Safe Software Updates via Multi-version Execution

Petr Hosek Cristian Cadar
Department of Computing
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 one; by carefully coordinating their executions and selecting the behaviour of the more reliable version when they diverge, we create a more secure and dependable multi-version application.

We implemented this technique in MX, a system targeting Linux applications running on multi-core processors, and show that it can be applied successfully to several real applications such as *Coreutils*, a set of user-level UNIX applications; *Lighttpd*, a popular web server used by several high-traffic websites such as Wikipedia and YouTube; and *Redis*, an advanced key-value data structure server used by many well-known services such as GitHub and Flickr.

***Index Terms*—multi-version execution, software updates, surviving software crashes.**

I. Introduction

In this paper, we propose a novel technique for improving the reliability and security of software updates, which takes advantage of the idle resources made available by multi-core platforms. 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a high failure rate, with many users and administrators refusing to upgrade their software and relying instead on outdated versions, which often leaves them exposed to critical bugs and security vulnerabilities.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 performing a software upgrade, regardless of their experience level [13].

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 the code. For example, a recent study of software updates in commercial and open-source operating systems has shown that at least 14.8% to 25% of fixes are incorrect and have affected end-users [32]. 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 one. 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 and secure as each individual version by itself.

We implemented this approach in a prototype system called MX, which targets crash bugs in Linux applications running on multi-core processors. MX allows a new and an old version of an application to run concurrently, without requiring any modifications to the application itself or the operating system, nor any input from the user. The synchronisation of the two versions is performed at the system call level, using system call interposition and synchronisation. When one of the versions crashes, MX transparently restarts it via a lightweight checkpointing mechanism and often allows it to survive the bug by using the code of the other version.

We evaluate MX by showing that it can successfully survive crashes in several real applications, specifically several *Coreutils* utilities and two popular servers, *Lighttpd* and *Redis*. In summary, the main contributions of this paper are:

1) A novel software update approach based on multi-version execution which allows applications to survive crash errors introduced by incorrect software updates;
2) A study of the evolution of application external behaviour confirming the feasibility of our approach;
3) A prototype system for multi-core x86 and x86-64 Linux systems which implements this approach without requiring any changes to the application binaries, nor the operating system kernel;
4) An evaluation of our prototype on several real applications: *Coreutils*, a set of user-level UNIX utilities; *Lighttpd*, a popular web server used by several high-traffic websites such as Wikipedia and YouTube; and *Redis*, an advanced key-value data structure server used by many well-known services such as GitHub and Flickr.

The rest of this paper is organised as follows. §II introduces our approach through a real update scenario in *Lighttpd*. Then, §III presents a study analysing the feasibility of our approach, §IV describes our MX prototype targeting Linux applications running on multi-core processors, and §V presents our experience applying MX to several real applications. Finally, §VI discusses the different trade-offs involved in our approach, §VII presents related work and §VIII concludes.

II. Motivating Example

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

978-1-4673-3076-3/13 © 2013 IEEE

ICSE 2013, San Francisco, CA, USA

Lighttpd[1] is a popular open-source web-server used by several high-traffic websites such as Wikipedia and YouTube. 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 precisely, *Lighttpd*'s support for HTTP compression uses caching to avoid re-compressing files which have not changed since the last access. To determine whether the cached file is still valid, *Lighttpd* internally 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 bottom line is that for about one year, users affected by this buggy patch 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.

Our approach provides users with a third choice; when a new version arrives, instead of replacing the old version, we run both versions in parallel. In our example, consider that we are using MX to run a version of *Lighttpd* from March 2009. When the buggy April 2010 version is released, MX runs it in parallel with the old one. As the two versions execute:

- As long as the two versions have the same external behaviour (e.g. they write the same values into the same files, or send the same data over the network), they are run side-by-side and MX ensures that they act as one to the outside world (see §IV-A);
- When one of the versions crashes (e.g. the new version executes the buggy patch), MX will patch the crashing version at runtime using the behaviour of the non-crashing version (see §IV-B). In this way, MX can successfully survive crash bugs in both the old and the new version, increasing the reliability and availability of the overall application;
- When a non-crashing divergence is detected, MX will discard one of the versions (by default the old one, but other heuristics can be used). The other version can be later restarted at a convenient synchronisation point (e.g. at the beginning of the dispatch loop of a network server).

From the user's point of view, this process is completely transparent and does not cause any interruption in service. In our example, this effectively eliminates the bug in *Lighttpd*, while still allowing users to use the latest features and bug fixes of the recent versions.

III. Feasibility Study

Our approach is based largely on the assumption that during software evolution, the changes to the external behaviour of an application are relatively small. In the context of Linux applications, the external behaviour of an application consists of its sequence of system calls, which are the primary mechanism for an application to change the state of its environment. Note that the key insight here is that we are only concerned with *externally observable behaviour*, and are oblivious to the way the external behaviour 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.

To verify this assumption, we compared 164 successive revisions of the *Lighttpd* web server, namely revisions in the range 2379–2635 of branch `lighttpd-1.4.x`, 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. During this period, code patches in *Lighttpd* varied between 1 and 2959 LOC, with a median value of 33 LOC.

To compare the external behaviour of each version, we traced the system calls made by these versions using the `strace`[6] tool, while running all the tests from the *Lighttpd* regression suite targeting the core functionality (a total of seven tests, but each test contains a large number of test cases issuing 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.

The system call traces were further normalised and post-processed. We first split the original trace on a per-process basis, and normalised all differences caused by timing (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,

[1] http://www.lighttpd.net/
[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
[6] http://sourceforge.net/projects/strace/

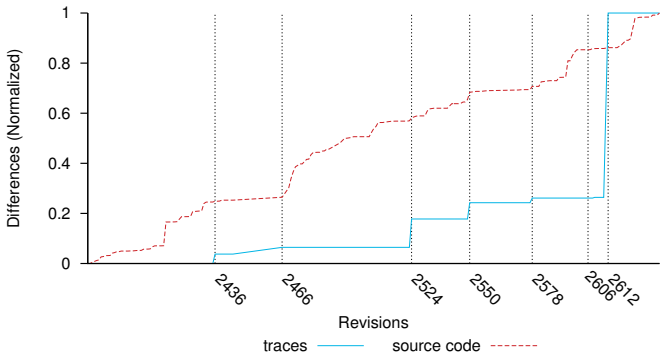

Fig. 1. Correlation of differences in post-processed system call traces with differences in source code across 164 revisions of *Lighttpd*. The seven named revisions are the only ones introducing external behaviour changes.

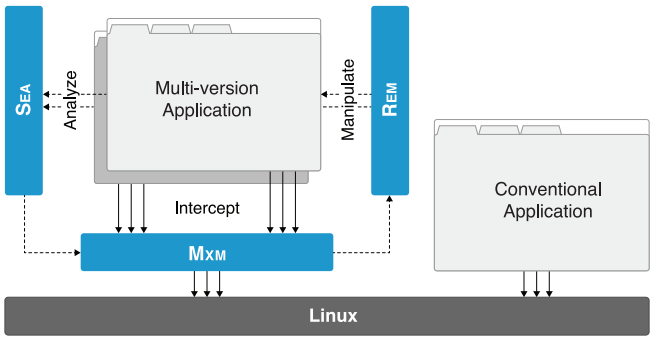

Fig. 2. MX system architecture.

but we performed it because many system calls accept as arguments addresses of data structures residing in the virtual address space, and these addresses may differ across versions (but MX handles this while mediating the effect of system calls, as described in §IV-A). Finally, for each test case, we compared the traces of consecutive *Lighttpd* versions using the edit distance.

Our results are shown in Figure 1, which correlates the differences in post-processed system call traces with the source code changes. The graph shows that changes in externally observable behaviour occur only sporadically. In fact, 156 versions (which account for around 95% of all the versions considered) introduce *no changes* in external behaviour. In particular, the revision which introduced the bug described in §II is one of the versions that introduces no changes, yet this revision is responsible for a critical crash bug.

IV. MX Prototype System

We have implemented our approach in a prototype system called MX, targeted at multi-core processors running Linux. Currently, MX works with only two application versions, but we are adding support for arbitrarily many versions. The system works directly on application binaries, making it easy to deploy it and possibly integrate it with existing software package managers such as `apt` or `yum`.

On a platform using MX, 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. 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 of Figure 2. The input to MX consists of the binaries of two versions of an application, which we will 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 analysed by the SEA (Static Executable Analyser) component, which constructs a mapping from the control flow graph (CFG) of the old version to the CFG of the new version (§IV-C). The two versions are then passed to MXM (Multi-eXecution Monitor), whose job is to run the two versions in parallel, synchronise their execution, virtualise their interaction with the outside environment, and detect any divergences in their external behaviour (§IV-A). Once a divergence is detected, it is resolved by REM (Runtime Execution Manipulator), which selects between the available behaviours, and resynchronises the two versions after the divergence (§IV-B). 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, synchronise their executions, and detect any divergences in their external behaviour. 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 synchronised fashion and act as one to the outside world.

MXM is implemented using the `ptrace` interface provided by the Linux kernel. 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 synchronise applications at the level of system calls [20], [24].

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 (§IV-B).

Otherwise, if the two versions perform the same system call with the same arguments, MXM virtualises their interaction with the environment. If the operation performed by the system call has no side effects and does not involve virtualised state (e.g. `sysinfo`), MXM allows both processes to execute it independently. Otherwise, it executes the system call on their behalf and copies its results into the address spaces of both versions.

MXM must also enforce deterministic execution across versions. This consists mainly of intercepting instructions that may produce non-deterministic results, and returning the same result in both versions. Examples of such non-deterministic operations include random number generators (e.g. read calls to `/dev/[u]random`), date and time (e.g. read calls to `/etc/localtime`), and access to files and network (e.g. file descriptor consistency). Note that non-deterministic effects resulting from allocating memory objects at different addresses in memory or randomly arranging memory areas via address space layout randomisation (ASLR) do not pose any problems: MXM understand the semantics of individual system calls and rather than directly comparing memory addresses (which might be different in each executed version), it compares the actual values stored at those memory locations.

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 complicated semantics to pass values to the operating system kernel, as in the case of `ioctl` or `futex`. To be able to compare the parameters of these system calls and copy back their results, MXM needs to understand the semantics of these structures. However, there are only a relatively small number of system calls in Linux, and once the support for handling them is implemented, it can be reused across applications. MXM currently implements 131 system calls (out of the 311 provided by Linux x86-64 3.1.9), which was enough to allow us to run MX on our benchmarks (§V).

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 contents pointed to by these structures to its own address space in order to perform their comparison. The `ptrace` interface on x86-64 only allows to copy one quadword per system call, which is very expensive. Previous approaches either used various ad-hoc optimisations [24] such as named pipes or shared memory with custom shellcode, or a modified kernel [20] to overcome this limitation. Instead, MXM uses *cross memory attach*, a new mechanism for fast interprocess communication which has been recently added to the Linux kernel [10].

Finally, a particular challenge arises in the context of multi-process and multithreaded applications. Using a single monitor instance to intercept both versions and their child processes (or threads) would eliminate any advantage that these applications derive from using concurrency. Therefore, MXM uses a new monitor thread for each set of child processes (or threads) spawned by the application. 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.

MXM does not enforce deterministic execution across multiple versions of multithreaded programs (which may diverge if race conditions can lead to different external behaviour across executions), although we could overcome this limitation by combining it with recently proposed deterministic multithreading systems such as DTHREADS [18].

B. REM: *Runtime Execution Manipulator*

At the core of our system lies the REM component, which is invoked by MXM whenever a divergence is detected. REM has two main jobs: (1) to decide whether to resolve the divergence in favour of the old or the new version; and (2) to allow the other version to execute through the divergence and resynchronise the execution of the two versions after the divergence. As discussed before, in this paper we focus our attention on 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 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 at the same time).

We emphasise that we apply our approach only to crash errors (those raising a `SIGSEGV` signal), and not to other types of program termination, such as `abort`. This is important from a security perspective, because when a vulnerability is discovered, but a proper solution is not yet known, developers often fail-stop the program rather than letting it continue and allowing the attacker to compromise the system.

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 do a form of *runtime code patching*, in which we use the code of the non-crashing version to execute over the buggy code in the crashing version.

Our exact recovery mechanism is illustrated in Figure 3. 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 3, 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 buggy code using v_1's code (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

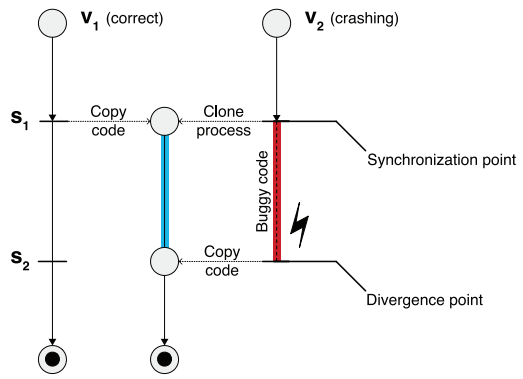

Fig. 3. REM's recovery mechanism uses the code of the non-crashing version to run through the buggy code.

of the `objcopy` tool.[7] However, in the future we plan to implement this transparently by using the cross-memory attach mechanism used by MXM.

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 (§IV-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 buggy code, REM performs the same operations in reverse: it redirects execution to v_2's 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.

Note that MX cannot currently 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 the system with the ability to perform full stack reconstruction [19] and automatically infer basic data structure changes at the binary-level [12].

Our approach of using the code of the non-crashing version to survive failures in the crashing one may potentially leave the recovered version in an inconsistent state. However, MX is able to discover most internal state inconsistencies by comparing whether the two versions have the same external behaviour. When the behaviour of the recovered version starts to differ, MX will immediately discard it and continue with only one version to ensure correctness. The discarded version can be later restarted at a convenient synchronisation point. This restarting functionality is not currently implemented in MX, but we plan to add it as a future extension.

C. SEA: Static Executable Analyser

The SEA component statically analyses the binaries of the two versions to obtain information needed at runtime by the MXM and REM components. SEA is invoked only once, when the multi-version application is assembled from its component versions.

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 functions 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 (§IV-B).

Second, SEA computes a mapping from all possible return addresses in one version to the corresponding return addresses in the other version. 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 them. The implementation is based on the `libbfd` and `libopcodes` libraries, 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 functions, SEA uses the `libbf` library,[10] built on top of `libopcodes`.

V. EVALUATION

To evaluate our approach, we show that MX can survive crash bugs in several real applications: *GNU Coreutils* (§V-A), *Redis* (§V-B) and *Lighttpd* (§V-C). We then examine the question

[7] http://sourceware.org/binutils/docs/binutils/objcopy.html
[8] http://www.nongnu.org/libunwind/
[9] http://www.gnu.org/software/binutils/
[10] http://github.com/petrh/libbf

TABLE I
UTILITIES FROM *GNU Coreutils*, THE CRASH BUGS USED, AND THE VERSIONS IN WHICH THESE BUGS WERE INTRODUCED AND FIXED. WE GROUP TOGETHER UTILITIES AFFECTED BY THE SAME OR SIMILAR BUGS.

Utility	Bug description	Bug span
`md5sum` `sha1sum`	Buffer underflow	v5.1 – v6.11
`mkdir` `mkfifo` `mknod`	`NULL`-pointer dereference	v5.1 – v6.11
`cut`	Buffer overflow	v5.3 – v8.11

of how far apart can be the versions run by MX (§V-D), and discuss MX's performance overhead (§V-E).

A. Coreutils

As an initial evaluation of MX's ability to survive crashes, we have used applications from the *GNU Coreutils* utility suite,[11] which provides the core user-level environment on most UNIX systems. We have selected a number of bugs reported on the *Coreutils* mailing list, all of which trigger segmentation faults. The bugs are described in Table I, together with the utilities affected by each bug and the versions in which they were introduced and fixed.

For all these bugs, we configured MX to run the version that fixed the bug together with the one just before. MX successfully intercepted the crash and recovered the execution by using the strategy described in §IV-B.

B. Redis

Redis is an advanced key-value data structure server,[12] used by many well-known services such as GitHub and Flickr. Because the whole dataset is held in memory, reliability is critically important, as a crash could result in total data loss. 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 the 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 lookup on line 1 is successful, 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 losing some or even all of the stored data. The bug was introduced in April 2010, diagnosed and reported only half a year later in October 2010 and then fixed after fifteen days.

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 §IV-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 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 §IV-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.

Since the checkpoint was performed right after the execution of the 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 synchronising their global state (see §IV-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 §II, 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.

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

`for (k=0; k < pattern_len; k++)` should have been

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

The bug seems to have been present since the very first version added to the repository. 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

[11] http://www.gnu.org/software/coreutils/

[12] http://redis.io/

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

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

```
robj *o = lookupKeyRead(c->db, c->argv[1]);
if (o == NULL) {
  addReplySds(c,sdscatprintf(sdsempty(),"*
  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(),"*
```

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(),"*
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*.

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 shown 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 does not. One important question is how far apart can be the versions run by MX. To answer this question, we determined for each of the bugs discussed above the most distant revisions that can be run together to survive the bug.

For the *Coreutils* benchmarks, we are able to run versions which are hundreds of revisions apart: 1,124 revisions (corresponding to over one year and seven months of development time) for the `md5sum`/`sha1sum` bug; 2,937 revisions (over four years of development time) for the `mkdir`/`mkfifo`/`mknod` bug; and 1,201 revisions (over two years and three months of development time) for the `cut` bug.

The most distant versions for the first *Lighttpd* bug are approximately two months apart and have 87 revisions in-between, while the most distant versions for the second *Lighttpd* bug are also approximately two months apart but have only 12 revisions in-between. Finally, the most distant versions for the *Redis* bug are 27 revisions and 6 days apart.

Of course, it is difficult to draw any general conclusions from only this small number of data points. Instead, we focus on understanding the reasons why MX couldn't run farther apart versions for the bugs in *Lighttpd* and *Redis* (we ignore *Coreutils*, for which we can run very distant versions). 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()` to allow *Lighttpd* to start for a non-root user with GID 0. MX currently does not support changes to the order of system calls, but we believe this limitation could be overcome by using peephole-style optimisations [1], which would allow MX to recognise 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 `/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 optimised by allowing MX to recognise 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. 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

We ran our experiments on a four-core server with 3.50 GHz Intel Xeon E3 and 16 GB of RAM running 64-bit Linux v3.1.9.

SPEC CPU2006. To measure the performance overhead of our prototype, we first used the standard *SPEC CPU2006*[15] benchmark suite. Figure 4 shows the performance of MX running two instances of the same application in parallel, compared to a native system. The execution time overhead of MX varies from 3.43% to 105.16% compared to executing just a single version, with the geometric mean at 17.91%.

Coreutils. The six *Coreutils* applications discussed in §V-A are mostly used in an interactive fashion via the command-line interface (CLI). For such applications, a high performance overhead is acceptable as long as it is not perceptible to the user; prior studies have shown that response times of less than 100ms typically feel instantaneous [7]. In many common use cases (e.g. creating a directory, or using `cut` on a small text file), the overhead of MX was imperceptible—e.g. creating a directory takes around 1ms natively and 4ms with MX. For the three utilities that process files, we calculated the maximum file size for which the response time with MX stays under the 100ms threshold. For `cut`, the maximum file size is 1.10MB (with an overhead of 14.08×), for `md5sum` 1.25MB (16.23× overhead), and for `sha1sum` 1.22MB (12.00× overhead).

***Redis* and *Lighttpd*.** To measure the performance overhead for *Redis*, we used the `redis-benchmark`[16] utility, which is part of the standard *Redis* distribution and simulates `GET`/`SET`

[15] http://www.spec.org/cpu2006/

[16] http://redis.io/topics/benchmarks

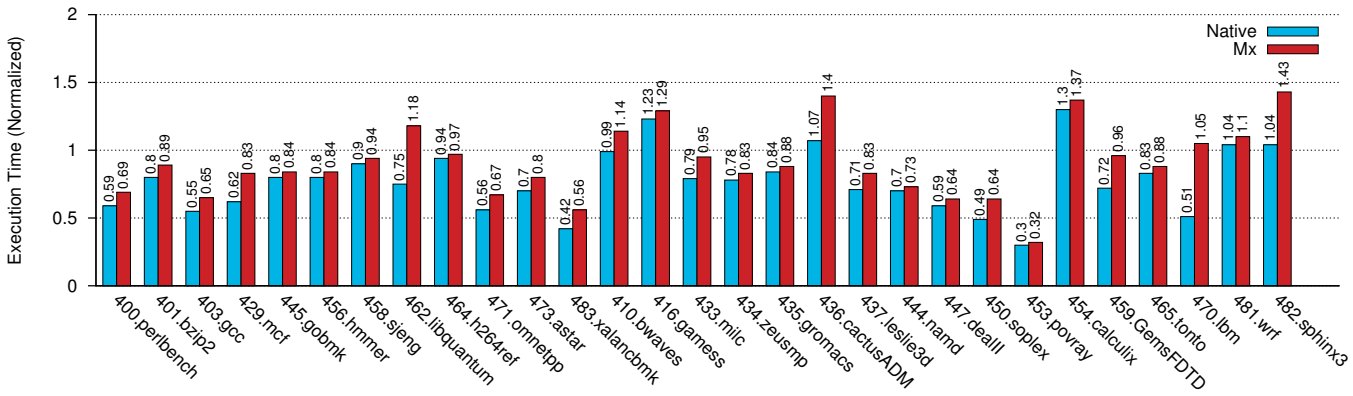

Fig. 4. Normalised execution times for the *SPEC CPU2006* benchmark suite running under MX.

operations done by N clients concurrently, with default workload. For *Lighttpd*, we used the `http_load`[17] multiprocessing test client that is also used by the *Lighttpd* developers. Both of these standard benchmarks measure the end-to-end time as perceived by users. As a result, we performed two sets of experiments: (1) with the client and server located on the same machine, which represents the worst case performance-wise for MX; and (2) with the client and server located on different continents (one in England and the other in California), which represents the best case.

The overhead for *Redis* varies, depending on the operation being performed, from $1.00\times$ to $1.05\times$ in the remote scenario, and from $3.74\times$ to $16.72\times$ in the local scenario. The overhead for *Lighttpd* varies from $1.01\times$ to $1.04\times$ in the remote scenario, and from $2.60\times$ to $3.49\times$ in the local scenario. Despite the relatively large overhead in the local experiments, the remote overhead is negligible because times are dominated by the network latency (which in our case is over 150ms).

As a result, we believe MX is most suitable for scenarios for which its execution overhead does not degrade the performance of the end-to-end task, such as the remote *Redis* and *Lighttpd* scenarios discussed above, or interactive tasks such as those performed using command-line utilities, where users would not notice the overhead as long as the response time stays within a certain range.

Finally, we would like to emphasise that our current prototype has not been optimised for performance, and we believe its overhead can still be significantly reduced. For example, we could synchronise versions at a coarser granularity, by using an epoch-based approach [29], or we could improve our checkpointing mechanism by implementing it as a loadable kernel module that only stores the part of the state needed for recovery [27].

VI. Discussion

This section discusses in more detail the scope of our approach with regard to the type of software updates suitable to multi-version execution and the different trade-offs involved.

[17]http://www.acme.com/software/http_load/

Types of code changes. In order for MX to be successful, the external behaviour of the versions that are run in parallel has to be similar enough to allow us to synchronise their execution. Our empirical study in §III shows that changes to the external behaviour of an application are often minimal, so our approach should work well with versions that are not too distant from one another. Similarly, our system relies on the assumption that versions re-converge to the same behaviour after a divergence. As a result, we believe MX would be a good fit for 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. Our approach is also suitable to local code changes, which have small propagation distances, thus ensuring that the different versions will eventually re-converge to the same behaviour.

Trade-offs involved. Our approach is targeted toward scenarios where the availability, reliability and security of a software system is more important than strict correctness, high performance and low energy consumption.

In terms of correctness guarantees, MX is similar to previous approaches such as failure oblivious computing [23] which may sacrifice strict correctness for increased availability and security (see §IV-B for details regarding possible problems caused by MX). However, MX alleviates this problem by using a previously correct piece of code to execute through the crash, and by discovering most potential problems by regularly checking if the two versions have the same external behaviour. Finally, note that MX always reverts to running a single version when a non-resolvable divergence is detected.

MX incurs a performance overhead, as discussed in §V-E. In our experience, MX is readily deployable to interactive applications such as command-line utilities, text editors and other office tools, where the performance degradation is not noticeable to the user. We believe it is also applicable to server applications where availability is more important than high performance. MX is not applicable to patches that fix performance bugs, as the system runs no faster than the slowest version.

Our approach of using idle CPU time to run additional versions also increases energy consumption. However, it is

interesting to note that idle CPUs are not "free" either: even without considering the initial cost of purchasing the cores left idle, an energy-efficient server consumes half its full power when doing virtually no work—and for other servers, this ratio is usually much worse [2].

VII. Related Work

We have introduced the idea of multi-version software updates in a HotSWUp workshop position paper [5].

N-version programming paradigm. The original idea of concurrently running multiple versions of the same application was first explored in the context of N-version programming, a software development approach introduced in the 1970s in which multiple teams of programmers independently develop functionally equivalent versions of the same program in order to minimise the risk of having the same bugs in all versions [8]. During runtime, these versions are executed in parallel and majority voting is used to continue in the best possible way when a divergence occurs.

Cook and Dage [9] proposed a multi-version framework for upgrading components. Users formally specify the specific input subdomain that each component version should handle, after which versions are run in parallel and the output of the version whose domain includes the current input is selected as the overall output of the computation. The system was implemented at the level of leaf procedures in the Tcl language. The key difference with MX is that this framework requires a formal description of what input domain should be handled by each version; in comparison, MX targets crash bugs and is fully automatic. Moreover, MX's goal is to have all versions alive at all times, so crash recovery plays a key role. Finally, MX has to carefully synchronise access to shared state, which is not an issue at the level of Tcl leaf procedures.

More recently, researchers have proposed additional techniques that fit within the N-version programming paradigm, e.g. by using heap over-provisioning and full randomisation of object placement and memory reuse to run multiple replicas and reduce the likelihood that a memory error will have any effect [4], employing complementary thread schedules to survive concurrency errors [29], or using genetic programming to automatically generate a large number of application variants that can be combined to reduce the probability of failure or improve various non-functional requirements [15]. We have also argued that automatically generated software variants are a good way for exploiting the highly parallel nature of modern hardware platforms [6].

Cox et al. [11] 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 in which variants are run on different machines, and one in which they are run on the same machine and synchronised at the system call level, using a modified Linux kernel. Within this paradigm, the Orchestra framework [24] uses a modified compiler to produce two versions of the same application with stacks growing in opposite directions, runs them in parallel on top of an unprivileged user-space monitor, and raises an alarm if any divergence is detected to protect against stack-based buffer overflow attacks.

There are two key differences between our approach and the work discussed in the last two paragraphs. First, we do not rely on automatically-generated variants, but instead run in parallel existing software versions, which raises a number of different technical challenges. Second, this body of work has mostly focused on detecting divergences, while our main concern is to *survive* them (keeping all versions alive), in order to increase both the security and availability of the overall application.

An approach closer to the original N-version programming paradigm is Cocktail [31], which proposes the idea of running different web browsers in parallel under the assumption that any two of them are unlikely to be vulnerable to the same attacks. Compared to MX and other techniques inspired by the N-version programming paradigm, Cocktail's task is simplified by exclusively targeting web browsers, which implement common web standards.

Online and offline testing. Back-to-back testing [30], where the same input is sent to different variants or versions of an application and their outputs compared for equivalence, has been used since the 1970s. More recently, delta execution [28] proposes to run two different versions of a single application, splitting the execution at points where the two versions differ, and comparing their behaviour to test the patch for errors and validate its functionality. Band-aid patching [25] proposes an online patch testing system that also splits execution before a patch, and then retroactively selects one code version based on certain criteria. Similarly, Tachyon [20] is an online patch testing system in which the old and the new version of an application are run concurrently; when a divergence is detected, the options are to either halt the program, or to create a manual rewrite rule specifying how to handle the divergence.

The idea of running multiple executions concurrently has also been used in an offline testing context. For instance, d'Amorin et al. [14] optimise the state-space exploration of object oriented code by running the same program on multiple inputs simultaneously, while Kim et al. [17] improve the testing of software product lines by sharing executions across a program family.

By comparison with this body of work, our focus is on managing divergences across software versions at runtime in order to keep the application running, and therefore runtime deployment and automatic crash recovery play a central role in MX.

Software updating. Dynamic software updating (DSU) systems such as Ginseng [21], UpStare [19] or Kitsune [16] are concerned with the problem of updating programs while they are running. As opposed to MX, the two versions co-exist only for the duration of the software update, but DSU and the REM component of MX face similar challenges when switching execution from one version to another. We hope that some of the technique developed in DSU research will also benefit the recovery mechanism of MX and vice versa.

Other prior work on improving software updating has looked at different aspects related to managing and deploying new software versions. For example, Beattie et al. [3] have considered the issue of timing the application of security updates, while Crameri et al. [13] 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. In relation to this work, MX tries to encourage users to always apply a software update, but it would still benefit from effective strategies to decide what versions to keep when resources are limited.

Surviving software failures. MX's main focus is on surviving errors. Prior work in this area has employed several techniques to accomplish this goal. For example, Rx [22] helps applications recover from software failures by rolling back the program to a recent checkpoint upon a software failure, and then re-executing it in 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 two approaches are complementary, and could be easily combined to support a larger number of errors and application types.

Failure-oblivious computing [23] helps software 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, execution transactions [26] help survive software bugs by terminating the function in which the bug has occurred and continuing to execute the code immediately following the corresponding function call. 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 when using the other version's code. However, by using a previously correct piece of code to execute through the crash and regularly checking for divergences in the external behaviour, our approach provides stronger guarantees than those obtained by fabricating read values or terminating the function in which the bug occurred.

VIII. 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 have proposed a novel multi-version execution approach for improving the software update process. 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 one, and carefully synchronise their execution to create a more secure and reliable multi-version application.

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.

Acknowledgements

We would like to thank our reviewers and colleagues for their feedback on the paper. Petr Hosek is a recipient of the Google Europe Fellowship in Software Engineering, and this research is supported in part by this Google Fellowship.

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] L. A. Barroso and U. Hölzle, "The case for energy-proportional computing," *Computer*, vol. 40, pp. 33–37, 2007.
[3] S. Beattie, S. Arnold, C. Cowan, P. Wagle, and C. Wright, "Timing the application of security patches for optimal uptime," in *LISA'02*.
[4] E. D. Berger and B. G. Zorn, "Diehard: probabilistic memory safety for unsafe languages," in *PLDI'06*.
[5] C. Cadar and P. Hosek, "Multi-version software updates," in *HotSWUp'12*.
[6] C. Cadar, P. Pietzuch, and A. L. Wolf, "Multiplicity computing: A vision of software engineering for next-generation computing platform applications," in *FoSER'10*.
[7] S. Card, T. Moran, and A. Newell, "The model human processor: an engineering model for human performance," *Handbook of Perception and Human Performance*, vol. 2, pp. 1–35, 1986.
[8] L. Chen and A. Avizienis, "N-version programming: A fault-tolerance approach to reliability of software operation," in *FTCS'78*.
[9] J. E. Cook and J. A. Dage, "Highly reliable upgrading of components," in *ICSE'99*.
[10] J. Corbet, "Cross memory attach," http://lwn.net/Articles/405346/, 2010.
[11] 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 *USENIX Security'06*.
[12] A. Cozzie, F. Stratton, H. Xue, and S. T. King, "Digging for data structures," in *OSDI'08*.
[13] O. Crameri, N. Knezevic, D. Kostic, R. Bianchini, and W. Zwaenepoel, "Staged deployment in Mirage, an integrated software upgrade testing and distribution system," in *SOSP'07*.
[14] M. d'Amorim, S. Lauterburg, and D. Marinov, "Delta execution for efficient state-space exploration of object-oriented programs," in *ISSTA'07*.
[15] M. Harman, W. Langdon, Y. Jia, D. White, A. Arcuri, and J. Clark, "The GISMOE challenge: constructing the pareto program surface using genetic programming to find better programs," in *ASE'12*.
[16] C. M. Hayden, E. K. Smith, M. Denchev, M. Hicks, and J. S. Foster, "Kitsune: Efficient, general-purpose dynamic software updating for C," in *OOPSLA'12*.
[17] C. H. P. Kim, S. Khurshid, and D. Batory, "Shared execution for efficiently testing product lines," in *ISSRE'12*.
[18] T. Liu, C. Curtsinger, and E. D. Berger, "DTHREADS: Efficient deterministic multithreading," in *SOSP'11*.
[19] K. Makris and R. A. Bazi, "Immediate multi-threaded dynamic software updates using stack reconstruction," in *USENIX ATC'09*.
[20] M. Maurer and D. Brumley, "TACHYON: Tandem execution for efficient live patch testing," in *USENIX Security'12*.
[21] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol, "Practical dynamic software updating for C," in *PLDI'06*.
[22] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou, "Rx: treating bugs as allergies—a safe method to survive software failures," in *SOSP'05*.
[23] 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 *OSDI'04*.
[24] B. Salamat, T. Jackson, A. Gal, and M. Franz, "Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space," in *EuroSys'09*.
[25] S. Sidiroglou, S. Ioannidis, and A. D. Keromytis, "Band-aid patching," in *HotDep'07*.
[26] S. Sidiroglou and A. D. Keromytis, "Execution transactions for defending against software failures: Use and evaluation," *IJIS*, vol. 5, no. 2, pp. 77–91, 2006.
[27] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou, "Flashback: A lightweight extension for rollback and deterministic replay for software debugging," in *USENIX ATC'04*.
[28] J. Tucek, W. Xiong, and Y. Zhou, "Efficient online validation with delta execution," in *ASPLOS'09*.
[29] K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy, "Detecting and surviving data races using complementary schedules," in *SOSP'11*.
[30] M. A. Vouk, "Back-to-back testing," *IST*, vol. 32, pp. 34–45, Jan.-Feb. 1990.
[31] H. Xue, N. Dautenhahn, and S. T. King, "Using replicated execution for a more secure and reliable web browser," in *NDSS'12*.
[32] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram, "How do fixes become bugs?" in *ESEC/FSE'11*.