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 transforma-

tions [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.

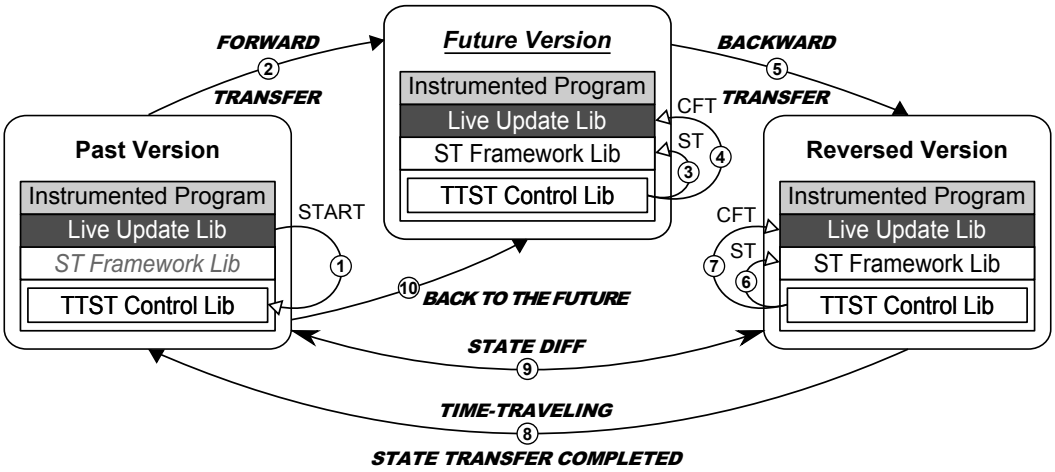

Figure 1: Time-traveling state transfer overview. The numbered arrows indicate the order of operations.

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 ← addr_start
    while a < shadow_start do
        m1 ← IS_MAPPED_WRITABLE(a, pid1)
        m2 ← IS_MAPPED_WRITABLE(a, pid2)
        if m1 or m2 then
            if m1 ≠ m2 then
                return true
            if MEMPAGECMP(a, pid1, pid2) ≠ 0 then
                return true
        a ← 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

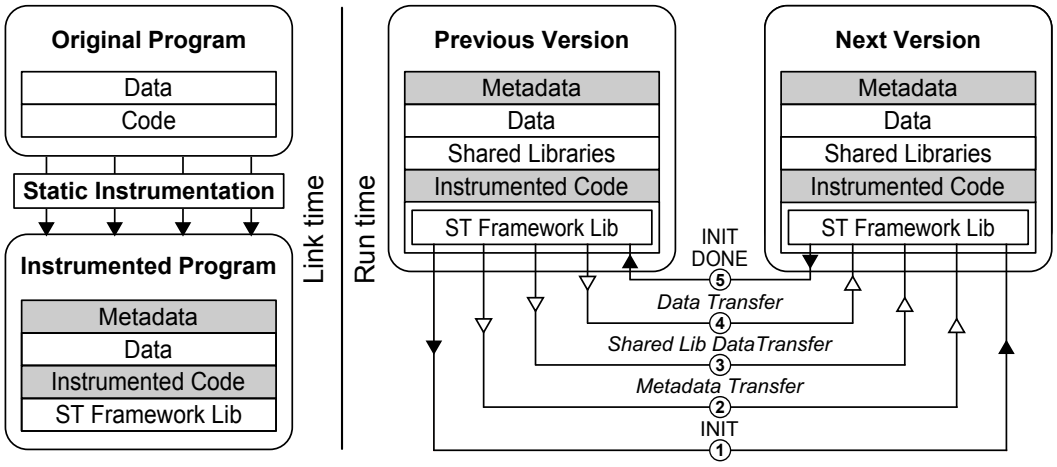

Figure 3: State transfer framework overview.

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-

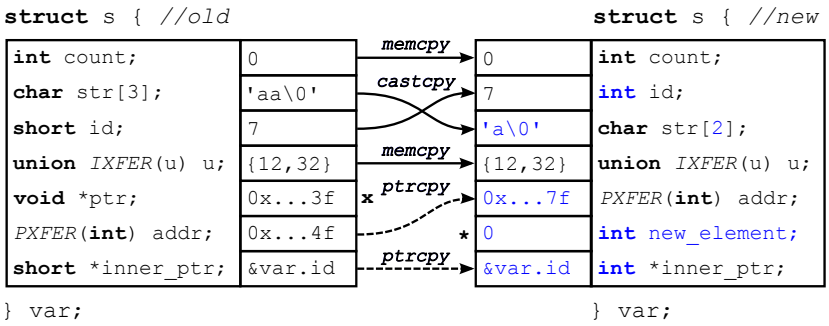

Figure 4: Automated state transfer example for the data structure presented in Listing 2.

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-

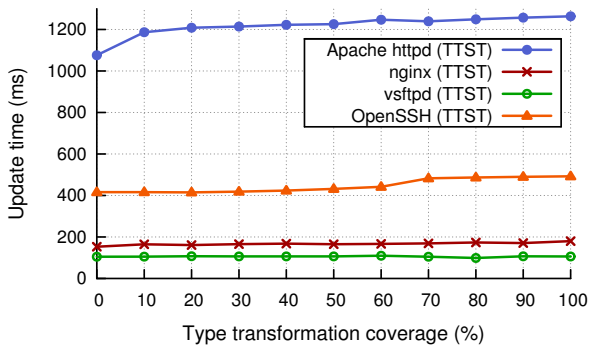

Figure 5: Update time vs. type transformation coverage.

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

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.*

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.

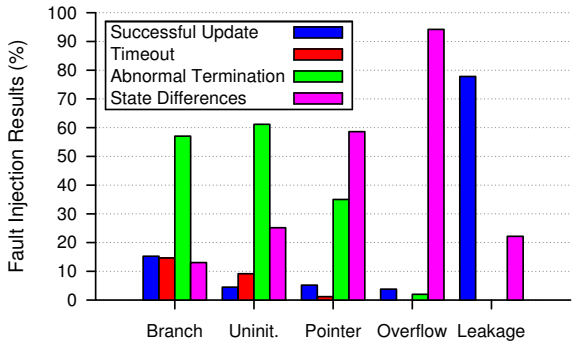

Figure 6: TTST *behavior in our automated fault injection experiments for varying fault types.*

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.