Vrije Universiteit Amsterdam

Master's Thesis

Safe and Automatic Live Update for Unix Applications

Author:
Călin Iorgulescu

Supervisor:
Professor Andrew S. Tanenbaum

Second Reader:
Cristiano Giuffrida

Secure and Reliable Systems
Department of Computer Sciences

August 2013

“The last thing one knows in constructing a work is what to put first.”

Blaise Pascal

VRIJE UNIVERSITEIT AMSTERDAM

Abstract

Safe and Automatic Live Update for Unix Applications

by Călin IORGULESCU

The accelerated pace of software development is continuously shaping the software engineering world, with an ever-growing stream of updates being released every minute. However, the integration of an update (even one that addresses a crucial problem) is still slow and cumbersome due, in great part, to the potentially high risk of service disruption and downtime, which is unacceptable for many systems. For example, a study from 2002 [1] showed that over 10% of security patches introduce new software bugs of their own.

Live update is a technique that aims to eliminate the overhead and downtime of applying an update, making the process seamless to both the clients of the service and the system administrators. Unfortunately, current state-of-the-art techniques require considerable manual effort, only support trivial updates, or are prone to errors and faults. Making live update safe and reliable while offering support for generic programs and updates is paramount for its widespread adoption.

This work proposes a new live update solution for generic Unix programs which minimizes the engineering effort needed to deploy updates, while maximizing the fault tolerance of this process.

We validate our design by implementing it as a framework and evaluating the top 4 most commonly used applications in the live update literature (Apache httpd, nginx, sshd and vsftpd). Our measurements indicate that the framework does not violate any operating system assumptions (e.g., no timeouts or dropped connections) and that the update time is low given the complex nature of the tested programs (i.e., at most 1.2s update time for the most complex updates). Furthermore, it correctly recovers from several common fault types, achieving a 94% detection rate in the case of buffer overflows.

Based on our experiments, we observed that the implementation of some applications can reduce the engineering effort needed for live update. Thus, we also formulate several guidelines for developers that wish to make their applications more live update friendly.

Contents

Chapter 1

Background

The sheer volume of software updates today is only matched by the integration effort they require. For example, in the case of the Linux kernel, there are on average 5.6 (with a maximum of 6.8) patches applied to the trunk per hour [2]. Unfortunately, because of the need to restart a service, or even the entire operating system, many organizations cannot cope with the resulting downtime. This results in wider update windows that can leave many systems dangerously exposed to security attacks: once a vulnerability is disclosed, the number of attacks usually increases by up to 5 orders of magnitude [3]. "*Rolling upgrades*" is an existing enterprise solution which deploys an upgrade incrementally in a large environment while masking service downtime through replication [4]. However, this solution requires both hardware redundancy and engineering effort, and still cannot handle services that need persistent program state. Furthermore, the clashes between different running software versions can lead to conflicts, unless properly handled [5].

1.1 The Live Update Problem

Live update is a solution which allows a user to properly face the increasing stream of updates without losing existing work, while offering a revert mechanism in case undesirable behavior is encountered. The idea is to upgrade a running system on the fly, thus removing the problem of downtime. However, in order for widespread adoption to be possible, the update process needs to be reliable and easy to use. Unfortunately, most of the existing techniques make strong assumptions about the updates and the system. For example, Ksplice[6] focuses primarily on security patches for the kernel, without automating state transfer directly. Other techniques such as Ginseng [7], Stump[8] or Kitsune[9] require a considerable amount of manual effort in order to properly support updates that exhibit data layout modifications, and scale poorly for very complex updates. The consequences are two-fold: the release of an update is a lot more difficult, and also the process is a lot less reliable.

In the following, we propose a model which takes into account the design of commodity operating systems while properly handling the constraints imposed by the application. We implement this model as a framework for generic C 32 bit programs and we evaluate it for 4 common UNIX server applications (*the Apache* ***httpd*** *Web server*, *the* ***nginx*** *Web server*, *the* ***OpenSSH*** *server* and *the* ***vsftpd*** *server*). We also perform an in-depth analysis on the engineering effort required to deploy our technique in Section 2.8 and in Section 3.6.

1.2 The PROTEOS Model

Our work was done in the context of the PROTEOS[10] project, which is a POSIX-compliant research operating system written with live update support in mind. Updates are performed at *process level*, allowing for arbitrarily complex changes to the code and the data. It also introduces the concept of *state quiescence* which provides a safe and stable environment in which to deploy these updates. The system server programs are event driven, allowing for a clear demarcation of the main request processing loop, as well as the initialization routines. Program *state transfer* is based on link-time instrumentation which also allows for common type transformations to be handled automatically. Finally, the system provides program version transparency as well as a well defined interprocess communication mechanism, which allow the update to occur seamlessly to the user.

The application code is instrumented using an LLVM pass which generates precise type information for memory objects. The *state transfer* part of the update process behaves as a precise tracing *garbage collector*, relocating and readjusting memory objects as dictated by the data layout of the new version. Since PROTEOS does not use shared libraries, the only cases not covered precisely are those of C *unions* containing pointers, both explicit (i.e., regular `union`) and implicit (i.e., pointer stored as an integer variable). For them, code annotations need to be created so that the framework may correctly handle those memory objects.

The framework also keeps track of dynamically allocated memory objects at runtime by intercepting the standard allocator calls. It properly reallocates them in the new version, taking note of any size transformation that might have occurred. Dynamic memory objects are identified by allocation site and only those objects whose call-sites exist in the new version are transferred, thus eliminating possible garbage objects.

State quiescence is achieved by the annotation of safe update points, which are best known by the developer. The design of PROTEOS allows for seamless integration of these update points at the top of the main processing loop by using a single point of entry for both update events and system events. Such behavior allows for significant reduction of the execution overhead by removing the necessity of full stack instrumentation. Instead, it is only necessary to provide type information for the frames of the long-lived function. Furthermore, the interleaving of update events and system events ensures that the state of the process is safe and consistent for an update.

1.3 Key Challenges on Commodity Operating Systems

In order to support generic C programs, we had to tackle several commonly occurring issues, while no longer being able to rely on specialized support from the kernel. We wanted to alter the target operating system as little as possible so as not to introduce any potential cause for instability. What is more, we needed to extend the framework to encompass different types of programming models, not just the event-driven one present in the PROTEOS servers. We briefly mention bellow the most important challenges we faced:

- **Version Transparency:** We needed to compensate for the lack of a unified IPC mechanism with which to perform delegation of control. Further, we also needed to account for unique process identifiers that discriminated between program versions.
- **Shared Libraries:** Most applications make use of shared libraries, either OS or vendor provided. As such, we could not assume that instrumenting them was sometimes (if ever) possible, so we needed to deal with a lot less precise view of the memory objects.
- **Multithreading Support:** The use of threads leads to several non-obvious complications in our design. We needed to account for scheduler non-determinism, as well as for inter-thread synchronization mechanisms. Moreover, we addressed the problem of possible deadlocks caused by resource contention during updating.
- **Custom Memory Allocators:** Many applications provide an internal implementation to improve the performance and usability of memory objects. Unfortunately, this has a serious impact on our static analysis and can often lead to the loss of necessary type information.

- **Fast and Secure Memory Transfer:** Without explicit support from the operating system, we designed our own method to transfer memory objects across processes without exposing a process' address space in an insecure fashion.

1.4 Our Approach

Our Live Update framework is implemented as a dynamic library that is meant to be preloaded by the target application, and that provides the runtime tools needed. In order to address the aforementioned challenges, it implements a four-stage protocol: **State Quiescence**, **Control Flow Transfer**, **State Transfer** and **State Validation**. We briefly discuss each of these stages, both with respect to PROTEOS, as well as to the key assumptions behind them.

Whereas reaching **State Quiescence** was straightforward in PROTEOS due to its event-driven design and singular event entry point, this was not the case for other operating systems or for multi-process, multi-threaded applications. Therefore, there are 2 aspects we considered: *establishing an update point* and *reaching quiescence in a bounded time.* The difficulty in the former emerged from the lack of demarcation between initialization code and event processing code. We automated this process by using an *offline profiler* capable of determining the long-lived tasks and the long-lived call stacks of each such task. The profiler also determines the idle execution points that map over the long-lived calls. These points would normally be blocking system calls used either for receiving external events (e.g., waiting for incoming connections) or for synchronization. Each such call would then have a special hook injected which would notify the framework of a task's quiescence. Furthermore, to eliminate the need for specialized support from the kernel, our instrumentation transforms all such blocking system calls into their non-blocking counterparts without exposing this to the application. This makes it possible for the framework to temporarily suppress incoming external events, thus preserving state consistency during an update.

The later aspect was complicated due the risk of deadlocking: simply blocking all tasks at the predetermined quiescence points did not guarantee that all synchronization events had completed. If such a deadlock were to occur, the new version would inherit the inconsistencies of the old version and would not be able to correctly resume execution. Our solution was the implementation of a *runtime quiescence algorithm* – basically a particular case of *Termination Detection* – which ensured all synchronization events were successfully completed before blocking, or that the update process was aborted if quiescence was not reached within a given time limit. The algorithm used several Read-Copy-Update cycles to guarantee deadlock-free quiescence.

Once the running application had reached quiescence, the new version process image would be executed. However, we needed to ensure that the new version correctly resumed execution from the same place as the old. This was achieved through **Control Flow Transfer**. While PROTEOS correctly handled this part using the well defined event entry point model, we needed to use **State Quiescence** to achieve execution synchronization: we would allow the new version to initialize and then wait for it to reach quiescence without allowing any external events to be received first. This also guaranteed that the new version would recreate its own process model. Once this was done, the processes were paired based on a unique *callsite identifier* created from the callstack hash of the parent process.

During this stage, we also inherited the *immutable objects* from the old version. *Immutable objects* are objects that cannot be altered across versions; they need to maintain the same properties. For example, when updating an application that communicates over a socket, the new version needs to use the same socket as the one opened in the old version. In this case we say that the socket needs to be *immutable.* Other types of immutable objects are: long-lived file descriptors, shared memory segments, memory objects that may have *likely pointers* to them (we will discuss this later on). To facilitate this, we implemented a basic *record and replay* protocol which records the creation of such objects during the initialization of the old version. When the new version initializes, our framework seamlessly replays the results by intercepting API calls and returning recorded values where needed.

The **State Transfer** implementation from PROTEOS could not work correctly due to the loss of type information for uninstrumented shared libraries internally managed memory objects (i.e., by use of custom memory allocator). For the latter, the framework also provides support for manual annotations by the developer. However, in order to further automate the process, we implemented a best-effort strategy: *conservative likely pointer analysis.* This technique scans the known memory objects (both with and without precise type information) looking for values that could be valid pointers and marks valid target objects as *immutable.* This ensures that they will not be relocated in the new version, preserving the consistency of the found likely pointers. Those pointers about which we already have precise information are not considered for the analysis. Finally, this also solves the problem of typeless C unions, as pointer consistency is guaranteed.

We classified *immutable memory objects* into 3 categories of interest: *static objects* (i.e., defined in the `.data`, `.rodata` or `.bss` segments), *dynamic mapped objects* (e.g., memory mapped files) and *dynamic allocated objects* (i.e., allocated through the system allocator). The *static objects* are fixed to their correspondent address using link-time placement, and the *dynamic mapped objects* are fixed at runtime using the provided POSIX API. In the case of *dynamic allocated objects*, the address fixing proved to be non-trivial due to the allocator implementation which, in general, does not expose the underlying allocation logic to the application. We chose to treat the allocator implementation as a blackbox, in order to support a wider variety of programs. Inheriting the memory objects was problematic as well, since replaying the heap allocations at initialization time required creating said objects before the process model of the new version was formed. To mitigate this, we implemented a *heap merging strategy* which combined all the immutable allocated objects across all the processes into a set of coherent objects that could be properly replayed.

The **State Validation** stage detects and automatically recovers from arbitrary runtime and memory errors. It relies on 2 elements: a *detection algorithm* and a *reliable computing base.* For the former we relied on the fact that all transformation operations done at update time can be reversed, and so we run a second update cycle after the first completes (but without resuming execution yet). The second cycle updates the new version of the application with the old version image, creating a *reversed version.* Because of the assumptions of the update process, the reversed version should exhibit the same memory layout as the original version, provided there were no errors. Therefore, we simply do a linear memory comparison of the reversed version and the old version. If we find differences, we can infer that there has been state corruption and we abort the update. The later part of the strategy enforces the requirement that, in order to be reliable, the running application cannot fail in case the update process itself fails. We achieve this by running a minimal amount of code on the old version during the update process, which constitutes our *reliable computing base.* In our implementation, the *RCB* is only 6% of the entire computing base.

If all steps complete and no errors are detected, the framework signals the new version to resume execution and the old version to exit gracefully. In case the update is aborted, for whatever reason, the old version simply resumes execution, while the new version is destroyed.

1.5 Contribution

Due to the size and complexity of the project, the resulting work is a product of the collaboration of multiple authors. Further we clearly and explicitly state the key contributions of the Thesis author:

- The **blackbox heap merging strategy** and **immutable memory object inheritance algorithm**, as described in Section 2.5 under *Immutable state objects*.
- The **immutable object marking algorithm** and the implementation of the **likely pointer analysis**, as described in Section 2.6 under *Conservative GC-style tracing*.
- The **memory object pairing algorithm**, as described in Section 2.6 under *Precise GC-style tracing*.
- The design and implementation of the **fault-tolerant live update framework**, as described in Section 3.3, in Section 3.4 and in Section 3.5 under *Shared Libraries*.
- The design and implementation of the **safe and fast inter-process memory transfer protocol**, as described in Section 3.4.
- The implementation of **unique identifiers for externally allocated memory objects**.

1.6 Outline

The following 2 chapters of the Thesis represent the full, unedited and unabridged contents of the 2 peer-reviewed conference papers that have resulted from this work. At the time of writing, "Mutable Checkpoint-Restart: Automating Live Update for Generic Long-running C Programs" is under submission to the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) 2014, while "Back to the Future: Fault-tolerant Live Update with Time-travelling State Transfer" is to appear in proceedings of the 27th International Conference on Large Installation and System Administration (LISA) 2013.

Chapter 2 describes the design and implementation of the **State Quiescence**, **Control Flow Transfer** and **State Transfer** stages, with context and evaluation. Chapter 3 focuses primarily on the **State Validation** stage, presenting the assumptions and fault model considered. It also briefly redescribes key issues from the **State Transfer** stage in order to provide proper context.

Chapter 2

Mutable Checkpoint-Restart: Automating Live Update for Generic Long-running C Programs

Mutable Checkpoint-Restart: Automating Live Update for Generic Long-running C Programs

Cristiano Giuffrida
Vrije Universiteit, Amsterdam
giuffrida@cs.vu.nl

Călin Iorgulescu
Vrije Universiteit, Amsterdam
calin.iorgulescu@cs.vu.nl

Andrew S. Tanenbaum
Vrije Universiteit, Amsterdam
ast@cs.vu.nl

Abstract

The pressing demand to deploy software updates without stopping running programs has fostered much research on live update systems in the past decades. Prior solutions, however, either make strong assumptions on the nature of the update or require extensive manual effort, factors which ultimately discourage widespread adoption of live update techniques.

This paper presents Mutable Checkpoint-Restart (MCR), a new live update technique for generic (multiprocess and multithreaded) long-running C programs. Unlike prior solutions, our techniques can support arbitrary software updates and automate most of the common live update operations. In particular, MCR seeks to address the key issues which required significant manual effort in prior solutions using three novel ideas: (i) Profile-guided quiescence detection; (ii) State-driven mutable record-replay; (iii) Mutable garbage collection-style tracing. Experimental results confirm that our techniques can effectively automate problems previously deemed difficult at the cost of modest performance and memory overhead. Our results also confirm that many programs are more "live update-friendly" than others and cases that are inherently hard to automate can be easily solved if live update becomes a driving factor in future software design.

1. Introduction

The fast-paced evolution of modern software is on a collision course with the pressing demand for highly available systems that guarantee nonstop operation. Live update—also known as *dynamic software updating* [55]—has increasingly gained momentum as a solution to the *update-without-downtime* problem, that is, deploying software updates without stopping running programs. Compared to the most common alternative—that is, *rolling upgrades* [26]—live update systems require no redundant hardware or software and can automatically preserve program state across versions. Ksplice [12] is perhaps the best known live update success story. According to its official website, the Ksplice Uptrack tool has already been used to deploy more than 2 million live updates on over 100,000 productions systems at more than 700 companies.

Despite decades of research in the area—with the first paper on the subject dating back to 1976 [27]—existing live update systems still have important limitations. *In-place* live update solutions [9,12,22,54,55] can transparently replace individual functions in a running program, but are inherently limited in the types of updates they can support without significant manual effort. Ksplice, for instance, is explicitly tailored to small security patches [4]. Prior *whole-program* live update solutions [29,35,48], in turn, can efficiently support several classes of updates, but require a nontrivial annotation effort which increases the maintenance burden and ultimately discourages widespread adoption of live update techniques.

This paper presents *Mutable Checkpoint-Restart* (*MCR*), a new live update technique for arbitrary long-running C programs. Drawing inspiration from traditional checkpoint-restart, MCR checkpoints (i.e., *freezes*) the running version, allows the new version to restart in a controlled way, and remaps the old execution state into the new one. This approach builds on kernel support for emerging userspace checkpoint-restart techniques [2] and can allow arbitrary updates without altering the structure of the program or its process model. Unlike traditional checkpoint-restart techniques [2,3,5,11,34], however, mutability induced by version updates requires remapping program checkpoints after restart, a hard problem [33] which we seek to automate in the common cases. The MCR model, in particular, helps pinpoint the three key challenges which required extensive manual effort in prior work: (i) how to obtain consistent checkpoints which are intuitively "*easy to remap*" after restart; (ii) how to remap the checkpointed control flow; (iii) how to remap the checkpointed program state.

To automate these tasks, we introduce three novel ideas. *Profile-guided quiescence detection* allows all the program threads to safely block in a known control-flow configuration with no code annotations required. To the best of our knowledge, ours is the first automated quiescence detection protocol for generic programs which is at the same time deadlock-free and provides fast convergence. *State-driven mutable record-replay* allows the new version to reinitialize in a controlled way and remap the checkpointed control flow without interfering with the old version or violating remapping invariants. To the best of our knowledge, ours is the first automated control-flow transfer strategy for generic programs that can also tolerate changes in the process model. *Mutable garbage collection-style tracing* allows the new version to remap the checkpointed state even with partial information on global data structures. To the best of our knowledge, ours is the first automated state transfer strategy for generic programs that can conservatively handle state changes without user-maintained annotations.

We have implemented our ideas in a MCR prototype for long-running Linux C programs. Our evaluation on four popular server programs shows that our techniques yield: (i) low engineering effort (334 LOC to prepare our programs for MCR), (ii) realistic update times (< 1 s); (iii) low run-time performance overhead in the default configuration (0-5%).

2. Background and Related Work

In the following, we focus on *local* live update solutions for operating systems and long-running C programs, referring the reader to [7, 26, 44, 66] for live update for distributed systems.

Quiescence detection. MCR relies on *quiescence* as a way to restrict the number of valid control-flow configurations at checkpointing time. Some approaches [48] relax this constraint, but then automatically remapping all the possible checkpointed control-flow configurations or simply allowing mixed-version execution [21, 22, 49] becomes quickly intractable without extensive user intervention. Quiescence detection algorithms proposed in prior work operate at the level of individual functions [9, 12, 28, 32] or generic events [14, 15, 29, 54, 55, 60]. The former approach—known for its weak consistency guarantees [29, 36]—relies on passive *stack inspection* [9, 12, 28, 32], which cannot guarantee convergence in bounded time [48, 49]. The latter approach relies on either update-friendly system design [14, 29, 60]—rarely an option for existing C programs—or explicit per-thread *update points* [35, 48, 54, 55]—typically annotated at the top of long-running loops. Two update point-based quiescence detection strategies are dominant: *free riding* [48, 54]—allow threads to run until they all happen to reach a valid update point at the same time–and *barrier synchronization* [35]—block each thread at the next valid update point. The first strategy cannot guarantee convergence in bounded time. To mitigate this problem, prior solutions suggest expanding the number of update points using static analysis [54] or per-function update points [48]. Both solutions can introduce substantial overhead yet they still fail to guarantee convergence. The second strategy, on the other hand, offers better convergence guarantees but is inevitably deadlock prone [54]. In addition, all the prior update point-based strategies require interrupting blocking calls, which would otherwise delay quiescence indefinitely. To address this problem, prior solutions suggest a combination of annotations and either signals [35] or file descriptor injection [48]. The former strategy is more general, but inherently race-prone and potentially disruptive for the program. Our *profile-guided quiescence detection* protocol, in contrast, requires no code annotations and is designed to provide efficient, race-free, and deadlock-free quiescence in bounded time.

Control-flow transfer. MCR relies on *control-flow transfer* as a way to remap the checkpointed control-flow configuration after restart. Prior in-place live update models [9, 12, 14, 21, 22, 49, 54, 55] provide no support for control-flow transfer, implicitly forbidding particular types of updates. Ksplice [12], for instance, cannot easily support a simple update to a global flag that changes the conditions under which kernel threads enter a particular fast path. Failure to remap the latter may, for instance, introduce silent data corruption or synchronization issues—such as, deadlocks. Prior whole-program live update models, in turn, implement control-flow transfer using system design [29, 60], *stack reconstruction* [48], or manual control migration [35]. The first option is overly restrictive for many C programs. The second option forces the user to perform the heroic effort of manually remapping all the possible control-flow configurations across versions. The last option, finally, reduces the effort by encouraging existing code path reuse, but still delegates control-flow transfer completely to the user. Our *state-driven mutable record-replay* strategy, in contrast, automatically reuses existing code paths and frees the user from the burden of reconstructing control flow and preserving all the necessary remapping invariants.

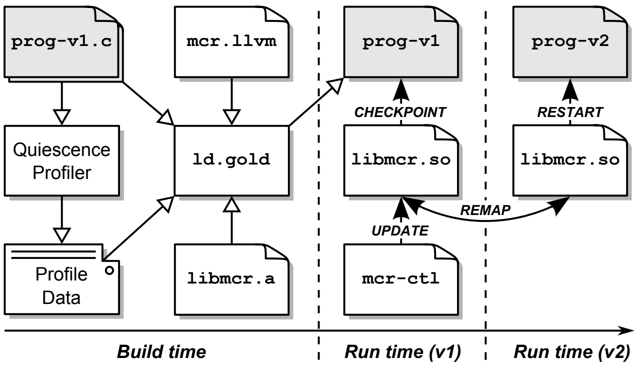

Figure 1: MCR overview.

State transfer. MCR relies on *state transfer* as a way to remap the checkpointed program state after restart and apply the necessary data structure transformations. Prior in-place live update models either delegate state transfer entirely to the user [9, 12, 14, 21, 22, 49] or provide simple type transformers with no support for pointer transformations [54, 55]. Such restrictions are inherent to the in-place live update model, which advocates "*patching*" the existing state to directly adapt it to the new version. Prior whole-program live update models, in turn, either delegated mapping functions to the user [48, 60] or attempted to automatically reconstruct the state in the new version using *precise* pointer traversal [29, 35]. The latter strategy, however, requires a nontrivial annotation effort to unambiguously identify all the global pointers correctly. Our *mutable GC-style tracing* strategy, in contrast, does not require state annotations and can gracefully handle uninstrumented shared libraries and custom memory allocation schemes.

3. Overview

Figure 1 illustrates the typical MCR workflow. In a preliminary step, users allow our quiescence profiler to run the program, identify all its *quiescent points*, and generate profile data required by our quiescence detection protocol. This is a relatively infrequent operation which should only be repeated when the quiescent behavior of the program changes—we envision programmers simply integrating quiescence profiling as part of their standard regression test suite. Building the program requires specifying standard compiler flags which instruct the GNU gold linker (`ld.gold`) to link the original code against our static library (`libmcr.a`) and use an LLVM-based link-time plugin [47] (`mcr.llvm`) for static instrumentation purposes. The latter instruments the quiescent points identified in the profile data and prepares the program for our mutable GC-style tracing strategy. Running the program requires preloading our dynamic instrumentation shared library (`libmcr.so`), which complements static instrumentation with information available only at runtime (i.e., shared libraries) and implements our mutable checkpoint-restart techniques. When an update is available, the user can signal the running version using a simple command-line tool (`mcr-ctl`). In response to the event, our shared library checkpoints the running program and allows the new version to restart from the old checkpoint. This is done by (i) running the quiescence detection protocol on the running version to obtain a consistent checkpoint; (ii) starting the new version with all the information required to perform state-driven mutable record-replay and allowing the program to reinitialize; (iii) running the quiescence detection protocol on the new version to synchronize; (iv) remap the program state from the old version using mutable GC-style tracing; (v) resume execution. Failure to complete the restart phase due to unexpected errors simply causes the old version to resume execution from the last checkpoint. Note that this is in stark contrast to prior solutions for generic userspace C programs [9, 22, 35, 48, 54, 55], which cannot match MCR's strong *atomicity* and *isolation* guarantees.

4. Profile-guided Quiescence Detection

Our profile-guided quiescence detection strategy stems from two simple observations. First, the problem of transparently synchronizing multiple threads in bounded time and in a deadlock-free fashion is undecidable—that is, easily reducible to the halting problem—absent extra information on thread behavior. Second, every long-running program has a number of natural *execution-stalling points* [43]—that are obvious choices to identify a globally quiescent configuration. The key idea is to profile the program at runtime and automatically identify *quiescent points* from all the stalling points observed. We note a number of interesting stalling point properties in server programs. First, they always originate from long-lived blocking calls (e.g., `accept`) with well-known semantics. This allows us to gather extra information on a stalling thread and carefully control its behavior. Second, stalling points are often found at the top of long-running loops, which prior work has already largely recognized as ideal update points [35, 54, 55]. Third, even when stalling points are deeper in the call stack, fine-grained control over them is clearly crucial to reach quiescence, a common problem in prior work [35, 48].

Quiescent points. To detect stalling points and the corresponding long-lived loops, our profiler relies on standard profiling techniques. Detecting long-lived loops is important to identify all the long-lived stack variables that might carry state information which needs to be remapped in the new version after restart. Our profiling strategy leverages static instrumentation to intercept all the function calls, library calls, and loop entries/exits at runtime. Dynamic instrumentation tracks all the processes and threads in the program and identify all the classes of threads with the same stalling behavior. To detect all the stalling points correctly, we rely on a test workload able to drive the program into all the potential *stalling states* (e.g., open idle connections, large file transfer, etc.). In our experience, this workload is typically domain-specific—can be reused across several applications of the same class—and often trivial to extrapolate from existing regression test suites. Even for very complex programs that may exhibit several possible stalling states, we expect this approach to be more intuitive, less error-prone, and more maintainable than manually annotated update points used in prior work [35, 54, 55].

Per-thread stalling points are detected using statistical profiling of library calls. Intuitively, a stalling point is simply identified by the long-lived blocking call where a given thread spends most of its time during the test workload. Loop profiling is used to detect every thread's deepest long-lived loop that never terminates during the test workload. At the end of the test run, our profiler produces not only instrumentation-ready profile data, but also a human-readable report with all the short-lived and long-lived classes of threads identified, their deepest long-lived loops, and their stalling points. Each stalling point is automatically classified as *persistent* or *volatile*—that is,whether it is already visible or not right after initialization—and as *external* or *internal*—that is, whether the corresponding blocking call is listening for external events (e.g., `select`) or not (e.g., `pthread_cond_wait`). In addition, a policy decides how each stalling point participates in our quiescence detection protocol. Three options are possible: (i) *quiescent point*—marks a valid quiescent state for a given thread to actively participate in our protocol; (ii) *blocking point*—allows execution to stall indiscriminately before reaching the next quiescent point; (iii) *cancellation point*—allows returning an error (e.g., `EINTR`) to the program at quiescence detection time. The default policy is to promote all the persistent stalling points to quiescent points and all the volatile ones to blocking points. The rationale is to allow all the checkpointed control-flow configurations that can be remapped in a fully automated way using state-driven mutable record-replay after restart.

Instrumentation. Our static instrumentation relies on profile data to transform all the stalling points identified in the dynamic call graph of the program. We currently use thread-local flags to propagate call graph information to every long-lived blocking call site and instrument stalling points correctly. In particular, each call site is changed to invoke a wrapper function in our static library—which currently provides support for many common blocking `libc` functions—in a way that it allows what we refer to as *unblockification*. Unblockification

```
1:  procedure COORDINATOR
2:      $Q \leftarrow 1$
3:      repeat
4:          $A \leftarrow 0$
5:          SYNCHRONIZE_RCU()
6:          SYNCHRONIZE_RCU()
7:      until $A \neq 0$
8:      $Q \leftarrow 2$
9:      SYNCHRONIZE_RCU()
10:     SYNCHRONIZE_RCU()
```

```
1:  procedure QUIESCENTPOINT
2:      if $Q > 0$ then
3:          if $Active$ then
4:              $A \leftarrow 1$
5:          if $Initiator$ or $Q == 2$ then
6:              RCU_THREAD_OFFLINE()
7:              THREAD_BLOCK()
8:              RCU_THREAD_ONLINE()
9:              THREAD_UNBLOCKED()
10:     RCU_QUIESCENT_STATE()
```

Figure 2: Pseudocode of our quiescence detection protocol.

exposes the same library call semantics to the program, but guarantees that every long-lived call never blocks execution for more than a predetermined time, while periodically calling our own hook at quiescence detection time. Prior work used a similar wrapping strategy [48], but only as an alternative to signals to unblock I/O calls on demand. Our goal, in contrast, is to ensure that all the blocking calls are *short-lived* and fully *controllable* by design at quiescence detection time.

Our unblockification design fulfills three key goals: (i) efficiency; (ii) low CPU utilization; (iii) low quiescence detection latency. To implement our strategy efficiently, we rely on standard timeout-based versions of library calls (e.g., `sem_timedwait`) and simply loop through repeated call invocations until control must be given back to the program. When a timeout-based version of the call is not available, we resort to the nonblocking version of the call (e.g., nonblocking `accept`) followed by a generic timeout-based call listening for the relevant events (e.g., `select`). The latter strategy guarantees a minimal number of mode switches are typically incurred when the program is under heavy load and thus on a performance-sensitive path. Our other goals highlight the evident tradeoff between unblockification latency and CPU utilization. In other words, short timeouts translate to very fast loop iterations and frequent invocations of our hooks, but also to high CPU utilization. To address this problem, our implementation dynamically adjusts the unblockification latency, using low values that guarantee fast convergence at quiescence detection time—currently 1 ms—and higher, more conservative values—currently 100 ms, which resulted in no visible CPU utilization increase in our test programs—otherwise.

We note that unblockification is a semantics-preserving transformation of the original program which ensures three important properties. First, it guarantees that stalling point execution always revolves around short-lived loops with bounded iteration latency even when a thread is blocked indefinitely. Second, it provides a straightforward way to enforce our stalling point policies (e.g., allow blocking behavior in case of blocking points or call our hooks at the top of each short-lived loop iteration in case of quiescent points). Third, it can unambiguously identify internal or external events received by long-lived blocking calls and pass this knowledge to our hooks at quiescence detection time. These properties all serve as a basis for our quiescence detection protocol.

Quiescence detection. Our quiescence detection protocol is based on two key observations. First, long-running programs are naturally structured to allow threads waiting for external events (e.g., a new service request or a timeout) to block indefinitely. Second, in the face of no external events, well-formed programs must normally reach a global quiescent state—all the threads stalling at quiescent points—in bounded time. Building on these observations, our protocol enforces simple *barrier synchronization* for all the threads blocked on external events—that is, *initiator threads*—and waits for all the threads processing internal events—that is, *internal threads*—to complete before detecting quiescence. When quiescence is detected, no new event can be initiated and all the threads can be safely blocked at their quiescent points. The next question is how long to wait for internal events to complete without blocking threads in a deadlock-prone fashion.

The naive solution is to scan the call stack of all the processes and threads to verify they have all reached their quiescent points. This strategy, however, is not race-free in absence of a consistent view of all the running threads. Worse, even a globally consistent snapshot of all the call stacks is not sufficient in the presence of asynchronous thread interactions. Suppose a thread *A* signals a thread *B* blocked on a condition variable and then reaches its next quiescent point. Before *B* gets a chance to unblock and process the event, a global call stack snapshot might mistakenly conclude that both threads are idle at their quiescent points and detect quiescence.

This race, known as the "*launch-in-transit hazard*" [23], is a recurring problem in the *Distributed Termination Detection* (*DTD*) literature [23, 42, 52]. All the DTD solutions to this problem rely on explicit event tracking [52], a costly solution in a local context partially explored in prior work [48]. Fortunately, unlike in DTD, we found that avoiding event tracking is possible, given that local events propagate in bounded time.

The key idea is to wait for all the threads to reach a quiescent point with no event received since the last quiescent point. This strategy effectively reduces our original global quiescence detection problem to a local quiescence detection problem—that is, quiescing short-lived loop iterations. To address the latter, we rely on RCU [50], a scalable, low-latency, and deadlock-free local quiescence detection scheme. RCU-like solutions to the problem of global quiescence detection were attempted before [14, 15], but in much less ambitious architectures that simply disallowed long-lived threads. Our implementation is based on `liburcu` QSBR [24], the fastest known userspace implementation for local quiescence detection with nearly zero overhead. The implementation provides a `synchronize_rcu` primitive, which allows a *controller thread* to wait for one quiescent period—that is, for all the threads to reach a *quiescent state* at least once from the beginning of the period [24].

Our RCU-based instrumentation ensures threads atomically enter a nonquiescent state at creation time (i.e., `pthread_create` blocks waiting for the new thread to complete RCU registration), atomically traverse a quiescent state at each quiescent point right before reaching the designated

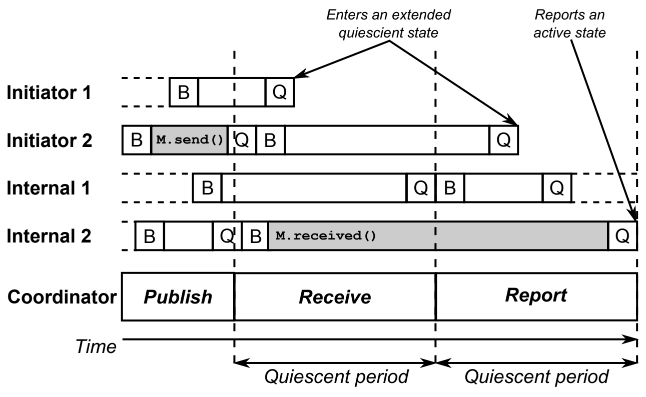

Figure 3: A sample run of our quiescence detection protocol.

blocking call, and enter an *extended quiescent state* [24] at destroy time or when our quiescence detection protocol dictates them. This strategy allows our protocol to transparently deal with an arbitrary number of short-lived and long-lived threads. Figure 2 illustrates the simplified steps of our protocol.

The coordinator publishes a quiescence detection protocol event ($Q = 1$) and sets a global active counter to 0. Next, it waits for a first quiescent period to ensure the protocol is visible to *all* the initiator and internal threads and a second quiescent period to give *any* thread a chance to report an active state—whether the last blocking (or thread creation) call received an event. The entire sequence is repeated until quiescence is detected, that is, no thread was found active in the last quiescent period. In the second phase, the coordinator publishes a barrier event ($Q = 2$) and waits for 2 more periods to ensure all the threads are safely blocked at their quiescent points. Our quiescent point instrumentation, in turn, implements the thread-side protocol logic. When the protocol is in progress ($Q > 0$), our hook reports an active state to the coordinator (if any) and blocks the current thread if it is an initiator thread or a barrier event is in progress. Lines 6–7 allow the current thread to enter an extended quiescent state and block on a condition variable. Lines 8–9, in contrast, allow the current thread to leave an extended quiescent state and synchronize before resuming execution—in case the coordinator decides to abort the protocol, for example after a predetermined timeout. Note the `rcu_quiescent_state` call at the bottom, the only step executed also during regular execution, to mark all the quiescent state transitions correctly. Figure 3 shows a sample run of the first phase of our protocol ($Q = 1$), with two threads reacting to the published protocol event after 2 grace periods.

Our protocol provides race-free and deadlock-free quiescence detection in only $2q+2$ quiescent periods (with $q = 1$ if the program is already globally quiescent and otherwise bounded by the length of the maximum internal event chain). Our strategy leverages two well-known RCU uses: publish-subscribe and "*waiting for things to finish*" [51]. A current limitation of `liburcu` is its inability to support multiprocess `synchronize_rcu` semantics. To address this issue, MCR uses a process-shared active counter and requests a controller thread in each process to complete the first phase of the protocol. In this phase, newly created processes simply cause the entire protocol to restart. When all the per-process threads complete, MCR transitions to the second phase of the protocol and waits for all the controller threads to report quiescence.

5. State-driven Mutable Record-replay

Our control-flow transfer strategy faces the major challenge of seamlessly remapping the control-flow configuration checkpointed at quiescence time. Further, our design goals dictate support for generic multiprocess/multithreaded programs and version updates that may introduce changes in the process model or long-lived thread behavior. To address this challenge, the key observation is that programs tend to naturally reconstruct their process hierarchy and control-flow configuration correctly at initialization time. Following this intuition, the idea is to allow the new version to reinitialize in a controlled way and exploit existing code paths to remap the checkpointed control flow correctly. Further, we note that initialization code can often naturally reconstruct shared library state and reinitialize new data structures that were not part of the old version. The latter observation shows that initialization code reuse is beneficial to also minimize manual state transfer effort in the common case of additive state changes introduced by updates.

Remapping control flow. Our strategy raises two main challenges: (i) how to *synchronize* the initialization process and avoid exposing the new version to external events which would violate our atomicity and reversibility guarantees; (ii) how to *control* the initialization process to prevent the new version from clashing or destroying state inherited from the checkpoint. MCR addresses the first challenge by allowing a controller thread to reinitiate the quiescence detection protocol *before* starting initialization. This forces all the long-lived threads to safely block at their quiescent points without receiving new external events. To address the second challenge, MCR relies on record-replay of initialization-time operations. This is marginally intrusive compared to full-execution record-replay used in prior work for state reconstruction [57, 64, 65] or multiversion execution [40]. Further, unlike traditional record-replay [10, 31, 46, 56, 61, 62], MCR does not attempt to deterministically replay execution, a strategy which would otherwise forbid any initialization-time changes. The goal is to replay the minimum number of operations to allow the new version to preserve the checkpointed state, while executing live the rest of the—arbitrarily different—initialization code.

We term this strategy *state-driven mutable record-replay*, which draws inspiration from recent mutable record-replay strategies [45, 67] but with at least two key differences. First, our strategy is state-driven, in that we only replay operations associated to immutable state objects inherited from the checkpoint (e.g., file descriptors). This eliminates the need for in-kernel replay to support transitions to live execution. Our record-replay implementation—part of our preloaded library—is simply based on library call interception at initialization time. Second, we allow nondeterministic multithreaded execution not to restrict behavioral changes across versions and enforce partial ordering of related operations similar to [67] only when strictly necessary—currently only for file descriptor operations used for synchronization purposes.

Mapping operations. Our record-replay strategy adopts a conservative mapping and conflict resolution strategy. For instance, if the new version is changed to omit a previously recorded operation to replay, we immediately flag a conflict. This strategy aims to unambiguously reinitialize the state while conservatively detecting complex changes that inevitably require user intervention—since the replay surface is small, we expect unnecessary conflicts to be minimal. This is in contrast to prior techniques that rely on best-fit exploration strategies to map record-replay operations and resolve conflicts [67].

To enforce a conservative mapping strategy in presence of reordering of operations due to nondeterminism or arbitrary version changes, MCR relies on *call stack IDs*. The latter are version-agnostic hashes—combined to per-call stack incremental counters—obtained from the call stack of a thread performing an operation considered for replay. Call stack IDs can conservatively discriminate individual thread operations and are more robust to addition/deletion/reordering of library calls—and changes to their arguments—than mapping schemes based on global or partial orderings of operations. The tradeoff is that unnecessary conflicts may arise in case of initialization function refactoring (e.g., renaming). In our experience, these cases are relatively rare. In addition, best-fit matching strategies may quickly suggest to the user how to resolve the conflict. To detect and tolerate benign changes to library call arguments across versions, MCR follows pointer arguments similar to [67], but relies, when possible, on tracking information provided by our mutable GC-style tracing instrumentation to better recognize equivalent call arguments.

Immutable state objects. Our state-driven mutable record-replay strategy currently records all the initialization-time operations and replays only those affecting *immutable state objects* after restart. Immutable state objects are objects inherited from the checkpoint that carry state information which must be conservatively preserved after restart. In other words, these are the only objects allowed to violate the mutable MCR semantics. Our current implementation supports three main classes of immutable objects: (i) file descriptors inherited from the checkpoint—immutable since they may carry associated in-kernel state; (ii) immutable memory objects identified by our mutable GC-style tracing strategy—immutable due to partial knowledge on global pointers; (iii) process and thread IDs—immutable since they carry process-specific information that may be stored in memory. Others are possible as well.

Mapping and preserving immutable objects inherited from the checkpointed version at replay time is challenging in a multiprocess context. The problem is exacerbated by the need to avoid unnecessary—and potentially expensive—object tracking during normal execution. Consider the naive solution for file descriptors—but similar considerations apply to other immutable objects as well. This may allow every process in the new version to simply inherit all the file descriptors from its old checkpointed counterpart at process creation time. There are two main problems with this approach which we found to be unacceptably common causes of unnecessary conflicts or ambiguity. First, the multiprocess nature of the initialization process may result in a checkpointed file descriptor ID clashing with a file descriptor ID already inherited from the parent process at initialization time. Second, file descriptors IDs may be reused during or after initialization, which means we can no longer unambiguously determine whether a checkpointed file descriptor ID matches an ID logged in the record phase when enforcing our state-driven replay strategy.

MCR addresses these challenges by enforcing two key principles: *global inheritance* and *global separability*. Global inheritance allows the first process in the new version to inherit *all* the immutable objects from the checkpointed process hierarchy before starting the initialization process. The idea is to preallocate all the necessary immutable objects to avoid identifier clashing and seamlessly propagate all the objects down the new process hierarchy for replay purposes. All the immutable objects that do not participate in replay operations in a given process are simply cleaned up at the end of the initialization. Global separability, in turn, allows all the immutable objects created at initialization time to have globally unique identifiers, preventing the ambiguity introduced by reuse. Note that this is not necessary for immutable objects created after initialization, which are not target of replay operations and can simply be inherited from the checkpoint.

MCR enforces these properties in different ways for different classes of immutable objects. Immutable static memory objects (e.g., global variables)—preinherited using a linker script—naturally guarantee global inheritance and separability by design. Immutable dynamic memory objects (e.g., heap objects) are inherited using global reallocation—as detailed later. Separability is enforced by deferring global deallocations at the end of initialization and explicitly flagging initialization-time allocations in allocator metadata maintained by our mutable GC-style tracing strategy—which also provides support for deferred deallocations. Immutable file descriptors are inherited using UNIX domain sockets. Separability is enforced by intercepting initialization-time file descriptor (`fd`) creation events to (i) allocate new `fd` IDs in a reserved range at the end of the `fd` space and (ii) prevent initialization-time reuse. Immutable process and thread IDs are handled similarly to file descriptors, except they cannot be simply inherited from the checkpoint or their creation operations simply replayed. To enforce global inheritance, MCR intercepts initialization-time process and thread creation events and relies on Linux namespaces [17] to force the kernel to assign a specific ID. This strategy follows the same approach adopted by emerging userspace checkpoint-restart techniques for Linux programs [2].

Another key challenge is how to implement global reallocation of immutable dynamic memory objects, which need to preserve their memory address after restart. MCR addresses this challenge using different strategies, coalescing overlapping memory objects into superobjects at reallocation time—deallocated later when no longer in use. Shared libraries are copied and prelinked—simply using the `prelink` tool [41]—in a separate directory before restart. We instruct the dynamic linker to use our copies, allowing the libraries to be remapped at the same virtual address in spite of address space layout randomization (*ASLR*). This also allows MCR to reallocate all the dynamically loaded libraries correctly using `dlopen`. Memory mapped objects are remapped at the same address using standard library interfaces (e.g., `MAP_FIXED`). Note that to provide the strongest isolation guarantees, we also envision memory mappings shared with the checkpoint to be "*shadowed*" during initialization and remapped correctly at the end, a strategy that our current prototype does not yet fully support.

Global reallocation of heap objects poses the greatest challenge, given that standard allocator interfaces provide no support for this purpose. MCR addresses this problem by leveraging the intuition that common allocator implementations behave similarly to a buffer allocator for an ordered sequence of allocations in a fresh heap state. MCR implements this strategy for `ptmalloc` [30]—the standard `glibc` allocator—using a single `malloc` arena, but we believe a relatively allocator-independent implementation is possible assuming predictable allocation behavior and `malloc` header size—currently inferred by gaps between dummy allocations performed at startup. We also envision this abstraction to become part of standard allocator interfaces once MCR is deployed—similar to `ptmalloc`'s existing `get_state` and `set_state` primitives for traditional process checkpoint-restart.

6. Mutable GC-style Tracing

Our state transfer strategy faces the major challenge of remapping all the state objects (i.e., data structures) from the checkpointed state in presence of even complex state transformations. Further, our goals dictate eliminating the need for annotations in all the common real-world C programs. To address this challenge, we make three key observations. First, annotations in prior program-level state transfer work [29, 35] were necessary to compensate for C's lack of rigorous type semantics, which prevents accurate program state identification. Not surprisingly, prior work has demonstrated that annotationless program-level state transfer is possible for managed languages like Java [63]. Second, similar problems are already well-understood in the garbage collection (GC) literature [13, 37, 58]. In particular, the problem of remapping the program state in face of full-coverage state transformations faces the very same challenges of a *precise* and *moving* tracing garbage collector for C [58]. By precise, we refer to the ability to accurately identify object types, necessary to apply on-the-fly type transformations. By moving, we refer to the ability to relocate objects, necessary to support arbitrary state changes

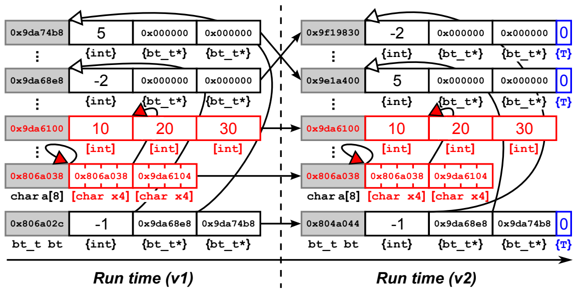

Figure 4: State remapping using mutable GC-style tracing.

in the new version—introduced by state transformations, compiler optimizations, or ASLR. Prior work [58] identified many real-world scenarios in which annotations are necessary in this context, such as: explicit or implicit `unions`, nonstandard allocation schemes, uninstrumented libraries, pointers as integers. Third, *conservative* garbage collectors are well-known solutions to all these problems in the GC literature [18, 19], in that they do not require any explicit type information at the cost, however, of being unable to relocate objects.

Remapping program state. Our observations hint at the key idea behind *mutable GC-style tracing*: trace all the state objects to remap using a precise GC-style strategy when possible, resort to a conservative GC-style strategy in face of incomplete or ambiguous type information. To implement this strategy, MCR gracefully relaxes the original full-coverage state transformation requirement, marking the necessary objects as immutable—they cannot be relocated after restart—and forbidding updates to certain objects when ambiguous type information is found. This strategy allows the user to tradeoff the annotation effort against the number of data structure changes that can be seamlessly remapped by MCR. When unsupported state change are detected, MCR raises a conflict that must be manually resolved by the user. We envision users deploying an annotationless version of MCR at first, and then incrementally add annotations only on the data structures that change more often if their experience with the system generates an undesirable number of conflicts. Even with a fully annotated state, our conservative strategy can help the user identify missing annotations or other problematic cases.

Our mutable GC-style tracing strategy is exemplified in Figure 4. MCR relies on the accurate type information available to *precisely* reconstruct the binary tree `bt` in the checkpoint and remap all the nodes and pointers with their new types (`T`) correctly after restart. The array `a`, in turn, is *conservatively* scanned for pointers, which are found to point into a heap-allocated array and `a` itself. As a result, both arrays are marked as immutable and remapped at the same locations after restart.

Precise GC-style tracing. There are two main strategies to implement precise GC-style tracing: (i) type-aware traversal functions generated by the frontend compiler [13, 35, 37] or (ii) data type tags [29]—hybrid approaches are also possible [58]. The former is generally more space- and time-efficient, but the latter can better deal with polymorphic behavior and provide more flexible run-time type management. MCR implements the latter strategy to simplify type management and seamlessly switch from precise to conservative tracing when needed.

Similar to prior precise strategies based on data type tags [29,58], we rely on static instrumentation to store tracking and type information for all the relevant static objects (i.e., static/global variables, constants, functions, etc.) and change all the allocator invocations to call wrapper functions in our static library and maintain data type tags in in-band metadata. Static analysis determines the allocated type on a per-callsite basis, similar to [58]. We also borrowed the tracking technique for generic stack variables, maintaining a stack-allocated linked list of overlay metadata nodes [58]. While inspired by prior approaches, our instrumentation has a number of unique properties. First, ambiguous cases like `unions` require no explicit annotations [29] or tagging [58]—which may not be sufficient in many real programs—given that our tracing strategy can be made conservative when needed. Similarly, we do not require full allocator instrumentation for complex custom allocation schemes. Our precise analysis can currently only support standard allocator abstractions (i.e., `malloc`) or—if annotations are provided—simple region-based allocation schemes [16]. For more complex unsupported allocator abstractions, our static type analysis resorts to fully conservative behavior. Finally, stack variable tracking—expensive for full-execution coverage—is limited to all the functions that profiling found active on the long-lived call stack of some quiescent thread.

The tracing strategy is implemented in our preloaded library. It operates in each new quiescent process after restart, parallelizing the state transfer operations in a multiprocess context. Each process requests a central coordinator to connect to its checkpointed counterpart (if any) identified by the same call stack ID. Once a pipe is established with the checkpointed process, MCR creates a fast read-only shared memory channel to transfer over all the tracking and type information from the old version. Starting from root data and stack objects, MCR traces pointer chains to reconstruct the entire checkpointed state and remap each object found in the traversal to the new version—while reallocating objects and applying type transformations as needed, similar to [29, 35]. We also allow user-specified traversal callbacks to handle complex state transformations, similar to [29]. Unlike prior approaches, however, the MCR model dictates a more comprehensive cross-version object matching strategy (i.e., variable `x` in the checkpoint should be remapped to variable `x` in the new version). We use symbol names to match static objects and allocation site information to match dynamic objects that need to be reallocated in the new version. Dynamic objects already reallocated at initialization time are matched by their call stack ID. Individual threads, finally, are matched based on their long-lived loops and their stack variables remapped using symbol names.

Conservative GC-style tracing. Our conservative GC-style strategy operates obliviously to its precise counterpart. The idea is to first perform a (partly) conservative analysis to identify all the remapping invariants and later allow precise GC-style tracing to implement state transfer without worrying about type ambiguity—pointers not explicitly exposed by the type information available are never traversed and the data simply copied over as is. Our conservative strategy seeks to identify two possible remapping invariants for every object in the old version: *immutability*—the object cannot be relocated after restart—and *nonupdatability*—the object cannot be automatically type-transformed by our precise tracing strategy after restart (a conflict is generated in case of type changes).

To identify such invariants, MCR operates similarly to a conservative garbage collector [18, 19], scanning opaque memory areas looking for *likely pointers*—that is, aligned memory words that point to a valid live object in memory. Objects referenced by likely pointers are marked as immutable and nonupdatable—we could restrict the latter to only interior pointers, but we have not implemented this option yet. Objects that contain likely pointers are marked as nonupdatable—again, we could restrict the latter to only certain type changes, but we have not implemented this option yet. Note that, unlike prior approaches, our strategy is only partly conservative: MCR traverses the state using our precise GC-style strategy by default and switches to a conservative analysis only when it encounters opaque memory areas. Further, when possible, our dynamic points-to analysis uses the precise pointed-object information to reject illegal (i.e., unaligned) likely pointers.

Run-time policies decide when a traversed memory area must be treated as opaque. Our default is to do so for explicit `unions`, pointer-sized integers, `char` arrays—often used to implement implicit `unions`—and uninstrumented allocator operations, but different program-driven policies are possible. Currently, we do not conservatively analyze nor transfer shared library state by default, since we have observed that most real-world programs already reinitialize shared libraries and their state correctly at initialization time. Nonetheless, the user can instruct MCR to transfer–and conservatively analyze—the static/dynamic state of particular uninstrumented shared libraries in an opaque way, when needed. Dynamic instrumentation included in our preloaded library implements tracking for shared libraries and their dynamically allocated objects.

Our tracing strategy raises two issues: *accuracy*—how conservative is the analysis in determining updatability coverage—and *timing*—when to perform the analysis. In our experience, the former is rarely an issue in real-world programs. Prior work has reported that even fully conservative GC rarely suffers from type-accuracy problems on 64-bit architectures—although more issues have been reported on 32-bit architectures [38]. Other studies confirm type accuracy is marginal compared to liveness accuracy [39]. In our context, liveness accuracy problems are only to be expected for uninstrumented allocator abstractions that aggressively use free lists—or other forms of reuse. These cases can be easily identified and compensated by annotations/instrumentation, if necessary. As for the latter, our analysis should be normally performed after

	Quiescence profiling						Updates		Changes			Engineering effort	
	SL	LL	Ext	Int	Per	Vol	Num	LOC	Fun	Var	Type	Ann LOC	ST LOC
Apache httpd	2	8	6	2	5	3	5	10,844	829	28	48	18	302
nginx	1	2	1	1	2	0	25	9,681	711	51	54	22	335
vsftpd	0	5	5	0	1	4	5	5,830	305	121	35	0	21
OpenSSH	3	3	3	0	1	2	5	14,370	894	84	33	0	135

Table 1: Overview of all the programs and updates used in our evaluation.

checkpointing the old version. This strategy, however, would block the running version for the time to relink the program and prelink the shared libraries to remap nonrelocatable immutable objects (e.g., global variables). Fortunately, we have observed very stable immutable behavior for such objects. As a result, our current strategy is to simply run the analysis and the relinking operations offline. If a mismatch is ever found after quiescence—although we have never encountered this scenario in practice—we could expand the set of immutable objects, resume execution, allow relinking operations in the background, and repeat the entire procedure until convergence.

7. Violating Assumptions

We report on the key issues that might allow programs found "*in the wild*" to violate MCR's annotationless semantics—excluding annotations required by complex semantic updates. The intention is to foster future research in the field, but also allow programmers to design more "*live update-friendly*" (and better) software. Profile-guided quiescence might require extra manual effort in the following cases: (i) missing stalling points in profile data (i.e., not covered by the test workload)—weakens convergence guarantees; (ii) misclassified stalling points in profile data (e.g., an external library call used to synchronize internal events)—weakens convergence or deadlock guarantees; (iii) overly conservative stalling point policies (i.e., promoting a semi-persistent stalling point to a blocking point)—weakens convergence guarantees. The latter is the only case we found to be relatively common in practice. In the worst case, this requires extra control-flow remapping operations not automatically performed by MCR. A possible solution is to extend our record-replay strategy to code paths leading to volatile quiescent points, but this may also introduce nontrivial run-time overhead. While annotations are possible, we believe these cases are better dealt with at design time. Purely event-driven servers (e.g., nginx) are an example, with only persistent quiescent points allowed during execution.

Further, state-driven mutable record-replay might require extra manual effort in the following cases: (i) unsupported immutable objects (e.g., process-specific IDs with no namespace support, such as System V shared memory IDs, stored into a global variable); (ii) nondeterministic process model behavior (e.g.,, a server dynamically adjusting worker processes depending on the load); (iii) nonreplayed operations actively trying to violate MCR semantics (e.g., a server aborting initialization when detecting another running instance). We believe these cases to be relatively common, the last two in particular—Apache httpd being an example. While the last case is trivial to address at design time, the others require better run-time support and more sophisticated process mapping strategies.

Finally, mutable GC-style tracing shares a number problematic cases that require extra manual effort with prior GC strategies for C [58]. Examples include storing a pointer on persistent storage or relying on specialized encoding to store pointer values in memory. In the MCR model, these cases are best described as examples of immutable objects not supported by our run-time system. While seemingly uncommon and easy to tackle at design time, we found 1 real-world program (i.e., nginx) using pointer encoding in our evaluation.

8. Evaluation

We have implemented MCR on Linux (x86), with support for generic userspace C programs. Static instrumentation—implemented in C++ using the LLVM v3.3 API [47]—accounts for 728 (quiescence profiler) and 8064 LOC [1] (other MCR components). MCR instrumentation relies on a static library, implemented in C in 4,531 LOC. Dynamic instrumentation—implemented in C in a preloaded shared library—accounts for 3,476 (quiescence profiler) and 21,133 LOC (other MCR components). The `mcr-ctl` tool, which allows users to signal updates to the MCR backend using UNIX domain sockets, is implemented in C in 493 LOC.

We evaluated MCR on a workstation running Linux v3.5.0 (x86) and equipped with a 4-core 3.0 Ghz AMD Phenom II X4 B95 processor and 8 GB of RAM. For our evaluation, we considered the two most popular open-source web servers—Apache httpd (v.2.2.23) and nginx (v0.8.54)—and, for comparison purposes, a popular ftp server—vsftpd (v1.1.0)—and a popular ssh server—the OpenSSH daemon (v3.5p1). The former [22, 35, 36, 48, 55] and the latter [22, 36, 55] are by far the most used programs (and versions) in prior live update solutions. We configured our programs (and benchmarks) with their default settings and instructed Apache httpd to use the worker module with 2 servers and 25 worker threads without dynamically adjusting its process model. We benchmarked our programs using the Apache benchmark (AB) [1] (Apache httpd and nginx), the FTP benchmark included in pyftpdlib [6] (vsftpd), and the built-in test suite (OpenSSH). We repeated all our experiments 11 times and report the median.

[1] Source lines of code reported by David Wheeler's SLOCCount.

	Precise pointers						**Likely pointers**					
	Total	**Static**		**Dynamic**		**Lib**	**Total**	**Static**		**Dynamic**		**Lib**
	Ptr	**Src**	**Targ**	**Src**	**Targ**	**Targ**	**Ptr**	**Src**	**Targ**	**Src**	**Targ**	**Targ**
Apache httpd	2,373	2,272	2,151	101	219	3	16,252	185	2,050	16,067	14,201	1
nginx	1,242	1,226	1,214	16	26	2	4,049	51	293	3,998	3,755	1
nginx$_{reg}$	2,049	1,226	1,455	823	592	2	3,522	51	149	3,471	3,372	1
vsftpd	149	148	131	1	4	14	6	6	0	0	6	0
OpenSSH	237	226	211	11	19	7	56	5	16	51	32	8

Table 2: Mutable GC-style tracing statistics aggregated after the execution of our benchmarks.

Our evaluation answers 4 key questions: (i) *Engineering effort*: How much engineering effort is required to adopt MCR? (ii) *Performance*: Does MCR yield low run-time overhead? (iii) *Update time*: Does MCR yield reasonable update time? (iv) *Memory usage*: How much memory does MCR use?

Engineering effort. To evaluate the engineering effort required to deploy our techniques, we first prepared our test programs for MCR and profiled their quiescent points. To put together an appropriate execution-stalling workload for our quiescence profiler, we used three simple test scripts. The first script—used for Apache httpd and nginx—opens a number of long-lived HTTP connections and issues one HTTP request for a very large file in parallel. The second and third scripts—used for OpenSSH and vsftpd, respectively—open a number of long-lived SSH (or FTP) connections—in authentication/post-authentication state–and, for vsftpd, issue one FTP request for a very large file in parallel. Note that our workload is not meant to be necessarily general—Apache httpd, for instance, supports plugins that can potentially create an arbitrary number of new volatile stalling points—but rather to cover all the common stalling points stressed by the execution of our benchmarks. Next, we considered a number of incremental releases following our original program versions, and prepared them for MCR. In particular, we selected 5 updates for Apache httpd (v2.2.23-v2.3.8), vsftpd (v1.1.0-v2.0.2), and OpenSSH (v3.5-v3.8), and 25 updates for nginx (v0.8.54-v1.0.15)—nginx's tight release cycle generally produces incremental patches that are much smaller than those of all the other programs considered. Table 1 presents our findings.

The first six grouped columns summarize the data generated by our quiescence profiler. The first two columns detail the number of short-lived and long-lived thread classes identified during the test workload. The short-lived thread classes detected derive from deamonification (all the programs except vsftpd), initialization tasks (httpd), or `exec()`ing other helper programs (OpenSSH). The long-lived thread classes detected, in turn, originated a total of 18 stalling points, 15 of which are external (*Ext*). OpenSSH and vsftpd's simple process model resulted in no internal stalling point (*Int*) and only 1 persistent stalling point (*Per*) associated to the master process. Finally, all the server programs reported volatile stalling points (*Vol*) with the exception of nginx, given its rigorous event-driven programming model. The profile data reported was used as is for our quiescence instrumentation without extra annotations.

The second two grouped columns provide an overview of the updates considered for each program and the number of LOC added, deleted, or modified by them. As shown in the table, we manually processed more than 40,000 LOC across the 40 updates considered. The third group shows the number of functions, variables, and types changed (i.e., added, deleted, or modified) by the updates, with a total of 2,739, 284, and 170 changes (respectively). The fourth group, finally, shows the engineering effort (LOC) in terms of annotations required to prepare our programs using the default stalling point policies and the extra state transfer code required by our updates.

As shown in the table, the annotation effort required by MCR is low. Apache httpd required only 8 LOC to prevent the server from aborting prematurely after actively detecting its own running instance and 10 LOC to ensure deterministic custom allocation behavior. Both changes were necessary to allow our state-driven mutable record-replay to complete correctly. Further, nginx required 22 LOC to annotate a number of global pointers using special data encoding—storing metadata in the 2 least significant bits. The latter is necessary for our mutable GC-style tracing strategy to interpret pointer values correctly. Supporting all the other nonpersistent quiescent points profiled with no application redesign, on the other hand, required an extra 82 LOC for vsftpd, 49 LOC for OpenSSH, and 163 LOC for Apache httpd. In addition, we had to manually write 793 LOC to allow state transfer to complete correctly across all the updates considered. The extra code was necessary to implement complex state changes that could not be automatically remapped by MCR. Moreover, two of our test programs rely on custom allocation schemes. nginx uses regions [16] and slab-like allocations [20]. Apache httpd uses nested regions [16]. Instrumenting custom allocation schemes—other than regular *libc* allocations—increases updatability, but also introduces extra complexity and overhead. To analyze the tradeoff, we allowed MCR to instrument only nginx's region allocator using 1 extra annotation—nested regions and slabs are not yet supported by our instrumentation—and instructed our tracing strategy to produce aggregated quiescent-time statistics—for both precisely and conservatively identified program pointers—after the execution of our benchmarks (Table 2).

In the two cases, the table reports the total number of pointers detected (*Ptr*), per-region source pointers (*Src*), and per-

	Unblock	+SInstr	+DInstr	+QDetect
Apache httpd	0.977	1.040	1.043	1.047
nginx	1.000	1.000	1.000	1.000
nginx$_{\mathbf{reg}}$	1.000	1.175	1.192	1.186
vsftpd	1.024	1.027	1.028	1.028
OpenSSH	0.999	0.999	1.001	1.001

Table 3: Benchmark run time normalized against the baseline.

region pointed target objects (*Targ*). Objects are classified into *Static* (e.g., global variables, but also strings, which attracted the vast majority of likely pointers into static objects), *Dynamic* (e.g., heap objects), *Lib* (i.e., static/dynamic shared library objects). We draw three main conclusions from our analysis. First, there are many (23,885) legitimate cases of likely pointers—we sampled a number of cases to check for accuracy—which cannot be ignored at state transfer time. Prior whole-program strategies would delegate this heroic effort entirely to the user. Second, we note a number of program pointers into shared library state (28+11). This confirms the importance of marking shared library objects as immutable if library state transfer is desired. Finally, our results confirm the impact of allocator instrumentation. Apache httpd's uninstrumented allocations produce the highest number of likely pointers (16,067), with nginx following with 3,998. Our (partial) allocator instrumentation on nginx (nginx$_{reg}$) can mitigate, but not eliminate this problem (3,471 likely pointers). Further, even in the case of a fully instrumented allocator (vsftpd and OpenSSH), we still note a number of likely pointers originating from legitimate type-unsafe idioms (6 and 56, respectively), which suggests annotations in prior solutions can hardly be eliminated even in the optimistic cases. Overall, we regard MCR as a major step forward over prior solutions: (i) much less annotation effort is required to deploy MCR and support updates; (ii) much less inspection effort is required to identify issues with pointers, allocators, and shared libraries.

Performance. To evaluate the run-time overhead imposed by MCR, we measured the time to complete the execution of our benchmarks compared to the baseline. We configured the Apache benchmark to issue 100,000 requests and retrieve a 1 KB HTML file. We configured the pyftpdlib benchmark to allow 100 users and retrieve a 1 MB file. In all the experiments, we observed marginal CPU utilization increase (i.e., $< 3\%$). Run-time overhead results, in turn, are shown in Table 3. We comment on results for uninstrumented region allocators first. As expected, unblockification alone (*Unblock*) introduces marginal run-time overhead (2.4% in the worst case for vsftpd). The reported speedups are well within the noise caused by memory layout changes [53]. When combined with our static instrumentation (+*SInstr*), the overhead is somewhat more visible (4% worst-case overhead for Apache httpd). The latter originates from our allocator instrumentation, which maintains in-band metadata for mutable GC-style tracing. The overhead is fairly stable when adding our dynamic instrumentation (+*DInstr*)—which also tracks all the allocations

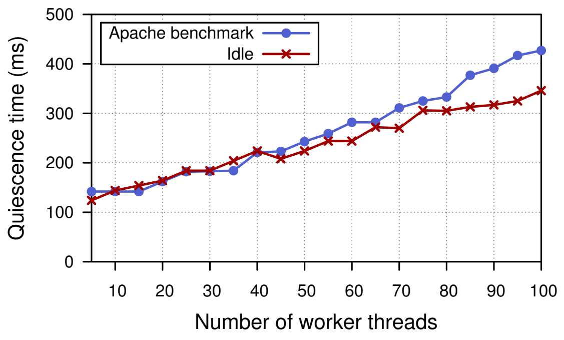

Figure 5: Quiescence time vs. number of worker threads.

from shared libraries, other than maintaining process hierarchy metadata. Finally, our quiescence detection instrumentation (+*QDetect*)—which essentially only introduces extra RCU calls to mark per-thread quiescent states—introduces, as expected, marginal overhead. This translates to the final 4.7% worst-case overhead (Apache httpd) for the entire solution.

To further investigate the overhead on allocator operations, we instrumented all the SPEC CPU2006 benchmarks with our static and dynamic allocator instrumentation. We reported a 5% worst-case overhead across all the benchmarks, with the exception of `perlbench` (36% worst-case overhead), a memory-intensive benchmark which essentially provides a microbenchmark for our instrumentation. Our results confirm the impact of allocator instrumentation on run-time performance. This is also evidenced by the cost of our region instrumentation on nginx, which incurs 19.2% overhead in the worst case (nginx$_{\mathbf{reg}}$ in Table 3). While our implementation may be poorly optimized for nginx's allocation behavior, this extra cost does evidence the tradeoff between precision of our GC-style tracing strategy and run-time performance, which MCR users should take into account when deploying our solution.

Our results demonstrate that MCR overhead is generally lower [49] or comparable [35, 54, 55] to prior solutions. The extra costs (unblockification and allocator instrumentation) provide much better quiescence guarantees and drastically simplify state transfer. For example, the tag-free traversal strategy proposed in [35] would eliminate the overhead on allocator operations, but at the cost of no support for interior or `void*` pointers without pervasive user annotations.

Update time. To evaluate the update time—the time the program is unavailable during the update—we analyzed its three main components in detail: (i) quiescence time; (ii) control-flow transfer time; (ii) state transfer time. To evaluate quiescence time, we allowed our quiescence detection protocol to complete during the execution of our benchmarks or during idle time. We found that programs with only external quiescent points—vsftpd and OpenSSH—or rarely activated internal points—nginx, whose master process is only activated for crash recovery purposes—always converge in comparable time in a workload-independent way (around 125 ms, with the first 100 ms directly attributable to our default unblockifi-

	Static	Run-time	Update-time
Apache httpd	2.187	2.100	7.685
nginx	2.358	4.111	4.656
nginx$_{reg}$	2.358	4.330	4.829
vsftpd	3.352	5.836	14.170
OpenSSH	2.480	3.047	11.814

Table 4: Memory usage normalized against the baseline.

cation latency), given that our protocol is essentially reduced to barrier synchronization. Apache httpd is more interesting, with several live internal points interacting across its worker threads. Figure 5 depicts the time Apache httpd requires to quiesce for an increasing number of worker threads, resulting in a maximum quiescent time of 184 ms with 25 threads (default value) and 427 ms with 100 threads (Apache httpd's recommended maximum value). The figure confirms our protocol scales well with the number of threads and converges quickly even under heavy load once external events are blocked. Both properties stem from our RCU-based design.

To evaluate control-flow transfer time, we measured the time to complete state-driven mutable record-replay across versions. We found that both the record and replay phase complete in comparable time (less than 40 ms), with modest overhead (1-45%) compared to the original initialization time across all our test programs. Finally, to evaluate state transfer time, we allowed a user to connect to our test programs—similar to the experimental setup adopted in prior solutions [35]—and measured the time to remap the state across versions using mutable GC-style tracing. Figure 6 depicts the resulting time as a function of the number of type transformations (%) we artificially injected into the new program version—using a source-to-source transformation, similar to [29]. Despite an average of 365,830 type transformations operated by our precise GC-style tracing strategy at 100% coverage, we observed a relatively low impact on state transfer time (462 ms in the worst case for Apache httpd). This behavior stems from optimizations operated in our tracing strategy, which relies on lookup tables and splay trees—an idea borrowed by bounds checkers [8, 25, 59]—to efficiently remap objects/types and pointers, respectively. While generally higher than prior solutions, we believe our update time to be sustainable for most programs and more optimizations possible. The benefit is full-coverage multiprocess state transfer able to conservatively handle C's ambiguous type semantics.

Memory usage. MCR instrumentation leads to larger binary sizes and run-time memory footprints. This stems from mutable GC-style tracing metadata, process hierarchy metadata, the in-memory log used for state-driven mutable record-replay, and the libraries required to support all our techniques.

Table 4 evaluates the MCR impact on our test programs. The static memory overhead (235.2% worst-case overhead for vsftpd) measures the impact of our static instrumentation on the original binary size. The run-time overhead (483.6% worst-case overhead for vsftpd), in turn, measures the impact of static

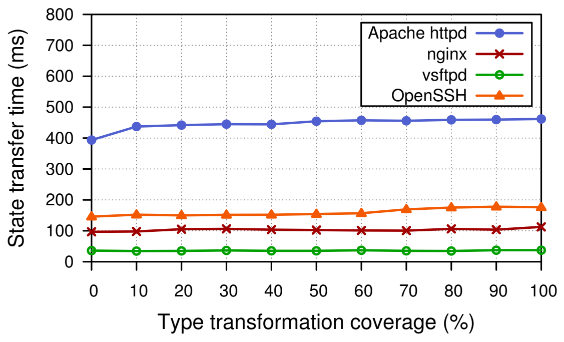

Figure 6: State transfer time vs. type transformation coverage.

and instrumentation (and support libraries) on the resident set size (RSS) observed at runtime, right after initialization—we found the overhead to be lower or comparable during the execution of our benchmarks. The update-time overhead, finally, shows the maximum RSS overhead we observed at update time, accounting for an extra running instance of the program and auxiliary data structures allocated for mutable GC-style tracing (1317.0% worst-case overhead for vsftpd).

As expected, MCR requires more memory than prior in-place live update solutions, while being, at the same time, comparable to other whole-program solutions that rely on data type tags. A tag-free tracing implementation such as the one proposed in [35] would help reduce the overhead in this case as well, but also impose all the important limitations discussed earlier. MCR favors annotationless semantics over memory usage, given the increasingly low cost of RAM in these days.

9. Conclusion

This paper presented *Mutable Checkpoint-Restart* (*MCR*), a new live update technique for generic long-running C programs. MCR's design goals dictate support for arbitrary software updates and minimal annotation effort for real-world multiprocess and multithreaded programs. To achieve these ambitious goals, the MCR model carefully decomposes the live update problem into three well-defined tasks: (i) checkpoint the running version; (ii) remap control flow after restart; (iii) remap program state after restart. For each of these tasks, MCR introduces novel ideas to drastically reduce the number of annotations and provide effective solutions to previously deemed difficult problems. *Profile-guided quiescence detection* relies on long-lived blocking call profiling to identify quiescent points in the program and implement the first race-free and deadlock-free generic quiescence detection protocol with convergence guarantees. *State-driven mutable record-replay* builds on well-established record-replay techniques to reuse existing code paths and implement the first automated control-flow transfer strategy that tolerates changes to the process model and long-lived thread behavior. *Mutable GC-style tracing* combines well-established precise and conservative garbage collection techniques to implement the first automated state transfer strategy that can safely reconstruct and transform the program state after restart even without full annotation and instrumentation coverage. Our experience with programs found "*in the wild*" shows that our techniques are practical, efficient, and significantly raise the bar in terms of deployability, reliability, and maintenance effort over all the prior solutions.

References

[1] Apache benchmark (AB) http://httpd.apache.org/docs/2.0/programs/ab.html.

[2] CRIU. http://criu.org.

[3] Cryopid2. http://sourceforge.net/projects/cryopid2.

[4] Ksplice performance on security patches. http://www.ksplice.com/cve-evaluation.

[5] OpenVZ. http://wiki.openvz.org.

[6] pyftpdlib. https://code.google.com/p/pyftpdlib.

[7] Sameer Ajmani, Barbara Liskov, Liuba Shrira, and Dave Thomas. Modular software upgrades for distributed systems. In *Proc. of the 20th European Conf. on Object-Oriented Programming*, pages 452–476, 2006.

[8] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proc. of the 18th USENIX Security Symp.*, pages 51–66, 2009.

[9] Gautam Altekar, Ilya Bagrak, Paul Burstein, and Andrew Schultz. OPUS: Online patches and updates for security. In *Proc. of the 14th USENIX Security Symp.*, pages 19–19, 2005.

[10] Gautam Altekar and Ion Stoica. ODR: Output-deterministic replay for multicore debugging. In *Proc. of the 22nd ACM Symp. on Operating Systems Principles*, pages 193–206, 2009.

[11] Jason Ansel, Kapil Arya, and Gene Cooperman. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *Proc. of the IEEE Int'l Symp. on Parallel and Distributed Processing*, pages 1–12, 2009.

[12] Jeff Arnold and M. Frans Kaashoek. Ksplice: Automatic rebootless kernel updates. In *Proc. of the Fourth ACM European Conf. on Computer Systems*, pages 187–198, 2009.

[13] J. Baker, A. Cunei, T. Kalibera, F. Pizlo, and J. Vitek. Accurate garbage collection in uncooperative environments revisited. *Concurr. Comput.: Pract. Exper.*, 21(12):1572–1606, 2009.

[14] Andrew Baumann, Jonathan Appavoo, Robert W. Wisniewski, Dilma Da Silva, Orran Krieger, and Gernot Heiser. Reboots are for hardware: Challenges and solutions to updating an operating system on the fly. In *Proc. of the USENIX Annual Tech. Conf.*, pages 1–14, 2007.

[15] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W. Wisniewski, and Jeremy Kerr. Providing dynamic update in an operating system. In *Proc. of the USENIX Annual Tech. Conf.*, page 32, 2005.

[16] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Reconsidering custom memory allocation. In *Proc. of the 17th ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–12, 2002.

[17] E. W. Biederman. Multiple instances of the global Linux namespaces. In *Proc. of the Linux Symposium*, 2006.

[18] Hans-J. Boehm. Bounding space usage of conservative garbage collectors. In *Proc. of the 29th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 93–100, 2002.

[19] Hans-Juergen Boehm. Space efficient conservative garbage collection. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 197–206, 1993.

[20] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *Proc. of the USENIX Summer Tech. Conf.*, page 6, 1994.

[21] Haibo Chen, Rong Chen, Fengzhe Zhang, Binyu Zang, and Pen-Chung Yew. Live updating operating systems using virtualization. In *Proc. of the Second Int'l Conf. on Virtual Execution Environments*, pages 35–44, 2006.

[22] Haibo Chen, Jie Yu, Rong Chen, Binyu Zang, and Pen-Chung Yew. POLUS: A POwerful live updating system. In *Proc. of the 29th Int'l Conf. on Software Eng.*, pages 271–281, 2007.

[23] Ronald F. DeMara, Yili Tseng, and Abdel Ejnioui. Tiered algorithm for distributed process quiescence and termination detection. *IEEE Trans. Parallel Distrib. Syst.*, 18(11):1529–1538, 2007.

[24] Mathieu Desnoyers, Paul E. McKenney, Alan S. Stern, Michel R. Dagenais, and Jonathan Walpole. User-level implementations of read-copy update. *IEEE Trans. Parallel Distrib. Syst.*, 23(2):375–382, 2012.

[25] Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for C with very low overhead. In *Proc. of the 28th Int'l Conf. on Software Eng.*, pages 162–171, 2006.

[26] Tudor Dumitras and Priya Narasimhan. Why do upgrades fail and what can we do about it?: Toward dependable, online upgrades in enterprise system. In *Proc. of the 10th Int'l Conf. on Middleware*, pages 1–20, 2009.

[27] R. S. Fabry. How to design a system in which modules can be changed on the fly. In *Proc. of the Second Int'l Conf. on Software Eng.*, pages 470–476, 1976.

[28] Ophir Frieder and Mark E. Segal. On dynamically updating a computer program: From concept to prototype. *J. Syst. Softw.*, 14(2):111–128, 1991.

[29] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Safe and automatic live update for operating systems. In *Proc. of the 18th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 279–292, 2013.

[30] Wolfram Gloger. ptmalloc. http://www.malloc.de/en.

[31] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. R2: An application-level kernel for record and replay. In *Proc. of the 8th USENIX Symp. on Operating Systems Design and Implementation*, page 193–208, 2008.

[32] Deepak Gupta and Pankaj Jalote. On-line software version change using state transfer between processes. *Softw. Pract. and Exper.*, 23(9):949–964, 1993.

[33] Deepak Gupta, Pankaj Jalote, and Gautam Barua. A formal framework for on-line software version change. *IEEE Trans. Softw. Eng.*, 22(2):120–131, 1996.

[34] Paul H. Hargrove and Jason C. Duell. Berkeley lab checkpoint/restart (BLCR) for Linux clusters. *Journal of Physics: Conference Series*, 46(1):494, 2006.

[35] C. M Hayden, E. K Smith, M. Denchev, M. Hicks, and J. S Foster. Kitsune: Efficient, general-purpose dynamic software updating for C. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, 2012.

[36] C.M. Hayden, E.K. Smith, E.A. Hardisty, M. Hicks, and J.S. Foster. Evaluating dynamic software update safety using systematic testing. *IEEE Trans. Softw. Eng.*, 38(6):1340–1354, 2012.

[37] Fergus Henderson. Accurate garbage collection in an uncooperative environment. In *Proc. of the 3rd Int'l Symp. on Memory management*, pages 150–156, 2002.

[38] Martin Hirzel and Amer Diwan. On the type accuracy of garbage collection. In *Proc. of the 2nd Int'l Symp. on Memory Management*, pages 1–11, 2000.

[39] Martin Hirzel, Amer Diwan, and Johannes Henkel. On the usefulness of type and liveness accuracy for garbage collection and leak detection. *ACM Trans. Program. Lang. Syst.*, 24(6):593–624, 2002.

[40] Petr Hosek and Cristian Cadar. Safe software updates via multi-version execution. In *Proc. of the Int'l Conf. on Software Engineering*, pages 612–621, 2013.

[41] Jakub Jelinek. Prelink http://people.redhat.com/jakub/prelink.pdf.

[42] Paul Johnson and Neeraj Mittal. A distributed termination detection algorithm for dynamic asynchronous systems. In *Proc. of the 29th IEEE Int'l Conf. on Distributed Computing Systems*, pages 343–351, 2009.

[43] Clemens Kolbitsch, Engin Kirda, and Christopher Kruegel. The power of procrastination: Detection and mitigation of execution-stalling malicious code. In *Proc. of the 18th ACM Conf. on Computer and Communications Security*, pages 285–296, 2011.

[44] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. *IEEE Trans. Softw. Eng.*, 16(11):1293–1306, 1990.

[45] Ilia Kravets and Dan Tsafrir. Feasibility of mutable replay for automated regression testing of security updates. In *Workshop on Runtime Environments, Systems, Layering and Virtualized Environments*, 2012.

[46] Oren Laadan, Nicolas Viennot, and Jason Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *Proc. of the Int'l Conf. on Measurement and Modeling of Computer Systems*, pages 155–166, 2010.

[47] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. of the Int'l Symp. on Code Generation and Optimization*, page 75, 2004.

[48] K. Makris and R. Bazzi. Immediate multi-threaded dynamic software updates using stack reconstruction. In *Proc. of the USENIX Annual Tech. Conf.*, pages 397–410, 2009.

[49] Kristis Makris and Kyung Dong Ryu. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *Proc. of the Second ACM European Conf. on Computer Systems*, pages 327–340, 2007.

[50] Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Proc. of the 10th Int'l Conf. Parallel and Distributed Computing and Systems*, pages 509–518, 1998.

[51] Paul E. McKenney and Jonathan Walpole. What is RCU, fundamentally? http://lwn.net/Articles/262464.

[52] Neeraj Mittal, S. Venkatesan, and Sathya Peri. A family of optimal termination detection algorithms. *Distributed Computing*, 20(2):141–162, 2007.

[53] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proc. of the 14th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 265–276, 2009.

[54] Iulian Neamtiu and Michael Hicks. Safe and timely updates to multi-threaded programs. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 13–24, 2009.

[55] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical dynamic software updating for C. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 72–83, 2006.

[56] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. PRES: Probabilistic replay with execution sketching on multiprocessors. In *Proc. of the 22nd ACM Symp. on Operating Systems Principles*, pages 177–192, 2009.

[57] Shaya Potter and Jason Nieh. Reducing downtime due to system maintenance and upgrades. In *Proc. of the 19th USENIX Systems Administration Conf.*, pages 6–6, 2005.

[58] Jon Rafkind, Adam Wick, John Regehr, and Matthew Flatt. Precise garbage collection for C. In *Proc. of the Int'l Symp. on Memory management*, pages 39–48, 2009.

[59] Olatunji Rowase and Monica S. Lam. A practical dynamic buffer overflow detector. In *Proc. of the 11th Annual Symp. on Network and Distr. System Security*, pages 159–169, 2004.

[60] Maxim Siniavine and Ashvin Goel. Seamless kernel updates. In *Proc. of the 43rd Int'l Conf. on Dependable Systems and Networks*, 2013.

[61] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: a lightweight extension for rollback and deterministic replay for software debugging. In *Proc. of the USENIX Annual Tech. Conf.*, page 3, 2004.

[62] Dinesh Subhraveti and Jason Nieh. Record and transplay: Partial checkpointing for replay debugging across heterogeneous systems. In *Proc. of the Int'l Conf. on Measurement and Modeling of Computer Systems*, pages 109–120, 2011.

[63] Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. Dynamic software updates: a VM-centric approach. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 1–12, 2009.

[64] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. *ACM Trans. Comput. Syst.*, 24(4):333–360, 2006.

[65] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. *ACM Trans. Comput. Syst.*, 23(1):77–110, 2005.

[66] Yves Vandewoude, Peter Ebraert, Yolande Berbers, and Theo D'Hondt. Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. *IEEE Trans. Softw. Eng.*, 33(12):856–868, 2007.

[67] Nicolas Viennot, Siddharth Nair, and Jason Nieh. Transparent mutable replay for multicore debugging and patch validation. In *Proc. of the 18th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 127–138, 2013.

Chapter 3

Back to the Future: Fault-tolerant Live Update with Time-travelling State Transfer

Back to the Future: Fault-tolerant Live Update with Time-traveling State Transfer

Cristiano Giuffrida Călin Iorgulescu Anton Kuijsten Andrew S. Tanenbaum
Vrije Universiteit, Amsterdam
{giuffrida, calin.iorgulescu, akuijst, ast}@cs.vu.nl

Abstract

Live update is a promising solution to bridge the need to frequently update a software system with the pressing demand for high availability in mission-critical environments. While many research solutions have been proposed over the years, systems that allow software to be updated on the fly are still far from reaching widespread adoption in the system administration community. We believe this trend is largely motivated by the lack of tools to automate and validate the live update process. A major obstacle, in particular, is represented by state transfer, which existing live update tools largely delegate to the programmer despite the great effort involved.

This paper presents *time-traveling state transfer*, a new automated and fault-tolerant live update technique. Our approach isolates different program versions into independent processes and uses a semantics-preserving state transfer transaction—across multiple *past*, *future*, and *reversed* versions—to validate the program state of the updated version. To automate the process, we complement our live update technique with a generic state transfer framework explicitly designed to minimize the overall programming effort. Our time-traveling technique can seamlessly integrate with existing live update tools and automatically recover from arbitrary run-time and memory errors in any part of the state transfer code, regardless of the particular implementation used. Our evaluation confirms that our update techniques can withstand arbitrary failures within our fault model, at the cost of only modest performance and memory overhead.

1 Introduction

In the era of pervasive and cloud computing, we are witnessing a major paradigm shift in the way software is developed and released. The growing demand for new features, performance enhancements, and security fixes translates to more and more frequent software updates made available to the end users. In less than a decade, we quickly transitioned from Microsoft's "*Patch Tuesday*" [39] to Google's "*perpetual beta*" development model [67] and Facebook's tight release cycle [61], with an update interval ranging from days to a few hours.

With more frequent software updates, the standard halt-update-restart cycle is irremediably coming to an impasse with our growing reliance on nonstop software operations. To reduce downtime, system administrators often rely on "*rolling upgrades*" [29], which typically update one node at a time in heavily replicated software systems. While in widespread use, rolling upgrades have a number of important shortcomings: (i) they require redundant hardware, which may not be available in particular environments (e.g., small businesses); (ii) they cannot normally preserve program state across versions, limiting their applicability to stateless systems or systems that can tolerate state loss; (iii) in heavily replicated software systems, they lead to significant update latency and high exposure to "*mixed-version races*" [30] that can cause insidious update failures. A real-world example of the latter has been reported as "*one of the biggest computer errors in banking history*", with a single-line software update mistakenly deducting about $15 million from over 100,000 customers' accounts [43].

Live update—the ability to update software on the fly while it is running with no service interruption—is a promising solution to the update-without-downtime problem which does not suffer from the limitations of rolling upgrades. A key challenge with this approach is to build trustworthy update systems which come as close to the usability and reliability of regular updates as possible. A significant gap is unlikely to encourage adoption, given that experience shows that administrators are often reluctant to install even regular software updates [69].

Surprisingly, there has been limited focus on automating and validating generic live updates in the literature. For instance, traditional live update tools for C programs seek to automate only basic type transformations [62,64], while more recent solutions [48] make little effort to spare the programmer from complex tasks like *pointer transfer* (§5). Existing live update validation tools [45–47], in turn, are only suitable for *offline* testing, add no *fault-tolerant* capabilities to the update process, require *manual* effort, and are inherently *update timing*-centric. The typical strategy is to verify that a given test suite completes correctly—according to some manually selected [45, 46] or provided [47] specification—regardless of the particular time when the update is applied. This testing method stems from the extensive focus on live update timing in the literature [44].

Much less effort has been dedicated to automating and validating *state transfer* (*ST*), that is, initializing the state of a new version from the old one (§2). This is somewhat surprising, given that ST has been repeatedly recognized as a challenging and error-prone task by many researchers [13, 22, 23, 57] and still represents a major obstacle to the widespread adoption of live update systems. This is also confirmed by the commercial success of Ksplice [11]—already deployed on over 100,000 production servers [4]—explicitly tailored to small security patches that hardly require any state changes at all (§2).

In this paper, we present *time-traveling state transfer* (TTST), a new live update technique to automate and validate generic live updates. Unlike prior live update testing tools, our validation strategy is *automated* (manual effort is never strictly required), *fault-tolerant* (detects and immediately recovers from any faults in our fault model with no service disruption), *state-centric* (validates the ST code and the full integrity of the final state), and *blackbox* (ignores ST internals and seamlessly integrates with existing live update tools). Further, unlike prior solutions, our fault-tolerant strategy can be used for *online* live update validation in the field, which is crucial to automatically recover from unforeseen update failures often originating from differences between the testing and the deployment environment [25]. Unlike commercial tools like Ksplice [11], our techniques can also handle complex updates, where the new version has significantly different code and data than the old one.

To address these challenges, our live update techniques use two key ideas. First, we confine different program versions into independent processes and perform *process-level* live update [35]. This strategy simplifies state management and allows for automated state reasoning and validation. Note that this is in stark contrast with traditional *in-place* live update strategies proposed in the literature [10–12,22,23,58,62,64], which “glue” changes directly into the running version, thus mixing code and data from different versions in memory. This mixed execution environment complicates debugging and testing, other than introducing address space fragmentation (and thus run-time performance overhead) over time [35].

Second, we allow two process-level ST runs using the time-traveling idea. With time travel, we refer to the ability to navigate backward and forward across program state versions using ST. In particular, we first allow a *forward* ST run to initialize the state of the new version from the old one. This is already sufficient to implement live update. Next, we allow a second *backward* run which implements the reverse state transformation from the new version back to a copy of the old version. This is done to validate—and safely rollback when necessary—the ST process, in particular to detect specific classes of programming errors (i.e., memory errors) which would otherwise leave the new version in a corrupted state. To this end, we compare the program state of the original version against the final state produced by our overall transformation. Since the latter is semantics-preserving by construction, we expect differences in the two states *only* in presence of memory errors caused by the ST code.

Our contribution is threefold. First, we analyze the state transfer problem (§2) and introduce *time-traveling state transfer* (§3, §4), an automated and fault-tolerant live update technique suitable for online (or offline) validation. Our TTST strategy can be easily integrated into existing live update tools described in the literature, allowing system administrators to seamlessly transition to our techniques with no extra effort. We present a TTST implementation for user-space C programs, but the principles outlined here are also applicable to operating systems, with the process abstraction implemented using lightweight protection domains [72], software-isolated processes [53], or hardware-isolated processes and microkernels [50, 52]. Second, we complement our technique with a TTST-enabled state transfer framework (§5), explicitly designed to allow arbitrary state transformations and high validation surface with minimal programming effort. Third, we have implemented and evaluated the resulting solution (§6), conducting fault injection experiments to assess the fault tolerance of TTST.

2 The State Transfer Problem

The state transfer problem, rigorously defined by Gupta for the first time [41], finds two main formulations in the literature. The traditional formulation refers to the live initialization of the data structures of the new version from those of the old version, potentially operating structural or semantic data transformations on the fly [13]. Another formulation also considers the execution state, with the additional concern of remapping the call stack and the instruction pointer [40, 57]. We here adopt the former definition and decouple *state transfer* (*ST*) from *control-flow transfer* (*CFT*), solely concerned with the execution state and subordinate to the particular update mechanisms adopted by the live update tool

```
--- a/drivers/md/dm-crypt.c
+++ b/drivers/md/dm-crypt.c
@@ -690,6 +690,8 @@ bad3:
 bad2:
     crypto_free_tfm(tfm);
 bad1:
+    /* Must zero key material before freeing */
+    memset(cc, 0, sizeof(*cc) + cc->key_size * sizeof(u8));
     kfree(cc);
     return -EINVAL;
 }
@@ -706,6 +708,9 @@ static void crypt_dtr(...)
         cc->iv_gen_ops->dtr(cc);
     crypto_free_tfm(cc->tfm);
     dm_put_device(ti, cc->dev);
+
+    /* Must zero key material before freeing */
+    memset(cc, 0, sizeof(*cc) + cc->key_size * sizeof(u8));
     kfree(cc);
 }
```

Listing 1: A security patch to fix an information disclosure vulnerability (CVE-2006-0095) in the Linux kernel.

```
--- a/example.c
+++ b/example.c
@@ -1,13 +1,12 @@
 struct s {
     int count;
-    char str[3];
-    short id;
+    int id;
+    char str[2];
     union u u;
-    void *ptr;
     int addr;
-    short *inner_ptr;
+    int *inner_ptr;
 } var;

 void example_init(char *str) {
-    snprintf(var.str, 3, "%s", str);
+    snprintf(var.str, 2, "%s", str);
 }
```

Listing 2: A sample patch introducing code and data changes that require state transfer at live update time.

considered—examples documented in the literature include manual control migration [40, 48], adaptive function cloning [58], and stack reconstruction [57].

We illustrate the state transfer problem with two update examples. Listing 1 presents a real-world security patch which fixes an information disclosure vulnerability (detailed in CVE-2006-0095 [5]) in the *md* (Multiple Device) driver of the Linux kernel. We sampled this patch from the dataset [3] originally used to evaluate Ksplice [11]. Similar to many other common security fixes, the patch considered introduces simple code changes that have no direct impact on the program state. The only tangible effect is the secure deallocation [24] of sensitive information on cryptographic keys. As a result, no state transformations are required at live update time. For this reason, Ksplice [11]—and other similar in-place live update tools—can deploy this update online with no state transfer necessary, allowing the new version to reuse the existing program state as is. Redirecting function invocations to the updated functions and resuming execution is sufficient to deploy the live update.

Listing 2 presents a sample patch providing a reduced test case for common code and data changes found in real-world updates. The patch introduces a number of type changes affecting a global `struct` variable (i.e., `var`)—with fields changed, removed, and reordered—and the necessary code changes to initialize the new data structure. Since the update significantly changes the in-memory representation of the global variable `var`, state transfer—using either automatically generated mapping functions or programmer-provided code—is necessary to transform the existing program state into a state compatible with the new version at live update time. Failure to do so would leave the new version in an invalid state after resuming execution. Section 5 shows how our state transfer strategy can effectively automate this particular update, while traditional live update tools would largely delegate this major effort to the programmer.

State transfer has already been recognized as a hard problem in the literature. Qualitatively, many researchers have described it as "*tedious implementation of the transfer code*" [13], "*tedious engineering efforts*" [22], "*tedious work*" [23]. Others have discussed speculative [14, 16, 37, 38] and practical [63] ST scenarios which are particularly challenging (or unsolvable) even with programmer intervention. Quantitatively, a number of user-level live update tools for C programs (Ginseng [64], STUMP [62], and Kitsune [48]) have evaluated the ST manual effort in terms of lines of code (LOC). Table 1 presents a comparative analysis, with the number of updates analyzed, initial source changes to implement their live update mechanisms (LU LOC), and extra LOC to apply all the updates considered (ST LOC). In the last column, we report a normalized ST impact factor (Norm ST IF), measured as the expected ST LOC necessary after 100 updates normalized against the initial LU LOC.

As the table shows, the measured impacts are comparable (the lower impact in Kitsune stems from the greater initial annotation effort required by program-level updates) and demonstrate that ST increasingly (and heavily) dominates the manual effort in long-term deploy-

	#Upd	LU LOC	ST LOC	Norm ST IF
Ginseng	30	140	336	8.0*x*
STUMP	13	186	173	7.1*x*
Kitsune	40	523	554	2.6*x*

Table 1: State transfer impact (normalized after 100 updates) for existing user-level solutions for C programs.

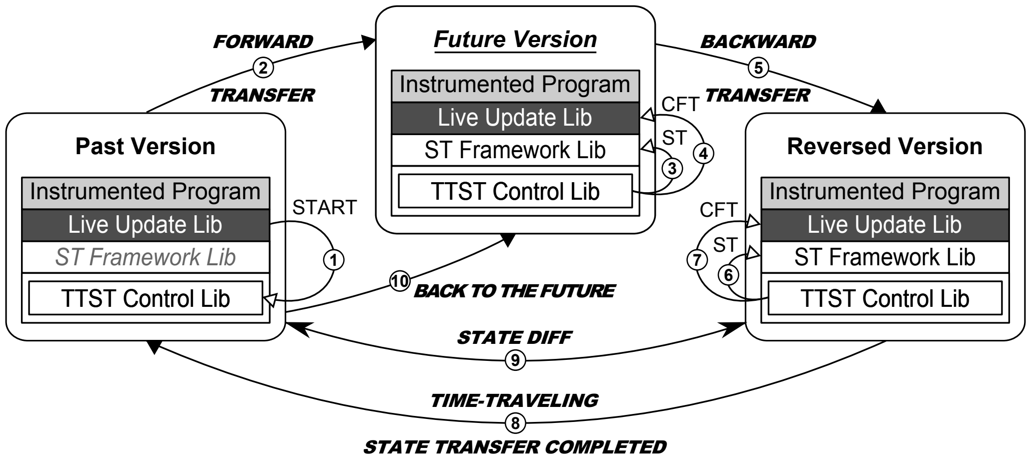

Figure 1: Time-traveling state transfer overview. The numbered arrows indicate the order of operations.

ment. Worse yet, any LOC-based metric underestimates the real ST effort, ignoring the atypical and error-prone programming model with nonstandard entry points, unconventional data access, and reduced testability and debuggability. Our investigation motivates our focus on automating and validating the state transfer process.

3 System Overview

We have designed our TTST live update technique with portability, extensibility, and interoperability in mind. This vision is reflected in our modular architecture, which enforces a strict separation of concerns and can support several possible live update tools and state transfer implementations. To use TTST, users need to statically instrument the target program in preparation for state transfer. In our current prototype, this is accomplished by a link-time transformation pass implemented using the LLVM compiler framework [56], which guarantees pain-free integration with existing GNU build systems using standard `configure` flags. We envision developers of the original program (i.e., users of our TTST technique) to gradually integrate support for our instrumentation into their development model, thus releasing live update-enabled software versions that can be easily managed by system administrators using simple tools. For this purpose, our TTST prototype includes `ttst-ctl`, a simple command-line tool that transparently interacts with the running program and allows system administrators to deploy live updates using our TTST technique with minimal effort. This can be simply done by using the following command-line syntax:

```
$ ttst-ctl `pidof program` ./new.bin
```

Runtime update functionalities, in turn, are implemented by three distinct libraries, transparently linked with the target program as part of our instrumentation process. The *live update library* implements the update mechanisms specific to the particular live update tool considered. In detail, the library is responsible to provide the necessary update timing mechanisms [46] (e.g., start the live update when the program is *quiescent* [46] and all the external events are blocked) and CFT implementation. The *ST framework library*, in turn, implements the logic needed to automate state transfer and accommodate user-provided ST code. The *TTST control library*, finally, implements the resulting *time-traveling state transfer* process, with all the necessary mechanisms to coordinate the different process versions involved.

Our TTST technique operates across three process instances. The first is the original instance running the old software version (*past version*, from now on). This instance initiates, controls, and monitors the live update process, in particular running the only trusted library code in our architecture with respect to our fault model (§4). The second is a newly created instance running the new software version (*future version*, from now on). This instance is instructed to reinitialize its state from the past version. The third process instance is a clone of the past version created at live update time (*reversed version*, from now on). This instance is instructed to reinitialize its state from the future version. Figure 1 depicts the resulting architecture and live update process.

As shown in the figure, the update process is started by the live update library in the past version. This happens when the library detects that an update is available and all the necessary update timing restrictions (e.g., *quiescence* [46]) are met. The *start* event is delivered to the past version's TTST control library, which sets out to initiate the *time-traveling state transfer* transaction. First, the library locates the new program version on the file system and creates the process instances for the future and reversed versions. Next, control is given to the future version's TTST control library, requesting to complete a *forward* state transfer run from the past version. In response, the library instructs the live update and ST framework libraries to perform ST and CFT, respectively. At the end of the process, control is given to the reversed version, where the TTST control library repeats the same steps to complete a *backward* state transfer run from the future version. Finally, the library notifies back the past version, where the TTST control library is waiting for TTST events. In response, the library performs *state differencing* between the past and reversed version to validate the TTST transaction and detect state corruption errors violating the semantics-preserving nature of the transformation. In our fault model, the past version is always immutable and adopted as a oracle when comparing the states. If the state is successfully validated (i.e., the past and reversed versions are identical), control moves *back to the future* version to resume execution. The other processes are automatically cleaned up.

When state corruption or run-time errors (e.g., crashes) are detected during the TTST transaction, the update is immediately aborted with the past version cleaning up the other instances and immediately resuming execution. The immutability of the past version's state allows the execution to resume exactly in the same state as it was right before the live update process started. This property ensures instant and transparent recovery in case of arbitrary TTST errors. Our recovery strategy enables fast and automated offline validation and, more importantly, a fault-tolerant live update process that can immediately and automatically rollback failed update attempts with no consequences for the running program.

4 Time-traveling State Transfer

The goal of TTST is to support a truly fault-tolerant live update process, which can automatically detect and recover from as many programming errors as possible, seamlessly support several live update tools and state transfer implementations, and rely on a minimal amount of trusted code at update time. To address these challenges, our TTST technique follows a number of key principles: a well-defined *fault model*, a large *state validation surface*, a *blackbox validation* strategy, and a generic *state transfer interface*.

Fault model. TTST assumes a general fault model with the ability to detect and recover from arbitrary *run-time* errors and *memory* errors introducing state corruption. In particular, run-time errors in the future and reversed versions are automatically detected by the TTST control library in the past version. The process abstraction allows the library to intercept abnormal termination errors in the other instances (e.g., crashes, panics) using simple tracing. Synchronization errors and infinite loops that prevent the TTST transaction from making progress, in turn, are detected with a configurable update timeout (5s by default). Memory errors, finally, are detected by state differencing at the end of the TTST process.

Our focus on memory errors is motivated by three key observations. First, these represent an important class of nonsemantic state transfer errors, the only errors we can hope to detect in a fully automated fashion. Gupta's formal framework has already dismissed the possibility to automatically detect semantic state transfer errors in the general case [41]. Unlike memory errors, semantic errors are consistently introduced across forward and backward state transfer runs and thus cannot automatically be detected by our technique. As an example, consider an update that operates a simple semantic change: renumbering all the global error codes to use different value ranges. If the user does not explicitly provide additional ST code to perform the conversion, the default ST strategy will preserve the same (wrong) error codes across the future and the reversed version, with state differencing unable to detect any errors in the process.

Second, memory errors can lead to insidious latent bugs [32]—which can cause silent data corruption and manifest themselves potentially much later—or even introduce security vulnerabilities. These errors are particularly hard to detect and can easily escape the specification-based validation strategies adopted by all the existing live update testing tools [45–47].

Third, memory errors are painfully common in pathologically type-unsafe contexts like state transfer, where the program state is treated as an opaque object which must be potentially reconstructed from the ground up, all relying on the sole knowledge available to the particular state transfer implementation adopted.

Finally, note that, while other semantic ST errors cannot be detected in the general case, this does not preclude individual ST implementations from using additional knowledge to automatically detect some classes of errors in this category. For example, our state transfer framework can detect all the semantic errors that violate automatically derived *program state invariants* [33] (§5).

State validation surface. TTST seeks to validate the largest possible portion of the state, including state objects (e.g., global variables) that may only be accessed much later after the live update. To meet this goal, our state differencing strategy requires valid forward and backward transfer functions for each state object to validate. Clearly, the existence and the properties of such functions for every particular state object are subject to the nature of the update. For example, an update dropping a global variable in the new version has no defined backward transfer function for that variable. In other cases, forward and backward transfer functions exist but cannot be automatically generated. Consider the error code renumbering update exemplified earlier. Both

State	Diff	Fwd ST	Bwd ST	Detected
Unchanged	✓	STF	STF	Auto
Structural chg	✓	STF	STF	Auto
Semantic chg	✓	User	User [1]	Auto [1]
Dropped	✓	-	-	Auto
Added	✗	Auto/User	-	STF

[1]Optional

Table 2: State validation and error detection surface.

the forward and backward transfer functions for all the global variables affected would have to be manually provided by the user. Since we wish to support fully automated validation by default (mandating extra manual effort is likely to discourage adoption), we allow TTST to gracefully reduce the state validation surface when backward transfer functions are missing—without hampering the effectiveness of our strategy on other fully transferable state objects. Enforcing this behavior in our design is straightforward: the reversed version is originally cloned from the past version and all the state objects that do not take part in the backward state transfer run will trivially match their original counterparts in the state differencing process (unless state corruption occurs).

Table 2 analyzes TTST's state validation and error detection surface for the possible state changes introduced by a given update. The first column refers to the nature of the transformation of a particular state object. The second column refers to the ability to validate the state object using state differencing. The third and fourth column characterize the implementation of the resulting forward and backward transfer functions. Finally, the fifth column analyzes the effectiveness in detecting state corruption. For unchanged state objects, state differencing can automatically detect state corruption and transfer functions are automatically provided by the state transfer framework (STF). Note that unchanged state objects do not necessarily have the same representation in the different versions. The memory layout of an updated version does not generally reflect the memory layout of the old version and the presence of pointers can introduce representation differences for some unchanged state objects between the past and future version. State objects with structural changes exhibit similar behavior, with a fully automated transfer and validation strategy. With structural changes, we refer to state changes that affect only the type representation and can be entirely arbitrated from the STF with no user intervention (§5). This is in contrast with semantic changes, which require user-provided transfer code and can only be partially automated by the STF (§5). Semantic state changes highlight the tradeoff between state validation coverage and the manual effort required by the user. In a traditional live update scenario, the user would normally only provide a forward transfer function. This behavior is seamlessly supported by TTST, but the transferred state object will not be considered for validation. If the user provides code for the reverse transformation, however, the transfer can be normally validated with no restriction. In addition, the backward transfer function provided can be used to perform a cold rollback from the future version to the past version (i.e., live updating the new version into the old version at a later time, for example when the administrator experiences an unacceptable performance slowdown in the updated version). Dropped state objects, in turn, do not require any explicit transfer functions and are automatically validated by state differencing as discussed earlier. State objects that are added in the update (e.g., a new global variable), finally, cannot be automatically validated by state differencing and their validation and transfer is delegated to the STF (§5) or to the user.

Blackbox validation. TTST follows a blackbox validation model, which completely ignores ST internals. This is important for two reasons. First, this provides the ability to support many possible updates and ST implementations. This also allows one to evaluate and compare different STFs. Second, this is crucial to decouple the validation logic from the ST implementation, minimizing the amount of trusted code required by our strategy. In particular, our design goals dictate the minimization of the *reliable computing base* (*RCB*), defined as the core software components that are necessary to ensure correct implementation behavior [26]. Our fault model requires four primary components in the RCB: the update timing mechanisms, the TTST arbitration logic, the run-time error detection mechanisms, and the state differencing logic. All the other software components which run in the future and reversed versions (e.g., ST code and CFT code) are fully untrusted thanks to our design.

The implementation of the update timing mechanisms is entirely delegated to the live update library and its size subject to the particular live update tool considered. We trust that every reasonable update timing implementation will have a small RCB impact. For the other TTST components, we seek to reduce the code size (and complexity) to the minimum. Luckily, our TTST arbitration logic and run-time error detection mechanisms (described earlier) are straightforward and only marginally contribute to the RCB. In addition, TTST's semantics-preserving ST transaction and structural equivalence between the final (reversed) state and the original (past) state ensure that the memory images of the two versions are always identical in error-free ST runs. This drastically simplifies our state differencing strategy, which can be implemented using trivial word-by-word memory comparison, with no other knowledge on the ST code and marginal RCB impact. Our comparison strategy examines all the

```
function STATE_DIFF($pid1, pid2$)
    $a \leftarrow addr\_start$
    while $a < shadow\_start$ do
        $m1 \leftarrow$ IS_MAPPED_WRITABLE($a, pid1$)
        $m2 \leftarrow$ IS_MAPPED_WRITABLE($a, pid2$)
        if $m1$ or $m2$ then
            if $m1 \neq m2$ then
                return $true$
            if MEMPAGECMP($a, pid1, pid2$) $\neq 0$ then
                return $true$
        $a \leftarrow a + page\_size$
    return $false$
```

Figure 2: State differencing pseudocode.

writable regions of the address space excluding only private shadow stack/heap regions (mapped at the end of the address space) in use by the TTST control library. Figure 2 shows the pseudocode for this simple strategy.

State transfer interface. TTST's state transfer interface seeks to minimize the requirements and the effort to implement the STF. In terms of requirements, TTST demands only a *layout-aware* and *user-aware* STF semantic. By layout-aware, we refer to the ability of the STF to preserve the original state layout when requested (i.e., in the reversed version), as well as to automatically identify the state changes described in Table 2. By user-aware, we refer to the ability to allow the user to selectively specify new forward and backward transfer functions and candidate state objects for validation. To reduce the effort, TTST offers a convenient STF programming model, with an error handling-friendly environment—our fault-tolerant design encourages undiscriminated use of assertions—and a generic interprocess communication (IPC) interface. In particular, TTST implements an IPC *control* interface to coordinate the TTST transaction and an IPC *data* interface to grant read-only access to the state of a given process version to the others. These interfaces are currently implemented by UNIX domain sockets and POSIX shared memory (respectively), but other IPC mechanisms can be easily supported. The current implementation combines fast data transfer with a secure design that prevents impersonation attacks (access is granted only to the predetermined process instances).

5 State Transfer Framework

Our state transfer framework seeks to automate all the possible ST steps, leaving only the undecidable cases (e.g., semantic state changes) to the user. The implementation described here optimizes and extends our prior work [33–36] to the TTST model. We propose a STF design that resembles a *moving*, *mutating*, and *interprocess* garbage collection model. By moving, we refer to the ability to relocate (and possibly reallocate) static and dynamic state objects in the next version. This is to allow arbitrary changes in the memory layout between versions. By mutating, we refer to the ability to perform on-the-fly type transformations when transferring every given state object from the previous to the next version. Interprocess, finally, refers to our process-level ST strategy. Our goals raise 3 major challenges for a low-level language like C. First, our moving requirement requires precise object and pointer analysis at runtime. Second, on-the-fly type transformations require the ability to dynamically identify, inspect, and match generic data types. Finally, our interprocess strategy requires a mechanism to identify and map state objects across process versions.

Overview. To meet our goals, our STF uses a combination of static and dynamic ST instrumentation. Our static instrumentation, implemented by a LLVM link-time pass [56], transforms each program version to generate *metadata* information that surgically describes the entirety of the program state. In particular, static metadata, which provides *relocation* and *type* information for all the static state objects (e.g., global variables, strings, functions with address taken), is embedded directly into the final binary. Dynamic metadata, which provides the same information for all the dynamic state objects (e.g., heap-allocated objects), is, in turn, dynamically generated/destroyed at runtime by our allocation/deallocation site instrumentation—we currently support `malloc/mmap-like` allocators automatically and standard region-based allocators [15] using user-annotated allocator functions. Further, our pass can dynamically generate/destroy local variable metadata for a predetermined number of functions (e.g., `main`), as dictated by the particular update model considered. Finally, to automatically identify and map objects across process versions, our instrumentation relies on *version-agnostic* state IDs derived from unambiguous *naming* and *contextual* information. In detail, every static object is assigned a static ID derived by its source name (e.g., function name) and scope (e.g., static variable module). Every dynamic object, in turn, is assigned a static ID derived by allocation site information (e.g., caller function name and target pointer name) and an incremental dynamic ID to unambiguously identify allocations at runtime.

Our ID-based naming scheme fulfills TTST's layout-awareness goal: static IDs are used to identify state changes and to automatically reallocate dynamic objects in the future version; dynamic IDs are used to map dynamic objects in the future version with their existing counterparts in the reversed version. The mapping policy to use is specified as part of generic *ST policies*, also implementing other TTST-aware extensions: (i) *randomization* (enabled in the future version): perform

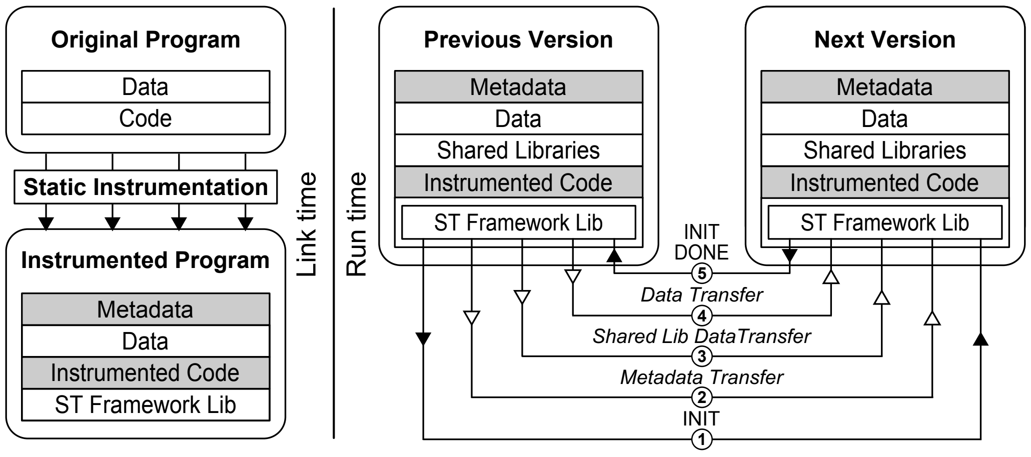

Figure 3: State transfer framework overview.

fine-grained address space randomization [34] for all the static/dynamically reallocated objects, used to amplify the difference introduced by memory errors in the overall TTST transaction; (ii) *validation* (enabled in the reversed version): zero out the local copy of all the mapped state objects scheduled for automated transfer to detect missing write errors at validation time.

Our dynamic instrumentation, included in a preloaded shared library (ST framework library), complements the static pass to address the necessary run-time tasks: type and pointer analysis, metadata management for shared libraries, error detection. In addition, the ST framework library implements all the steps of the ST process, as depicted in Figure 3. The process begins with an initialization request from the TTST control library, which specifies the ST policies and provides access to the TTST's IPC interface. The next *metadata transfer* step transfers all the metadata information from the previous version to a metadata cache in the next version (local address space). At the end, the local state objects (and their metadata) are mapped into the external objects described by the metadata cache and scheduled for transfer according to their state IDs and the given ST policies. The next two *data transfer* steps complete the ST process, transferring all the data to reinitialize shared library and program state to the next version. State objects scheduled for transfer are processed one at a time, using metadata information to locate the objects and their internal representations in the two process versions and apply pointer and type transformations on the fly. The last step performs cleanup tasks and returns control to the caller.

State transfer strategy. Our STF follows a well-defined automated ST strategy for all the mapped state objects scheduled for transfer, exemplified in Figure 4. As shown in the figure—which reprises the update example given earlier (§ 2)—our type analysis automatically and recursively matches individual type elements between object versions by *name* and *representation*, identifying added/dropped/changed/identical elements on the fly. This strategy automates ST for common structural changes, including: primitive type changes, array expansion/truncation, and addition/deletion/reordering of `struct` members. Our pointer analysis, in turn, implements a generic pointer transfer strategy, automatically identifying (base and interior) pointer targets in the previous version and reinitializing the pointer values correctly in the next version, in spite of type and memory layout changes. To perform efficient pointer lookups, our analysis organizes all the state objects with address taken in a splay tree, an idea previously explored by bounds checkers [9, 27, 70]. We also support all the special pointer idioms allowed by C (e.g., guard pointers) automatically, with the exception of cases of "*pointer ambiguity*" [36].

To deal with ambiguous pointer scenarios (e.g., `unions` with inner pointers and pointers stored as integers) as well as more complex state changes (e.g., semantic changes), our STF supports user extensions in the form of preprocessor annotations and callbacks. Figure 4 shows an example of two ST annotations: `IXFER` (force memory copying with no pointer transfer) and `PXFER` (force pointer transfer instead of memory copying). Callbacks, in turn, are evaluated whenever the STF maps or traverses a given object or type element, allowing the user to override the default mapping behavior (e.g., for renamed variables) or express sophisticated state transformations at the object or element level. Callbacks can be also used to: (i) override the default validation policies, (ii) initialize new state objects; (iii) instruct the STF to checksum new state objects after initialization to detect memory errors at the end of the ST process.

Shared libraries. Uninstrumented shared libraries (*SLs*) pose a major challenge to our pointer transfer strategy. In particular, failure to reinitialize SL-related pointers correctly in the future version would introduce er-

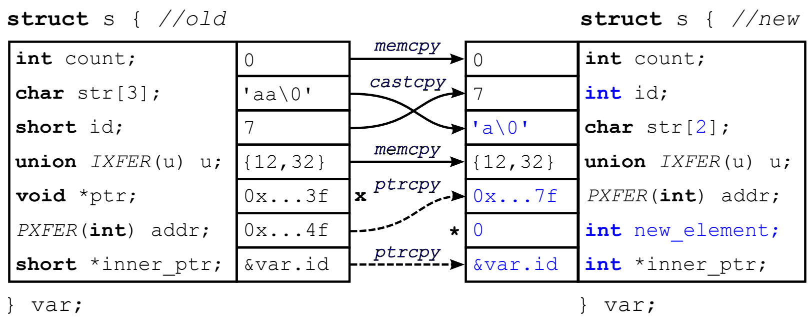

Figure 4: Automated state transfer example for the data structure presented in Listing 2.

rors after live update. To address this challenge, our STF distinguishes 3 scenarios: (i) program/SL pointers into static SL state; (ii) program/SL pointers into dynamic SL state; (iii) SL pointers into static or dynamic program state. To deal with the first scenario, our STF instructs the dynamic linker to remap all the SLs in the future version at the same addresses as in the past version, allowing SL data transfer (pointer transfer in particular) to be implemented via simple memory copying. SL relocation is currently accomplished by prelinking the SLs on demand when starting the future version, a strategy similar to "*retouching*" for mobile applications [19]. To address the second scenario, our dynamic instrumentation intercepts all the memory management calls performed by SLs and generates dedicated metadata to reallocate the resulting objects at the same address in the future version. This is done by restoring the original heap layout (and content) as part of the SL data transfer phase. To perform heap randomization and type transformations correctly for all the program allocations in the future version, in turn, we allow the STF to deallocate (and reallocate later) all the non-SL heap allocations right after SL data transfer. To deal with the last scenario, we need to accurately identify all the SL pointers into the program state and update their values correctly to reflect the memory layout of the future version. Luckily, these cases are rare and we can envision library developers exporting a public API that clearly marks long-lived pointers into the program state once our live update technique is deployed. A similar API is desirable to mark all the process-specific state (e.g., *libc*'s cached pids) that should be restored after ST—note that shareable resources like file descriptors are, in contrast, automatically transferred by the `fork/exec` paradigm. To automate the identification of these cases in our current prototype, we used conservative pointer analysis techniques [17, 18] under stress testing to locate long-lived SL pointers into the program state and state differencing at `fork` points to locate process-specific state objects.

Error detection. To detect certain classes of semantic errors that escape TTST's detection strategy, our STF enforces *program state invariants* [33] derived from all the metadata available at runtime. Unlike existing *likely* invariant-based error detection techniques [6,28,31,42,68], our invariants are conservatively computed from static analysis and allow for no false positives. The majority of our invariants are enforced by our dynamic pointer analysis to detect semantic errors during pointer transfer. For example, our STF reports invariant violation (and aborts ST by default) whenever a pointer target no longer exists or has its address taken (according to our static analysis) in the new version. Another example is a transferred pointer that points to an illegal target type according to our static pointer cast analysis.

6 Evaluation

We have implemented TTST on Linux (x86), with support for generic user-space C programs using the ELF binary format. All the platform-specific components, however, are well isolated in the TTST control library and easily portable to other operating systems, architectures, and binary formats other than ELF. We have integrated address space randomization techniques developed in prior work [34] into our ST instrumentation and configured them to randomize the location of all the static and dynamically reallocated objects in the future version. To evaluate TTST, we have also developed a live update library mimicking the behavior of state-of-the-art live update tools [48], which required implementing preannotated per-thread update points to control update timing, manual control migration to perform CFT, and a UNIX domain sockets-based interface to receive live update commands from our `ttst-ctl` tool.

We evaluated the resulting solution on a workstation running Linux v3.5.0 (x86) and equipped with a 4-core 3.0Ghz AMD Phenom II X4 B95 processor and 8GB of RAM. For our evaluation, we first selected Apache httpd (v.2.2.23) and nginx (v0.8.54), the two most popular open-source web servers. For comparison purposes, we also considered vsftpd (v1.1.0) and the OpenSSH dae-

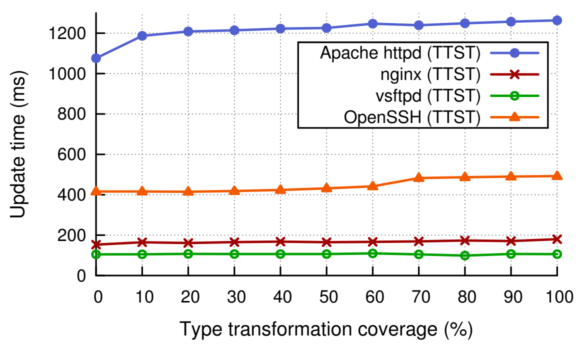

Figure 5: Update time vs. type transformation coverage.

Type	**httpd**	**nginx**	**vsftpd**	**OpenSSH**
Static	2.187	2.358	3.352	2.480
Run-time	3.100	3.786	4.362	2.662
Forward ST	3.134	5.563	6.196	4.126
TTST	3.167	7.340	8.031	5.590

Table 3: TTST*-induced memory usage (measured statically or at runtime) normalized against the baseline.*

mon (v3.5p1), a popular open-source ftp and ssh server, respectively. The former [23,45,48,49,57,63,64] and the latter [23,45,64] are by far the most used server programs (and versions) in prior work in the field. We annotated all the programs considered to match the implemented live update library as described in prior work [45, 48]. For Apache httpd and nginx, we redirected all the calls to custom allocation routines to the standard allocator interface (i.e., `malloc/free` calls), given that our current instrumentation does not yet support custom allocation schemes based on nested regions [15] (Apache httpd) and slab-like allocations [20] (nginx). To evaluate our programs, we performed tests using the Apache benchmark (AB) [1] (Apache httpd and nginx), dkftpbench [2] (vsftpd), and the provided regression test suite (OpenSSH). We configured our programs and benchmarks using the default settings. We repeated all our experiments 21 times and reported the median—with negligible standard deviation measured across multiple test runs.

Our evaluation answers five key questions: (i) *Performance*: Does TTST yield low run-time overhead and reasonable update times? (ii) *Memory usage*: How much memory do our instrumentation techniques use? (iii) *RCB size*: How much code is (and is not) in the RCB? (iv) *Fault tolerance*: Can TTST withstand arbitrary failures in our fault model? (v) *Engineering effort*: How much engineering effort is required to adopt TTST?

Performance. To evaluate the run-time overhead imposed by our update mechanisms, we first ran our benchmarks to compare our base programs with their instrumented and annotated versions. Our experiments showed no appreciable performance degradation. This is expected, since update points only require checking a flag at the top of long-running loops and metadata is efficiently managed by our ST instrumentation. In detail, our static metadata—used only at update time—is confined in a separate ELF section so as not to disrupt locality. Dynamic metadata management, in turn, relies on in-band descriptors to minimize the overhead on allocator operations. To evaluate the latter, we instrumented all the C programs in the SPEC CPU2006 benchmark suite. The results evidenced a 4% average run-time overhead across all the benchmarks. We also measured the cost of our instrumentation on 10,000 `malloc/free` and `mmap/munmap` repeated `glibc` allocator operations—which provide worst-case results, given that common allocation patterns generally yield poorer locality. Experiments with multiple allocation sizes (0-16MB) reported a maximum overhead of 41% for `malloc`, 9% for `free`, 77% for `mmap`, and 42% for `munmap`. While these microbenchmark results are useful to evaluate the impact of our instrumentation on allocator operations, we expect any overhead to be hardly visible in real-world server programs, which already strive to avoid expensive allocations on the critical path [15].

When compared to prior user-level solutions, our performance overhead is much lower than more intrusive instrumentation strategies—with worst-case macrobenchmark overhead of 6% [64], 6.71% [62], and 96.4% [57]—and generally higher than simple binary rewriting strategies [10, 23]—with worst-case function invocation overhead estimated around 8% [58]. Unlike prior solutions, however, our overhead is strictly isolated in allocator operations and never increases with the number of live updates deployed over time. Recent program-level solutions that use minimal instrumentation [48]—no allocator instrumentation, in particular—in turn, report even lower overheads than ours, but at the daunting cost of annotating all the pointers into heap objects.

We also analyzed the impact of process-level TTST on the update time—the time from the moment the update is signaled to the moment the future version resumes execution. Figure 5 depicts the update time—when updating the master process of each program—as a function of the number of type transformations operated by our ST framework. For this experiment, we implemented a source-to-source transformation able to automatically change 0-1,327 type definitions (adding/reordering `struct` fields and expanding arrays/primitive types) for Apache httpd, 0-818 type definitions for nginx, 0-142 type definitions for vsftpd, and 0-455 type definitions for OpenSSH between versions. This forced our ST framework to operate an average of 1,143,981, 111,707,

Component	**RCB**	**Other**
ST instrumentation	1,119	8,211
Live update library	235	147
TTST control library	412	2,797
ST framework	0	13,311
`ttst-ctl` tool	0	381
Total	1,766	26,613

Table 4: Source lines of code (LOC) and contribution to the RCB size for every component in our architecture.

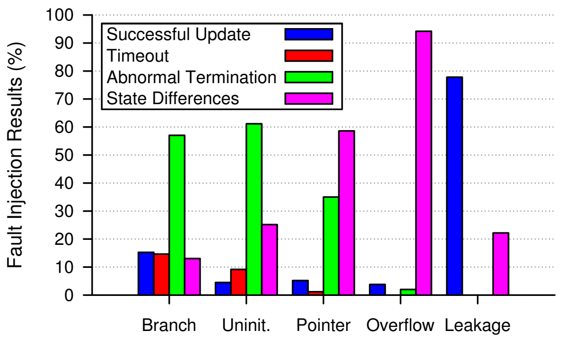

Figure 6: TTST *behavior in our automated fault injection experiments for varying fault types.*

1,372, and 206,259 type transformations (respectively) at 100% coverage. As the figure shows, the number of type transformations has a steady but low impact on the update time, confirming that the latter is heavily dominated by memory copying and pointer analysis—albeit optimized with splay trees. The data points at 100% coverage, however, are a useful indication of the upper bound for the update time, resulting in 1263 ms, 180 ms, 112 ms, and 465 ms (respectively) with our TTST update strategy. Apache httpd reported the longest update times in all the configurations, given the greater amount of state transferred at update time. Further, TTST update times are, on average, 1.76x higher than regular ST updates (not shown in figure for clarity), acknowledging the impact of backward ST and state differencing on the update time. While our update times are generally higher than prior solutions, the impact is bearable for most programs and the benefit is stateful *fault-tolerant* version updates.

Memory usage. Our state transfer instrumentation leads to larger binary sizes and run-time memory footprints. This stems from our metadata generation strategy and the libraries required to support live update. Table 3 evaluates the impact on our test programs. The static memory overhead (235.2% worst-case overhead for vsftpd) measures the impact of our ST instrumentation on the binary size. The run-time overhead (336.2% worst-case overhead for vsftpd), in turn, measures the impact of instrumentation and support libraries on the virtual memory size observed at runtime, right after server initialization. These measurements have been obtained starting from a baseline virtual memory size of 234 MB for Apache httpd and less than 6 MB for all the other programs. The third and the fourth rows, finally, show the maximum virtual memory overhead we observed at live update time for both regular (forward ST only) and TTST updates, also accounting for all the transient process instances created (703.1% worst-case overhead for vsftpd and TTST updates). While clearly program-dependent and generally higher than prior live update solutions, our measured memory overheads are modest and, we believe, realistic for most systems, also given the increasingly low cost of RAM in these days.

RCB size. Our TTST update technique is carefully designed to minimize the RCB size. Table 4 lists the LOC required to implement every component in our architecture and the contributions to the RCB. Our ST instrumentation requires 1,119 RCB LOC to perform dynamic metadata management at runtime. Our live update library requires 235 RCB LOC to implement the update timing mechanisms and interactions with client tools. Our TTST control library requires 412 RCB LOC to arbitrate the TTST process, implement run-time error detection, and perform state differencing—all from the past version. Our ST framework and `ttst-ctl` tool, in contrast, make no contribution to the RCB. Overall, our design is effective in producing a small RCB, with only 1,766 LOC compared to the other 26,613 non-RCB LOC. Encouragingly, our RCB is even substantially smaller than that of other systems that have already been shown to be amenable to formal verification [54]. This is in stark contrast with all the prior solutions, which make no effort to remove *any* code from the RCB.

Fault tolerance. We evaluated the fault tolerance of TTST using software-implemented fault injection (SWIFI) experiments. To this end, we implemented another LLVM pass which transforms the original program to inject specific classes of software faults into predetermined code regions. Our pass accepts a list of target program functions/modules, the fault types to inject, and a fault probability ϕ—which specifies how many fault locations should be randomly selected for injection out of all the possible candidates found in the code. We configured our pass to randomly inject faults in the ST code, selecting $\phi = 1\%$—although we observed similar results for other ϕ values—and fault types that matched common programming errors in our fault model. In detail, similar to prior SWIFI strategies that evaluated the effectiveness of fault-tolerance mechanisms against state corruption [65], we considered generic *branch* errors (branch/loop condition flip or stuck-at errors) as well as

	Updates		Changes			Engineering effort		
	#	**LOC**	**Fun**	**Var**	**Ty**	**ST Ann LOC**	**Fwd ST LOC**	**Bwd ST LOC**
Apache httpd	5	10,844	829	28	48	79	302	151
nginx	25	9,681	711	51	54	24	335	0
vsftpd	5	5,830	305	121	35	0	21	21
OpenSSH	5	14,370	894	84	33	0	135	127
Total	40	40,725	2,739	284	170	103	793	299

Table 5: Engineering effort for all the updates analyzed in our evaluation.

common memory errors, such as *uninitialized reads* (emulated by missing initializers), *pointer corruption* (emulated by corrupting pointers with random or off-by-1 values), *buffer overflows* (emulated by extending the size passed to data copy functions, e.g., `memcpy`, by 1-100%), and *memory leakage* (emulated by missing deallocation calls). We repeated our experiments 500 times for each of the 5 fault types considered, with each run starting a live update between randomized program versions and reporting the outcome of our TTST strategy. We report results only for vsftpd—although we observed similar results for the other programs—which allowed us to collect the highest number of fault injection samples per time unit and thus obtain the most statistically sound results.

Figure 6 presents our results breaking down the data by fault type and distribution of the observed outcomes—that is, update succeeded or automatically rolled back after *timeout*, *abnormal termination* (e.g., crash), or past-reversed *state differences* detected. As expected, the distribution varies across the different fault types considered. For instance, branch and initialization errors produced the highest number of updates aborted after a timeout (14.6% and 9.2%), given the higher probability of infinite loops. The first three classes of errors considered, in turn, resulted in a high number of crashes (51.1%, on average), mostly due to invalid pointer dereferences and invariants violations detected by our ST framework. In many cases, however, the state corruption introduced did not prevent the ST process from running to completion, but was nonetheless detected by our state differencing technique. We were particularly impressed by the effectiveness of our validation strategy in a number of scenarios. For instance, state differencing was able to automatically recover from as many as 471 otherwise-unrecoverable buffer overflow errors. Similar is the case of memory leakages—actually activated in 22.2% of the runs—with any extra memory region mapped by our metadata cache and never deallocated immediately detected at state diffing time. We also verified that the future (or past) version resumed execution correctly after every successful (or aborted) update attempt. When sampling the 533 successful cases, we noted the introduction of irrelevant faults (e.g., missing initializer for an unused variable) or no faults actually activated at runtime. Overall, our TTST technique was remarkably effective in detecting and recovering from a significant number of observed failures (1,967 overall), with no consequences for the running program. This is in stark contrast with all the prior solutions, which make *no* effort in this regard.

Engineering effort. To evaluate the engineering effort required to deploy TTST, we analyzed a number of official incremental releases following our original program versions and prepared the resulting patches for live update. In particular, we considered 5 updates for Apache httpd (v2.2.23-v2.3.8), vsftpd (v1.1.0-v2.0.2), and OpenSSH (v3.5-v3.8), and 25 updates for nginx (v0.8.54-v1.0.15), given that nginx's tight release cycle generally produces incremental patches that are much smaller than those of the other programs considered. Table 5 presents our findings. The first two grouped columns provide an overview of our analysis, with the number of updates considered for each program and the number of lines of code (LOC) added, deleted, or modified in total by the updates. As shown in the table, we manually processed more than 40,000 LOC across the 40 updates considered. The second group shows the number of functions, variables, and types changed (i.e., added, deleted, or modified) by the updates, with a total of 2,739, 284, and 170 changes (respectively). The third group, finally, shows the engineering effort in terms of LOC required to prepare our test programs and our patches for live update. The first column shows the one-time annotation effort required to integrate our test programs with our ST framework. Apache httpd and nginx required 79 and 2 LOC to annotate 12 and 2 `unions` with inner pointers, respectively. In addition, nginx required 22 LOC to annotate a number of global pointers using special data encoding—storing metadata information in the 2 least significant bits. The latter is necessary to ensure precise pointer analysis at ST time. The second and the third column, in turn, show the number of lines of state transfer code we had to manually write to complete forward ST and backward ST (respectively) across all the updates considered. Such ST extensions were necessary to implement complex state changes that could not be automatically handled by our ST framework.

A total of 793 forward ST LOC were strictly necessary to prepare our patches for live update. An extra 299 LOC, in turn, were required to implement backward ST. While optional, the latter is important to guarantee full validation surface for our TTST technique. The much lower LOC required for backward ST (37.7%) is easily explained by the additive nature of typical state changes, which frequently entail only adding new data structures (or fields) and thus rarely require extra LOC in our backward ST transformation. The case of nginx is particularly emblematic. Its disciplined update strategy, which limits the number of nonadditive state changes to the minimum, translated to no manual ST LOC required to implement backward ST. We believe this is particularly encouraging and can motivate developers to deploy our TTST techniques with full validation surface in practice.

7 Related Work

Live update systems. We focus on *local* live update solutions for generic and widely deployed C programs, referring the reader to [7, 8, 29, 55, 74] for distributed live update systems. LUCOS [22], DynaMOS [58], and Ksplice [11] have applied live updates to the Linux kernel, loading new code and data directly into the running version. Code changes are handled using binary rewriting (i.e., trampolines). Data changes are handled using shadow [11, 58] or parallel [22] data structures. OPUS [10], POLUS [23], Ginseng [64], STUMP [62], and Upstare [57] are similar live update solutions for user-space C programs. Code changes are handled using binary rewriting [10, 23], compiler-based instrumentation [62,64], or stack reconstruction [57]. Data changes are handled using parallel data structures [23], type wrapping [62, 64], or object replacement [57]. Most solutions delegate ST entirely to the programmer [10, 11, 22, 23, 58], others generate only basic type transformers [57, 62, 64]. Unlike TTST, none of these solutions attempt to fully automate ST—pointer transfer, in particular—and state validation. Further, their in-place update model hampers isolation and recovery from ST errors, while also introducing address space fragmentation over time. To address these issues, alternative update models have been proposed. Prior work on process-level live updates [40, 49], however, delegates the ST burden entirely to the programmer. In another direction, Kitsune [48] encapsulates every program in a hot swappable shared library. Their state transfer framework, however, does not attempt to automate pointer transfer without user effort and no support is given to validate the state or perform safe rollback in case of ST errors. Finally, our prior work [34, 35] demonstrated the benefits of process-level live updates in component-based OS architectures, with support to recover from run-time ST errors but no ability to detect a corrupted state in the updated version.

Live update safety. Prior work on live update safety is mainly concerned with safe update timing mechanisms, neglecting important system properties like fault tolerance and RCB minimization. Some solutions rely on *quiescence* [10–13] (i.e., no updates to active code), others enforce *representation consistency* [62, 64, 71] (i.e., no updated code accessing old data). Other researchers have proposed using transactions in local [63] or distributed [55, 74] contexts to enforce stronger timing constraints. Recent work [44], in contrast, suggests that many researchers may have been overly concerned with update timing and that a few predetermined update points [34, 35, 48, 49, 62, 64] are typically sufficient to determine safe and timely update states. Unlike TTST, none of the existing solutions have explicitly addressed ST-specific update safety properties. Static analysis proposed in OPUS [10]—to detect unsafe data updates—and Ginseng [64]—to detect unsafe pointers into updated objects—is somewhat related, but it is only useful to *disallow* particular classes of (unsupported) live updates.

Update testing. Prior work on live update testing [45–47] is mainly concerned with validating the correctness of an update in all the possible update timings. Correct execution is established from manually written specifications [47] or manually selected program output [45, 46]. Unlike TTST, these techniques require nontrivial manual effort, are only suitable for offline testing, and fail to validate the entirety of the program state. In detail, their state validation surface is subject to the coverage of the test programs or specifications used. Their testing strategy, however, is useful to compare different update timing mechanisms, as also demonstrated in [45]. Other related work includes online patch validation, which seeks to efficiently compare the behavior of two (original and patched) versions at runtime. This is accomplished by running two separate (synchronized) versions in parallel [21, 51, 59] or a single hybrid version using a split-and-merge strategy [73]. These efforts are complementary to our work, given that their goal is to test for errors in the patch itself rather than validating the state transfer code required to prepare the patch for live update. Complementary to our work are also efforts on upgrade testing in large-scale installations, which aim at creating sandboxed deployment-like environments for testing purposes [75] or efficiently testing upgrades in diverse environments using staged deployment [25]. Finally, fault injection has been previously used in the context of update testing [29, 60, 66], but only to emulate upgrade-time operator errors. Our evaluation, in contrast, presents the first fault injection campaign that emulates realistic programming errors in the ST code.

8 Conclusion

While long recognized as a hard problem, state transfer has received limited attention in the live update literature. Most efforts focus on automating and validating update timing, rather than simplifying and shielding the state transfer process from programming errors. We believe this is a key factor that has discouraged the system administration community from adopting live update tools, which are often deemed impractical and untrustworthy.

This paper presented *time-traveling state transfer*, the first fault-tolerant live update technique which allows generic live update tools for C programs to automate and validate the state transfer process. Our technique combines the conventional forward state transfer transformation with a backward (and logically redundant) transformation, resulting in a semantics-preserving manipulation of the original program state. Observed deviations in the reversed state are used to automatically identify state corruption caused by common classes of programming errors (i.e., memory errors) in the state transfer (library or user) code. Our process-level update strategy, in turn, guarantees detection of other run-time errors (e.g., crashes), simplifies state management, and prevents state transfer errors to propagate back to the original version. The latter property allows our framework to safely recover from errors and automatically resume execution in the original version. Further, our modular and blackbox validation design yields a minimal-RCB live update system, offering a high fault-tolerance surface in both online and offline validation runs. Finally, we complemented our techniques with a generic state transfer framework, which automates state transformations with minimal programming effort and can detect additional semantic errors using statically computed invariants. We see our work as the first important step toward truly practical and trustworthy live update tools for system administrators.

9 Acknowledgments

We would like to thank our shepherd, Mike Ciavarella, and the anonymous reviewers for their comments. This work has been supported by European Research Council under grant ERC Advanced Grant 2008 - R3S3.

References

[1] Apache benchmark (AB). `http://httpd.apache.org/docs/2.0/programs/ab.html`.

[2] dkftpbench. `http://www.kegel.com/dkftpbench`.

[3] Ksplice performance on security patches. `http://www.ksplice.com/cve-evaluation`.

[4] Ksplice Uptrack. `http://www.ksplice.com`.

[5] Vulnerability summary for CVE-2006-0095. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2006-0095`.

[6] ADVE, S. V., ADVE, V. S., AND ZHOU, Y. Using likely program invariants to detect hardware errors. In *Proc. of the IEEE Int'l Conf. on Dependable Systems and Networks* (2008).

[7] AJMANI, S., LISKOV, B., AND SHRIRA, L. Scheduling and simulation: How to upgrade distributed systems. In *Proc. of the Ninth Workshop on Hot Topics in Operating Systems* (2003), vol. 9, pp. 43–48.

[8] AJMANI, S., LISKOV, B., SHRIRA, L., AND THOMAS, D. Modular software upgrades for distributed systems. In *Proc. of the 20th European Conf. on Object-Oriented Programming* (2006), pp. 452–476.

[9] AKRITIDIS, P., COSTA, M., CASTRO, M., AND HAND, S. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proc. of the 18th USENIX Security Symp.* (2009), pp. 51–66.

[10] ALTEKAR, G., BAGRAK, I., BURSTEIN, P., AND SCHULTZ, A. OPUS: Online patches and updates for security. In *Proc. of the 14th USENIX Security Symp.* (2005), vol. 14, pp. 19–19.

[11] ARNOLD, J., AND KAASHOEK, M. F. Ksplice: Automatic rebootless kernel updates. In *Proc. of the Fourth ACM European Conf. on Computer Systems* (2009), pp. 187–198.

[12] BAUMANN, A., APPAVOO, J., WISNIEWSKI, R. W., SILVA, D. D., KRIEGER, O., AND HEISER, G. Reboots are for hardware: Challenges and solutions to updating an operating system on the fly. In *Proc. of the USENIX Annual Tech. Conf.* (2007), pp. 1–14.

[13] BAUMANN, A., HEISER, G., APPAVOO, J., DA SILVA, D., KRIEGER, O., WISNIEWSKI, R. W., AND KERR, J. Providing dynamic update in an operating system. In *Proc. of the USENIX Annual Tech. Conf.* (2005), p. 32.

[14] BAZZI, R. A., MAKRIS, K., NAYERI, P., AND SHEN, J. Dynamic software updates: The state mapping problem. In *Proc. of the Second Int'l Workshop on Hot Topics in Software Upgrades* (2009), p. 2.

[15] BERGER, E. D., ZORN, B. G., AND MCKINLEY, K. S. Reconsidering custom memory allocation. In *Proc. of the 17th ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications* (2002), pp. 1–12.

[16] BLOOM, T., AND DAY, M. Reconfiguration and module replacement in Argus: Theory and practice. *Software Engineering J. 8*, 2 (1993), 102–108.

[17] BOEHM, H.-J. Space efficient conservative garbage collection. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation* (1993), pp. 197–206.

[18] BOEHM, H.-J. Bounding space usage of conservative garbage collectors. In *Proc. of the 29th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages* (2002), pp. 93–100.

[19] BOJINOV, H., BONEH, D., CANNINGS, R., AND MALCHEV, I. Address space randomization for mobile devices. In *Proc. of the Fourth ACM Conf. on Wireless network security* (2011), pp. 127–138.

[20] BONWICK, J. The slab allocator: An object-caching kernel memory allocator. In *Proc. of the USENIX Summer Technical Conf.* (1994), p. 6.

[21] CADAR, C., AND HOSEK, P. Multi-version software updates. In *Proc. of the Fourth Int'l Workshop on Hot Topics in Software Upgrades* (2012), pp. 36–40.

[22] CHEN, H., CHEN, R., ZHANG, F., ZANG, B., AND YEW, P.-C. Live updating operating systems using virtualization. In *Proc. of the Second Int'l Conf. on Virtual Execution Environments* (2006), pp. 35–44.

[23] CHEN, H., YU, J., CHEN, R., ZANG, B., AND YEW, P.-C. POLUS: A POwerful live updating system. In *Proc. of the 29th Int'l Conf. on Software Eng.* (2007), pp. 271–281.

[24] CHOW, J., PFAFF, B., GARFINKEL, T., AND ROSENBLUM, M. Shredding your garbage: Reducing data lifetime through secure deallocation. In *Proc. of the 14th USENIX Security Symp.* (2005), pp. 22–22.

[25] CRAMERI, O., KNEZEVIC, N., KOSTIC, D., BIANCHINI, R., AND ZWAENEPOEL, W. Staged deployment in Mirage, an integrated software upgrade testing and distribution system. In *Proc. of the 21st ACM Symp. on Operating Systems Principles* (2007), pp. 221–236.

[26] DÖBEL, B., HÄRTIG, H., AND ENGEL, M. Operating system support for redundant multithreading. In *Proc. of the 10th Int'l Conf. on Embedded software* (2012), pp. 83–92.

[27] DHURJATI, D., AND ADVE, V. Backwards-compatible array bounds checking for C with very low overhead. In *Proc. of the 28th Int'l Conf. on Software Eng.* (2006), pp. 162–171.

[28] DIMITROV, M., AND ZHOU, H. Unified architectural support for soft-error protection or software bug detection. In *Proc. of the 16th Int'l Conf. on Parallel Architecture and Compilation Techniques* (2007), pp. 73–82.

[29] DUMITRAS, T., AND NARASIMHAN, P. Why do upgrades fail and what can we do about it?: Toward dependable, online upgrades in enterprise system. In *Proc. of the 10th Int'l Conf. on Middleware* (2009), pp. 1–20.

[30] DUMITRAS, T., NARASIMHAN, P., AND TILEVICH, E. To upgrade or not to upgrade: Impact of online upgrades across multiple administrative domains. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Appilcations* (2010), pp. 865–876.

[31] ERNST, M. D., COCKRELL, J., GRISWOLD, W. G., AND NOTKIN, D. Dynamically discovering likely program invariants to support program evolution. In *Proc. of the 21st Int'l Conf. on Software Eng.* (1999), pp. 213–224.

[32] FONSECA, P., LI, C., AND RODRIGUES, R. Finding complex concurrency bugs in large multi-threaded applications. In *Proc. of the Sixth ACM European Conf. on Computer Systems* (2011), pp. 215–228.

[33] GIUFFRIDA, C., CAVALLARO, L., AND TANENBAUM, A. S. Practical automated vulnerability monitoring using program state invariants. In *Proc. of the Int'l Conf. on Dependable Systems and Networks* (2013).

[34] GIUFFRIDA, C., KUIJSTEN, A., AND TANENBAUM, A. S. Enhanced operating system security through efficient and fine-grained address space randomization. In *Proc. of the 21st USENIX Security Symp.* (2012), p. 40.

[35] GIUFFRIDA, C., KUIJSTEN, A., AND TANENBAUM, A. S. Safe and automatic live update for operating systems. In *Proceedings of the 18th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems* (2013), pp. 279–292.

[36] GIUFFRIDA, C., AND TANENBAUM, A. Safe and automated state transfer for secure and reliable live update. In *Proc. of the Fourth Int'l Workshop on Hot Topics in Software Upgrades* (2012), pp. 16–20.

[37] GIUFFRIDA, C., AND TANENBAUM, A. S. Cooperative update: A new model for dependable live update. In *Proc. of the Second Int'l Workshop on Hot Topics in Software Upgrades* (2009), pp. 1–6.

[38] GIUFFRIDA, C., AND TANENBAUM, A. S. A taxonomy of live updates. In *Proc. of the 16th ASCI Conf.* (2010).

[39] GOODFELLOW, B. Patch tuesday. `http://www.thetechgap.com/2005/01/strongpatch_tue.html`.

[40] GUPTA, D., AND JALOTE, P. On-line software version change using state transfer between processes. *Softw. Pract. and Exper. 23*, 9 (1993), 949–964.

[41] GUPTA, D., JALOTE, P., AND BARUA, G. A formal framework for on-line software version change. *IEEE Trans. Softw. Eng. 22*, 2 (1996), 120–131.

[42] HANGAL, S., AND LAM, M. S. Tracking down software bugs using automatic anomaly detection. In *Proc. of the 24th Int'l Conf. on Software Eng.* (2002), pp. 291–301.

[43] HANSELL, S. Glitch makes teller machines take twice what they give. *The New York Times* (1994).

[44] HAYDEN, C., SAUR, K., HICKS, M., AND FOSTER, J. A study of dynamic software update quiescence for multithreaded programs. In *Proc. of the Fourth Int'l Workshop on Hot Topics in Software Upgrades* (2012), pp. 6–10.

[45] HAYDEN, C., SMITH, E., HARDISTY, E., HICKS, M., AND FOSTER, J. Evaluating dynamic software update safety using systematic testing. *IEEE Trans. Softw. Eng. 38*, 6 (2012), 1340–1354.

[46] HAYDEN, C. M., HARDISTY, E. A., HICKS, M., AND FOSTER, J. S. Efficient systematic testing for dynamically updatable software. In *Proc. of the Second Int'l Workshop on Hot Topics in Software Upgrades* (2009), pp. 1–5.

[47] HAYDEN, C. M., MAGILL, S., HICKS, M., FOSTER, N., AND FOSTER, J. S. Specifying and verifying the correctness of dynamic software updates. In *Proc. of the Fourth Int'l Conf. on Verified Software: Theories, Tools, Experiments* (2012), pp. 278–293.

[48] HAYDEN, C. M., SMITH, E. K., DENCHEV, M., HICKS, M., AND FOSTER, J. S. Kitsune: Efficient, general-purpose dynamic software updating for C. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Appilcations* (2012).

[49] HAYDEN, C. M., SMITH, E. K., HICKS, M., AND FOSTER, J. S. State transfer for clear and efficient runtime updates. In *Proc. of the Third Int'l Workshop on Hot Topics in Software Upgrades* (2011), pp. 179–184.

[50] HERDER, J. N., BOS, H., GRAS, B., HOMBURG, P., AND TANENBAUM, A. S. Reorganizing UNIX for reliability. In *Proc. of the 11th Asia-Pacific Conf. on Advances in Computer Systems Architecture* (2006), pp. 81–94.

[51] HOSEK, P., AND CADAR, C. Safe software updates via multi-version execution. In *Proc. of the Int'l Conf. on Software Engineering* (2013), pp. 612–621.

[52] HÄRTIG, H., HOHMUTH, M., LIEDTKE, J., WOLTER, J., AND SCHÖNBERG, S. The performance of microkernel-based systems. In *Proc. of the 16th ACM Symp. on Oper. Systems Prin.* (1997), pp. 66–77.

[53] HUNT, G. C., AND LARUS, J. R. Singularity: Rethinking the software stack. *SIGOPS Oper. Syst. Rev. 41*, 2 (2007), 37–49.

[54] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., SEWELL, T., TUCH, H., AND WINWOOD, S. seL4: Formal verification of an OS kernel. In *Proc. of the 22nd ACM Symp. on Oper. Systems Prin.* (2009), pp. 207–220.

[55] KRAMER, J., AND MAGEE, J. The evolving philosophers problem: Dynamic change management. *IEEE Trans. Softw. Eng. 16*, 11 (1990), 1293–1306.

[56] LATTNER, C., AND ADVE, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. of the Int'l Symp. on Code Generation and Optimization* (2004), p. 75.

[57] MAKRIS, K., AND BAZZI, R. Immediate multi-threaded dynamic software updates using stack reconstruction. In *Proc. of the USENIX Annual Tech. Conf.* (2009), pp. 397–410.

[58] MAKRIS, K., AND RYU, K. D. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *Proc. of the Second ACM European Conf. on Computer Systems* (2007), pp. 327–340.

[59] MAURER, M., AND BRUMLEY, D. TACHYON: Tandem execution for efficient live patch testing. In *Proc. of the 21st USENIX Security Symp.* (2012), p. 43.

[60] NAGARAJA, K., OLIVEIRA, F., BIANCHINI, R., MARTIN, R. P., AND NGUYEN, T. D. Understanding and dealing with operator mistakes in internet services. In *Proc. of the 6th USENIX Symp. on Operating Systems Design and Implementation* (2004), pp. 5–5.

[61] NEAMTIU, I., AND DUMITRAS, T. Cloud software upgrades: Challenges and opportunities. In *Proc. of the Int'l Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems* (2011), pp. 1–10.

[62] NEAMTIU, I., AND HICKS, M. Safe and timely updates to multi-threaded programs. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation* (2009), pp. 13–24.

[63] NEAMTIU, I., HICKS, M., FOSTER, J. S., AND PRATIKAKIS, P. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation* (2008), pp. 37–49.

[64] NEAMTIU, I., HICKS, M., STOYLE, G., AND ORIOL, M. Practical dynamic software updating for C. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation* (2006), pp. 72–83.

[65] NG, W. T., AND CHEN, P. M. The systematic improvement of fault tolerance in the Rio file cache. In *Proc. of the 29th Int'll Symp. on Fault-Tolerant Computing* (1999), p. 76.

[66] OLIVEIRA, F., NAGARAJA, K., BACHWANI, R., BIANCHINI, R., MARTIN, R. P., AND NGUYEN, T. D. Understanding and validating database system administration. In *Proc. of the USENIX Annual Tech. Conf.* (2006), pp. 213–228.

[67] O'REILLY, T. What is Web 2.0. `http://oreilly.com/pub/a/web2/archive/what-is-web-20.html`.

[68] PATTABIRAMAN, K., SAGGESE, G. P., CHEN, D., KALBARCZYK, Z. T., AND IYER, R. K. Automated derivation of application-specific error detectors using dynamic analysis. *IEEE Trans. Dep. Secure Comput. 8*, 5 (2011), 640–655.

[69] RESCORLA, E. Security holes... who cares? In *Proc. of the 12th USENIX Security Symp.* (2003), vol. 12, pp. 6–6.

[70] ROWASE, O., AND LAM, M. S. A practical dynamic buffer overflow detector. In *Proc. of the 11th Annual Symp. on Network and Distr. System Security* (2004), pp. 159–169.

[71] STOYLE, G., HICKS, M., BIERMAN, G., SEWELL, P., AND NEAMTIU, I. Mutatis mutandis: Safe and predictable dynamic software updating. *ACM Trans. Program. Lang. Syst. 29*, 4 (2007).

[72] SWIFT, M. M., BERSHAD, B. N., AND LEVY, H. M. Improving the reliability of commodity operating systems. *ACM Trans. Comput. Syst. 23*, 1 (2005), 77–110.

[73] TUCEK, J., XIONG, W., AND ZHOU, Y. Efficient online validation with delta execution. In *Proc. of the 14th Int'l Conf. on Architectural support for programming languages and operating systems* (2009), pp. 193–204.

[74] VANDEWOUDE, Y., EBRAERT, P., BERBERS, Y., AND D'HONDT, T. Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. *IEEE Trans. Softw. Eng. 33*, 12 (2007), 856–868.

[75] ZHENG, W., BIANCHINI, R., JANAKIRAMAN, G. J., SANTOS, J. R., AND TURNER, Y. JustRunIt: Experiment-based management of virtualized data centers. In *Proc. of the USENIX Annual Tech. Conf.* (2009), p. 18.

Bibliography

[1] S. Beattie, S. Arnold, C. Cowan, P. Wagle, C. Wright, and A. Shostack. *Timing the application of security patches for optimal uptime*. *Proceedings of LISA*, 2(16): 233–242, 2002.

[2] Jonathan Corbet, Greg Kroah-Hartman, and Amanda McPherson. *Linux Kernel Development – How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It*. Technical report, The Linux Foundation, March 2012. URL http://go.linuxfoundation.org/who-writes-linux-2012.

[3] Leyla Bilge and Tudor Dumitras. Before we knew it: an empirical study of zero-day attacks in the real world. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 833–844. ACM, 2012.

[4] Tudor Dumitraş and Priya Narasimhan. *Why do upgrades fail and what can we do about it?: toward dependable, online upgrades in enterprise system*. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, page 18. Springer-Verlag New York, Inc., 2009.

[5] Tudor Dumitras, Priya Narasimhan, and Eli Tilevich. *To upgrade or not to upgrade: impact of online upgrades across multiple administrative domains*. In *ACM Sigplan Notices*, volume 45, pages 865–876. ACM, 2010.

[6] Jeff Arnold and M Frans Kaashoek. Ksplice: Automatic rebootless kernel updates. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 187–198. ACM, 2009.

[7] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. *Practical dynamic software updating for C*, volume 41. ACM, 2006.

[8] Iulian Neamtiu and Michael Hicks. Safe and timely updates to multi-threaded programs. In *ACM Sigplan Notices*, volume 44, pages 13–24. ACM, 2009.

[9] Christopher M Hayden, Edward K Smith, Michail Denchev, Michael Hicks, and Jeffrey S Foster. Kitsune: Efficient, general-purpose dynamic software updating for

c. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 249–264. ACM, 2012.

[10] Cristiano Giuffrida, Anton Kuijsten, and Andrew S Tanenbaum. Safe and automatic live update for operating systems. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, pages 279–292. ACM, 2013.