Safe and Automatic Live Update for Operating Systems

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

Abstract

Increasingly many systems have to run all the time with no downtime allowed. Consider, for example, systems controlling electric power plants and e-banking servers. Nevertheless, security patches and a constant stream of new operating system versions need to be deployed without stopping running programs. These factors naturally lead to a pressing demand for live update—upgrading all or parts of the operating system without rebooting. Unfortunately, existing solutions require significant manual intervention and thus work reliably only for small operating system patches.

In this paper, we describe an automated system for live update that can safely and automatically handle major upgrades without rebooting. We have implemented our ideas in PROTEOS, a new research OS designed with live update in mind. PROTEOS relies on system support and nonintrusive instrumentation to handle even very complex updates with minimal manual effort. The key novelty is the idea of *state quiescence*, which allows updates to happen only in safe and predictable system states. A second novelty is the ability to automatically perform transactional live updates at the *process level*, ensuring a safe and *stable* update process. Unlike prior solutions, PROTEOS supports automated state transfer, state checking, and *hot rollback*. We have evaluated PROTEOS on 50 real updates and on novel live update scenarios. The results show that our techniques can effectively support both simple and complex updates, while outperforming prior solutions in terms of flexibility, security, reliability, and stability of the update process.

Categories and Subject Descriptors D.4.7 [*Operating Systems*]: Organization and Design

General Terms Design, Experimentation, Reliability

Keywords Live update, Automatic updates, State transfer, State checking, Update safety, Operating systems

1. Introduction

Modern operating systems evolve rapidly. Studies on the Linux kernel have shown that its size has more than doubled in the last 10 years, with a growth of more than 6 MLOC and over 300 official versions released [64]. This trend leads to many frequently released updates that implement new features, improve performance, or fix important bugs and security vulnerabilities. With today's pervasive demand for 24/7 operation, however, the traditional patch-install-reboot cycle introduces unacceptable downtime for the operating system (OS) and all the running applications. To mitigate this problem, enterprise users often rely on "*rolling upgrades*" [27]—upgrading one node at a time in a highly replicated software system—which, however, require redundant hardware (or virtualized environments), cannot normally preserve program state across system versions, and may introduce a very large update window with high exposure to mixed version races [29].

Live update (sometimes also called *hot* or *dynamic* update) is a potential solution to this problem, due to its ability to upgrade a running system on the fly with no service interruption. To reach widespread adoption, however, a live update solution should be practical and trustworthy. We found that existing solutions for operating systems [12–14, 20, 56] and generic C programs [10, 21, 55, 55, 59, 60] meet these requirements only for simple updates. Not surprisingly, many live update solutions explicitly target small security patches [10, 12]. While security patches are a critical application for live update—as also demonstrated by the commercial success of solutions like Ksplice [12]—we believe there is a need for solutions that can effectively handle more complex updates, such as upgrading between operating system versions with hundreds or thousands of changes.

We note two key limiting factors in existing solutions. First, they scale poorly with the size and complexity of the update. This limitation stems from the limited system support to ensure update safety and transfer the run-time state from one system version to another. Existing solutions largely delegate these challenging tasks to the programmer. When applied to updates that install a new OS version with major code and data structure changes, this strategy requires an unbearable effort and is inevitably error prone.

Second, they scale poorly with the number of live updates applied to the system. This limitation stems from the update mechanisms employed in existing solutions, which assume a rigid address space layout, and glue code and data changes directly into the running version. This strategy typically leads to an *unstable* live update process, with memory leakage and performance overhead growing linearly over time (§2.3). We found that both limiting factors introduce important reliability and security issues, which inevitably discourage acceptance of live update.

This paper presents PROTEOS, a new research operating system designed to safely and automatically support many classes of live updates. To meet this challenge, PROTEOS introduces several novel techniques. First, it replaces the long-used notion of *function* [12] and *object* [14] *quiescence* with the more general idea of *state quiescence*, allowing programmer-provided *state filters* to specify constraints on the update state. The key intuition is that operating systems quickly transition through many states with different properties. Restricting the update to be installed only in specific states dramatically simplifies reasoning on update safety.

Further, PROTEOS employs transactional *process-level* live updates, which reliably replace entire processes instead of individual objects or functions. To increase the update surface and sup-

ASPLOS'13, March 16–20, 2013, Houston, Texas, USA.

port complex updates, we explore this idea in a design with all the core OS subsystems running as independent event-driven processes on top of a minimal message-passing substrate running in kernel mode. Using processes as updatable units ensures a *stable* update process and eliminates the need for complex patch analysis and preparation tools. In addition, PROTEOS uses hardware-isolated processes to sandbox the state transfer execution in the new process version and support *hot rollback* in case of run-time errors.

Finally, PROTEOS relies on compiler-generated instrumentation to automate state transfer (migrating the state between processes), state checking (checking the state for consistency), and tainted state management (recovering from a corrupted state), with minimal run-time overhead. Our state transfer framework is designed to *automatically* handle common structural state changes (e.g., adding a new field to a `struct`) and recover from particular tainted states (i.e., memory leakage), while supporting a convenient programming model for extensions. As an example, programmers can register their own callbacks to handle corrupted pointers or override the default transfer strategy for state objects of a particular type.

Our current PROTEOS implementation runs on the x86 platform and supports a complete POSIX interface. Our state management framework supports C and assembly code. Its instrumentation component is implemented as a link-time pass using the LLVM compiler framework [50].

We evaluated PROTEOS on 50 real updates (randomly sampled in the course of over 2 years of development) and novel live update scenarios: *online diversification*, *memory leakage reclaiming*, and *update failures* (§6.3). Our results show that: (i) PROTEOS provides an effective and easy-to-use update model for both small and very complex updates. Most live updates required minimal effort to be deployed, compared to the "*tedious engineering effort*" reported in prior work [20]; (ii) PROTEOS is reliable and secure. Our state transfer framework reduces manual effort to the bare minimum and can safely rollback the update when detecting unsafe conditions or run-time errors (e.g., crashes, timeouts, assertion failures). Despite the complexity of some of the 50 updates analyzed, live update required only 265 lines of custom state transfer code in total. (iii) The update techniques used in PROTEOS are *stable* and efficient. The run-time overhead is well isolated in allocator operations and only visible in microbenchmarks (6-130% overhead on allocator operations). The service disruption at update time is minimal (less than 5% macrobenchmark overhead while replacing an OS component every 20s) and the update time modest (3.55s to replace all the OS components). (iv) The update mechanisms used in PROTEOS significantly increase the update surface and enable novel live update scenarios. In our experiments, we were able to update *all* the OS components in a single fault-tolerant transaction and completely *automate* live update of as many as 4,873,735 type transformations throughout the entire operating system (§6.3).

Contribution. This paper makes several contributions. First, we identify the key limitations in existing live update solutions and present practical examples of reliability and security problems. Second, we introduce a new update model based on *state quiescence*, which generalizes existing models but allows updates to be deployed only in predictable system states. Third, we introduce transactional process-level updates, which allow safe *hot rollback* in case of update failures, and present their application to operating systems. Fourth, we introduce a new reliable and secure state transfer framework that automates state transfer, state checking, and tainted state management. Finally, we have implemented and evaluated these ideas in PROTEOS, a new research operating system designed with live update in mind. We believe our work raises several important issues on existing techniques and provides effective solutions that can drive future research in the field.

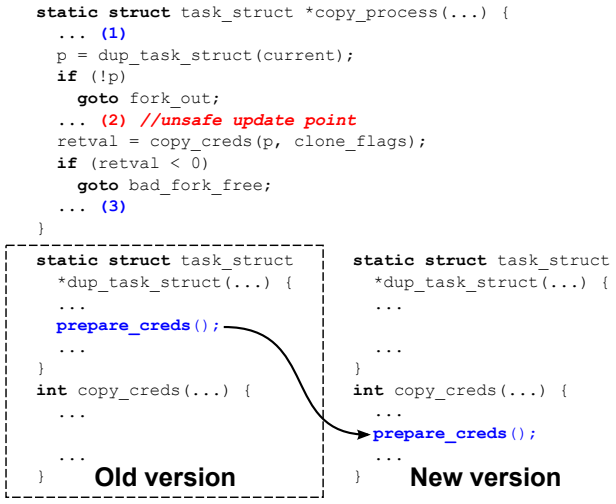

Figure 1. An unsafe live update using function quiescence.

2. Background

Gupta has determined that the validity of a live update applied in an arbitrary state S and using a state transfer function T is undecidable in the general case [37]. Hence, system support and manual intervention are needed. Unfortunately, existing solutions offer both limited control over the update state S and poor support to build the state transfer function T.

2.1 Safe Update State

Prior work has generally focused on an update-agnostic definition of a safe update state. A number of solutions permit both the old and the new version to coexist [20, 21, 56], many others disallow updates to active code [10, 12, 14, 31, 36]. The first approach yields an highly unpredictable update process, making it hard to give strong safety guarantees. The second approach relies on the general notion of *function* (or *object*) *quiescence*, which only allows updates to functions that are not on the call stack of some active thread.

Figure 1 demonstrates that quiescence is a weak requirement for a safe update state. The example proposed (inspired by real code from the Linux `fork` implementation) simply moves the call `prepare_creds()` from the function `dup_task_struct` to the function `copy_creds`. Since `copy_process` is unchanged, function quiescence would allow the update to happen at any of the update points (1, 2, 3). It is easy to show, however, that the update point (2) is unsafe, since it may allow a single invocation of the function `copy_process()` to call (i) the old version of the function `dup_task_struct()` and (ii) the new version of the function `copy_creds()`. Due to the nature of the update, the resulting execution would *incorrectly* call `prepare_creds()` twice—and not once, as expected during normal update-free execution.

To address this problem, prior live update solutions have proposed pre-annotated transactions [61], update points [60], or static analysis [59]. These strategies do not easily scale to complex operating system updates and always expose the programmer to the significant effort of manually verifying update correctness in all the possible system states. PROTEOS addresses this problem using our new notion of *state quiescence*, which generalizes prior update safety mechanisms and allows the programmer to dynamically express update constraints on a per-update basis. In the example, the programmer can simply specify a *state filter* (§4.3) requesting no `fork` to be in progress at update time.

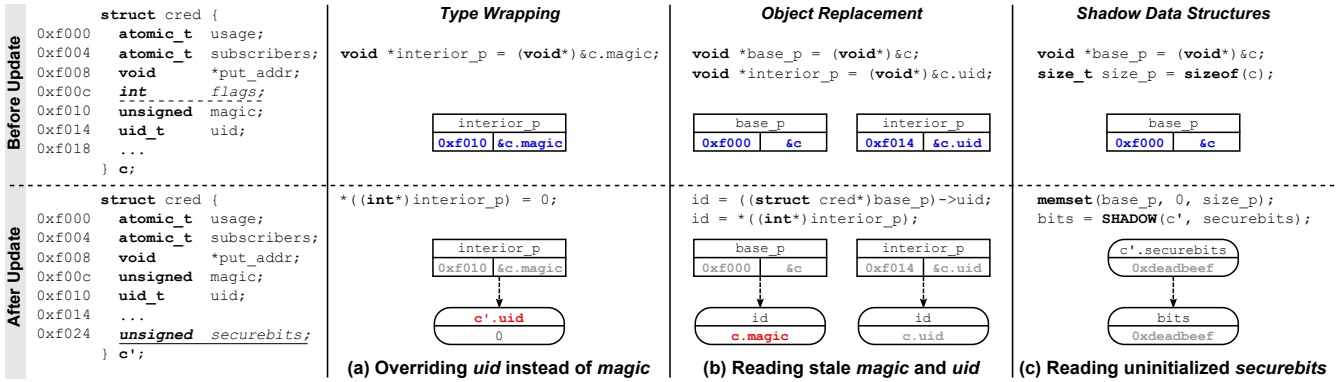

Figure 2. Examples of live update vulnerabilities introduced by unhandled pointers into updated data structures: (a) Type-unsafe memory writes; (b) Misplaced reads of stale object data; (c) Uninitialized reads.

2.2 State Transfer

Prior work has generally focused on supporting data type transformations in a rigid address space organization. Three approaches are dominant: *type wrapping* [59, 60], *object replacement* [14, 20, 21, 55], and *shadow data structures* [12, 56]. Type wrapping instruments data objects with extra padding and performs in-place type transformations. Object replacement dynamically loads the new objects into the address space and transfers the state from the old objects to the new ones. Shadow data structures are similar, but preserve the old objects and only load the new fields of the new objects. While some have automated the generation of type transformers [59, 60], *none* of the existing live update solutions for C provides automated support for transforming pointers and reallocating dynamic objects. Figure 2 demonstrates that failure to properly handle pointers into updated objects can introduce several problems, ranging from subtle logical errors to security vulnerabilities. Type wrapping may introduce type-unsafe memory reads/writes for stale interior pointers into updated objects. This is similar to a typical dangling pointer vulnerability [8], which, in the example, causes the pointer `interior_p` to erroneously write into the field `uid` instead of the field `magic`. Object replacement may introduce similar vulnerabilities for stale base pointers to updated objects. In the example, this causes the pointer `base_p` to erroneously read from the field `magic` in the old object instead of the field `uid` in the new one. It may also introduce misplaced reads/writes for stale interior pointers into updated objects. In the example, this causes the pointer `interior_p` to read the field `uid` from the old object instead of the new one. Finally, shadow data structures may introduce missing read/write errors for nonupdated code accessing updated objects as raw data. This may, for example, lead to uninitialized read vulnerabilities, as shown in the example for the field `securebits`.

Prior solutions have proposed static analysis to identify all these cases correctly [60]. This strategy, however, requires sophisticated program analysis that scales poorly with the size of the program, limits the use of some legal C idioms (e.g., `void*` pointers), and only provides the ability to *disallow* updates as long as there are some live pointers into updated objects. Thus, extensive *manual* effort is still required to locate and transfer all the pointers correctly in the common case of long-lived pointers into updated data structures. In our experience, this effort is unrealistic for nontrivial state changes. PROTEOS addresses this problem by migrating the *entire state* from one process version to another, automating pointer transfer and dynamic object reallocation with none of the limitations above. This is possible using our run-time state introspection strategy implemented on top of LLVM-based instrumentation (§5.2).

2.3 Stability of the update process

We say that a live update process is *stable* if version τ of the system with no live update applied behaves no differently than version $\tau - k$ of the same system after k live updates. This property is crucial for realistic long-term deployment of live update. Unfortunately, the update mechanisms used in existing live update solutions for C repeatedly violate the stability assumption. This is primarily due to the rigid address space organization used, with every update loading new code and data directly into the running version. This *in-place* strategy typically introduces memory leakage (due to the difficulties to reclaim dead code and data) and poorer spatial locality (due to address space fragmentation). For example, prior work on server applications reported 40% memory footprint increase and 29% performance overhead after 10 updates [60]. Further, solutions that redirect execution to the new code via binary rewriting [10, 12, 21, 56] introduce a number of trampolines (and thus overhead) that grows linearly with the number and the size of the updates. Finally, shadow data structures change the code representation and force future updates to track all the changes previously applied to the system, complicating version management over time. PROTEOS' *process-level* updates eliminate all these issues and ensure a *stable* live update process (§5).

3. Overview

Our design adheres to 3 key principles: (i) *security and reliability*: updates are only installed in predictable system states and the update process is safeguarded against errors and unsafe conditions; (ii) *large update surface*: no constraints on the size, complexity, and number of updates applied to the system; (iii) *minimal manual effort*: state filters minimize code inspection effort to ensure safety; automated state transfer minimizes programming effort for the update; process-level updates make deploying live updates as natural as installing a new release, with no need for specialized toolchains or complex patch analysis tools.

3.1 Architecture

Figure 3 shows the overall architecture of PROTEOS. Our design uses a minimalistic approach with a thin kernel only managing the hardware and providing basic IPC functionalities. All the core operating system subsystems are confined into hardware-isolated processes, including drivers, scheduling, process management, memory management, storage, and network stack. The OS processes communicate through message passing and adhere to a

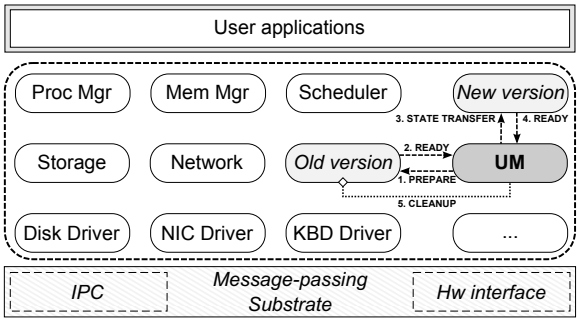

Figure 3. The architecture of PROTEOS.

well-defined event-driven model. This design is advantageous for a number of reasons. First, it introduces clear module boundaries and interfaces to simplify updatability and reasoning on update safety. Second, live updates are installed by replacing entire processes, with a new code and data representation that is no different from a freshly installed version of the system. This strategy fulfills the stability requirement and simplifies deployment of live updates. Third, the MMU-based isolation sandboxes the execution of the *entire* state transfer code in the new version, simplifying detection and isolation of run-time errors and allowing for safe *hot rollback* and no memory leakage. Finally, our event-driven update model facilitates state management and allows the system to actively *cooperate* at update time, a strategy which translates to a much more predictable and controllable live update process [33].

The update process is orchestrated by the *update manager* (UM), which provides the interface to deploy live updates for all the OS processes (including itself). When an update is available, the update manager loads the new process instances in memory and requests all the processes involved in the update to converge to the required update state. When done, every process reports back to UM and blocks. At the end of the *preparation phase*, the update manager *atomically* replaces all the processes with their new counterparts. The new processes perform state transfer and report back to the update manager when done. At the end of the *state transfer phase*, the old processes are cleaned up and the new processes are allowed to resume execution. Synchronization between the update manager and the OS processes is entirely based on message passing. Live updates use atomic transactions: the update manager can safely abort and rollback the update during any stage of the update process, since *no* changes are made to the original process. Figure 3 depicts the steps of the update process for single-component live updates (multicomponent updates are discussed in §4.5).

3.2 Update example

In PROTEOS, building a live update is as simple as recompiling all the updated components using our LLVM compiler plugin. To apply the update, programmers use `prctl`, a simple command-line utility that interfaces directly with the update manager. For example, the following command instructs the update manager to install a new version of the memory manager in the default live update state (*no event in progress*):

```
% prctl update mm /bin/mm.new
```

In our evaluation, we used this update to apply important changes to page fault handling code. An example of a multicomponent update is the following:

```
% prctl mupdate net /bin/net.new \
    -state 'num_pending_writes == 0'
% prctl mupdate e1000 /bin/e1000.new
% prctl mupdate-start
```

In our evaluation, we used this update to change the interface between the network stack and the network drivers. The state filter for the variable `num_pending_writes` is used to ensure that no affected interface interactions are in progress at update time. In our experiments, this change was applied automatically, with no manual effort required. Without the filter, the change would have required several lines of manual and error-prone state transfer code. Since interface changes between versions are common in modern operating systems [62, 63], we consider this an important improvement over the state of the art. While it should be clear that not all the updates can be so smoothly expressed with a simple state filter, this example does show that, when the state is well-captured in the form of global data structures, programmers can much more easily reason on update safety in terms of state quiescence, which frees them from the heroic effort of validating the update in many transient (and potentially unsafe) system states. In our model, identifying a single and well-defined safe update state is sufficient to guarantee a predictable and reliable update process.

3.3 Limitations

The OS design adopted in PROTEOS is not applicable as-is to commodity operating systems. Nonetheless, our end-to-end design can be easily applied to: (i) microkernel architectures used in common embedded OSes, such as L4 [49], Green Hills Integrity [3], and QNX [45]; (ii) research OSes using process-like abstractions, such as Singularity [47]; (iii) commodity OS subsystems running in user space, such as filesystems [2] and user-mode drivers in Windows [57] or Linux [19]; (iv) long-running user-space C programs. We make no claim that our OS design is the *only* possible design for a live update system. PROTEOS merely illustrates one way to implement several novel techniques that enable *truly* safe and automatic live updates. For instance, our single-component live update strategy could be also applied to monolithic OS architectures, using shadow kernel techniques [24] to enable state transfer between versions. The reduced modularity, however, would complicate reasoning on update safety for nontrivial updates. Failure to provide proper process-like isolation for the state transfer code, in turn, would lower the dependability of our hot rollback strategy.

We stress that the individual techniques described in the paper (e.g., state quiescence, automated state transfer, and automated state checking) have general applicability, and we expect existing live update solutions for commodity OSes or user-space programs to directly benefit from their integration. To encourage adoption and retrofit existing OSes and widely deployed applications, we explicitly tailored our techniques to the C programming language.

A practical limitation of our approach is the need for annotations to handle ambiguous pointer transfer scenarios (§5.3). Our experience, however, shows that the impact of these cases is minimal in practice (§6.1). Moreover, we see this as a feature rather than a limitation. Annotations compensate for the effort to manually perform state transfer and readjust all the pointers. Failing to do so leads to the reliability and security problems pointed out earlier.

Finally, a limitation of our current implementation is the inability to live update the message-passing substrate running in kernel mode. Given its small size and relatively stable code base, we felt this was not a feature to particularly prioritize. The techniques presented here, however, are equally applicable to the kernel code itself. We expect extending our current implementation to pose no more challenges than enabling live update for the update manager, which PROTEOS already supports in its current form (§4.5).

```
static int my_init() {
  ... //initialization code
  return 0;
}
int main() {
  event_eh_t my_ehs = {init : my_init};
  sys_startup(&my_ehs);
  while(1) { // event loop
    msg_t m;
    sys_receive(&m);
    process_msg(&m);
  }
  return 0;
}
```

Figure 4. The event-driven programming model.

4. Live Update Support

This section describes the fundamental mechanisms used to implement safe and automatic live update in PROTEOS.

4.1 Programming model

Figure 4 exemplifies the event-driven model used in our OS processes. The structure is similar to a long-running server program, but with special *system events* managed by the run-time system—implemented as a library transparently linked against every OS process as part of our instrumentation strategy. At startup, each process registers any custom *event handlers* and gives control to the runtime (i.e., `sys_startup()`).

At boot time, the runtime transparently invokes the *init* handler (`my_init` in the example) to run regular initialization code. In case of live update, in contrast, the runtime invokes the *state transfer* handler, responsible for initializing the new process from the old state. The default *state transfer* handler (also used in the example) *automatically* transfers all the old state to the new process, following a default state transfer strategy (§5.4). This is done by applying LLVM-based state instrumentation at compile time and automatically migrating data between processes at runtime.

After startup, each process enters an endless *event loop* to process IPC messages. The call `sys_receive()` dispatches regular messages to the loop, while transparently intercepting the special system events part of the update protocol and handling all the interactions with the update manager. The event loop is designed to be short lived, thanks to the extensive use of asynchronous IPC. This ensures scalability and fast convergence to the update state. Fast *state quiescence* is important to replace many OS processes in a single atomic transaction, eliminating the need for unsafe cross-version execution in complex updates. Note that this property does not equally apply to function *quiescence*, given that many OS subsystems never quiesce [56]. In PROTEOS, *all* the nonquiescent subsystems are isolated in *event loops* with well-defined mappings across versions. This makes it possible to update *any* nonquiescent part of the OS with no restriction. The top of the loop is the only possible *update point*, with an update logically transferring control flow from an invocation of `sys_receive()` in the old process to its counterpart in the new process (and back in case of *hot rollback*).

4.2 Virtual IPC endpoints

Two properties make it possible to support transactional process-level updates for the entire OS. First, updates are *transparent* to any nonupdated OS process or user program. Second, updates are *atomic*: only one version at the time is logically visible to the rest of the system. To meet these goals, PROTEOS uses *virtual endpoints* in its IPC implementation. A virtual endpoint is a unique version-agnostic IPC identifier assigned to the *only* active instance of an OS process. At update time, the kernel atomically rebinds all the virtual endpoints to the new instances. The switchover, which occurs at the end of the preparation phase, transparently redirects all the IPC invocations to the new version.

4.3 State filters

Unlike prior solutions, PROTEOS relies on *state quiescence* to detect a safe update state. This property allows updates to be installed only when particular constraints are met by the global state of the system. *State filters* make it possible to specify these constraints on a per-update basis. A state filter is a generic boolean expression written in a C-like language and evaluated at runtime. Our state filter evaluator supports the arithmetic, comparison, and logical operators allowed by C. It can also handle pointers to dynamically allocated objects, compute the value of any global/static variable (and subelements), and invoke read-only functions with a predetermined naming scheme. State filters reflect our belief that specifying a safe update state should be as easy as writing an assertion to check the state for consistency. Our evaluator is implemented as a simple extension to our state management framework (§5), which already provides the ability to perform run-time state introspection.

At the beginning of the preparation phase, every to-be-updated OS process receives a string containing a state filter, which is installed and transparently evaluated at the end of every following event loop iteration. When the process transitions to the required state, the expression evaluates to `true`, causing the process to report back to the update manager and block at the top of the event loop. The default state filter forces the process to block immediately. To support complex state filters that cannot be easily specified in a simple expression, PROTEOS can automatically compile generic state filter functions (written in C) into binary form. This is simply accomplished by generating *intermediate* process versions that only differ from the old ones by a new filter function `sf_custom`. Since the change is semantics-preserving, the intermediate versions can be automatically installed in the default update state before the actual update process takes place. State filter functions give the programmer the flexibility to express complex state constraints using any valid C code. On the other hand, regular state filters, are a simpler and smoother solution for online development and fast prototyping. They are also safer, since the state expression is checked for correctness by our state transfer framework.

4.4 Interface filters

Our short-lived event loop design is not alone sufficient to guarantee convergence to the update state in the general case, especially when the system is under heavy load. To give stronger convergence guarantees in particular scenarios, PROTEOS supports (optional) *interface filters* for every to-be-updated OS process. Each filter is transparently installed into the kernel at the beginning of the preparation phase. Its goal is to monitor the incoming IPC traffic and temporarily block delivery of messages that would otherwise delay state quiescence. Programmers can specify *filtering rules* similar to those used in packet filters [40], to selectively blacklist or whitelist delivery of particular IPC messages by source or type.

4.5 Multicomponent updates

Changes that affect IPC interactions require the system to atomically update multiple processes in a single update transaction. To support multicomponent updates, the update manager orderly runs the preparation protocol with every to-be-updated OS process. The overall preparation phase is strictly *sequential*, namely the process

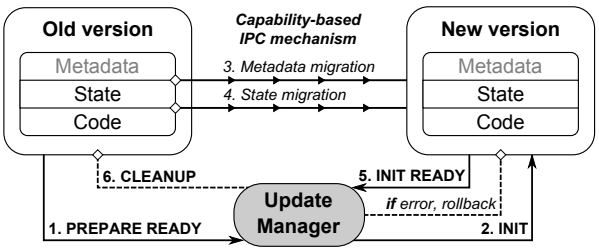

Figure 5. The state transfer process.

i in the update transaction is only requested to start the preparation phase after the process $i-1$ has already reached state quiescence and blocked. The state transfer phase is, in contrast, completely *parallel*. Parallelism is allowed to avoid placing any restrictions on state transfer extensions that require updated processes to initialize some mutual state. Our sequential preparation strategy, in turn, ensures a predictable live update process and gives the programmer full control over the update transaction, while preserving the ability to safely and automatically rollback the update in case of programming errors (i.e., deadlocks or other synchronization issues). Our design introduces a new structured definition of the live update problem: a live update is feasible if it is possible to identify a sequence of state and interface filters able to drive the system into a state with a valid mapping—and state transfer function—in the new version. Our experience shows that this approach is effective and scales to complex updates. For instance, following a top-down update strategy, we were successfully able to implement a *fault-tolerant* update transaction that atomically replaces *all* the OS processes, *including the update manager itself*.

To update the update manager, PROTEOS uses two simple ideas. First, the update manager is constrained to be the last process in the update transaction to obey the semantics of the update process. At the end of the preparation phase, kernel support allows the update manager to block and atomically yield control to its new process version. Second, the new version completes the update process as part of its own state transfer phase. Once the automated state transfer process completes (§5.1), the new manager updates its state to account for its own update and normally waits for the other OS processes to synchronize. This simple strategy added less than 200 lines of code to our original update manager implementation.

4.6 Hot rollback

In case of unexpected errors, *hot rollback* enables the update manager to abort the update process and safely allow the old version to resume execution. Our manager can detect and automatically recover from the following errors: (i) timeouts in the preparation phase (e.g., due to broken dependencies in the update transaction or poorly designed state/interface filters which lead to deadlocks or other synchronization errors); (ii) timeouts in the state transfer phase (e.g., due to synchronization errors or infinite loops); (iii) fatal errors in the state transfer phase (e.g., due to crashes, panics, or error conditions automatically detected by our state checking framework). The MMU-based protection prevents any run-time errors from propagating back to the old version. Fatal errors are ultimately intercepted by the kernel, which simply notifies the update manager—or its old instance, which is automatically revived by the kernel when the update manager itself is updating—to perform rollback. To atomically rollback the update during the state transfer phase, the update manager simply requests the kernel to freeze all the new instances, rebind all the virtual endpoints to the old instances, and unblock them. The new instances are cleaned up next in *cooperation* with the old version of the system.

5. State Management

To automate process-level updates, PROTEOS needs to automatically migrate the state between the old and the new process. Our migration strategy makes no assumptions about compiler optimizations or number of code or data structures changed between versions. In other words, the two processes are allowed to have *arbitrarily different* memory layouts. This allows us to support arbitrarily complex state changes with no impact on the stability of the update process. To address this challenge, PROTEOS implements *precise* run-time state introspection, which makes it possible to automate pointer transfer and dynamic object reallocation even in face of type changes. Our goal is to require help from the programmers only in the *undecidable* cases, for example, ambiguous pointer scenarios (§5.3), semantic changes that cannot be automatically settled by our state mapping and migration strategy (e.g., an update renumbering the error codes stored in global variables), and changes that also require updating external state (e.g., an update modifying the representation of some on-disk data structures).

5.1 State transfer

To support run-time state introspection, every OS process is instrumented using an LLVM link-time pass, which embeds *state metadata* in a predefined section of the final ELF binary. The metadata contains the relocation and type information required to introspect all the state objects in the process at runtime. The metadata structures use a fixed layout and are located in a randomized location only known to the process and the kernel.

Figure 5 depicts the state transfer process. The migration phase starts with the state transfer framework transferring all the metadata from the old version to the new version (local address space). This is done using a capability-based design, with the kernel granting (only) the new process read-only access to the address space of the old process. At the end of the metadata migration phase, both the old and the new metadata are available locally. This allows the framework to introspect both the old and the new state and remap all the state objects across versions. The mapping relies on a version-agnostic *naming scheme* established at compile time. This enables the framework to unambiguously pair functions, variables, strings, and dynamic objects across versions.

At the end of the pairing phase, all the paired objects are scheduled for transfer by default. Programmers can register extensions to change the pairing rules (e.g., in case of variable renaming) or instruct the framework to avoid transferring particular objects (§5.4).

In the data migration phase, the framework traverses all the old state objects (and their inner pointers) scheduled for transfer and ordinately migrates the data to their counterparts in the new version. Our traversal strategy is similar, in spirit, to a precise garbage collector that relocates objects [67]. There are, however, important differences to point out. First, all the dynamic (and static) objects are reallocated (loaded) in the new process. Second, our event loop design allows no state objects on the stack at update time. This eliminates the need to create dynamic metadata for all the local variables, which would degrade performance. Note that, to encourage adoption of our state transfer framework in other update and execution contexts (e.g., multithreaded server applications), however, our instrumentation can already support dynamic metadata generation for local variables (disabled in PROTEOS), using stack instrumentation strategies similar to those adopted by garbage collectors [67]. Finally, objects are possibly reallocated (or loaded) with a different run-time type. Unlike prior solutions, our framework applies type transformations (for both objects and pointers) on-the-fly, analyzing the type differences between paired objects at runtime. This eliminates the need for complex patch analysis tools and exposes a powerful programming model to state transfer exten-

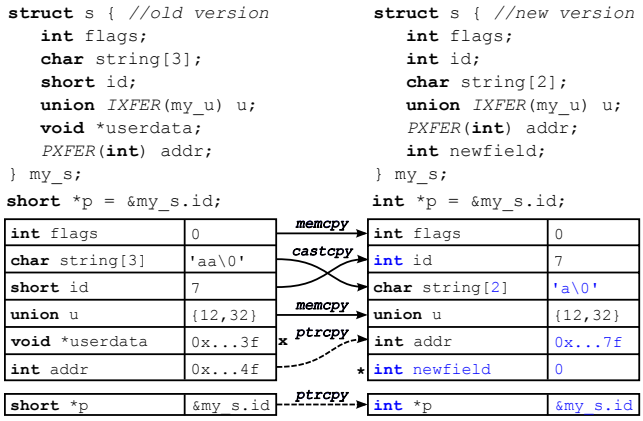

Figure 6. Automating type and pointer transformations.

sions. We clarify this claim with an example. To deploy a live update that added a new field in the middle of the `struct buf_desc` (a core data structure of the buffer cache) in our evaluation, we only had to write a simple type-based state transfer callback (§5.4) that reinitialized the new field in every object in the new version. The latter is a programmer-provided function automatically invoked by the framework on every transferred object of the requested type (e.g., `struct buf_desc`). This allows the programmer to focus on the data transformation logic while the framework automatically performs dynamic object reallocation and updates all the live pointers into the new objects. Since this `struct` was used in complex data structures like hash tables and linked lists (chained together by several base and inner pointers), this is a significant improvement over existing techniques, which would have required extensive and error-prone manual effort to implement state transfer.

5.2 Metadata instrumentation

Our LLVM transformation pass operates at the LLVM IR level and generates metadata for global/static variables (and constants), functions, and strings. Although functions and strings need not be normally transferred to the new version, their metadata is necessary to transfer pointers correctly. For each object, the pass records information on the address, the name, and the type. To create unique and version-coherent identifiers to pair static state objects across versions, our pass uses both *naming* (e.g., global variable name) and *contextual* (e.g., module name for static functions/variables) information derived from debug symbols. Note that this strategy does not prevent debug symbols from being completely stripped with no restriction from the final binary. To create metadata for dynamically allocated objects, in turn, the pass analyzes and instruments each allocation site found in the original code. Our static analysis can automatically identify `malloc/free` and `mmap/munmap` allocator abstractions, which PROTEOS natively supports for every OS process. For each allocation site, the pass records the name (derived from the caller and the allocation variable), the allocator name, and the static type. The name and the allocator name are used to pair (and reallocate) allocation sites across versions. The static type is used to dynamically determine the run-time type of every allocated object. For example, an allocation of the form `ptr = malloc(sizeof(msg_t)*4)` will be associated a static type `msg_t` and a run-time type `[4 x msg_t]`. The pass replaces every allocation/deallocation call with a call to a wrapper function responsible to dynamically create/destroy metadata for every dynamic object. To minimize the performance impact, the wrappers normally use in-band descriptors to store the metadata for the dynamic state objects. The allocators, however, support special flags to let the programmer control the allocation behavior (e.g., use out-of-band metadata for special I/O regions, or remap DMA buffers at state transfer time instead of explicitly reallocating them).

5.3 Pointer transfer

Pointers pose a fundamental challenge to automating state transfer for C programs. To transfer base and interior pointers correctly, our framework implements dynamic *points-to* analysis on top of the precise type information provided by our instrumentation. Our analysis is cast-insensitive and does not forbid or limit the use of any legal C programming idiom (e.g., `void*`), a problem in prior work [59, 60]. Our pointer transfer strategy follows 5 steps (an example is presented in Figure 6): (i) locate the target object (and the inner element, for interior pointers); (ii) locate the target object counterpart in the new version according to the output of the pairing phase; (iii) remap the inner element counterpart in case of type changes; (iv) reinitialize the pointer according to the target object (and element) counterpart identified; (v) schedule the target object for transfer. The last step is necessary to preserve the shape of arbitrarily complex data structures. In addition, the traversal allows our framework to structurally prevent any memory leakages (i.e., unreachable dynamic objects) in the old version from propagating to the new version. Note that our pointer traversal strategy relies only on the run-time type of the target object (and element), with no assumptions on the original pointer type. This strategy can seamlessly support generic `void*` pointers and eliminates the need to explicitly deal with pointer casting. Our framework can also automatically handle pointers with special integer values (e.g., `NULL` or `MAP_FAILED` (-1)) and guard pointers that mark buffer boundaries. Uninitialized pointers are structurally prevented in the allocators and dangling pointers disallowed by design. While our pointer analysis can handle all these common scenarios automatically, we have identified practical cases of *pointer ambiguity* that always require (typically one-time) user intervention, pointers stored as integers and `unions` with inner pointers, in particular. Manually handling these cases via annotations or callbacks (§5.4) is necessary to ensure precise pointer analysis. More details on our points-to analysis and our pointer transfer strategy are published elsewhere [32].

5.4 Transfer strategy

Our framework follows a well-defined default state transfer strategy, while allowing programmer-provided extensions to arbitrarily override the default behavior.

Figure 6 shows an example of the transfer strategy followed by our framework for a simple update. All the objects and the pointers are automatically transferred (and reallocated on demand) to the new version in spite of type changes. Our default transfer strategy automates state transfer for many common structural changes, such as: (i) primitive type transformations, (ii) array truncation/expansion, and (iii) addition/deletion/reordering of `struct` fields. Extensions can be used to handle more complex state changes (and cases of pointer ambiguity) with minimal effort. The latter are supported in the form of *type-based* or *object-based* annotations or callbacks, evaluated every time the framework traverses or remaps the intended type (or object). Annotations are implemented at the preprocessor level with no changes in the compiler. Figure 6 shows an example, with the `IXFER` and `PXFER` type-based annotations forcing the framework to `memcpy` the `union u` (without introspecting it) and perform pointer transfer of the integer `addr`.

Programmer-provided state transfer callbacks, in turn, provide a more generic extension mechanism to override the default state transfer behavior during any stage of the state transfer process and

			Update LOC			Changes			Manual effort			Time (ms)
Category	**#**	**Multi**	**Total**	**Median**	**90thP**	**Fun**	**Var**	**Ty**	**Ann**	**SF**	**ST LOC**	**Med Upd**
Bug fixes	15	4	1593	18	1231	27	2	2	-	2	55	397
Maintenance	12	5	2206	62	872	16	7	8	-	1	16	230
New features	19	6	10122	195	2435	199	45	101	-	1	63	202
Performance	4	1	652	179	291	10	2	7	-	0	131	358
Total	50	16	14573	63	709	252	56	118	14	4	265	272

Table 1. Overview of all the updates analyzed in our evaluation.

at several possible levels of abstraction. For instance, programmers can register object-level callbacks and element-level callbacks—evaluated when the framework performs a particular action on a given object or an element part of an object, respectively. To specify the trigger entity in the most flexible way, callbacks can be registered by object/element storage (e.g., data, heap), object/element name (e.g., `my_var_namespace_*`), and object/element type (e.g., `struct my_struct_s`), or using any combination thereof. To support many possible trigger events, programmers can register object/element pairing callbacks (to override the default name-based pairing strategy adopted by the framework), object/element transfer callbacks (to override the default transfer strategy or selectively schedule individual objects for transfer), and pointer transfer callbacks (to override the default pointer transfer strategy). Note that the callbacks are automatically and logically chained together by the framework. For example, a user-defined element pairing callback that remaps a `struct` field in a nonstandard way in the new version is automatically invoked by the framework when either transferring the original field to the new version or remapping an inner pointer to the field into an updated object. The callbacks all run in the context of the new process version after completing the metadata migration phase, allowing the programmer to seamlessly access objects in the old and the new version (and their metadata information) with no restriction. The callbacks are written directly in C, providing the ability to operate arbitrary transformations in the state transfer code—even changing external state on the disk, for example. In addition, this allows the programmer to remap complex data structures that significantly change their representation across versions (e.g., a hash table transformed into multiple balanced BSTs) and cannot be automatically paired (nor transferred) by our framework. Even in such complex state transformation scenarios, our programming model can provide a generic callback-driven interface to locate and traverse all the objects (and pointers) to transfer, allowing the programmer to select the best level of abstraction to operate and concentrate on data transformations rather than on manual and error-prone state introspection.

5.5 State checking

Our state management framework supports automated state checking using generic *state invariants*. The idea is to detect an invalid state when conservatively defined invariants are violated. *Target-based* invariants are naturally enforced by our *points-to* analysis (i.e., a pointer not pointing to any valid object is invalid). Other invariants are determined by static analysis. We support *value-based* invariants (derived from value set analysis of integer variables) and *type-based* invariants, which verify that a pointer points to a target of a valid type at runtime. This is done by recording metadata on all the valid implicit and explicit pointer casts (i.e., `bitcast` and `inttoptr` LLVM instructions). State checking is performed on the old version before the transfer and on the new version after the transfer. In both cases, the transfer is atomically aborted when invariants violations are found (unless extensions change the default behavior). Checking both the old and the new state allows the framework to detect: (i) a tainted state in the old version (i.e., arbitrary memory corruption) and possibly let extensions recover from it; (ii) corruption in the new state introduced by the state transfer code itself; (iii) violating assumptions in the state transfer process. An example in the latter category is the attempt to transfer a pointer to an old object that no longer exists (or no longer has its address taken) in the new version.

6. Evaluation

We have implemented PROTEOS on the x86 platform. The current PROTEOS implementation is a major rewrite of the original MINIX 3 microkernel-based operating system, which only provided process-based isolation for all the core OS components and restartability support for stateless device drivers [43]. Our current prototype includes 22 OS processes (8 drivers and 14 servers) and supports a complete POSIX interface. The static instrumentation is implemented as an LLVM pass in 6550 LOC [1]. The state management framework is implemented as a static library written in C in 8840 LOC. We evaluated PROTEOS on a workstation equipped with a 12-core 1.3Ghz AMD Opteron processor and 4GB of RAM. For evaluation purposes, we ported the C programs in the SPEC CPU 2006 benchmark suite to PROTEOS. We also put together an *sdtools* macrobenchmark, which emulates a typical syscall-intensive workload with common development operations (compilation, text processing, copy, delete) performed on the entire OS source tree. We repeated all our experiments 21 times and reported the median. Our evaluation focuses on 4 key aspects: (i) *Experience*: Can PROTEOS effectively support both simple and complex updates with minimal effort? (ii) *Performance*: Do our techniques yield low runtime overhead and realistic update times? (iii) *Service disruption*: Do live updates introduce low service disruption? (iv) *Memory footprint*: How much memory do our techniques use?

6.1 Experience

To evaluate the effort in deploying live updates, we randomly sampled 50 real updates produced by the team of core MINIX 3 developers in the course of over 2 years. The live update infrastructure, in turn, was developed independently to ensure a fair and realistic update evaluation. We carefully analyzed each update considered, prepared it for live update, and finally deployed it online during the execution of our SPEC and *sdtools* benchmarks. We successfully deployed all the live updates considered and checked that the system was fully functional before and after each experiment. In 4 cases, our first update attempt failed due to bugs in the state transfer code. The resulting (pointer) errors, however, were immediately detected by our state transfer framework and the update safely rolled back with no consequences for the system. We also

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

	PROTEOS	Linux
`malloc`	2.30	1.41
`free`	1.19	1.09
`mmap`	1.41	1.77
`munmap`	1.06	1.42

Table 2. Execution time of instrumented allocator operations normalized against the baseline.

verified that the update process was stable (no performance/space overhead increase over time) and that our live update infrastructure could withstand arbitrary compiler optimization changes between versions (e.g., from `-O1` to `-O3`). Table 1 presents our findings.

The first three grouped columns provide an overview of all the updates analyzed, with the number of updates considered per category and the number of updates that involved multiple OS processes. The *New features* category has the highest number of updates, given that MINIX 3 is under active development. Of the 50 updates considered, 16 involved multiple OS processes. This confirmed the importance of supporting multicomponent live updates. The second group of columns shows the number of lines of code (total, median, 90th percentile) changed across all the updates, with a total of nearly 15,000 LOC. The third group shows the number of functions, variables, and types changed (i.e., added/deleted/modified). For example, *New features* updates introduced 199 function changes, 45 variable changes, and 101 type changes. The fourth group, in turn, shows the manual effort required in terms of annotations, state filters, and lines of code for state transfer extensions. We only had to annotate 14 declarations (using 3 object-based annotations, 10 type-based annotations, and 1 type-based callback) throughout the entire PROTEOS source code. Encouragingly, this was a modest *one-time* effort that required less than 1 man week. The annotations were only necessary for `unions` with inner pointers and special nontransferrable state objects (e.g., allocator variables). Custom state filters, in turn, were only required for 4 updates. We found that, for most updates, our event loop design gave sufficient predictability guarantees in the default update state. In the remaining cases, however, we faced complex interface changes that would have required extensive manual effort without support for custom state filters. From empirical evidence, we also believe that more than half of the updates would have been extremely hard to reason about using only function quiescence. Despite the many variable and type changes, all the updates required only 265 LOC of state transfer extensions. We found that our state transfer framework was able to fully automate most data structure changes (i.e., addition/removal). In addition, our type-based state transfer callbacks minimized the effort to handle cross-cutting type changes. Finally, the last column reports the median update time, with a value of 272ms across all the updates. We also measured a maximum update time of 3550ms for cross-cutting updates that replaced *all* the OS processes (individually or in a single multicomponent and fault-tolerant transaction).

We now compare our results with prior solutions. Before ours, Ksplice was the only OS-level live update solution evaluated with a comprehensive list of updates over a time interval [12]. Compared to ours, however, their evaluation is based on security patches of much smaller size. Their median patch size is less than 5 LOC, and the 90th percentile less than 30 LOC. As Table 1 demonstrates, PROTEOS was evaluated with much more complex updates, while only requiring 265 LOC for state transfer extensions (compared to 132 LOC for Ksplice's 64 small patches [1]). Many other live update solutions for C present only case studies [13, 14, 20, 56] or lack a proper quantitative analysis of the manual effort required [10, 21, 55]. Many researchers, however, have reported "*tedious implementation of the transfer code*" [13], "*tedious engineering efforts*" [20], "*tedious work*" [21], and "*an arduous testing process that spanned several weeks of concentrated work*" [10]. In contrast, we found our techniques to reduce the live update effort to the bare minimum. In particular, our entire update evaluation required only 10 man days. Ginseng [60] and Stump [59] are the only prior solutions for C that provide quantitative measurements for the manual effort. Ginseng (*Stump*) required 140 (*186*) source changes and 336 (*173*) state transfer LOC to apply 30 (*13*) server application updates introducing 21919 (*5817*) new LOC in total. While it is difficult to directly compare their results on server applications with ours, we believe that our techniques applied to the same set of updates would have significantly reduced the effort, avoiding manual inspection or code restructuring to eliminate unsupported C idioms, posing no restriction on the nature of the data structure changes, and assisting the programmer in challenging tasks like heap traversal, pointer transfer, and state checking.

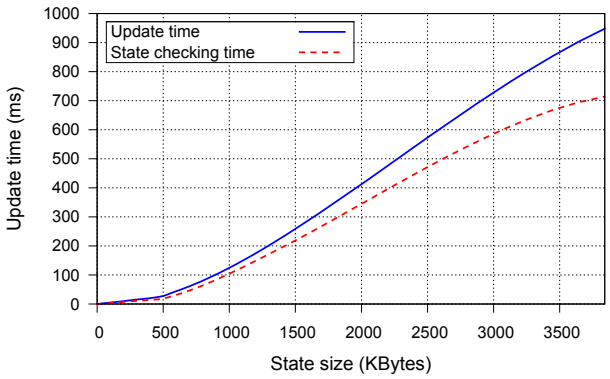

Figure 7. Update time vs. run-time state size.

6.2 Performance

We evaluated the run-time overhead imposed by the update mechanisms used in PROTEOS. Virtual endpoints introduce only update-time costs and no extra run-time overhead on IPC. Transparent interception of special system events introduces only 3 additional cycles per event loop iteration. An important impact comes also from the microkernel-based design itself. Much prior work has been dedicated to improving the performance of IPC [52] and microkernel-based systems in general [39, 53]. Our focus here is on the update techniques rather than on microkernel performance. For instance, our current measurements show that the `gettimeofday`, `open`, `read`, `write`, `close` system calls are 1.05-8.27x slower than on Linux due to our microkernel design. These numbers are, however, pessimistic, given that we have not yet operated many optimizations described in the literature [39, 52, 53].

Much more critical is to assess the cost of our state instrumentation, which directly affects the applicability of our techniques to other OS architectures or user-space applications. To this end, we first ran our SPEC and *sdtools* macrobenchmarks to compare the base PROTEOS implementation with its instrumented version. Our repeated experiments reported no noticeable performance degradation. This is expected since static metadata, used only at update time, is isolated in a separate ELF section with no impact on spatial locality. The use of in-band descriptors to generate dynamic metadata, in turn, minimizes the run-time overhead on allocator operations. To isolate this overhead, we measured the cost of our instrumentation on 10,000 `malloc`/`free` and `mmap`/`munmap` repeated allocator operations. We ran the experiments for multiple allocation sizes (0-16MB) and reported the median overhead

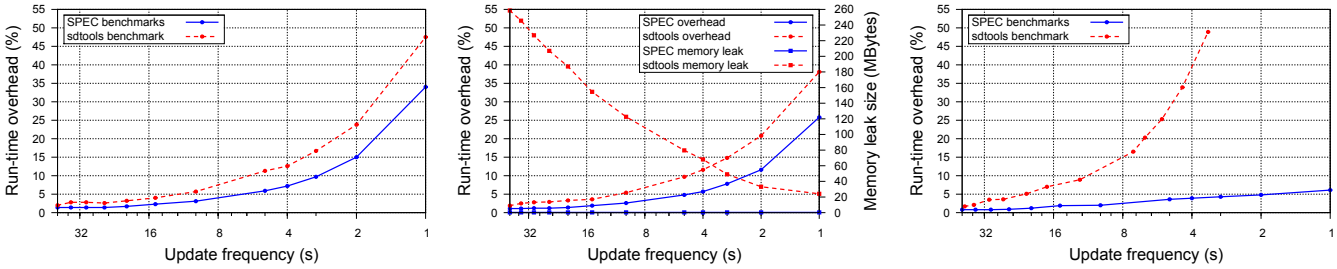

Figure 8. Run-time overhead vs. update frequency. Results are reported for our benchmarks in 3 different live update scenarios: (a) online diversification; (b) memory leakage reclaiming; (c) update failures.

for `malloc`/`free` (the overhead does not generally depend on the allocation size) and the maximum overhead for `mmap`/`munmap` (the overhead generally decreases with the allocation size). For comparison, we also ported our instrumentation to Linux user-space programs and ran the same microbenchmarks on Ubuntu 10.04 LTS 32-bit (`libc` allocators). Table 2 depicts our results, with a different (but comparable) impact of our state instrumentation in the two allocator implementations. The highest overheads in PROTEOS and Linux are incurred by `malloc` (130%) and `mmap` (77%), respectively. Note that these results are overly pessimistic, since common allocation patterns typically yield poorer spatial locality, which will likely mask the overhead on allocator operations further.

We now compare our results with prior live update techniques. Live update solutions based on (more intrusive) instrumentation strategies have reported macrobenchmark results with worst-case overheads of 6% [60], 6.71%[59], and 96.4% [55]. Solutions based on binary rewriting, in turn, have reported microbenchmark results with 1%-8% invocation overhead [56] for updated functions. Unlike all the existing techniques, our overhead is well-isolated in allocator operations and never increases with the number and the size of the updates applied to the system (stability assumption).

To assess the impact of live updates on the system, we also analyzed the distribution of the update time in more detail. Figure 7 depicts the update time (the time from the moment the update is signaled to the moment the new version resumes execution) as a function of the run-time state size (total size of all the static and dynamic state objects). These (interpolated) results reflect average measurements obtained during the execution of our macrobenchmarks for all the single-component updates used in our evaluation. The figure shows that the update time grows approximately linearly with the state size. This behavior stems from the fact that the update time is heavily dominated by state transfer and state checking (isolated in the figure). The time to load the new processes in memory and complete the preparation phase is normally marginal. We experimented with many state filters to quiesce all the common OS process interactions and found that the time to reach state quiescence was only a few milliseconds in the worst case. This property makes any overhead associated to evaluating state and interface filters in the preparation phase *marginal*. While our overall update times are generally higher than prior solutions for simple updates (since we replace entire processes instead of individual functions), the resulting impact is still orders of magnitude shorter than any reboot and bearable for most systems.

6.3 Service disruption

To substantiate our last claim, we evaluated the service disruption caused by live update. Figure 8 shows the run-time overhead incurred by our macrobenchmarks when periodically updating OS processes in a round-robin fashion. The overhead increases for shorter update intervals, with more disruption incurred by *sdtools*. The figures present three novel live update scenarios. Figure 8a presents results for an *online diversification* scenario—an idea we have also developed further in [34]. We implemented a source-to-source transformation able to *safely* and automatically change 3872 type definitions (adding/reordering `struct` elements and expanding arrays/primitive types) throughout the entire operating system. The changes were randomized for each generated operating system version, introducing a heavily diversified memory layout across updates. This scenario stressed the capabilities of our state transfer framework, introducing an average of 4,873,735 type transformations at each update cycle and approximating an upper bound for the service disruption. Figure 8b presents a *memory leakage reclaiming* scenario. Updates were performed between identical OS versions (approximating a lower bound for the service disruption), but we deliberately introduced a memory leak bug (similar to one found during development) that caused the virtual filesystem not to free the allocated memory at `exec()` time. Shorter update intervals increase the overhead but allow our state transfer framework to automatically repair leaks more quickly. The tradeoff is evident for *sdtools*, which `exec()`ed several programs during the experiment. Figure 8c presents an *update failures* scenario. We deliberately simulated state transfer crashes or 2-second timeouts (in equal measure) for each update, resulting in more severe service disruption for the syscall-intensive benchmark *sdtools*, but with no system-perceived impact. This scenario stressed the *unique* fault-tolerant capabilities of our live update infrastructure, able to withstand significant update failures and automatically rollback the entire update transaction with no consequences for the operating system and all the running programs. Overall, our experiments reported a negligible overhead for update intervals larger than 20s. Given that updates are relatively rare events, we expect the update-induced service disruption to be minimal in practice.

6.4 Memory footprint

Our metadata instrumentation naturally leads to a larger memory footprint at runtime. Our current implementation required an average of 65 bytes for each type, 27 extra bytes for each variable/constant/string, 38 extra bytes for each function with address taken, 10 extra bytes for each allocation, and 38 bytes for each allocation site. During the execution of our macrobenchmarks, we measured an average state overhead (i.e., metadata size vs. run-time state size) of 18% and an overall memory footprint overhead of 35% across all the operating system processes. While comparable to prior instrumentation-based live update techniques [56, 59, 60],

our memory footprint overhead never increases with the number and the size of the updates applied to the system. This is ensured by our stable live update strategy. For comparison, we also ported the Linux ACPI driver to PROTEOS. Despite the very complex code base, adding updatability only required 2 type-based callbacks for 2 `unions`. In this case, the state overhead and the overall memory footprint overhead measured were 41% and 37%, respectively.

7. Related work

Several live update solutions are described in the literature, with techniques targeting operating systems [12–14, 20, 56], C programs [10, 21, 55, 59, 60], object-oriented programs [46, 74], programming languages [11, 26, 73], database systems [18], and distributed systems [6, 7, 9, 16, 17, 27, 28, 48, 76]. We focus here on live update for operating systems and generic C programs, but we refer the interested reader to [5, 35, 44, 68] for more complete surveys.

K42 [13, 14, 72] is a research OS that supports live update functionalities using object-oriented design patterns. To update live objects, K42 relies on system-enforced quiescence, transparently blocking all the threads calling into updated objects. Unfortunately, this strategy leads to a poorly predictable update process, with hidden thread dependencies potentially leading to *unrecoverable* deadlocks. In contrast, PROTEOS gives programmers full control over the update process and can automatically recover from synchronization errors (e.g., deadlocks) introduced by poorly designed update transactions using a predefined timeout. In addition, K42 provides *no* support for automated state transfer. Unlike existing live update techniques for C, however, their object-oriented approach offers a solution to the stability problem. The downside is that their techniques are limited to object-oriented programs. In contrast, the techniques we propose have more general applicability. For instance, state filters and our state management framework can be used to improve existing live update solutions for the Linux kernel [12, 20, 56] and generic C programs [10, 21, 55, 59, 60]. Our framework could be, for example, integrated in existing solutions to automatically track pointers to updated data structures or abort the update in case of unsafe behavior. Our process-level updates, in turn, are an elegant solution to the stability problem for user-level live update solutions. Also note that, while explicitly conceived to simplify state management—no need for explicit update points and stack instrumentation—and minimize manual effort—simpler to reason on update safety and control multicomponent live update transactions—our event-loop design is not strictly required for the applicability of our techniques. For instance, our event-driven model can be easily extended to multithreaded execution using annotated per-thread update points, as previously suggested in [59]. When backward compatibility is not a primary concern, however, we believe our event-driven strategy to offer a superior design for safe and automatic live update. For this reason, we opted for a pure event-driven model for our current PROTEOS implementation.

DynaMOS [56] and LUCOS [20] are two live update solutions for the Linux kernel. They both apply code updates using binary rewriting techniques. To handle data updates, DynaMOS relies on shadow data structures, while LUCOS relies on virtualization to synchronize old and new data structure versions at each write access. Both solutions advocate running the old and the new version in parallel. Unlike ours, their cross-version execution strategy leads to a highly unpredictable update process. In addition, state transfer is delegated entirely to the programmer.

Ksplice [12] is an important step forward over its predecessors. Similar to DynaMOS [56], Ksplice uses binary rewriting and shadow data structures to perform live updates. Unlike all the other live update solutions for C, however, Ksplice prepares live updates at the object code layer. This strategy simplifies patch analysis and does not inhibit any compiler optimizations or language features. Process-level updates used in PROTEOS take these important guarantees one step further. Not only are the two versions allowed to have arbitrarily different code and data layout, but patch analysis and preparation tools are no longer necessary. The new version is compiled and deployed as-is, with changes between versions automatically tracked by our state transfer framework at runtime. Moreover, Ksplice does not support update states other than function quiescence and provides no support for automated state transfer, state checking, or hot rollback.

Related to OS-level live update solutions is also work on extensible operating systems [15, 22, 69, 70] (which only allow predetermined OS extensions), dynamic kernel instrumentation [58, 75] (which is primarily concerned with debugging and performance monitoring), microkernel architectures [23, 42, 49, 71] (which can replace OS subsystems but not without causing service loss [23]), and online maintenance techniques [54, 66] (which require virtualization and domain-specific migration tools).

Similar to OS-level solutions, existing live update techniques for user-space C programs all assume an in-place update model, with code and data changes loaded directly into the running version. Redirection of execution is accomplished with compiler-based techniques [59, 60], binary rewriting [10, 21], or stack reconstruction [55]. Some techniques assume quiescence [10], others rely on predetermined update points [55, 59, 60] or allow unrestricted cross-version execution [21]. Unlike PROTEOS, these solutions offer no support to specify safe update states on a per-update basis, do not attempt to fully automate state transfer or state checking, and fail to ensure a transactional and stable update process.

Recent efforts on user-space C programs by Hayden et al. [41], developed independently from our work, also suggest using entire programs as live updatable units. Unlike our process-level updates, however, their update strategy encapsulates every program version inside a shared library and allows the old and the new version to share the same process address space with no restriction at live update time. This strategy requires every program to be compiled with only position-independent code—which may be particularly inefficient on some architectures—and also fails to properly isolate, detect, and recover from errors in the state transfer code. In addition, their state transfer strategy does not support interior pointers and unrestricted use of `void*` pointers, nor does it attempt to automate pointer transfer for heap-allocated objects with no user intervention. Finally, their system includes xfgen, a tool to generate state transformers using a domain-specific language. While a high-level language may reduce the programming effort, we found much more natural to express state transfer extensions for C programs directly in C, using a convenient and well-defined callback interface.

The techniques used in PROTEOS draw inspiration from prior work in different research areas. Our state filters are inspired by DYMOS [51], an early dynamic modification system that allowed programmers to specify procedures required to be inactive at update time. State filters are more general and easier to use, allowing programmers to specify safe update states in the most natural way. The idea of state transfer between processes was first explored by Gupta [36], but his work assumed a fixed memory layout and delegated state transfer entirely to the programmer. Our state introspection strategy is inspired by garbage collector-style object tracking, a technique also explored in live update solutions for managed languages like Java [74]. Similarly, our update-time memory leakage reclaiming strategy is inspired by prior precise garbage collection techniques for C programs [67]. Finally, state checking is inspired by invariants-based techniques to detect anomalous program behavior [4, 25, 30, 38, 65, 77]. Unlike prior techniques, our state invariants are conservatively derived from static analysis, eliminating false positives that arise from learning *likely* invariants at runtime.

8. Conclusion

In this paper, we presented PROTEOS, a new research OS designed with live update in mind. Unlike existing solutions, the techniques implemented in PROTEOS can efficiently and reliably support several classes of updates with minimal manual effort. State and interface filters allow updates to happen only in predictable system states and give programmers full control over the update process. Process-level updates completely eliminate the need for complex toolchains, enable safe hot rollback, and ensure a stable update process. Our state management framework reduces the state transfer burden to the bare minimum, fully automating state transfer for common structural state changes and exposing a convenient programming model for extensions. Finally, our state checking framework can automatically identify errors in a tainted state and detect violating assumptions in the state transfer process itself.

Acknowledgments

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

References

[1] Ksplice performance record. `http://www.ksplice.com/cve-evaluation`, 2009.

[2] FUSE: Filesystem in userspace. `http://fuse.sourceforge.net/`, 2012.

[3] Green hills integrity. `http://www.ghs.com/products/rtos/integrity.html`, 2012.

[4] S. V. Adve, V. S. Adve, and Y. Zhou. Using likely program invariants to detect hardware errors. In *Proc. of the IEEE Int'l Conf. on Dependable Systems and Networks*, 2008.

[5] S. Ajmani. A review of software upgrade techniques for distributed systems, 2004.

[6] S. Ajmani, B. Liskov, and L. Shrira. Scheduling and simulation: How to upgrade distributed systems. In *Proc. of the Ninth Workshop on Hot Topics in Operating Systems*, volume 9, pages 43–48, 2003.

[7] S. Ajmani, B. Liskov, L. Shrira, and D. Thomas. Modular software upgrades for distributed systems. In *Proc. of the 20th European Conf. on Object-oriented Programming*, pages 452–476, 2006.

[8] P. Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *Proc. of the 19th USENIX Security Symp.*, page 12, 2010.

[9] J. P. A. Almeida, M. v. Sinderen, and L. Nieuwenhuis. Transparent dynamic reconfiguration for CORBA. In *Proc. of the Third Int'l Symp. on Distributed Objects and Applications*, pages 197–207, 2001.

[10] G. Altekar, I. Bagrak, P. Burstein, and A. Schultz. OPUS: Online patches and updates for security. In *Proc. of the 14th USENIX Security Symp.*, volume 14, pages 19–19, 2005.

[11] J. R. Andersen, L. Bak, S. Grarup, K. V. Lund, T. Eskildsen, K. M. Hansen, and M. Torgersen. Design, implementation, and evaluation of the resilient Smalltalk embedded platform. *Comput. Lang. Syst. Struct.*, 31(3-4):127–141, 2005.

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

[13] A. Baumann, G. Heiser, J. Appavoo, D. Da Silva, O. Krieger, R. W. Wisniewski, and J. Kerr. Providing dynamic update in an operating system. In *Proc. of the USENIX Annual Tech. Conf.*, page 32, 2005.

[14] A. Baumann, J. Appavoo, R. W. Wisniewski, D. D. Silva, O. Krieger, and G. 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] B. N. Bershad, S. Savage, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proc. of the 15th ACM Symp. on Oper. Systems Prin.*, volume 29, pages 267–284, 1995.

[16] T. Bloom. *Dynamic module replacement in a distributed programming system*. PhD thesis, MIT, Cambridge, MA, USA, 1983.

[17] T. Bloom and M. Day. Reconfiguration and module replacement in Argus: Theory and practice. *Software Engineering J.*, 8(2):102–108, 1993.

[18] C. Boyapati, B. Liskov, L. Shrira, C. Moh, and S. Richman. Lazy modular upgrades in persistent object stores. In *Proc. of the 18th ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 403–417, 2003.

[19] S. Boyd-Wickizer and N. Zeldovich. Tolerating malicious device drivers in Linux. In *Proc. of the USENIX Annual Tech. Conf.*, page 9, 2010.

[20] H. Chen, R. Chen, F. Zhang, B. Zang, and P. Yew. Live updating operating systems using virtualization. In *Proc. of the Second Int'l Conf. on Virtual Execution Environments*, pages 35–44, 2006.

[21] H. Chen, J. Yu, R. Chen, B. Zang, and P. Yew. POLUS: A powerful live updating system. In *Proc. of the 29th Int'l Conf. on Software Eng.*, pages 271–281, 2007.

[22] C. Cowan, T. Autrey, C. Krasic, C. Pu, and J. Walpole. Fast concurrent dynamic linking for an adaptive operating system. In *Proc. of the Third Int'l Conf. on Configurable Distributed Systems*, pages 108–115, 1996.

[23] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell. CuriOS: Improving reliability through operating system structure. In *Proc. of the 8th USENIX Symp. on Operating Systems Design and Implementation*, pages 59–72, 2008.

[24] A. Depoutovitch and M. Stumm. Otherworld: giving applications a chance to survive OS kernel crashes. In *Proc. of the 5th ACM European Conf. on Computer systems*, pages 181–194, 2010.

[25] M. Dimitrov and H. Zhou. 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*, pages 73–82, 2007.

[26] D. Duggan. Type-based hot swapping of running modules. In *Proc. of the Sixth ACM SIGPLAN Int'l Conf. on Functional programming*, pages 62–73, 2001.

[27] T. Dumitras and P. 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.

[28] T. Dumitras, J. Tan, Z. Gho, and P. Narasimhan. No more HotDependencies: Toward dependency-agnostic online upgrades in distributed systems. In *Proc. of the Third Workshop on Hot Topics in System Dependability*, page 14, 2007.

[29] T. Dumitras, P. Narasimhan, and E. Tilevich. 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*, pages 865–876, 2010.

[30] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proc. of the 21st Int'l Conf. on Software Eng.*, pages 213–224, 1999.

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

[32] C. Giuffrida and A. Tanenbaum. 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*, pages 16–20, 2012.

[33] C. Giuffrida and A. S. Tanenbaum. Cooperative update: A new model for dependable live update. In *Proc. of the Second Int'l Workshop on Hot Topics in Software Upgrades*, pages 1–6, 2009.

[34] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *Proc. of the 21st USENIX Security Symp.*, page 40, 2012.

[35] D. Gupta. *On-line software version change*. PhD thesis, Indian Institute of Technology Kanpur, 1994.

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

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

[38] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proc. of the 24th Int'l Conf. on Software Eng.*, pages 291–301, 2002.

[39] H. Hartig, M. Hohmuth, J. Liedtke, J. Wolter, and S. Schönberg. The performance of microkernel-based systems. In *Proc. of the 16th ACM Symp. on Oper. Systems Prin.*, pages 66–77, 1997.

[40] D. Hartmeier. Design and performance of the OpenBSD stateful packet filter (pf). In *Proc. of the USENIX Annual Tech. Conf.*, pages 171–180, 2002.

[41] 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 Appilcations*, 2012.

[42] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Reorganizing UNIX for reliability. In *Proc. of the 11th Asia-Pacific Conf. on Advances in Computer Systems Architecture*, pages 81–94, 2006.

[43] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Failure resilience for device drivers. In *Proc. of the Int'l Conf. on Dependable Systems and Networks*, pages 41–50, 2007.

[44] M. Hicks. *Dynamic software updating*. PhD thesis, Univ. of Pennsylvania, 2001.

[45] D. Hildebrand. An architectural overview of QNX. In *Proc. of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 113–126, 1992.

[46] G. Hjálmtýsson and R. Gray. Dynamic C++ classes: A lightweight mechanism to update code in a running program. In *Proc. of the USENIX Annual Tech. Conf.*, page 6, 1998.

[47] G. C. Hunt and J. R. Larus. Singularity: Rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, 2007.

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

[49] O. K. Labs. OKL4 community site. `http://wiki.ok-labs.com/`, 2012.

[50] C. Lattner and V. 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.

[51] I. Lee. *Dymos: A dynamic modification system*. PhD thesis, Univ. of Wisconsin-Madison, 1983.

[52] J. Liedtke. Improving IPC by kernel design. In *Proc. of the 14th ACM Symp. on Oper. Systems Prin.*, pages 175–188, 1993.

[53] J. Liedtke. On micro-kernel construction. In *Proc. of the 15th ACM Symp. on Oper. Systems Prin.*, pages 237–250, 1995.

[54] D. E. Lowell, Y. Saito, and E. J. Samberg. Devirtualizable virtual machines enabling general, single-node, online maintenance. In *Proc. of the 11th Int'l Conf. on Architectural support for programming languages and operating systems*, volume 39, pages 211–223, 2004.

[55] 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.

[56] K. Makris and K. D. 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.

[57] Microsoft. Windows User-Mode driver framework. `http://msdn.microsoft.com/en-us/windows/hardware/gg463294`, 2010.

[58] R. G. Minnich. A dynamic kernel modifier for Linux. In *Proc. of the LACSI Symposium*, 2002.

[59] I. Neamtiu and M. 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.

[60] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for C. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 72–83, 2006.

[61] I. Neamtiu, M. Hicks, J. S. Foster, and P. Pratikakis. 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*, pages 37–49, 2008.

[62] Y. Padioleau, J. L. Lawall, and G. Muller. Understanding collateral evolution in Linux device drivers. In *Proc. of the First ACM European Conf. on Computer Systems*, pages 59–71, 2006.

[63] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in Linux device drivers. In *Proc. of the Third ACM European Conf. on Computer Systems*, pages 247–260, 2008.

[64] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in Linux: Ten years later. In *Proc. of the 16th Int'l Conf. on Architectural support for programming languages and operating systems*, pages 305–318, 2011.

[65] K. Pattabiraman, G. P. Saggese, D. Chen, Z. T. Kalbarczyk, and R. K. Iyer. Automated derivation of application-specific error detectors using dynamic analysis. *IEEE Trans. Dep. Secure Comput.*, 8(5):640–655, 2011.

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

[67] J. Rafkind, A. Wick, J. Regehr, and M. Flatt. Precise garbage collection for C. In *Proc. of the 2009 Int'l Symp. on Memory management*, pages 39–48, 2009.

[68] M. E. Segal and O. Frieder. On-the-fly program modification: Systems for dynamic updating. *IEEE Softw.*, 10(2):53–65, 1993.

[69] M. Seltzer and C. Small. Self-monitoring and self-adapting operating systems. In *Proc. of the Sixth Workshop on Hot Topics in Operating Systems*, pages 124–129, 1997.

[70] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proc. of the Second USENIX Symp. on Operating Systems Design and Implementation*, pages 213–227, 1996.

[71] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: A fast capability system. In *Proc. of the 17th ACM Symp. on Oper. Systems Prin.*, pages 170–185, 1999.

[72] C. A. N. Soules, D. D. Silva, M. Auslander, G. R. Ganger, and M. Ostrowski. System support for online reconfiguration. In *Proc. of the USENIX Annual Tech. Conf.*, pages 141–154, 2003.

[73] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. Mutatis mutandis: Safe and predictable dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 29(4), 2007.

[74] S. Subramanian, M. Hicks, and K. S. McKinley. Dynamic software updates: A VM-centric approach. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, volume 44, pages 1–12, 2009.

[75] A. Tamches and B. P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proc. of the Third ACM Symp. on Oper. Systems Prin.*, pages 117–130, 1999.

[76] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt. Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. *IEEE Trans. Softw. Eng.*, 33(12):856–868, 2007.

[77] P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas. AccMon: Automatically detecting memory-related bugs via program counter-based invariants. In *Proc. of the 37th Annual IEEE/ACM Int'l Symp. on Microarchitecture*, pages 269–280, 2004.