Evaluating Dynamic Software Update Safety using Systematic Testing

Christopher M. Hayden, Edward K. Smith, Eric A. Hardisty,
Michael Hicks, and Jeffrey S. Foster

***Abstract*—Dynamic software updating (DSU) systems patch programs on the fly without incurring downtime. To avoid failures due to the updating process itself, many DSU systems employ *timing restrictions*. However, timing restrictions are theoretically imperfect, and their practical effectiveness is an open question.**

This paper presents the first significant empirical evaluation of three popular timing restrictions: *activeness safety* (AS), which prevents updates to active functions; *con-freeness safety* (CFS), which only allows modifications to active functions when doing so is provably type-safe; and *manual identification* of the event-handling loops during which an update may occur.

We evaluated these timing restrictions using a series of DSU patches to three programs: OpenSSH, vsftpd, and ngIRCd. We systematically applied updates at each distinct update point reached during execution of a suite of system tests for these programs to determine which updates pass and which fail. We found that all three timing restrictions prevented most failures, but only manual identification allowed none. Further, although CFS and AS allowed many more update points, manual identification still supported updates with minimal delay. Finally, we found that manual identification required the least developer effort. Overall, we conclude that manual identification is most effective.

***Index Terms*—Dynamic software updating, DSU, hot-swapping, software reliability, testing, program tracing**

I. Introduction

Over the last 30+ years, researchers and practitioners have been exploring means to *dynamically update* the software of a running system with new code and data. Dynamic updates make it possible to fix bugs or to add features conveniently, without incurring downtime. Support for dynamic software updating (DSU) takes many forms. “Fix-and-continue” development, in which one incrementally develops and tests an application as it runs, has long been common for Smalltalk and CLOS, and Sun’s HotSwap VM [20], [9] and Microsoft’s .NET Visual Studio for C# and C++ [11] both include special support for it. The Erlang programming language [3], designed by Ericsson for building phone switches and other event-driven software, provides DSU primitives that are regularly used to hot-patch fielded systems. Research DSU systems for other languages such as C, C++, and Java [28], [7], [21], [32] have been able to dynamically update legacy server systems with patches corresponding to dozens of releases collected over several years. DSU adoption is also beginning to impact the end-user. Ksplice [4] can hot-patch the Linux kernel, and, as lucidly suggested by Bracha [6], network applications in the style of Google Documents naturally benefit from DSU. Even iPhone and Android applications support dynamic updates: When a user navigates away from a running app its state may be checkpointed; if the user navigates back to the app after upgrading it, the new version may restore and transform the checkpointed state so as to pick up where the old version left off.

All authors are with the University of Maryland, College Park.

A. Controlling update timing effectively

While DSU can significantly improve application availability, it is not without risk. Even if the new version of an application runs correctly when started from scratch, the application could behave incorrectly when patched on the fly, depending on *when* the update takes effect. To see why, consider the example in Figure 1, which shows two versions of a simplified HTTP server. There are two semantics-preserving changes in the new version. First, the escape function used to take a single argument, but now has been changed to take two arguments (and the call to it from parse is updated accordingly). Second, the global cnt, which counts the number commands processed, is now updated in get_file prior to logging, rather than in parse.

Suppose the old program is running and a *dynamic patch* (a patch to be applied at runtime) based

```
main() { ...
  while (1) { /**2**/
    byte* packet = network_read();
    struct event *e = parse(packet); /**3**/
    switch (e→ kind) {
    case GET: get_file(e→ gete.fname); break;
    case PUT: put_file(e→ pute.data); break;
  }
}
struct event* parse(byte* pkt) { /**1**/
   pkt = escape(pkt);
  ... cnt++;
}
void get_file (char* name) {
  log (..., cnt ,...);   ...
}
char* escape(char* buf) { ... }
```

(a) Old version

```
main() { ...
  /*


     as before



  */
}
struct event *parse(byte* pkt) {
  pkt = escape(pkt,METHOD_1);
   ...
}
void get_file (char* name) {
  cnt++; log (..., cnt ,...);   ...
}
char* escape(char* buf, int mode) { ... }
```

(b) New version

Fig. 1. Two versions of a program

on the new program is ready to take effect as control enters parse, at the point marked /**1**/. In many DSU systems, functions running at the time of an update continue executing the old code, while subsequent function calls invoke the new version [3], [28], [7], [32], [19]. Thus, we would have a type error: the old parse would call the new escape with a single parameter, instead of two parameters as expected, which could lead to surprising behavior.

To avoid these and other problems, most DSU systems support mechanisms that constrain when a dynamic patch may be applied. The goal of this paper is to evaluate the *effectiveness* of the most well-studied approaches to controlling update timing. The question of effectiveness is gaining in importance as DSU technology makes its way inexorably toward the mainstream. We characterize effectiveness as having three facets. The primary criterion is *safety*: an effective approach to controlling DSU timing should rule out incorrect behavior, such as the type error described above. The flip side is *availability*: timing cannot be restricted so much as to preclude a dynamic update for an extended period. Finally, there is *usability*: an effective approach will not require a developer to perform significantly more or difficult work to add DSU support to her program.

For our evaluation, we considered three approaches from among the most common and/or mature systems in the literature, and we evaluated their effectiveness on real programs undergoing dynamic updates that correspond to actual releases. The approaches are:

Activeness safety (AS): In this approach, an update may be performed only if those functions changed by the update are not *active*, i.e., if changed functions are not on the activation stack of a running thread. AS prevents the update at location /**1**/ in our example by forbidding the update from taking effect in parse since it has changed. AS is probably the most popular approach, used by the commercial DSU system Ksplice [4]; the research systems Dynamic ML [33], K42 [21], OPUS [1], and Jvolve [32]; and advocated by Bracha [6] for web-based end-user apps.

Con-freeness safety (CFS): Stoyle et al. [31] observed that AS may be overly restrictive, and proposed a condition called *con-freeness* that allows updates to active code if the old code that executes after the update will never access data or call a function whose type signature has changed. As such, it would rule also out the problematic update point /**1**/ in the example, since escape's type signature has changed and escape would be called after the update takes place in parse. Unlike AS, however, CFS would allow an update *after* the call to escape since subsequent actions in parse do not involve code or data whose type has changed, e.g., cnt is still a variable of type int. In general, it has been proved that AS and CFS both guarantee that no updating execution will exhibit a type error [31]. CFS is used by Ginseng, a research system developed by the authors that has successfully supported dozens of dynamic updates to realistic programs [28].

Manually identified update points: Several DSU systems, including Erlang [3], UpStare [24], POLUS [7], DLpop [19], DYMOS [22], and Ekiden [18] impose no automatic timing restrictions. Instead, these systems rely on the programmer to identify legal update points, and thus put her firmly in the driver's seat to balance safety and availability. A common approach, e.g., advocated by Armstrong for Erlang [3], which we call *manual identification*, is to permit updates only at the start or end of event processing loops (e.g., at position /**2**/ in the example). In fact, doing so would help avoid a problem that both AS and CFS allow. Consider if the example update were performed at /**3**/, which is permitted by AS and CFS because main is the only active function, and is unchanged. By this point the program has called the old version of parse, which runs the statement cnt++. After the update, the program will call the new version of get_file, which contains the statement cnt++ where the old version did not. Thus the execution has increased cnt one too many times, resulting in the call to log being incorrect. The manually chosen point at /**2**/ avoids this problem by ensuring calls to functions will always go to the same code version when processing a single command. Identification of the event-processing loops for adding update points is a straightforward, almost mechanical process. The developer just finds the event loops that correspond to places where updates should be supported and adds an update point.

B. Empirical assessment of timing effectiveness

Our evaluation of timing controls is *empirical*: we studied how these approaches would fare for real systems with real updates applied to them, derived from the systems' actual evolution. Our results are important because they provide quantitative evidence for assessing arguments that, to this point, have been essentially qualitative.

To perform our study, we considered dynamic updates to three mature, open-source applications: vsftpd, a popular FTP server, OpenSSH daemon, a secure shell server, and ngIRCd, an IRC server. The first two applications have already been studied by several DSU systems [28], [7], [24], while the third is new to this study. Each of these programs is single-threaded, although both OpenSSH and vsftpd use multiple processes. For each application we selected a streak of releases, and for each release (after the first) we constructed a dynamic patch using Ginseng, adjusting it as needed for the timing approach under study. For OpenSSH we chose eleven straight releases over a three year period; for vsftpd we chose nine releases over three years; and for ngIRCd we chose eight releases over eight months. Though we use Ginseng for this study, we argue (in Section II-D) that our results generalize to many other DSU systems, since most adopt similar models and mechanisms.

For each release and patch we executed a suite of system tests (either provided with the application or written by us) and observed whether a test passed when a dynamic update was applied during the test's execution. Each system test induces many *update tests*, with one update test for each distinct moment during the test's execution at which the update could be applied. By exhaustively running all update tests we can directly assess effectiveness. In particular, we can assess whether an approach to controlling timing would permit a particular update test (availability), and if so, whether that test passes (safety). We can also look holistically at the allowed update points to assess whether they occur often enough at execution time to provide availability. We observe that even a few syntactic update points may be sufficient in practice as long as they are executed suitably often.

Running a test for every possible update point would be prohibitively expensive. Fortunately, many update tests are provably redundant: Suppose a dynamic patch does not change the code of function f. Then a test with an update point just before a call to f will behave identically to the same test with an update point just after that call to f. Therefore, we only need to run one of these two possible update tests, rather than both possible tests. In the implementation of our systematic testing framework, which extends Ginseng, we built on this intuition to produce a *test minimization* algorithm that dramatically reduces the number of tests we have to run while retaining the same coverage [15]. For our experiments in particular, we found that 95% of the update tests from OpenSSH, 96% of points from vsftpd, and 87% of points from ngIRCd could be eliminated.

C. Results

The results of our study convince us that the approach of manual update point identification is the most effective. In particular, this approach eliminates all failures, provides sufficient availability, and is relatively easy to use. The two automatic mechanisms we considered do not preclude all failures and required substantial manual effort.

Assessing *usability*, all three methods required some manual effort to *extract* certain blocks of code into separate functions. In particular, each server program contained a potentially infinite main loop that processed client requests. Since updates to functions do not take effect until the next time the function is called, any updates to the loop-containing function would never be realized. To remedy this problem, we extract the body of the loop into a separate function, so that changes to the loop itself effectively take effect on the next loop iteration [28].

AS required several additional extractions to be effective. In particular, because it precludes updates to active functions, dynamic patches that contain an update to main (or any other function on the stack when the infinite loop is executing) will never be applied. As it turns out, without further change to the program, *every update to every program we considered would be disallowed by AS*. To avoid this problem, we extracted the bodies of functions up to the one containing the main loop so that the extracted parts are not on-stack at update time. This transformation does not affect semantics because extracted code portions are never executed again by the server.

CFS required additional work of a different sort. Because it relies on a static analysis, conservatism in the analysis may preclude updates to certain data structures or prohibit updates at certain program points even when they are safe. It sometimes took considerable effort to identify the root of such problems and work around them by refactoring the code in various ways.

The manual approach required the least additional work. Following the direction of Armstrong mentioned above [3], we simply prescribed that an update may take place at the beginning of each event processing loop (prior to calling the extracted body). Indeed, we needed to identify such positions to extract loop bodies, so adding the manual update point required no additional work.

As for *safety*, we find that both AS and CFS are highly effective at avoiding failures, though AS does this better than CFS, and neither is perfect. With no safety checking, many updates fail: in total, 1.87M of the total 14.2M tested executions failed (13%). Using either AS or CFS dramatically reduces the number of failures to about 495 for AS (0.003%) and 49K for CFS (.34%). For the manual approach, we observed no failures whatsoever.

As for *availability*, we found that both AS and CFS are fairly permissive, though CFS is more permissive than AS. In total, CFS permitted 68% of the passing update points, while AS permitted 59% of them, a difference of about 2.1M update tests; roughly 55% of passing update points are allowed by both. Thus, AS's lower failure rates come at the cost of higher restrictiveness, compared to CFS. The manual approach admitted the least number of update points: about 17.7K, or 0.14% of the passing update points.

While the more allowed update points the better, in general we only need updates to occur reasonably often. We measured the potential delay to updating that would be introduced by updating only at manual points using our test suite and benchmark programs. We found that while AS or CFS may allow an update to occur more quickly than manual points (since they permit more potential update points), this delay is typically quite short, usually less than 1ms for our tests (although the delay can be longer for certain requests, e.g., large file downloads). In cases where a developer decides that manual update points are reached too infrequently, she can allow faster updates by adding a manual point to the loops that cause the delay. We also categorized the update points in each program by the program phase they occur in—startup, connection loop, transition, command loop, or shutdown—and found that a significant number of the failures occur in the startup and transition phases, providing further support that update points in loops seem the most reliable.

In summary, this paper presents the first substantial study of several proposed DSU timing restrictions. While others have argued for [31], [28], [26], [4], [33], [21], [1], [32], [6] and against [24], [3], [7], [19], [22], [18] these approaches, these arguments have previously been qualitative. This paper is the first to empirically consider the safety, availability, and usability of these approaches when applied realistic applications. Our in-depth analysis of the data—including a characterization of the failures allowed and disallowed by the checks and where those failures tend to occur—provides a valuable source of information for judging and motivating future developments and research in DSU systems.

II. Dynamic software updating background

We used Ginseng for our empirical study, so this section describes it in detail. We chose Ginseng because it has proven to be quite effective; e.g., published work describes how Ginseng has been used to update six open-source server programs, where updates correspond to actual releases taken from several years' worth of development [28], [26], upwards of 60 dynamic updates in all. Moreover, Ginseng's updating semantics are quite similar to the semantics of many other DSU systems. As such, we believe that results for Ginseng have broad applicability.

The next three subsections describe how Ginseng works, first considering its basic mechanisms, then discussing how it handles updates to active code via extraction, and finally considering how it implements timing controls. We close this section by considering how results for Ginseng could be interpreted with respect to other DSU systems.

A. Ginseng implementation basics

In Ginseng, an update's effects are observed at function calls—following the application of a patch, subsequent function calls reach the function's most recent version. To implement this semantics, Ginseng compiles programs to use an extra level of indirection. In particular, all direct function calls are made indirect via an introduced global function pointer. When the Ginseng run-time system loads a dynamic patch—which among other things contains new and changed function definitions—it redirects these global pointers to the updated versions.

A dynamic patch also contains user-defined *transformation functions* used to update affected data. Global state is initialized or updated by a *state transformation function* that is executed after the new code is loaded and installed. For example, if a simple event counter is replaced with a more full-featured logging feature, the state transformer will contain code to initialize the log. Type-level conversions are effected by *type transformation* functions. For example, if the old program contained definition **struct** entry { **int** key; **void** *value } and the new version modified this definition to be **struct** entry { **int** key; **int** priority ; **void** *value }, the user provides a function that can initialize a value of the new version's type given a value of the old version's type, e.g., by copying the values from unchanged fields key and value, and initializing the new field priority. The program is compiled so that type transformers are invoked on demand: each access to data is prefaced by a check of whether the data is up-to-date, and if not, the representation is converted. The Ginseng compiler inserts padding in updateable values so that their representation can grow over time. See Neamtiu et al. for more details [28].

B. Updating active code in Ginseng

Functions that are active during an update will complete execution at the same version at which they were initially invoked. However, in some cases we might like to update an active function so that it transitions to its new version immediately. To see why, consider the processing loop in main in Figure 1. Suppose a subsequent version changes the loop body, e.g., to add additional operations to the **switch** statement. Once the patch is applied, these changes will take effect the next time main is called. But, because main will never be called again, the effects of updates to code in this long-running loop are delayed indefinitely.

To avoid this problem, Ginseng provides annotations that can be used to refactor a long-running function into several shorter-running ones. For example, to ensure that an update during the event processing loop will transition to the new version on the next loop iteration, the developer can add an annotation to the program that causes the compiler to extract the loop body into a separate function whose arguments include all the local variables mentioned in the loop body (passed by reference). Now that it is in a separate function, each subsequent call to the loop body will reach the new version. Likewise, any code that follows the loop could be made into a new function, allowing the new version to be reached once the refactored loop exits. The developer may similarly want to extract the "continuations" (i.e., code that would be executed at the old version upon return) of functions that could be on the stack when a desirable update point (such as this loop) is reached.

A drawback of using code extraction is that developers must anticipate which code to extract before deploying the program. In ours and others' experience, it is sufficient to extract each long-running event loop and the teardown code that immediately follows [28], [26], [7].

C. *Controlling timing in Ginseng*

Ginseng supports all three timing control mechanisms described in the introduction: *activeness*, *con-freeness*, and *manual*. In Ginseng, a program calls the function `DSU_update()` to ask the run-time system whether a dynamic update is available. We refer to such calls as *update points*. If an update is available and is compatible with the safety check in use (AS, CFS, or neither), it is applied at this point; otherwise, it is delayed until the next update point is reached. Thus, to implement the manual approach for our evaluation we simply inserted calls to `DSU_update()` at the desired program points and disabled additional safety checks.

To implement AS in Ginseng, the developer can specify activeness as the additional safety check; activeness is implemented by walking the stack to find the current active functions and ensuring they are not changed by the available update (note that Ginseng only supports single-threaded programs). To simulate asynchronous updates (i.e., those that could take effect at any time), the Ginseng compiler accepts an option that will insert update points automatically according to some policy, e.g., one prior to each non-system function call in the program.

Ginseng implements CFS as a static analysis. The Ginseng compiler analyzes the program source code to determine, for each update point,[1] those definitions that could be used *concretely* beyond that point (function calls, dereferences of global variables, field accesses of structured types, etc.) by the current function or any function that could be on the stack. Then it stores the set of names of those definitions in a data structure at that point. At run-time when control reaches that point and an update is available, the patch will be compared against the set: if definitions changed by the update appear in the set but have not changed their type signature then the update is permitted, and otherwise it is delayed. In effect, this check allows updates to active functions, but only if Ginseng can prove those functions will not subsequently call functions or access any data whose type signatures have changed.

[1]Update points could be inserted manually or automatically

D. *Other DSU systems*

Because we evaluate the effectiveness of various timing mechanisms using Ginseng, an important question is whether our results generalize. Here we argue that they do, and explain exactly how behavior similar to what we observe for Ginseng would manifest in the other systems, based on how they differ semantically from Ginseng.

It is easy to argue that our results generalize to the approaches used by Ksplice [4], Jvolve [32], K42 [21], DLpop [19], Dynamic ML [33] and Bracha [6]. In terms of updating semantics, the main difference between these systems and Ginseng is that Ginseng applies type transformations lazily rather than all at update-time, and so timing-related errors could manifest in Ginseng that would not manifest in the other systems. However, for our experiments all type transformers are pure functions, so the effects of type transformation would be the same if they were applied at update-time.

POLUS [7] and Erlang [3] employ a slightly different updating model than Ginseng: after an update in these systems, the programmer can partially control whether a function call should reach the newest version or the contemporaneous one. If the programmer were to specify that all calls are to the most recent version, the results would be the same as those for Ginseng given here. We note that the use of versioned function calls can encode the manual approach. For example, the programmer could avoid the problems that occur due to updates at `/**1**/` and `/**3**/` in Figure 1 by specifying all calls but those to the extracted loop body to be contemporaneous calls; this is essentially the approach recommended by Erlang. Our results confirm the effectiveness of this approach.

UpStare [24] is strictly more expressive than Ginseng in that it permits a dynamic patch to transform the execution state (i.e., the PC and stack) of the program. An UpStare patch developer provides a mapping between PC locations in each changed function's old and new versions and writes a function to initialize the stack of the new version based on the stack of the running version. At update time, if a changed function is active at a PC specified in the mapping, the transformation function is used to initialize the stack, and then execution proceeds at the new version's corresponding PC. UpStare's execution state transformations are akin to code extractions for Ginseng and similar systems: they are used to ensure that the correct new code is reached following an update. Thus the failures due to timing that we observe in our Ginseng-based experiments would correspond to failures using UpStare assuming the developer wrote the stack mapping in a way that corresponded with our loop extraction. Although UpStare supports changes that Ginseng cannot (e.g., changing the ordering of functions on the stack), the patches in our experiments did not require its extra expressiveness and so we believe that UpStare mappings would aim to achieve the same results for these programs as our code extractions. As a result, developers using UpStare could use the manual identification strategy we evaluate here by limiting the mapped points to those at event loops. Any additional points allowed by the developer's mappings may or may not correspond to the AS, CFS, or unrestricted approaches that we evaluate, depending on the developer's choices. Many of the failures we observed, particularly those allowed by AS, could also occur under UpStare, and reflect the hazards of constructing mappings for it.

UpStare also provides some support for AS-like timing restrictions [23]. Patch developers can specify *update constraints* that preclude updates when particular changed functions are active. The UpStare manual indicates that these constraints are useful to reduce the effort in mapping program states between versions. We believe that our findings apply directly to the use of these constraints.

III. Testing dynamic updates

To evaluate the effectiveness of DSU timing controls, we need to establish which program executions in which an update takes place can be deemed correct, and which cause misbehavior. For the purposes of our experiments, we do so using testing. While testing is an incomplete measure of correctness, tests typically cover the most important program behaviors, and provide an easy-to-measure, practical assessment of whether an updated execution is valid.

We begin by outlining the basic testing procedure. Next we present the intuition behind our minimization algorithm, which eliminates tests of update timings whose outcome is provably equal to the outcome of other tests. Finally, we present details of our testing framework's implementation. [2]

A. Testing procedure

Our approach to update testing is as follows. Let P_0 and P_1 be two program versions, and let π be a patch that updates P_0 to P_1. To dynamically test π, we must run P_0, apply π at the allowable update points, and then decide whether the ensuing behavior is acceptable. We do this by deriving *update tests*, one per allowable update point, from selected tests t in the system test suites of P_0 and P_1. Here, we use the term *update point* in a dynamic sense: each time the same call to DSU_update() is reached during execution we consider it a separate update point. Assuming we have a deterministic, single-threaded program, the update points for an execution can be numbered unambiguously. Thus, we define t^i_π to be the update test that executes P_0 on t and applies π at the i^{th} update point; if the test passes, then we deem π to be correct for point i. Since t should terminate, there will be a finite number of induced update tests t^i_π for a fixed π. To run update tests, we modify the Ginseng runtime to delay patch application to the i^{th} update point reached. Our implementation handles some forms of non-determinism and multi-process (but not multi-threaded) programs, which we describe in Section III-C.

We select the system tests t from which to derive update tests from the test suites of the old and new program version. Let T_i be a suite of system tests for P_i, for $i \in \{0, 1\}$. We use all $t \in (T_0 \cap T_1)$: since they should pass for both P_0 and P_1, we expect all t^i_π for all i should pass no matter when the update happens during the test execution.

On the other hand, we cannot generally use tests $t \in (T_1 - T_0)$, which consider functionality relevant only to the new version, or tests in $t \in (T_0 - T_1)$, which likely consider deprecated functionality. For these tests, not all update points will necessarily

[2]This testing procedure was described in a workshop paper [15]; our presentation here is for completeness, and for archival purposes.

make sense. For example, suppose P_0 is an FTP server, P_1 adds support for a new command qux, and t tests the proper functioning of qux, by logging into the server and then performing the command. For update tests t^i_π where update point i occurs prior to the login procedure finishing, then we can imagine the test will pass. This is because the login procedure has not changed between the two versions and should work identically in both. On the other hand, for update points j that occur after that point, applying the update will be too late: the old version, which does not support qux, will by that point have rejected the command and terminated the test. In general, tests in $(T_1 - T_0)$ may have some preamble during which a dynamic update is legitimate. Likewise, tests in $(T_0 - T_1)$ may have some legitimate post-amble during which an update could occur. Either way, we cannot identify this preamble automatically, so for simplicity we simply do not consider these tests. In future work, we plan to explore more powerful ways of adapting existing single-version tests to check DSU correctness.

B. Update test suite minimization

The procedure just described lets us systematically derive update tests from existing system tests. Unfortunately, we have found this procedure vastly multiplies the number of tests to run. For example, our experiments with roughly 100 system tests applied for 10 patches of OpenSSH yielded more than 8 million update tests. We mitigate this increase in test suite size by developing an algorithm that eliminates all provably redundant tests, sometimes yielding a dramatic reduction in test suite size.

To illustrate our algorithm, consider the following code, assuming that f, g, and h call no other functions:

```
1  void main() { DSU_update();
2                f ();
3                DSU_update();
4                g ();
5                DSU_update();
6                h ();   }
```

Suppose a dynamic patch π_1 to this program contains only a modification to function h. Then whether the update is applied at line 1, 3, or 5, the behavior of the program is the same: the calls to f and g will be to the old version, which is the same as the new version, and the call to h will be to the new version. Thus, for patch π_1, update points $\{1, 2, 3\}$ form an equivalence class, and we need only test one of the three to cover the whole class.

However, suppose dynamic patch π_2 modifies f, g, and h. In this case, none of the update points are equivalent. If we update at line 1, we will call the new versions of all three functions. If we update at line 3, we will call the old version of f and the new versions of g and h. If the update happens at line 5, we will call the old f and g and the new h. All of these executions may produce reasonable behavior, but we have to test them to find out.

We take the following approach to find equivalence classes of update points with respect to a given patch π . We instrument the program so that when it runs it produces an *update trace* ν of relevant events; among other things, the trace contains functions called, global variables read or written, and update points reached (but not taken). We run the instrumented program as part of some test t, but do not update it. The resulting trace ν_t contains some number n of update-point events, which in turn induce a set of update tests $t^1_\pi \ldots t^n_\pi$. Our goal is to determine which of these update tests produce *equivalent* traces for a given patch π. By equivalent, we mean that although they vary in the update point taken, they read and write the same values to and from the same variables, call the same functions with the same parameters, etc.—in other words, their behavior is identical except for update timing. Then we can run a single representative test from each equivalence class while retaining full update coverage.

For each event in the trace, we determine whether it *conflicts* with patch π. In particular, if the event is a call, read, or write to F (where F is a function, global variable, or a value of named type) then the event conflicts with π if and only if F is changed by the patch. If F is a function, any change to its text constitutes a change to F. (Note that a change to a function called by F does not render F itself changed, by this definition). If F is a global variable, then either a change to its nominal type or a modification of its contents during state transformation constitutes a change. Finally, if F is a named type, then a modification of F's definition (e.g., changing a struct's set of fields or their nominal types, or changing the nominal type that underlies a typedef) constitutes a change. If there is no conflict, then the update π could be applied before or after the event and the semantics of the overall program trace would be the same. This makes intuitive sense: if we call a function G, but the patch does not change G, then whether we apply the update before or after calling G makes no difference; we will execute the same code for G. [3] On the other hand, if we did update G, then applying the update before the call will result in calling the new G, whereas applying the update after the call will result in calling the old G.

We compute the set S of update points to consider as follows. We start with the empty set S and analyze the trace. In addition, we maintain the index i of the most recently reached update point. When analyzing the trace, if we reach a conflicting event e we add i to our set of update points to test, since the semantics of e could change if the update happens before it. On the other hand, if we reach another update point $i+1$ without having found a conflicting event for update point i, then we merely update the index to $i+1$; thus we have determined that i need not be tested. The reason should be clear: none of the events between update points i and $i+1$ conflict with the patch, so applying the update at i would be equivalent to applying it at $i+1$.

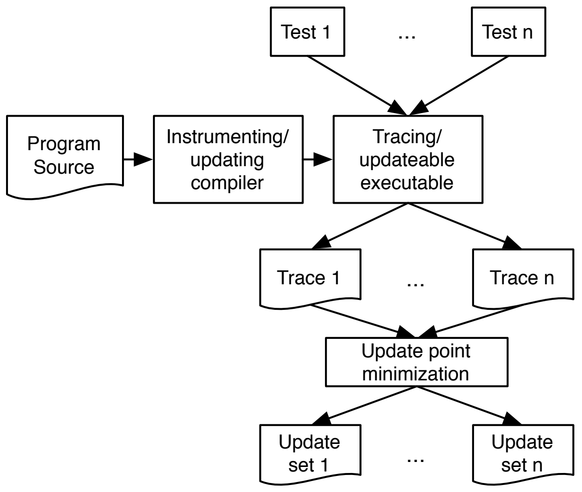
Let us reconsider the example at the start of this subsection. Running the program will produce the following trace:

$$\nu = \mathit{update}_1; \mathit{call}(f); \mathit{update}_2; \mathit{call}(g); \mathit{update}_3; \mathit{call}(h)$$

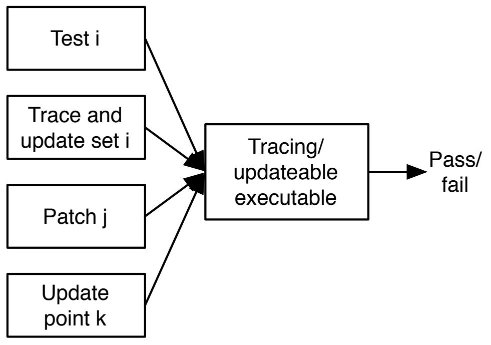
Consider patch π_1 in which only f is changed. Then the outcome of our minimization algorithm will be the set $S = \{1\}$: only the first update point needs to be tested. On the other hand, patch π_2 changed all three functions, so all three calls conflict, and thus each update point would be added to S, resulting in $S = \{1, 2, 3\}$.

We have formalized this minimization algorithm and proven it correct [15], [16]. In practice, the reductions for the three benchmark programs we assess in Section V were substantial: 95% of update points from OpenSSH, 96% of points from vsftpd, and 87% of points from ngIRCd could be eliminated as redundant. The absolute reduction in update tests was also significant: the initial number of update tests was very large, with over 8M for OpenSSH, 3.9M for vsftpd, and 2.2M for ngIRCd. Running the reduced test suite was time consuming, and would have been prohibitive without reduction. For example, testing OpenSSH with the minimized test suite still required approximately 600 CPU hours to complete. Extensive experimental results assessing the effectiveness of the technique for our benchmark programs are given in a technical report [16].

[3]Note that if G itself called some function that would be affected by the update, then this event will also appear in the trace subsequent to the call for G, and it would serve as the source of a conflict.


Test 1
...
Test n
Program Source
Instrumenting/ updating compiler
Tracing/ updateable executable
Trace 1
...
Trace n
Update point minimization
Update set 1
...
Update set n

(a) Instrumentation and trace gathering



(b) Running a test case

Fig. 2. DSU testing framework architecture

C. Implementation

We extended Ginseng to implement our testing framework. Our extended implementation, called DSUTest, works in two phases, illustrated in Figure 2(a) and (b), respectively. In the first phase, the DSUTest compiler instruments the program to log relevant events to a trace file, and then processes each file to find the minimal set of update points to test. In the second phase, the instrumented program replays a given test once per update point identified during the test's minimization, and tabulates the results.

The implementation was largely straightforward, except for two wrinkles: handling programs that fork child processes that themselves must be updated, and coping with non-determinism that arises during tracing.

Handling multiple processes: So far, we have assumed we could identify an update point by its position in the trace. However, this approach does not accommodate server programs that fork independent subprocesses that could themselves be updated. Even when forked processes do not communicate with each other in an interesting way, their logging output will be interleaved in the shared log file, and the particular interleaving can vary from run to run.

To compensate, we include the current process number when logging events, and count update points relative to a particular process. Since OS-supplied process identifiers vary between runs, we use our own process numbering scheme, being careful to deterministically choose numbers that are unique among related processes. We log the parent and child at each fork, and when we minimize a child process's trace, we may equate some of its initial update points with the parent's update point before the fork in the absence of intervening conflicting events in the child.

Non-determinism: Our basic methodology presumes that tests are deterministic. However, most programs, including our benchmark servers, exhibit some non-determinism, and thus different runs of the same test may produce slightly different traces. We have encountered non-determinism arising from three main causes. The first is I/O handling by the OS. The main connection loops of our servers block until they receive a command on a socket, carry out the appropriate behavior, and then continue with the loop. Sometimes the server can wake unpredictably though no I/O is available. In this case, the server "stutter steps" back to the top of the loop, but in doing so may call functions or access data, affecting the trace. Second, the exact timing of any signal handlers can vary between runs. Thus, trace events that occur within a signal handler could be spliced into a trace at different positions in different runs. Finally, some common functionality depends on the environment, such as the current system time, random numbers, and (for vsftpd) process IDs and memory addresses used as hash keys.

To keep update tests consistent with the initial trace, we check that each update test trace matches the original trace up to the chosen update point, and replay it if not. However, this approach fails to converge in the presence of highly non-deterministic events, e.g., the timing of signal handling and, in some cases, the occurrence of loop stutter steps. To compensate, we designate *ignore regions* of code in which the test trace need not match the original and within which updates are not tested. We still note accesses to changed code and data within ignore regions to ensure that update points separated by a region are not erroneously equated.

For the programs in our experiments, we found that it was usually straightforward to designate the code to include in ignore regions. The process entailed comparing several traces produced by executions of a system test. We found that the traces would largely match, except in a few places, as mentioned above. We would then look at the source code that produced the non-determinism and decide whether enclosing the code in an ignore region might mask interesting update behavior. In some cases we would add an ignore region; in others, we elected to leave in the non-determinism and rely on match-checking/replay to produce consistent executions. In some cases, several rounds of experimentation were required to get the ignore regions right. To be sure that our experiments are meaningful, we took pains to minimize the size and use of these regions.

Note that we currently limit our focus to single-threaded programs, making no attempt to account for non-determinism that would arise from thread scheduling. In future work, we may explore integrating our framework with techniques for systematically testing under different thread schedules [25], [29] to handle multi-threading.

IV. Experimental setup

Using our testing framework, we set out to empirically evaluate the effectiveness of DSU timing restrictions. Our chief goal in designing these experiments was to extensively test a large variety of patches and program functionality to ensure that our results will apply generally. This section describes our experimental setup: which applications we considered, which test suites we used, and how we ran the tests and gathered the data.

A. Test applications

We tested updates to three long-running server applications: OpenSSH, a widely used SSH server; vsftpd, a popular FTP server; and ngIRCd, an IRC server. Figure 3 summarizes the versions of each application that we consider. We largely re-use the OpenSSH and vsftpd dynamic patches used by Neamtiu et al. in their Ginseng work [28], with some changes that we describe in the next section. The OpenSSH releases range from Oct. 2002 to Sept. 2005, and the vsftpd releases range from July 2004 to Feb. 2008. We also developed patches for seven ngIRCd releases that range from Sept. 2002 to May 2003.

The patches to OpenSSH vary considerably in scope, from bug-fix–only releases (3.6.1p2, 3.8.1p1, 4.0p1) to ones that add significant functionality. Examples of added features include: new ciphers (3.7.1p1, 4.2p1), limits to the number of failed authentication attempts (3.9p1), and advance warning of account/password expiration (4.0p1). Many bugs were fixed over this stretch including memory leaks (3.7.1p1, 3.8p1) and buffer management errors (3.7.1p1). The Ginseng OpenSSH patches include *state and type transformation code* (see below) to add data for new features to tables of configuration options, ciphers, and command dispatch. Transformation code is also used to account for changes to implementation details, e.g., copying over the values of global integers that were moved into a global struct (2.6.1p2).

The patches to vsftpd also introduce many new features, which include: terminating a session after too many failed logins (2.0.5), locking of files being uploaded (2.0.4), and receiving connection options (OPTS) prior to login (2.0.6). These patches also contained a variety of bug-fixes, such as: corrected handling of * (match anything) in commands (2.0.4) and not sending duplicate responses to the "store unique" (STOU) command (2.0.6). State transformation for the vsftpd Ginseng patches required initializing fields added to the structure representing a session with a connected user and, as with OpenSSH, initialization of global tables of configuration operations.

Likewise, the patches to ngIRCd added new features, such as support for IRC commands, *TIME* to display the server time (0.6.0) and *HELP* to list available commands (0.7.0), as well as new configuration options, e.g., a configurable limit to the number of active connections (0.6.0). These patches also fixed bugs, including buffer overflows (0.5.2), format string errors (0.5.2), and attempts to write on a closed socket (0.5.1). The dynamic patches that we constructed performed transformations like adjusting the lengths of buffers (0.5.2) and modifying the C representation used to hold information about active connections (0.6.0).

To make it easy to refer to the versions in the subsequent discussion, we number them starting from 0. For each version, Figure 3 lists the total lines of code (measured with SLOCCount [34]), the number of update tests (described below), and the number of function signature changes, function body changes, and named type changes (structs, unions and typedefs), that are required to update to the next version. We provide the latter data as it is useful to help explain some of the failures we found, described in the next section.

B. Test suites

To perform our testing experiments, we required test suites for each program's core functionality in order to generate update tests. We wrote test suites for vsftpd and ngIRCd that cover all supported client operations, and reused the set of system tests distributed with OpenSSH. Each of the test suites exercise core program features and were developed independently of our evaluation.

We constructed update tests for OpenSSH from the suite of system tests that are distributed with OpenSSH's source code. Tests launch a server and communicate with it via an ssh client, exercising various connection parameters and/or executing remote commands, and judging success/failure on return codes and command output. We found that all supplied tests for version n also pass for version $n+1$. Thus, we used the full suite of version n's server tests to develop update tests for the patch to version $n+1$.

We made two minor changes to OpenSSH's test suite for efficiency. First, we reduced the timeout period of the *login-timeout* test, which tests that a server terminates its connection if a client takes too long to log in. Second, we split large tests with orthogonal components (e.g., the *try-ciphers* test)

				Tests			Δ to next ver		
	#	Version	LoC	Ct.	Line Cov. %	Func. Cov. %	Sig	Fun	Type
OpenSSH	0	3.5p1	46,735	75	46.1	61.4	3	98	5
	1	3.6.1p1	48,459	75	46.6	62.0	0	6	0
	2	3.6.1p2	48,473	76	46.4	61.4	5	238	11
	3	3.7.1p1	50,448	91	46.2	61.8	0	18	0
	4	3.7.1p2	50,460	91	46.3	61.8	13	172	10
	5	3.8p1	51,822	104	44.4	59.3	0	24	1
	6	3.8.1p1	51,838	104	44.4	59.4	6	257	10
	7	3.9p1	53,260	104	44.8	59.3	4	179	12
	8	4.0p1	56,068	105	44.5	59.9	0	72	3
	9	4.1p1	56,104	104	44.4	60.1	10	157	7
	10	4.2p1	57,294	(Not patched)					
vsftpd	0	2.0.0	13,048	27	61.1	75.5	0	6	0
	1	2.0.1	13,059	27	60.8	74.8	1	12	0
	2	2.0.2pre2	13,114	27	60.7	74.7	0	21	0
	3	2.0.2pre3	14,293	27	59.4	74.3	0	76	0
	4	2.0.2	16,970	27	60.8	74.7	0	10	1
	5	2.0.3	12,977	27	60.9	74.6	0	25	1
	6	2.0.4	14,427	27	60.5	74.5	0	100	2
	7	2.0.5	14,482	27	60.7	74.5	0	93	2
	8	2.0.6	14,785	(Not patched)					
ngircd	0	0.5.0	8,157	34	60.5	82.2	0	6	0
	1	0.5.1	8,160	34	60.5	82.2	0	23	1
	2	0.5.2	8,161	34	60.4	82.2	12	28	2
	3	0.5.3	8,178	34	60.6	82.2	1	17	2
	4	0.5.4	8,211	34	56.4	75.6	4	104	8
	5	0.6.0	9,302	34	56.0	75.6	0	24	0
	6	0.6.1	9,333	34	53.3	72.3	2	79	4
	7	0.7.0	10,043	(Not patched)					

Fig. 3. Version, patch, and test information

into many smaller tests, to reduce total testing time and permit parallel testing.

As vsftpd is not distributed with any system tests, we constructed 27 tests for core FTP operations, including connecting, uploading, and downloading files in binary and ASCII formats, and navigating remote FTP directories. These tests apply to all versions of the server, and exercise all of the FTP operations supported by version 2.0.0 of vsftpd.

We also developed a suite of 34 ngIRCd tests, exercising functionality including connecting, sending and receiving chat messages, joining and communicating through IRC channels, and querying the server for information such as the set of connected users and available channels. These tests exercise all operations that a client can perform when connected to version 0.5.0 of ngIRCd. All tests in this suite apply to all tested versions of ngIRCd.

Figure 3 shows the single-version line and function coverage information for each tested patch. Line coverage was in the mid-40% range for OpenSSH, and at around 60% for most versions of vsftpd and ngIRCd. Function coverage was in the low-60% range for OpenSSH, in the mid-70% range for vsftpd, and varied from the low-70% range to the low-80% range for ngIRCd. While these figures indicate that some functionality was not tested (e.g., the SSL capabilities of vsftpd, server-to-server connections in ngIRCd, and error handling code generally), our test suites exercise a large number of distinct operations. Even if our test suite does not exhibit every possible DSU timing error for these patches, each set of update tests induced by a system test provides a large set of common update points with which we compare the three timing restrictions. In total, across a variety of real-world patches and tests, we believe our results provide an extensive and realistic corpus of update points for comparison (over 14M in all). Since our goal is to uncover any errors that occur, the test scripts are written to check the correctness of as much of the external behavior as possible, including return codes, response messages, and other effects like downloaded files. Of course, more "blunt" criteria were also used, like ensuring that the program did not crash.

C. Running tests and tabulating results

As mentioned earlier, updates can take effect at calls to DSU_update(), where these calls can be inserted manually or automatically. For our tests, we directed DSUTest to automatically insert a call to DSU_update() prior to each function call, and systematically tested the outcome of performing an update at each of these points, with all safety checks disabled. We refer to this set of dynamic update points as *All Pts*. We used our test minimization algorithm (Section III-B) to determine which update tests should actually be performed and then scaled the results back up to the full set of points. For each test execution, we recorded whether the test passed or failed. We marked a run as failing if either the system test itself reports a failure, if the server unexpectedly terminates during the test, or if the test times out. We set the timeout for each run as the time required to gather the initial trace plus 10 seconds.

Having determined the effects of updating at all possible points, we can assess the availability and safety of the three timing control mechanisms by considering which update tests would have been permitted by each restriction.

V. Experimental results

This section presents the results of our empirical evaluation of the AS and CFS safety checks and manual update point identification. Our experiments seek to evaluate the effectiveness of these timing restrictions in terms of their usability (by considering the manual effort required to use them), safety (by judging their ability to prevent incorrect behavior), and availability (by ensuring that updates are allowed sufficiently often).

A. Usability

All three methods for controlling timing required some manual changes to the applications. These fall into two categories: update point selection, and code refactoring to ensure desired update semantics and availability.

For both AS and CFS, no programmer effort is required to select update points; these are inserted automatically. For the manual approach, we followed the recommended pattern of placing them at the outset of long-running event processing loops [3], [28], [19]. Note that, while we have referred to such update-point placement as "manually identified," it may be possible to automate parts of this relatively systematic procedure. Nevertheless, human judgment is probably necessary to distinguish event handling loops from other loops and to account for the program's update availability requirements. When preparing vsftpd and OpenSSH to support updating, Neamtiu et al. chose to place a single DSU_update() at the beginning of the loop that accepts new connections [28]. We placed an additional update point in each per-session command loop of the applications—some patches we consider add new command handling, and we wanted to allow those to be updated during an active session. OpenSSH provides two distinct command loops to handle different ssh protocol versions, while vsftpd uses only one; so vsftpd contained a total of two calls to DSU_update(), while OpenSSH had three. For ngIRCd there is only one event processing loop, so we placed a single point at its beginning. Following this pattern was quite straightforward: the only work involved was identifying the main loops.

As mentioned in Section II-B we must manually extract each connection/command loop and its cleanup code into separate functions so that each connection loop iteration executes the most recent code and cleans up the server state appropriately when it exits. This task is required for all three timing mechanisms, since in Ginseng (and indeed in nearly all other DSU systems) updates take effect at function calls. (UpStare would not require this effort at the outset, but as discussed in Section II-D, the programmer would have to do something similar when writing her dynamic patch so as to properly map between versions' execution contexts.)

AS required some additional manual effort. In particular, after some preliminary testing, we discovered a significant problem with the AS check. Recall that AS forbids updates to functions that are on the stack. It turns out that *this restriction forbids all updates from being applied to* OpenSSH, vsftpd, *and* ngIRCd, because each update included changes to main, which is always on the stack. Even excluding main, we found that AS very often forbids updates within the command loop. Schematically, the command loop is reached through a chain of function calls, starting from main, that look like the following:

```
void f() {
  ...         // startup code
  g();        // call next function, ultimately reaching
              //     the function containing the main loop
}
```

In many cases updates change the "startup" code in the functions in this chain (i.e., the code before the call to g() in the schematic), and thus AS would prevent those updates from being applied during the command loop. However, patches can be written to contain state transformation code to execute relevant changes to the startup code that would have been executed if the program were started from scratch. Therefore, we can (and did) reasonably relax the AS check by also extracting the startup code, so that it is no longer on the stack when the loop executes.[4]

CFS required additional effort as well, but of a different sort. To implement CFS, Ginseng uses a static analysis. Unfortunately, this analysis is conservative, and so it can overestimate the definitions that can be accessed concretely following an update point, spuriously preventing updates that are actually safe to allow. This problem can be overcome with some refactoring. For our experiments, the analysis over-approximated the set of possible calls through a table of function pointers, and as such spuriously forbids updates within the OpenSSH command loops. Therefore we performed some additional code extractions so that updates within the command loop would pass the CFS check.

Examining these costs in terms of code changed and programmer effort, the manual approach comes out on top. In particular, while the programmer must identify manual update points, these exactly coincide with the positions at which loops must be extracted, a task required by all three approaches. As such, the additional work required by AS and CFS makes those approaches a bit more expensive, especially since they require some amount of testing or interaction with the tool to figure out why certain updates are not being permitted.

[4]In actual fact, we opted to leave the code as-is and simulate the extraction: When we post-process the *All Pts* data set to determine which updates would be allowed by AS, we permit updates within the command loop even if they modify startup code in the functions leading up to the loop.

B. Update Safety

Figure 4 summarizes the number of update points allowed under each timing restriction for each patch to OpenSSH, vsftpd, and ngIRCd, and how many of those points resulted in a failing test.

The *All Pts* column of Figure 4 lists over 1.4M failing update points out of 8M total (17.8%) for OpenSSH, over 128K failing runs out of 3.9M total (3.2%) for vsftpd, and over 308K failing runs out of nearly 2.2M total (13.9%) for ngIRCd. This is clear evidence that applying updates indiscriminately is extremely risky, and thus timing restrictions are necessary.

The *CFS*, *AS*, and *Manual* columns of Figure 4 illustrate that all three timing restrictions disallow the vast majority of failing updates; however both automatic safety checks permit some unsafe updates. For all three programs, CFS allows the most failures, but manages to reduce the total number of failures from 1.4M to 46.8K (96.7% reduction) for OpenSSH, 128K to 2.2K (98.3% reduction) for vsftpd, and 308K to 98 (over 99.9% reduction) for ngIRCd. AS performed even better, allowing only 495 failures (well over 99.9% reduction) for OpenSSH and no failures for vsftpdand ngIRCd. Significantly, only *Manual* identification of update points exhibited no test failures.

Looking at the data we can make several high-level observations about the relationship between the patches and their failures. Comparing program versions, we see that updates containing few changes typically induce few failures. One particularly striking observation is that patches containing no type or function signature changes (OpenSSH patches 1→2 and 3→4, vsftpd patches 0→1, 2→3, and 3→4, and ngIRCd patches 0→1 and 5→6) exhibited almost no failures (ngIRCdpatch 5→6 exhibited 3 failures). Since both AS and CFS ensure updates are type-safe, it seems likely that a large portion of the failures are due to type errors. We manually examined several of the failures reported in *All Pts* and found type safety violations to be the most common cause. We also note that patches containing relatively few overall changes had fewer failures, while the largest updates, such as OpenSSH patches 2→3, 4→5, and 9→10, generally resulted in more failures. There are notable exceptions to this general trend, such as vsftpd patch 4→5, which contained few changes but resulted in the

	Update	All Pts		CFS		AS		Manual	
		Total	Failed	Total	Failed	Total	Failed	Total	Failed
OpenSSH	0→1	580,871	19,715	68,044	0	35,314	0	566	0
	1→2	705,322	0	705,322	0	587,578	0	630	0
	2→3	638,720	306,965	75,307	1,688	20,902	4	568	0
	3→4	772,198	0	772,198	0	638,803	0	783	0
	4→5	773,086	565,681	110,633	609	21,343	380	782	0
	5→6	878,235	10,703	130,000	0	111,950	0	860	0
	6→7	879,668	163,333	96,183	44,461	44,278	110	859	0
	7→8	918,717	11,380	80,070	1	100,854	1	850	0
	8→9	973,364	3	261,885	0	61,724	0	868	0
	9→10	933,514	357,919	121,337	24	61,051	0	833	0
	Total	**8,053,695**	**1,435,699**	**2,420,979**	**46,783**	**1,683,797**	**495**	**7,599**	**0**
vsftpd	0→1	437,910	0	437,910	0	209,441	0	154	0
	1→2	439,983	2,993	198,277	726	186,769	0	154	0
	2→3	470,494	0	470,494	0	179,726	0	155	0
	3→4	507,071	0	507,071	0	91,993	0	157	0
	4→5	486,927	119,922	19,297	1,468	6,365	0	155	0
	5→6	511,032	893	65,999	0	215,557	0	155	0
	6→7	529,845	1,270	29,339	0	27,020	0	155	0
	7→8	549,380	3,246	5,010	0	14,880	0	155	0
	Total	**3,932,642**	**128,324**	**1,733,397**	**2,194**	**931,751**	**0**	**1,240**	**0**
ngIRCd	0→1	291,331	0	291,331	0	152,830	0	372	0
	1→2	289,558	0	286,310	0	167,372	0	370	0
	2→3	289,650	204	2,007	0	443	0	375	0
	3→4	289,900	1,086	2,008	0	444	0	376	0
	4→5	281,684	138,105	1,987	95	328	0	260	0
	5→6	392,219	3	392,219	3	11,711	0	384	0
	6→7	392,309	169,064	860	0	452	0	384	0
	Total	**2,226,651**	**308,462**	**976,722**	**98**	**333,580**	**0**	**2,521**	**0**

Fig. 4. Points allowed/test failures

most vsftpd failures.

We investigate the causes of the failures that AS and CFS allow in Section V-D. The relationship between failures (and successes) allowed by both checks is tabulated in a technical report [17].

C. Update Availability

The most straightforward way to assess update availability to is measure which timing restrictions permit the most update points. Returning to Figure 4 we see that both AS and CFS allow many update points, though CFS is more permissive than AS. Both CFS and AS allow several orders of magnitude more update points than are allowed under Manual update point identification. When we consider only the passing update points, as shown in the right half of Figure **??**, the trend continues: In total, CFS permitted 68% of the passing update points, while AS permitted 59% of them, a difference of about 2.1M update tests; roughly 55% of passing update points are allowed by both. The manual approach admitted the least number of update points: about 17.7K, or 0.14% of the passing update points. Thus, across these three approaches to timing restriction, we observe that the lower failure rates of manual point identification (and to a lesser extent AS) come at the cost of fewer correct update points allowed.

Generally speaking, while allowing more correct update points is better than fewer, it also matters where those update points occur during program execution. In particular, since the majority of each server's execution takes place within one of a few long-running loops, it is crucial that a safe update point is reached on almost every iteration of these loops. Otherwise, we may be unable to update a program in a timely fashion. Assuming loops complete reasonably quickly, and the time transitioning between loops is also quick, just updating in loops may well be sufficient.

To get a more concrete idea how frequently updates would be permitted using the manual identification strategy, we timed how long server processing took during each iteration of the main loops for our subject programs (the first version of each) throughout execution of our test suites. This provides an indication how long an update might be delayed by server processing. For all three programs, we found that most loop iterations required less than 1ms to complete. For both vsftpd and ngIRCd, the longest loop iteration required less than 10ms (for vsftpd, this was the time required to download a 900kB file locally). The longest overall delays were OpenSSH tests that performed a sleep operation for 3 seconds on the server. Overall, we believe the delays to updating that we observed would be unlikely to make any difference in server operation. However, it is not difficult to imagine less trivial delays, e.g., large downloads by a remote client could delay an update to vsftpd for much longer. However, in this example, it is doubtful that updated functionality would be needed during a download—and if it is, the developer might choose to add a manual update point to the download loop. Based on this investigation, we believe that the delay due to manual update point identification will usually be inconsequential. When developers judge the delay to be significant, we suspect that it can often be ameliorated by adding manual update points at the long-running loops that cause the delay.

D. Failure examples

To help understand better where the automated checks fall short, we investigated several of the failures that are still allowed by the CFS and AS checks.

Failures allowed by CFS: The property that distinguishes CFS is that it will execute code that is active at the time of update at the old version, provided this execution will not violate type safety. However, as we mentioned in Section sec:intro, type-safe executions may nevertheless fail, and indeed we observed cases of this. We have found that sometimes executing a function (or part of it) at the old version and then executing a related function at the new version may induce a failure when the relationship between these two functions changes between versions. We generically refer to these problems as *version consistency errors* [27], since they involve executing the old version of some function and then executing the new version of another where there is a relationship between the two.

One example occurred while testing upload operations against the 1→2 patch to vsftpd. Figure 5 shows a simplified version of the relevant code. In this patch, the code that sends the FTP return code 226 indicating a successful transfer was moved from do_file_recv to handle_upload_common. If an update occurs after entering handle_upload_common,

```
void
handle_upload_common() {
  DSU_update();
  ret = do_file_recv ();
}
void do_file_recv() {
  ... // receive file
  if (ret == SUCCESS)
     write(226, "OK.");
  return ret;
}
```

(a) Version 1

```
void
handle_upload_common() {
  DSU_update();
  ret = do_file_recv ();
  if (ret == SUCCESS)
     write(226, "OK.");
}
void do_file_recv() {
  ... // receive file
  return ret;
}
```

(b) Version 2

Fig. 5. Skipped return code

```
void maincont() {
  DSU_update();
  serverloop2();
}
void serverloop2() {
  global_ptr = init ;
  tmp = (*global_ptr).pw;
}
```

(a) Version 4

```
void maincont() {
  global_ptr = init ;
  DSU_update();
  serverloop2();
}
void serverloop2() {
  tmp = (*global_ptr).pw;
}
```

(b) Version 5

```
void maincont() {
  extracted ();
  DSU_update();
  serverloop2();
}
void extracted() {
}
void serverloop2() {
  global_ptr = init ;
  tmp = (*global_ptr).pw;
}
```

(c) Ver. 4, after extraction

```
void maincont() {
  extracted ();
  DSU_update();
  serverloop2();
}
void extracted() {
  global_ptr = init ;
}
void serverloop2() {
  tmp = (*global_ptr).pw;
}
```

(d) Ver. 5, after extraction

Fig. 6. Skipped initialization error

but before calling do_file_recv, then the new version of do_file_recv executes and then returns to the old version of handle_upload_common—and thus the server will never write the return code. Eventually this causes the transfer to time out and fail. Though the code executed following the update in handle_upload_common is changed by the update, the execution is allowed by CFS as the function signatures have not changed. On the other hand, AS precludes the update (and thus, its failure) because handle_upload_common is active.

Failures allowed by CFS and AS: While AS prevents the version-consistency failure we just saw, it does not prevent such problems entirely. A particularly interesting example occurs in the

4→5 patch of OpenSSH. This example involves a problem that was not present in the original code, but was introduced via a code extraction step that is needed to permit many other, safe updates to occur.

Figures 6(a) and (b) show a highly simplified version of the relevant code for both versions. In version 4, a global pointer is initialized in the serverloop2 function, prior to entry into the command loop.

Version 5 moves this initialization earlier into maincont (a function we added during code extraction), prior to calling serverloop2. (In the actual code, the call to serverloop2 is further down the call chain.)

CFS will always allow this update to be applied, because it involves no type changes, and hence is type-safe. However, if the update indicated in Figure 6(a) is taken, then global_ptr will be uninitialized when dereferenced, leading to a crash. On the other hand, AS should prevent this update, because maincont is changed by the update and is active at the update point.

However, recall from Section IV that we extracted the "startup" code in all functions leading up to the command loops in our subject programs. Consider Figures 6(c) and (d), which show the two versions of the program after code extraction. Notice that the initialization of global_ptr is moved from serverloop2 to extracted. Thus, the update no longer changes maincont, and when the indicated update point is triggered in our experiments, AS actually allows the update. This example illustrates the tension between update availability and safety when applying AS, and cases like these show the fragility of automatic update safety checks.

In general, AS is also unable to prevent any version consistency problems where the old version of code involved is executed to completion and so is no longer on the stack. We observed a set of failures where this occurs in OpenSSH patch 2→3. This patch included a change to the format of a packet sent from the server to the client and then later sent back to the server. Version 2 included only a sequence number in the packet, while version 3 adds a count of blocks and packets. This change is manifested through a modification to two functions: mm_send_keystate and mm_get_keystate.

If an update occurs after a call to mm_send_keystate but before a call to mm_get_keystate, then the new version of mm_get_keystate is invoked and is unable to parse a packet generated by the old code version, causing a test failure.

These update points are allowed by CFS, which determines that the update cannot violate type safety. AS will also allow these failures as this version consistency error can occur at points when neither changed function is on the call stack. Typically, state transformation can be used to ensure that program state is updated to work with new code, but in this case the state of the packet is stored on the client, where it cannot easily be changed when the server is updated.

It is unlikely that an automatic check could effectively avoid failures such as these, since they are quite specific to the application (and, in the general case, the problem is undecidable [14]). On the other hand, the manual effort required to avoid these errors by placing a few update points in the program seems quite manageable.

VI. LIMITATIONS

Our study found that manual update point identification maximizes update safety and requires the least developer effort while providing sufficient update availability. We now discuss several potential threats to the validity of the study. First, the test suites we used for OpenSSH, vsftpd, and ngIRCd do not exercise all features of the applications, so we may be undercounting how many patches introduce failures into the programs. However, we did endeavor to choose tests that cover the core features of each application, and since we are interested in what the application is doing when it might be updated, we think these tests are representative. For this reason, we did not test cases where the program goes wrong independently of DSU (e.g., error-handling code that only runs prior to shutting the application). As we discussed in Section , our experiments used tests that exercise behavior that persists across the update (although the implementation of that behavior may have changed). This introduces the risk that tests for added/removed/changed behavior might have produced different results. However, defining correct behavior in such cases is not straightforward whereas correctness for our tests was obvious. A related point is that update points within ignore regions are not tested, so failures due to such points may be missed. We have checked for this possibility by minimizing the size and use of these regions and inspecting their effects. This threat could be completely mitigated by continuing to prevent updates within ignore regions after the application is deployed.

Second, our empirical study is limited to three applications and a hand-picked set of updates to them, so the results may not generalize to other applications or updates. This is always a danger with benchmarks. However, we have striven to consider a lengthy streak of updates, and have chosen applications that fit the general mold of single-threaded and/or multi-process server applications written in C. Our results do not directly speak to multi-threaded applications, but for these we note that the qualitative case to be made for manual update points is much stronger than for single-threaded applications, since there are many more application states the programmer must be concerned with [26].

Third, our results may be specific to Ginseng, and may not generalize to other updating systems. We think this threat is unlikely, as argued in Section II-D.

Fourth, our evaluation of the relative effort required to use each safety check is qualitative, rather than quantitative. This presents the risk that our effort comparisons may be biased or fail to generalize. However, it is critical to note that the effort for AS and CFS was strictly greater than the effort for Manual identification. Specifically, all three approaches require restructuring the code around the event loops to work well, but AS and CFS often require additional restructuring due to over-conservatism. In addition, AS and CFS require manually reasoning about the correctness of updating at many more points than Manual.

Finally, there is some discretion involved in how a programmer may extract application code, write transformer functions, etc. It is possible that different reasonable choices would produce different results. We believe that our manual modifications to these programs were dictated by the structure of the program and that other developers would have chosen the same modifications.

VII. Related work

While there is much prior work in developing DSU systems (much of which is cited and/or described in Section I and in Section II-D), this paper represents the first empirical study of the effectiveness of DSU controls to timing. Most prior work has focused on evaluating different implementation mechanisms (e.g., based on compilation or binary patching), and relatively little focus has been given to assessing the effectiveness of timing mechanisms, particularly for ensuring that updates are safe.

Some prior work has considered what it means for updates to be correct, and proposed timing restrictions that would ensure correctness. Gupta et al. [14] originally defined the *update validity* problem as showing, for a given program and patch, that after patching the old version its execution would eventually reach a state that could have been reached by executing the new version from scratch. Gupta et al. showed that this problem is in general undecidable, and then proposed a way to calculate a set of functions that must be inactive if the update were to be valid. However, Gupta's check only applies when a patch adds new functionality and programs do not use complex data types and pointers. Stoyle et al. [31] proved that CFS, and a flavor of AS, prevent type incorrect executions, but did not evaluate whether the allowed executions may be behaviorally incorrect, as was done in our study. As described in Section II-D, most practical DSU implementations use the AS check but do not evaluate its efficacy, or do so only cursorily. Some systems, such as DyMOS [22] and POLUS [7], permit fine-grained timing controls, but no means to evaluate their proper use is given. Our study is the first to provide empirical data on the effectiveness of common timing controls in a practical setting.

Our approach to generating update tests is related to Chess [25] and MultithreadedTC [29], which test multi-threaded programs by intelligently enumerating a program's potential thread schedules. At a high level, our technique for test minimization is like partial order reduction in model checking [2], which is used to avoid consideration of distinct program executions that result in the same states. Our minimization algorithm on traces is inspired by Neamtiu et al.'s observation that an update at two program points is equivalent if the activity between those two points is unaffected by the patch [27]. Neamtiu et al. applied this observation to a static analysis for implementing *update transactions* whose execution is version consistent (i.e., consisting of behavior entirely attributable to only one version), while we apply it to test case minimization. Our testing experiments use developer annotations to identify sources of non-determinism in code and compares program traces to be sure they match. Many other techniques have been proposed to provide deterministic replay including approaches based on libraries [13], [30], [12] and virtual machines [35], [5], [10].

The failure examples in Section V-D represent the first careful analysis of failures allowed by common DSU systems. The notion of version consistency was identified previously [27], but the relative frequency of version consistency errors was never studied empirically. Indeed, many DSU systems make an implicit assumption that version consistency errors are not a problem [21], [4], [1], [6].

VIII. Conclusions

We have presented an empirical evaluation of means to control the timing of a dynamic update. Such means restrict the application of an update to select program points. We evaluated the effectiveness of three ways in which these points are selected in typical DSU systems: (a) manually, according to a simple design pattern, (b) automatically, such that points do not occur in functions an update changes (referred to as activeness safety), or (c) automatically, such that execution in active code following the update will not access definitions whose type signature has changed (referred to as con-freeness safety). Our evaluation is based on systematically testing long streaks of updates to OpenSSH, vsftpd, and ngIRCd, three substantial, open source server applications. The systematic testing framework we developed is noteworthy in that it evaluates the effect of an update applied at essentially any point during a program's execution despite actually testing only a small fraction of such update points. We tabulated which update points are permitted by which mechanism, and whether tests of updates at these points succeeded or failed. We also assessed the programmer effort involved to use these mechanisms.

We found that all three timing mechanisms eliminated a substantial number of failures, but only the manual approach eliminated all failures. Also, while the automatic approaches allowed many more update points than the manual approach, updates should still happen often enough even in that case. Finally, we found that the programmer effort was highest for the automatic approaches, because programs needed to be refactored slightly to be compatible with them. The manual approach required programmers to identify update points, but this task was relatively easy, compared to the needed refactoring.

Our study suggests several lines of future work. Our study of the failures allowed by the automatic checks could provide a basis for devising a better automated approach. While such an approach is unlikely to be perfect in general, it may be possible for it to be perfect in limited cases, such as for small code changes for the purposes of security fixes [4], [1]. It might also be interesting to consider automating the placement of update points according to the manual pattern we followed; an analysis could be used to assess whether such update points can be reached frequently enough, e.g., applying techniques similar to those used to prove termination via reachability [8].

Acknowledgments This research was supported by the partnership between UMIACS and the Laboratory for Telecommunications Sciences. We would also like to thank Iulian Neamtiu for supporting our use of the Ginseng DSU system.

References

[1] G. Altekar, I. Bagrak, P. Burstein, and A. Schultz. Opus: online patches and updates for security. In *USENIX Security*, 2005.

[2] R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Partial-order reduction in symbolic state space exploration. In *CAV*, 1997.

[3] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. The Pragmatic Programmers, LLC, 2007.

[4] J. Arnold and F. Kaashoek. Ksplice: Automatic rebootless kernel updates. In *Eurosys*, 2009.

[5] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *VEE*, 2006.

[6] G. Bracha. Objects as software services. http://bracha.org/objectsAsSoftwareServices.pdf, Aug. 2006.

[7] H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew. Polus: A powerful live updating system. In *ICSE*, pages 271–281, 2007.

[8] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*, 2006.

[9] M. Dmitriev. Towards flexible and safe technology for runtime evolution of java language applications. In *Workshop on Engineering Complex OO Systems for Evolution*, 2001.

[10] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *OSDI*, 2002.

[11] Edit and continue. http://msdn2.microsoft.com/en-us/library/bcew296c.aspx.

[12] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay debugging for distributed applications. In *USENIX*, 2006.

[13] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. Frans, and K. Z. Zhang. R2: An application-level kernel for record and replay. In *OSDI*, 2008.

[14] D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *IEEE TSE*, 22(2), 1996.

[15] C. M. Hayden, E. A. Hardisty, M. Hicks, and J. S. Foster. Efficient Systematic Testing for Dynamically Updatable Software. In *HOTSWUP*, 2009.

[16] C. M. Hayden, E. A. Hardisty, M. Hicks, and J. S. Foster. A testing-based empirical study of dynamic software update safety restrictions. Technical Report CS-TR-4949, Department of Computer Science, the University of Maryland, College Park, 2010.

[17] C. M. Hayden, E. A. Hardisty, M. Hicks, and J. S. Foster. Evaluating dynamic software update safety using systematic testing. Technical Report CS-TR-4993, Department of Computer Science, the University of Maryland, College Park, 2011.

[18] C. M. Hayden, E. K. Smith, M. Hicks, and J. S. Foster. State Transfer for Clear and Efficient Runtime Updates. In *HOTSWUP*, 2011.

[19] M. Hicks and S. Nettles. Dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 27(6):1049–1096, 2005.

[20] Java platform debugger architecture. http://java.sun.com/j2se/1.4.2/docs/guide/jpda/.

[21] The K42 Project. http://www.research.ibm.com/K42/.

[22] I. Lee. *DYMOS: A Dynamic Modification System*. PhD thesis, Dept. of Computer Science, U. Wisconsin, Madison, 1983.

[23] K. Makris. Upstare manual. http://files.mkgnu.net/files/upstare/UPSTARE_RELEASE_0-12-8/manual/html-single/manual.html.

[24] K. Makris and R. Bazzi. Immediate multi-threaded dynamic software updates using stack reconstruction. In *USENIX ATC*, 2009.

[25] M. Musuvathi, S. Qadeer, and T. Ball. Chess: A systematic testing tool for concurrent software. Technical Report MSR-TR-2007-149, Microsoft Research, 2007.

[26] I. Neamtiu and M. Hicks. Safe and timely dynamic updates for multi-threaded programs. In *PLDI*, June 2009.

[27] I. Neamtiu, M. Hicks, J. S. Foster, and P. Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *POPL*, 2008.

[28] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for C. In *PLDI*, 2006.

[29] W. Pugh and N. Ayewah. Unit testing concurrent software. In *ASE*, 2007.

[30] Y. Saito. Jockey: A user-space library for record-replay debugging. In *AADEBUG*, 2005.

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

[32] S. Subramanian, M. Hicks, and K. S. McKinley. Dynamic software updates for Java: A VM-centric approach. In *PLDI*, 2009.

[33] C. Walton. *Abstract Machines for Dynamic Computation*. PhD thesis, University of Edinburgh, 2001. ECS-LFCS-01-425.

[34] D. A. Wheeler. Sloccount. http://www.dwheeler.com/sloccount/.

[35] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, B. Weissman, and V. Inc. Retrace: Collecting execution trace with virtual machine deterministic replay. In *MoBS*, 2007.

PLACE PHOTO HERE

Christopher M. Hayden is a doctoral student in the Department of Computer Science at the University of Maryland, College Park. His research interests are in tools and techniques to help developers safely apply dynamic software updating. In particular, his research aims to provide an improved understanding of the challenges posed by run-time updates, such as ensuring update correctness, and to reduce the developer effort required to address those challenges.

PLACE PHOTO HERE

Eric A. Hardisty is a doctoral student in the Department of Computer Science at the University of Maryland, College Park. His research interests are Natural Language Processing and Programming Languages. His current research focuses on sentiment analysis and the detection of persuasion.

PLACE PHOTO HERE

Edward K. Smith is an undergraduate in the Department of Computer Science at the University of Maryland, College Park.

PLACE PHOTO HERE

Michael Hicks is an Associate Professor in the Department of Computer Science and the University of Maryland Institute for Advanced Computer Studies (UMIACS) at the University of Maryland, College Park. His research focuses on developing systems that are reliable, available, and secure. His solutions often involve tools and techniques—such as new programming languages, compilers, run-time systems, and static analyses—that aim to make programmers more effective.

PLACE PHOTO HERE

Jeffrey S. Foster is an Associate Professor in the Department of Computer Science and the University of Maryland Institute for Advanced Computer Studies (UMIACS) at the University of Maryland, College Park. His research aims to give programmers practical new tools to help improve the quality and security of their programs. His research interests include programming languages, program analysis, constraint-based analysis, and type systems.