle such cases, REM inserts breakpoints to the first instruction of every function in v_2's original code. Then, when a breakpoint is encountered, REM is notified via a `SIGTRAP` signal, and redirects execution to the equivalent function in v_1's code (which is obtained from the SEA component) by simply changing the instruction pointer.

Finally, after executing through the critical code, REM performs the same operations in reverse: in redirects execution to v_2 original code, changes the return addresses on the stack to point to v_2's functions, and disables all breakpoints inserted in v_2's code. The one additional operation that is done at this point is to copy all the global data modified by v_1's code into the corresponding locations referenced by v_2's code. This is to a large extent an artifact of our current implementation (which links together the two versions as discussed above), so we do not provide a detailed description of this step.

Limitations and future work: Furthermore, our recovery mechanism only works when the two versions have relatively small changes in their external behavior, as discussed in more detail in Section II-C. Our initial case study in Section IV-A is encouraging, indicating that the external behavior of an application changes slowly over time, but this is still an important limitation of our approach.

Furthermore, we acknowledge that our approach of using the code of the non-crashing version to survive failures in the crashing one may leave the application in an inconsistent state, and thus may not be applicable for application in which absolute correctness and a fail-fast approach is more important than allowing the application to survive errors. However, Mx is usually able to discover most inconsistencies, since it regularly checks if the two versions have the same the external behavior.

REM's recovery mechanism also cannot handle major modifications to the layout of the data structures used by the code, including individual stack frames. While this still allows us to support several common software update scenarios, in future work we plan to improve REM with the ability to perform full stack reconstruction (which would allow it to support changes to stack layout), and with algorithms for inferring basic data structure changes in binaries [10], [18].

REM currently imposes a high performance overhead by creating a checkpoint after the execution of each system call in each version. To decrease this overhead, in future work we plan to perform these checkpoints at a lower frequency, and record the external behavior since the last checkpoint, so that it can be successfully replayed during recovery (e.g., as in [21]).

C. SEA*: Static Binary Analyzer*

The SEA component statically analyzes the binaries of the two versions to obtain information needed at runtime by

[7]http://sourceware.org/binutils/docs/binutils/objcopy.html
[8]http://www.nongnu.org/libunwind/

the MXM and REM components. SEA is invoked only once, when the multi-version application is assembled from its component versions. As mentioned in §III-B, we currently link together the two application versions after renaming all the symbols in one of them using a modified copy of the `objcopy` tool, although in the future we plan to do this linking dynamically by directly changing the code segment in each version.

The main goal of SEA is to create several mappings from the code of one version to the code of the other. First, SEA extracts the addresses of all function symbols in one version and maps them to the addresses of the corresponding functions in the other version. This mapping is used by REM to handle calls performed via function pointers (§III-B).

Second, SEA computes a mapping from all possible return addresses in one code to the corresponding return addresses in the other code. In order to allow for code changes, this mapping is done by computing an ordered list of all possible return addresses in each function. For example, if function `foo` in v_1 performs call instructions at addresses `0xabcd0000` and `0xabcd0100`, and function `foo` in v_2 performs call instructions at addresses `0xdcba0000` and `0xdcba0400`, then SEA will compute the mapping {`0xabcd0005` → `0xdcba0005`, `0xabcd0105` → `0xdcba0405`} (assuming each call instruction takes 5 bytes). This mapping is then used by REM to rewrite return addresses on the stack.

To construct these tables, SEA first needs to extract the addresses of all function symbols and then disassemble the code for each individual function in order to locate the call instructions within these functions. The implementation is based on the `libbfd` and `libopcodes` libraries, which are part of the GNU BINUTILS suite[9]. To obtain the addresses of all function symbols defined by the program, SEA uses `libbfd` to extract the static and dynamic symbol tables and relocation tables. To disassemble individual functions, SEA uses the `libopdis` library[10], a wrapper on top of `libopcodes`, which was modified to properly handle shared libraries, dynamic symbols and symbol relocations.

Limitations and future work: Like REM, SEA cannot handle major changes between the code of the two versions. In particular, SEA ignores any function whose name has changed from one version to the other, as well as more complex refactorings that involve splitting a function or merging together several different functions. If any of these functions are involved during recovery, the recovery process fails. Similarly, SEA can only handle cases where both versions perform function calls in the same order (although the code in-between these calls may change). If code in which the order of function calls has changed is involved during recovery, the recovery process similarly fails. In the future, we plan to use clone detection techniques [15], [17] to identify changes to function names and to sequences of function calls.

[9]http://www.gnu.org/software/binutils/

[10]http://mkfs.github.com/content/opdis/

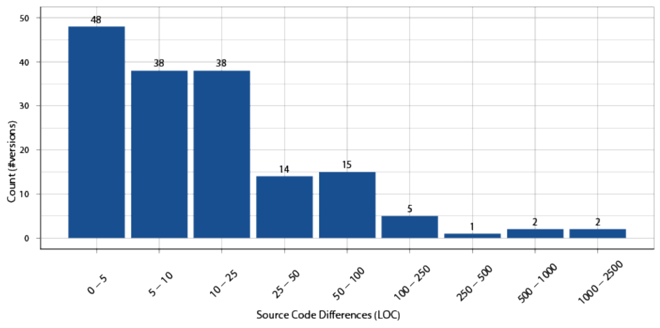

Figure 5. Source code differences across 164 versions of `lighttpd`.

IV. EVALUATION

To evaluate our prototype, we start by presenting a study focusing on how the external behavior of a real application evolves in practice (§IV-A). We then discuss our experience applying Mx to two real applications, `redis` (IV-B) and `lighttpd` (IV-C). We then examine the question of how far apart can be the versions run by Mx, and discuss Mx's performance overhead (§IV-E)

A. *Evolution of External Behavior*

Our approach is based largely on the assumption that during software evolution, changes to the externally observable behavior of an application (i.e., kernel system calls) are relatively small. To verify this assumption, we have compared 164 successive revisions of the `lighttpd` web server, namely revisions `2379`–`2635`, which were developed and released over a span of approximately ten months, from January to October 2009. To understand the amount of code changes in these versions, we computed the number of lines of code (LOC) that have changed from one version to the next. Figure 5 summarizes these differences. This graph shows that patches in `lighttpd` are relatively small, most of them affecting less than 30 LOC.

To compare the external behavior of each version, we have traced the system calls made by these versions using the strace[11] tool, while running all the test cases from the `lighttpd` regression suite targeting the core functionality (a total of seven test cases, but each test case contains a large number of HTTP requests). All tests were executed on a machine running a Linux 2.6.40.6 x86-64 kernel and the GNU C library 2.14.

To eliminate possible sources of non-determinism, we have disabled address-space randomization while running the tests. To further ensure the absence of non-deterministic

[11]http://sourceforge.net/projects/strace/

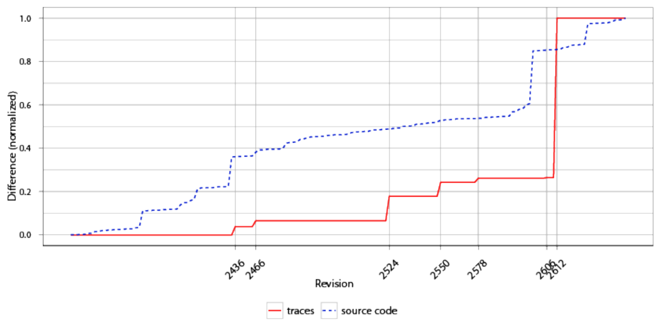

Figure 6. Correlation of differences in post-processed system call traces with differences in source code across 164 revisions of `lighttpd`.

behavior, we have repeated the tracing three times for each test case and compared the resulting traces across runs.

The system call traces were further normalized and post-processed. During normalization, we first split the original trace on a per-process basis, so that the trace of each different process used by `lighttpd` was stored in an individual file. We then normalized all differences caused by timing (and which would not affect Mx's operation), e.g., we collapsed all sequences of `accept-poll` system calls, which represent repeated polling operations.

Trace files were then post-processed by eliminating individual system call arguments and return values. This post-processing step might reduce the precision of our comparison, but we performed it because many system calls accept as arguments addresses of data structures residing in the process memory space, and these addresses may differ across versions (but Mx handles this while mediating the effect of system calls as described in §III-A).

Finally, we concatenated the traces of all `lighttpd` processes spawned by each version. In the end, we ended up with one trace for each run of a `lighttpd` version on a test case in the regression suite. For each test case, we compared the traces of consecutive `lighttpd` versions using the *Damerau-Levenshtein* edit distance.

Our results are shown in Figure 6, which correlates the differences in post-processed system call traces with the source code changes. The graph shows that while the code changes continuously across versions, changes in externally observable behavior occur only sporadically. In fact, 156 versions (which account for around 95% of all versions considered) introduce *no changes* in external behavior. In particular, the revision which introduced the bug described in detail in Section II-A, is one of the versions that introduces no observable differences in system call traces, yet this revision is responsible for a critical crash bug.

We believe this initial study supports our original assumption, and is encouraging for the viability of our approach. We next discuss our experience applying Mx to `redis` (§IV-B) and `lighttpd` (§IV-C).

B. Redis

`redis` is an advanced key-value data structure server[12], often refered to as one the most popular NoSQL databases. Due to its high-performance and low-resource requirements, `redis` is being used by many well-known services such as *GitHub*, *Digg* or *Flickr*.

Because the whole dataset is held in memory, reliability is critically important. However, like any other large software system, `redis` is often subject to crash bugs. Issue 344[13] is one such example. This issue causes `redis` to crash when the `HMGET` command is used with a wrong type. The bug was introduced during a code refactoring applied in revision `7fb16bac`. The original code of the problematic `hmgetCommand` function is shown in Listing 1, while the (buggy) refactored version is shown in Listing 2.

In the original code, if the key is found (line 1), but the type is not `REDIS_HASH` (line 9), the function returns after reporting an incorrect type (lines 10–11). However, in the refactored version (Listing 2), the `return` statement is missing, and after reporting an incorrect type (line 4), the function continues execution and crashes inside the `hashGet` function invoked on line 8. This is a critical bug, which may result in loosing some or even all of the stored data. The bug was introduced on April 13, 2010, diagnosed and reported only half a year later on October 12, 2010, and fixed on October 27, 2010.

Below, we describe how Mx can survive this bug while running in parallel the `redis` revision `a71f072f` (the *old version*, just before the bug was introduced) with revision `7fb16bac` (the *new version*, just after the bug). Mx first invokes SEA to perform a static analysis of the two binaries and construct the mappings described in Section III-C. Then, Mx invokes the MXM monitor, which executes both versions as child processes and intercepts their system calls.

When the new version crashes after issuing the problematic `HMGET` command, MXM intercepts the `SIGSEGV` signal which is being sent to the application by the operating system. At this point, REM starts the recovery procedure. First, REM sends a `SIGKILL` signal to the new version to terminate it. It then takes the last checkpoint of the new version, which was taken at the point of the last invoked system call, which in this case is an `epoll_ctl` system call. Then, REM uses the information provided by SEA to rewrite the stack of the new version, as detailed in Section III-B. In particular, REM replaces the return addresses of all functions in the new version with the corresponding addresses from the old version. REM also adds breakpoints at the beginning of all the functions in the code of the new version (to intercept indirect calls via function pointers), and then finally restores the original processor registers of the checkpointed process and restarts the execution of the (modified) new version.

[12]http://redis.io
[13]http://code.google.com/p/redis/issues/detail?id=344

```
robj *o = lookupKeyRead(c->db, c->argv[1]);
if (o == NULL) {
  addReplySds(c, sdscatprintf(sdsempty(), "*%d\r\n", c
      ->argc-2));
  for (i = 2; i < c->argc; i++) {
    addReply(c, shared.nullbulk);
  }
  return;
} else {
  if (o->type != REDIS_HASH) {
    addReply(c, shared.wrongtypeerr);
    return;
  }
}
addReplySds(c,sdscatprintf(sdsempty(),"*%d\r\n",c->
      argc-2));
```

Listing 1. Original (correct) version of the hmgetCommand function in redis

```
robj *o, *value;
o = lookupKeyRead(c->db, c->argv[1]);
if (o != NULL && o->type != REDIS_HASH) {
  addReply(c, shared.wrongtypeerr);
}
addReplySds(c, sdscatprintf(sdsempty(), "*%d\r\n", c->
      argc-2))
for (i = 2; i < c->argc; i++) {
  if (o != NULL && (value = hashGet(o, c->argv[i])) !=
        NULL) {
    addReplyBulk(c, value);
    decrRefCount(value);
  } else {
    addReply(c, shared.nullbulk);
  }
}
```

Listing 2. Refactored (buggy) version of the hmgetCommand function in redis

Since the checkpoint was performed right after the execution of system call `epoll_ctl`, the first thing that the code does is to return from the `libc` wrapper that performed this system call. This in turn will return to the corresponding code in the old version that invoked the wrapper, since all return addresses on the stack have been rewritten. From then on, the code of the old version is executed (but in the state of the new version), until the first system call is intercepted. In our example, the old and the new versions perform the same system call (and with the same arguments), so REM concludes that the two processes have re-converged, and thus restores back the code of the new version by performing the steps above in reverse, plus the additional step of synchronizing their global state (see §III-B). Finally, the control is handed back to the MXM monitor, which continues to monitor the execution of the two versions.

C. Lighttpd

To evaluate Mx on `lighttpd`, we have used two different crash bugs. The first bug is the one described in detail in Section II-A, related to the ETag and compression functionalities. As previously discussed, the crash is triggered by a very small change, which decrements the upper bound of a for loop by one. Mx successfully protects the application against this crash, and allows the new version to survive it by using the code of the old version. As we discuss in Section IV-D, Mx allows users to incorporate all the changes in the next 87 revisions following the buggy commit, while still protecting the overall application against this crash.

The other crash bug we reproduced[14] affects the URL rewrite functionality. This is also caused by an incorrect bound in a `for` loop. More precisely, the line:

```
for (k = 0; k < pattern_len; k++) ...
```

should have been

```
for (k = 0; k + 1 < pattern_len; k++) {
```

[14]http://redmine.lighttpd.net/projects/lighttpd/issues/2140

Application/Bug	Max distance	Time span
lighttpd/2169	87	2 months 2 days
lighttpd/2140	12	2 months 1 day
redis/344	27	6 days

Table I
THE MAXIMUM DISTANCE IN NO. OF VERSIONS AND THE TIME SPAN BETWEEN THE VERSIONS THAT CAN BE RUN BY MX FOR EACH BUG.

The bug seems to have been present since the initial codebase. It was reported in December 2009, and fixed one month later. As a result, we are running Mx using the last version containing the bug together with the one that fixed it. While this bug does not fit within the pattern targeted by Mx (where a newer revision introduces the bug), from a technical perspective it is equally challenging. Mx is able to successfully run the two versions in parallel, and help the old version survive the crash bug.

D. Ability to run distant versions

In the previous sections, we have showed how Mx can help software survive crash bugs, by running two *consecutive* versions of an application, one which suffers from the bug, and one which doesn't.

One important question is how far apart can the versions run by Mx be. To answer this question, we determined for each of the bugs discussed above the most distant versions that can be run together to survive the bug. Our results are shown in Table I. The most distant versions for the first `lighttpd` bug are approximately two months apart and have 87 versions in-between, while the most distant versions for the second `lighttpd` bug are approximately two months apart and have 12 versions in-between. Finally, the most distant versions for the `redis` bug are 27 versions apart and have 6 days in-between them. Of course, it is difficult to draw any general conclusions from only these three data points. Instead, we focus on understanding the reasons why Mx couldn't run farther apart versions for these bugs.

For `lighttpd` issue #2169, the lower bound is defined by a revision in which a pair of `geteuid()` and `getegid()` calls are replaced with a single call to `issetugid()` in order to allow `lighttpd` to start for a non-root user with GID 0. Mx cannot run this revision together with the one before it, because it does not support changes to the order of system calls. However, we believe this limitation could be overcome by using peephole optimizations [1], which would allow Mx to recognize that the pair `geteuid()` and `getegid()` could be matched with the call to `issetugid()`. The upper bound for `lighttpd` issue #2169 adds a `read` call to the `/dev/[u]random`, in order to provide a better entropy source for generating HTTP cookies. This additional `read` call changed the sequence of system calls, which Mx cannot handle.

For `lighttpd` issue #2140, both the lower and the upper bounds are caused by a change in a sequence of `read()` system calls. We believe this could be optimized by allowing Mx to recognize when two sequences of read system calls are used to perform the same overall read.

For the `redis` bug, the lower bound is given by the revision in which the `HMGET` command was first implemented. Since there was no support for `HMGET` before that version, Mx has no way to survive the crash caused by invoking `HMGET` with a wrong type (see §IV-B). The upper bound is defined by a revision which changes the way error responses are being constructed and reported, which results in a very different sequence of system calls.

E. Performance Overhead

To measure the performance overhead of our current implementation, we ran `lighttpd` and `redis` on their standard performance benchmarks. Mx incurs a very large performance overhead, of two to three orders of magnitude over native execution. However, our current prototype has not been optimized at all for performance, and we believe its overhead can be substantially reduced, as demonstrated by previous work using similar implementation-level mechanisms. For example, as we discuss in related work, our monitor MXM is very similar to the monitor used by Orchestra [25], which by employing various optimizations manages to obtain an average overhead of only about 15% when synchronizing two program variants at the level of system calls. In terms of checkpointing, the Rx system [21] implements a similar approach based on the Linux copy-on-write mechanism, and which through various optimizations manages to achieve a performance penalty of less than 5% when checkpointing every 200 milliseconds.

V. RELATED WORK

The idea of concurrently running multiple versions (or a *multi-version execution*) of the same application was first explored in the context of N-version programming, a software development methodology introduced in the 1970s in which multiple teams of programmers develop functionally equivalent versions of the same program in order to minimize the risk of having the same bugs in all versions [6].

Recently, N-version programing was explored by Cox et al. [9], who propose a general framework for increasing application security by running in parallel several automatically-generated *diversified variants* of the same program. The technique was implemented in two prototypes, one which runs the variants on different machines, and one which runs them on the same machine and synchronizes them at the system call level, using a modified Linux kernel.

Within this paradigm, the Orchestra framework [25], [26] uses a modified compiler to produce two versions of the same application with stacks growing in opposite directions, and runs them in parallel on top of an unprivileged user-space monitor, which synchronizes their execution at the level of system calls, and raises an alarm if any divergence is detected, which would have been triggered by a stack-based buffer overflow attack. Our MXM monitor (§III-A) has the same goals and a similar implementation to Orchestra's monitor.

Besides Orchestra, several other projects have proposed techniques that fit within the same paradigm. For example, Berger et al. use address space layout randomization to generate multiple replicas that are executed concurrently [3] and Shye et al. use multiple instances of the same application in order to overcome transient hardware failures [27].

There are two key differences between our approach and previous work in this space. First, we don't rely on automatically-generated variants, but instead propose to use multi-variant execution as a mechanism for improving software evolution. This also means that unlike previous solutions, the variants are not semantically equivalent—this eliminates the challenge of generating diversified variants and creates opportunities in terms of recovery from failures, but also introduces additional challenges in terms of synchronizing the execution of the different versions. Second, while previous work has focused on detecting divergences, our main concern is to *survive* them, in order to increase the security and availability of the overall application.

Previous work on improving the software update process has looked at different aspects related to managing and deploying new software versions. For example, Beattie et al. has looked at the issue of timing the application of security updates [2], while Crameri et al. has proposed a framework for staged deployment, in which user machines are clustered according to their environment and software updates are tested across clusters using several different strategies [11]. In relation to this work, Mx tries to encourage users to always apply a software update, but it would still benefit from effective update strategies in order to decide what versions to keep when resources are limited.

Mx's main focus in on surviving failures. Previous work in this area has employed several techniques to accomplish this goal. Rx [21] proposes an approach that helps programs recover from software failures by rolling back the program to a recent checkpoint upon a software failure, and then to re-executing it a modified environment. Mx similarly rolls back execution to a recent checkpoint, but instead of modifying the environment, it uses the code of a different version to survive the bug. The main limitation of Rx is that it can only recover from bugs which are (partly) caused by the interaction with the environment. In comparison, Mx does not have this drawback, but it places various limitations regarding the code changes between the two versions. The two approaches are complementary, and could be easily combined to support a larger number of errors and application types.

The Frost system [29] targets race errors, and shares similarities with both Rx and Mx. Like Rx, it changes the environment, in this case by using complementary thread schedules. Like Mx, it runs several versions in parallel and can survive errors by selecting the more reliable one according to some strategy. As previous multi-variant based approaches, Frost uses automatically-generated variants of the same program, while Mx uses different (manually-written) program versions.

Failure-oblivious computing [22], [23] helps applications survive memory errors by simply discarding invalid writes and fabricating values to return for invalid reads, enabling applications to continue their normal execution path. Similar to failure-oblivious computing, executions transactions help survive software bugs by terminating the function in which the bug has occurred and continuing on to execute the code immediately following the corresponding function call [28]. Our approach shares some of the philosophy of these two techniques, as we cannot always guarantee that the crashing version will correctly execute through the divergence by using the other version's code (see §III-B). However, by using a previously correct piece of code to execute through the crash, our approach provides stronger guarantees than those obtained by fabricating read values or terminating the function in which the bug occurred.

Research on automatic generation of filters based on vulnerability signatures [4], [7], [8] or on patch generation in deployed systems [13], [16], [20] also target applications with high-availability requirements, and the generated patches work by installing lightweight input filters or by changing the values of memory locations at runtime.

Recovery Oriented Computing [19] advocates the re-engineering of software systems to allow applications to recover from errors. Within this paradigm, microrebooting [5] proposes building systems out of individually recoverable components, which can be rebooted to survive bugs, without disturbing the rest of the application.

VI. Conclusion

Software updates are an important part of the software development and maintenance process. Unfortunately, they also present a high failure risk, and many users refuse to upgrade their software, relying instead on outdated versions, which often leave them exposed to known software bugs and security vulnerabilities.

In this paper we propose a novel approach designed to improve the software update process for applications with high availability and security requirements. Whenever a new program update becomes available, instead of upgrading the software to the newest version, we run the new version in parallel with the old, and carefully synchronize their execution to create a more secure and reliable multi-version application.

While our initial proof-of-concept prototype has several important limitations, our experience applying it to two real applications, `lighttpd` and `redis`, is encouraging, and we believe our approach can be improved to overcome many of the current limitations. Our ultimate goal is to enable users to benefit from the additional features and bug fixes provided by recent versions, without sacrificing the stability and security of older versions.

Acknowledgments

Petr Hosek is a recipient of the Google Europe Fellowship in Software Engineering, and this research is supported in part by this Google Fellowship. We thank Paul Marinescu for feedback on the text.

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Addison Wesley, 2006.

[2] S. Beattie, S. Arnold, C. Cowan, P. Wagle, and C. Wright, "Timing the application of security patches for optimal uptime," in *Proceedings of the 16th USENIX Conference on System Administration (LISA'02)*, Nov. 2002.

[3] E. D. Berger and B. G. Zorn, "Diehard: probabilistic memory safety for unsafe languages," in *Proc. of the Conference on Programing Language Design and Implementation (PLDI'06)*, Jun. 2006.

[4] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha, "Towards automatic generation of vulnerability-based signatures," in *Proc. of the IEEE Symposium on Security and Privacy (IEEE S&P'06)*, May 2006.

[5] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot—a technique for cheap recovery," in *Proc. of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI'04)*, Dec. 2004.

[6] L. Chen and A. Avizienis, "N-version programming: A fault-tolerance approach to reliability of software operation," in *Proc. of the 8th IEEE International Symposium on Fault Tolerant Computing (FTCS'78)*, Jun. 1978.

[7] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado, “Bouncer: securing software by blocking bad input,” in *Proc. of the 21st ACM Symposium on Operating Systems Principles (SOSP’07)*, Oct. 2007.

[8] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham, “Vigilante: end-to-end containment of Internet worms,” in *Proc. of the 20th ACM Symposium on Operating Systems Principles (SOSP’05)*, Oct. 2005.

[9] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, “N-variant systems: a secretless framework for security through diversity,” in *Proc. of the 15th USENIX Security Symposium (USENIX Security’06)*, July-August 2006.

[10] A. Cozzie, F. Stratton, H. Xue, and S. T. King, “Digging for data structures,” in *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI’08)*, Dec. 2008.

[11] O. Crameri, N. Knezevic, D. Kostic, R. Bianchini, and W. Zwaenepoel, “Staged deployment in mirage, an integrated software upgrade testing and distribution system,” in *Proc. of the 21st ACM Symposium on Operating Systems Principles (SOSP’07)*, Oct. 2007.

[12] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” in *Proc. of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI’04)*, Dec. 2004.

[13] B. Demsky and M. C. Rinard, “Automatic detection and repair of errors in data structures,” in *Proc. of the 18th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA’03)*, Oct. 2003.

[14] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, “Dryad: distributed data-parallel programs from sequential building blocks,” in *Proc. of the 2nd European Conference on Computer Systems (EuroSys’07)*, Mar. 2007.

[15] L. Jiang, G. Misherghi, and Z. Su, “Deckard: Scalable and accurate tree-based detection of code clones,” in *Proc. of the 29th International Conference on Software Engineering (ICSE’07)*, May 2007.

[16] H. Jula, D. M. Tralamazza, C. Zamfir, and G. Candea, “Deadlock immunity: Enabling systems to defend against deadlocks,” in *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI’08)*, Dec. 2008.

[17] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, “Cp-miner: Finding copy-paste and related bugs in large-scale software code,” *IEEE Transactions on Software Engineering*, vol. 32, pp. 176–192, 2006.

[18] Z. Lin, X. Zhang, and D. Xu, “Automatic reverse engineering of data structures from binary execution,” in *Proc. of the 17th Network and Distributed System Security Symposium (NDSS’10)*, February–March 2010.

[19] D. Patterson, A. Brown, P. Broadwell, G. Candea, G. C, M. Chen, J. Cutler, Armando, P. Enriquez, O. Fox, D. Oppenheimer, E. Kcman, M. Merzbacher, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft, “Recovery oriented computing (roc): Motivation, definition, techniques, and case studies,” UC Berkeley, Computer Science Dept., Tech. Rep., 2002.

[20] 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 22nd ACM Symposium on Operating Systems Principles (SOSP’09)*, Oct. 2009.

[21] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou, “Rx: treating bugs as allergies—a safe method to survive software failures,” in *Proc. of the 20th ACM Symposium on Operating Systems Principles (SOSP’05)*, Oct. 2005.

[22] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and J. William S. Beebee, “Enhancing server availability and security through failure-oblivious computing,” in *Proc. of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI’04)*, Dec. 2004.

[23] M. Rinard, C. Cadar, and H. H. Nguyen, “Exploring the acceptability envelope,” in *Companion to the 20th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA’05 Companion)*, Oct. 2005.

[24] D. T. Ross, “The AED free storage package,” *Communications of the Association for Computing Machinery (CACM)*, vol. 10, pp. 481–492, Aug. 1967.

[25] B. Salamat, T. Jackson, A. Gal, and M. Franz, “Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space,” in *Proc. of the 4th European Conference on Computer Systems (EuroSys’09)*, March-April 2009.

[26] B. Salamat, T. Jackson, G. Wagner, C. Wimmer, and M. Franz, “Run-time defense against code injection attacks using replicated execution,” *IEEE Transactions on Dependable and Secure Computing*, vol. 8, pp. 588–601, 2011.

[27] A. Shye, J. Blomstedt, T. Moseley, V. J. Reddi, and D. A. Connors, “PLR: A Software Approach to Transient Fault Tolerance for Multicore Architectures,” *IEEE Transactions on Dependable and Secure Computing*, vol. 6, pp. 135–148, 2009.

[28] S. Sidiroglou and A. D. Keromytis, “Execution transactions for defending against software failures: Use and evaluation,” *Springer International Journal of Information Security (IJIS)*, vol. 5, no. 2, pp. 77–91, 2006.

[29] K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy, “Detecting and surviving data races using complementary schedules,” in *Proc. of the 23rd ACM Symposium on Operating Systems Principles (SOSP’11)*, Oct. 2011.

[30] T. White, *Hadoop: The Definitive Guide*, 1st ed. O’Reilly Media, 2009.