Evaluating Dynamic Software Update Safety Using Efficient Systematic Testing

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

Abstract

Dynamic software updating (DSU) systems, which allow programs to be patched on the fly, often employ automatic safety checks to avoid applying a patch that may lead to incorrect behavior. This paper presents what we believe is the first significant empirical evaluation of two DSU safety checks: *activeness safety* (AS) and *con-freeness safety* (CFS). To measure the checks' effectiveness, we developed a novel approach to systematically test dynamic updates by forcing updates at each of the *update points* encountered during system test execution. To mitigate the increase in the number of tests, we developed an algorithm for *test suite minimization* which proved highly effective in our experiments. Using this approach, we systematically tested a series of dynamic patches to OpenSSH, vsftpd and ngIRCd. AS and CFS prevented most, but not all, dynamic update failures; CFS allowed more failures than AS, but AS was more restrictive, disallowing many more successful updates. Our results show that neither AS nor CFS can be completely relied upon to produce correct dynamic updates, and our investigation points to the reasons why. Our work represents an important step, and important insights, toward developing safe, easy-to-use DSU systems.

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 to fix bugs or add features without incurring downtime. Support for dynamic software updating (DSU) takes

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

many forms. Smalltalk and CLOS have long provided basic DSU support to enable "fix-and-continue" development, and the JVM and CLR now provide similar support [13], [9]. Unsanity's Application Enhancer [25] can update running Mac OS X applications, while the DSU capabilities of Ericsson's Erlang programming language [3] are regularly used to hot-patch fielded telecommunications systems. Research DSU systems for C, C++, and Java have been used to dynamically update servers and operating systems with patches ranging from security bug fixes [4], [1] to full releases [21], [7], [14], [24].

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. For example, the two versions of function bar in Figure 1 have identical semantics (we assume baz, not shown, is the same in both versions), but consider what could happen if the program is updated just as bar starts running. In many DSU systems [21], [7], [24], [12], functions running at the time of an update continue executing the old code, while subsequent function calls invoke the new version. Thus, we would have a type error: the old bar would call the new foo with the integer 0, instead of a pointer to an integer as expected. As a result, foo dereferences 0, causing the program to crash.

To avoid these and other problems, most DSU systems place restrictions on *when* a dynamic patch may be applied. Several systems have proposed mechanisms for manually imposing timing restrictions [15], [7], while automatically imposed restrictions fall into two categories:

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 neither running nor on the activation stack of a running thread. This ensures that, following an update, the program will

```
int foo(int x, int y) {
  return x + y;
}
void bar() {
  int z = 0;
  ...
  z = foo(z,5);
  baz();
}
```

(a) Old version

```
void foo(int *x, int y) {
  *x += y;
}
void bar() {
  int z = 0;
  ...
  foo(&z,5);
  baz();
}
```

(b) New version

Fig. 1. Two versions of a program

only execute the new version's code. Notice that AS prevents the problematic update location in our example by forbidding updates from taking effect in bar, since it has changed. AS is advocated by Bracha [6], and is used by DSU systems such as Dynamic ML [26], K42 [14], OPUS [1], Ksplice [4], and Jvolve [24].

Con-freeness safety (CFS): Stoyle et al. [23] proposed a condition called *con-freeness* that relaxes AS by allowing updates to active code, but only 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 out the problematic update point in the example, since foo's type signature has changed and foo would be called after the update takes place. However, unlike AS, CFS would allow an update *after* the call to bar, since the type signature of baz, which is called next, is not changed by the update. Ginseng [21] uses CFS to ensure update type safety.

Although AS and CFS are clearly useful, Gupta [10] has shown that no fully automatic safety check can be perfect: a check must be either too *permissive*, allowing some incorrect updates, or too *restrictive*, disallowing some correct updates, or both. Nevertheless, while no check can be perfect in theory, there may be a fully automatic check that is highly effective in practice, e.g., by disallowing all incorrect updates along with some that might actually be correct.

In this paper, we present what we believe to be the first significant empirical evaluation of the permissiveness and restrictiveness of AS and CFS when applied to real programs. The aim of our study is to help understand the advantages and limitations of these approaches, and ultimately to understand what it will take to develop a practical and safe DSU system.

To evaluate the two safety checks, we developed a novel dynamic update testing methodology and implemented it for Ginseng [21], a freely available DSU system for C programs. Our technique and its implementation are contributions in their own right, as they constitute the first practical means to systematically validate the likely-correctness of a dynamic patch. In our approach, a dynamic patch is tested against a given a suite of system tests. We run each test multiple times, applying the dynamic patch at a different program point in each run. Running a test for every possible update point would be prohibitively expensive, so we have developed a minimization algorithm by which we can avoid testing any update point that would produce provably identical behavior to other tested points. We find this algorithm to be highly effective in practice: overall, 95% of the update tests from OpenSSH, 86% of points from vsftpd, and 90% of points from ngIRCd could be eliminated.

To evaluate the AS and CFS checks, we used our framework to execute a suite of system tests and tracked whether, when executing a given test and applying a patch at each non-redundant update point, the test succeeds or fails. For each choice of tested update point we determined whether the update would have been allowed by the AS and/or CFS safety checks. In this way, we measured each check's permissiveness and restrictiveness based on the update test's outcome. We considered three-years' worth of updates to vsftpd (totaling eleven releases) and OpenSSH (totaling nine releases), two popular open-source server programs that have been extensively studied in the DSU literature [21], [7], [17]. We also considered eight months (totaling eight releases) of updates to ngIRCd, another popular open-source server application.

Our two key results are as follows. First, we find that both safety checks are highly effective at avoiding failures, though AS does this better than CFS. With no safety checking, many updates fail: in total, 1.59M of the total 10.5M tested executions failed (15%). Using either AS or CFS dramatically reduces the number of failures to about 495 for AS (0.2%) and 48K for CFS (3.2%). In addition to making such quantitative comparisons, Section VII-B investigates the causes of particular failures and why safety checking failed to prevent them. We expect these examples will be useful to DSU users and motivate future research.

Second, we found that both AS and CFS are fairly permissive, though CFS is more permissive than AS. In total, CFS permitted 76% of the passing update points, while AS permitted 61% of them, a difference of about 1.26M update tests; roughly 59% of passing update points are allowed by both. Thus, AS's lower failure rates come at the cost of higher restrictiveness, compared to CFS. Moreover, this restrictiveness could be much higher: initial experience with our framework suggested we needed to slightly refactor the test programs by extracting some code blocks into separate functions [21], or else AS would preclude all possible updates.

While overall more available updates is better, in general we only need updates to occur reasonably often. We 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. Since we likely only need to support updates during the loops, this is a positive result. Indeed, we found that restricting updates to just a few manually specified points in the loops eliminated all test failures.

In summary, this paper makes the following contributions:

- We present the first substantial empirical study of the practical effectiveness of DSU systems.

Our study evaluates the two safety checks most often employed and considers the actual evolution of real server systems. While many prior DSU systems have been proposed, this paper is the first to comprehensively consider the relative safety and availability of updates in such systems. 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 research.

• We present the first framework for systematically testing dynamic software updates.[1] This framework includes a novel algorithm for minimizing the number of update tests without compromising their checking power, which we find highly effective on our benchmark programs.

II. Dynamic Software Updating

This section describes the workings of modern dynamic updating systems followed by a detailed description of the two safety checks—Activeness Safety and Con-freeness Safety—these systems often use to avoid incorrect updates. Despite differences in the choice of mechanisms, many updating systems' semantics are quite similar. As we chose to use Ginseng for our study, we describe it in more detail, first considering its basic mechanisms and then how it handles updates to active code.

A. Basic DSU semantics and implementation

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. Ksplice [4], Jvolve [24] and K42 [14] take a similar approach. In some systems, including POLUS [7], DLpop [12], and Erlang [3], the programmer can partially control whether a function call should reach the newest version or the contemporaneous one.

To implement its updating 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

[1]A previously published, invited workshop paper discusses our testing methodology and evaluates our minimization algorithm on vsftpd and OpenSSH [11]. The current paper is intended as an archival version of that paper, and substantially expands it with the study of AS and CFS safety checks along with new results for ngIRCd.

to the updated versions. DLpop, Erlang, and K42 use a similar mechanism, while POLUS, Ksplice, and Jvolve achieve a similar effect by dynamically rewriting and recompiling parts of the program to redirect the calls.

Ginseng also executes user-defined *transformation functions* provided with a patch to update changed data, e.g., to convert values whose type definitions have changed between versions. Global data is updated by *state transformation functions* at update time, and type-level conversions are effected by *type transformation* functions as data is accessed by the program. Such on-demand transformation is enabled by special compilation: each access to data whose type could change is prefaced by code to check whether the data is up-to-date, and converts it if not. In Erlang and DLpop, data transformation is scheduled entirely by the programmer, while K42 similarly changes data on-demand, and Jvolve changes data by piggybacking on garbage collection. POLUS permits multiple views of data, depending on whether it is accessed by old or new code, and the programmer must ensure these views are coherent.

B. Updating active code

In Ginseng and the other systems we have discussed, 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 following function which implements a typical server's event processing loop:

```
void foo (...)  {
  // ... loop startup code
  while(1) {
    req = get_request();
    switch(req) {
      case OPERATION_1: // ... break
      case OPERATION_2: // ... break
    }
  }
  // ... loop cleanup code
}
```

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 foo is called. However, it could be that foo runs for a long time without exiting. Thus, the effects of updates to code in this long-running loop would be unduly delayed.

To avoid this problem, when using Ginseng (and most other systems) we can 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, we can extract the loop body into a separate function. Following an update, each subsequent call to the loop body will reach the new version. Likewise, the loop cleanup code can be made into a new function, allowing the new version to be reached once the refactored loop exits. We may similarly want to extract the continuations of functions that could be on the stack when a desirable update point (such as this loop) is reached. The Ginseng updating system provides *code extraction* features to support such refactorings [21]. These allow developers to annotate loops or blocks of code to be extracted, and the compiler will replace the code with a call to a function containing this code (abstracting the local state the loop depends on, which itself is subject to transformation at update time).

While efficacious, the drawback of using code extraction is that developers must anticipate which code to extract before deploying the program. In ours and others' experience with server programs, doing so is not difficult: typically, each long-running event loop, and potentially its subsequent teardown code, must be extracted [21], [19], [7]. UpStare [17], a recently developed DSU system, permits a programmer to include in a patch a mapping between a PC location in a changed function's old version and one in the new, as well as provide a function to initialize the stack frame of the new version based on the stack frame of the running version. At update time, if a changed function is active at a PC specified in the patch mapping, the transformation function is used to initialize the stack, and then execution proceeds at the new version's corresponding PC. In any case, one can think of results using Ginseng or similar systems on an extracted program as simulating UpStare's behavior on the same program without extractions.

C. DSU safety checks

As shown by the example in Figure 1 in the introduction, applying an update at an inopportune time can lead to incorrect program semantics. To avoid such problems, DSU systems often employ safety checks that automatically restrict when a patch can be applied. The two most popular checks, which we evaluate empirically in this paper, we dub *activeness safety* (AS) and *con-freeness safety* (CFS).

Activeness Safety (AS): AS is simple: it prevents application of an update if the patch changes *active* functions, i.e., functions that are either running or referenced via a return address from the stack of a running thread [4], [14], [1], [7].

Activeness Safety ensures the updated program's execution is *type safe* because, if a function f is called from a function g, and the type of f is changed in the new version, then g must have changed as well, to properly call it at the new type. A similar argument can be made for accesses to values whose representation changes, since the code generated for those accesses must also have changed.

However, an update applied at an activeness-safe point may still cause the program to fail, despite being type-correct. Such failures could arise when old functions that have already executed are followed by an update that changes related functions called soon thereafter such that the old and new versions make incompatible assumptions about the environment or state. We refer to these failures generically as *version consistency errors* [20].

Con-freeness safety (CFS): AS can sometimes be too restrictive. For example, imagine a server in which main parses the command-line options and concludes by calling a function like foo from Section II-B to start processing events. In a subsequent version, suppose main adds support for new command-line options. Even if the new option-processing code would have no effect on the updated execution, nevertheless the update will be indefinitely precluded because main is always active.

As a remedy to this problem, Stoyle et al. [23] proposed a more relaxed safety check called *con-freeness*. This check allows updates to active functions, but only if it can prove those functions will not subsequently call functions or access any data whose type signatures have changed. In other words, any code active on the stack must be free of *concrete* uses (function calls, dereferences, field accesses, etc.) of definitions that have changed in a type-incompatible way; hence the name, *con-freeness*. This restriction ensures that updated executions will always be type-correct. For example, in Figure 1, CFS would allow the update while bar is running, but only after it has called foo.

Ginseng implements CFS using a combination of static and dynamic analysis. We note one important implementation detail. In Ginseng, a program calls the function DSU_update() to ask the run-time system whether a dynamic update is available. If an update is available, and is compatible with the CFS check, it is applied at this point. The developer has the choice of inserting calls to DSU_update() manually, or having the compiler insert them automatically according to some policy, e.g., one prior to each non-system function call in the program. Unless specified otherwise, we assume the latter approach in our examples.

While very useful, CFS's extra permissiveness relative to AS presents additional opportunities for version consistency errors. In particular, CFS introduces the possibility for a function on the program stack at the time of the update to be involved in a version consistency error.

Prior to this work, we had some intuition how version consistency errors might be permitted by these safety checks, but little understanding of whether such problems were likely to occur in practice. Indeed, many DSU systems make an implicit assumption that version consistency errors are not a problem [14], [4], [1], [6]. We also did not know the magnitude of potential DSU problems overall, whether due to type errors or version consistency errors.

Therefore the main question we aim to address in this paper is: how often do problems arise when applying updates to practical programs, and when they arise, what form do they take? We also address the related question: how often are AS and CFS too permissive, and/or too restrictive? To answer these questions we developed a methodology for systematically testing dynamic updates, which we describe next.

III. Testing Dynamic Updates: Basic Procedure

To evaluate the effectiveness of automatic DSU safety checks, 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.

In what follows, we presume we can specifically enumerate those program points at which a particular patch can be applied during a program's execution. In DSU systems like DLpop [12] and Ginseng [21], programmers can provide a *whitelist* of program locations (e.g., line numbers) that are valid for an update, while DYMOS [15], POLUS [7], and others propose a *blacklist* (e.g., by indicating that certain functions must be inactive prior to updating). We define an *update point* to occur each time the program reaches a whitelisted or non-blacklisted location such that automatic safety checks (if any) are satisfied for that point and the given patch. In Ginseng, the whitelist is defined for the original program when it is deployed: updates can only occur at calls to a `DSU_update()` function, and then only for those that satisfy Ginseng's CFS check.

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. In particular, we define t_π^i 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. Such update tests are well-defined when t's execution is deterministic, since each update point i that arises during execution is unambiguous; we discuss how we handle non-determinism in Section V. To run update tests, we can easily modify the DSU run-time to delay patch application to the i^{th} update point reached. Since t presumably terminates, there will be a finite number of induced update tests t_π^i for a fixed π.

We select the system tests t to start from as follows. Let T_i be a suite of system tests for P_i, for $i \in \{0, 1\}$. All $t \in (T_0 \cap T_1)$ should pass for both P_0 and P_1, so all t_π^i for all i are reasonable update tests. On the other hand, tests $t \in (T_1 - T_0)$ are meant to test functionality that is new to P_1, else t would have also been in T_0. (If this is not the case, we can treat such a test as if it were in T_0 as well.) For such a test, not all t_π^i for all i will be reasonable update tests. To see why, suppose P_0 is an FTP server, and P_1 adds support for a new command `qux`. If t tests the proper functioning of `qux`, test t_π^i will fail if update point i arises too long after t sends the `qux` command to the server.

To address this situation we construct a *hybrid test* amenable to execution on either P_0 or P_1. In particular, we execute test t with P_0 and run it to completion without performing an update. We observe P_0's output, and then manually construct the hybrid test t' that modifies t to also allow this output. Thus t' will be considered as having passed if its output corresponds to the output of either P_0 or P_1. We then generate update tests for t'. At the tester's discretion, a hybrid test could also specify behaviors that are correct despite not exactly matching either P_0 or P_1.

If P_0 responds gracefully to test t, the hybrid test t' clearly makes sense. However, suppose t tests a bug fixed in P_1 that causes P_0 to crash. In this case we would deem the hybrid test based on t successful if the program either crashes or produces the correct output. While reasonable, the hybrid test may misattribute a crash to the expected behavior of P_0, when it could instead be due to an incorrectly written or ill-timed patch.

We can guard against this possibility in two ways. First, we can gain confidence in the patch overall through other tests in which the outcome is the same in both versions. Second, we can make sure that the *first* update point tested for t (i.e., induced test t^1_π) always produces P_1's behavior (since the overall execution should be identical to P_1), and then examine the execution of the first series of crashing update tests manually (e.g., via tracing) to ensure that the crash is not due to the update itself.

The last category of tests are those in $T_0 - T_1$, which are likely tests for deprecated functionality. In these cases, we might omit the test, since it does not apply to P_1. Alternatively, if P_1 handles deprecated features gracefully, e.g., it issues warning messages for unsupported commands, we could create hybrid tests for these cases as well.

IV. Update Test Suite Minimization

The procedure described in the previous section 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.

A. Formal language and traces

We present our algorithm in terms of the small formal language in Figure 2, meant to model our actual implementation described in the next section. In this language, expressions consist of constants c (e.g., integers, floating point numbers, etc.), variables x, function calls $f(e_1, ..., e_n)$, or sequences $s; e$, which execute s and then e, returning the result of the latter. Statements consist of assignment $x := e$, sequencing $s_1; s_2$, branching $\mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2$ (which executes s_1 if e evaluates to a non-zero integer, and s_2 otherwise), looping $\mathsf{while}\ e\ \mathsf{do}\ s$ (which repeatedly executes s until e evaluates to a non-zero integer) and the no-op skip. The statement update identifies a program point where a dynamic update is permitted to take place if a patch is available, akin to Ginseng's DSU_update() calls described above. We can apply our algorithm to other dynamic updating approaches as discussed in Section IV-D.

We model a program as a pair (H, s), where H is a *heap* containing bindings for functions and global variables, and s is the statement to be executed. A *binding* b maps an identifier x to a constant c or to $\lambda(x_1, ..., x_n).e$, which denotes a function with arguments x_1 to x_n and body e. When the function is called it returns the result of evaluating e with the formal parameters substituted by the actual arguments. A *patch* π is also a set of bindings, like the heap. When a patch is applied, its bindings add to or replace the corresponding bindings in the heap. Roughly speaking, we can model a C program in this language as $(H, \mathsf{fin} := \mathsf{main}(c_1, ..., c_n))$ where H contains the program's initial function and global variable bindings, the c_i represent the command-line arguments, and fin receives the final result.

We define our algorithm in terms of *event traces* induced by the statement or expression's execution. Event traces are defined at the bottom of Figure 2. Each event corresponds to the execution of one program construct, and individual events are concatenated using the ; operator. For example, suppose plus is a function that returns the sum of its arguments, and H is the heap $(\mathsf{x} \mapsto 4,\ \mathsf{plus} \mapsto \lambda(x_1, x_2).e)$.[2] Then if we evaluate the statement $\mathsf{x} := \mathsf{plus}(\mathsf{x}, 5)$, we produce the

[2]The code for plus is omitted, just notated here as e.

$$
\begin{array}{llcl}
\text{Expressions} & e & ::= & c \mid x \mid f(e_1, ..., e_n) \mid s; e \\
\text{Statements} & s & ::= & x := e \mid s_1; s_2 \mid \mathsf{skip} \\
 & & \mid & \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \mid \mathsf{update} \\
 & & \mid & \mathsf{while}\ e\ \mathsf{do}\ s \\
 \\
\text{Heap, patch} & H, \pi & ::= & \cdot \mid b, H \\
\text{Binding} & b & ::= & x \mapsto c \mid f \mapsto \lambda(x_1, ..., x_n).e \\
 \\
\text{Traces} & \nu & ::= & \nu; \nu \mid \mathit{skip} \mid \mathit{read}(x, c) \mid \mathit{write}(x, c) \\
 & & \mid & \mathit{call}(f(c_1, ..., c_n)) \\
 & & \mid & \mathit{noupdate} \mid \mathit{update}(\pi)
\end{array}
$$

Fig. 2. Syntax of programs and event traces

following trace ν and new heap H':

$$
\begin{aligned}
\nu &\equiv \mathit{read}(\mathsf{x}, 4); \mathit{call}(\mathsf{plus}(4, 5)); ...; \mathit{write}(\mathsf{x}, 9) \\
H' &\equiv (\mathsf{x} \mapsto 9,\ \mathsf{plus} \mapsto \lambda(x_1, x_2).e)
\end{aligned}
$$

That is, starting out with a heap that maps x to 4 and plus to an appropriate function, executing $\mathsf{x} := \mathsf{plus}(\mathsf{x}, 5)$ produces a heap that maps x to 9. The execution also yields an event trace ν indicating x was read, the function plus was called, its body executed (...), the function returned, and then x was written. We discuss *update*(π) and *noupdate* events shortly.

Formally, we can introduce an operational semantics for our language to generate traces from program executions. Our semantics defines a relation $(H, s) \longrightarrow^{\nu} H'$ meaning that, if H and s are the current heap and statement, then H' is the heap after s has been completely executed, and ν is the generated event trace. We also need a sibling relation $(H, e) \longrightarrow^{\nu} (H', c)$ for expressions, where c is the result of computing the expression e.

The operational semantics behaves as expected and is fairly straightforward, e.g., the skip statement generates a *skip* trace. The most interesting cases are for update points update, variable reads and writes, and function calls, i.e. expressions of the form $f(e_1, ..., e_n)$. update statements produce an *update*(π) trace event when an update is taken and *noupdate* otherwise. Compound statements/expressions concatenate the traces induced by each sub-expression based on the language's order of evaluation. For example, a function call produces a trace of the form:

$$\nu_1; ...; \nu_n; \mathit{call}(f(c_1, ..., c_n)); \nu$$

Where $\nu_1; ...; \nu_n$ represents the concatenation of the traces induced by evaluation of the arguments, $\mathit{call}(f(c_1, ..., c_n))$ is a trace element representing the function call, and ν represents the trace yielded when executing the function body. Other compound constructs such as sequencing and branching perform similar concatenation.

A full formal presentation of the semantics is shown in Figure 12 in the Appendix.

B. Finding equivalent update points for π

Let $t = (H, s)$ be a system test, i.e., the program code in H with a test driver s. Then if we run t, the resulting trace ν_t contains some number n of *noupdate* events, which in turn induce a set of update tests $t^1_\pi \dots t^n_\pi$. Our goal is to determine which of these update tests, if executed, would produce *equivalent* traces:

Definition 4.1: Traces ν and ν' are *π-equivalent for* (H, s) *iff* we have $(H, s) \longrightarrow^{\nu} H'$ and $(H, s) \longrightarrow^{\nu'} H'$ where

$$\begin{aligned} \nu &= \nu_1; \mathit{update}(\pi); \nu_2; \mathit{noupdate}; \nu_3 \\ \nu' &= \nu_1; \mathit{noupdate}; \nu_2; \mathit{update}(\pi); \nu_3 \end{aligned}$$

The key here is that H, H', s, ν_1, ν_2, and ν_3 are exactly the same in both ν and ν'. This means that they read and write the same values to and from the same variables, call the same functions with the same parameters, etc. The only difference is when the update is actually applied, but obviously this difference has no effect on the program's execution. Our algorithm identifies sets of update points that would produce equivalent traces; thus we can choose an update point i from each identified equivalence class and run only t^i_π.

We compute equivalent update points by applying the $\mathrm{gentests}$ function in Figure 3 to the original trace ν_t. The $\mathrm{gentests}$ function invokes $\mathrm{conflict}(\pi, \nu)$, which returns a boolean indicating whether actions in ν *conflict* with patch π. More precisely, if this function returns *false*, then applying π any time during a run that generates ν will not affect the generated trace (and therefore will not affect the program's behavior). If the function returns *true*, then applying the patch may affect the program's behavior.

Function $\mathrm{conflict}(\pi, \nu)$ is defined at the top of Figure 3. Given a call, read, or write to x, there is a conflict with π if and only if $x \in \mathrm{dom}(\pi)$. There are no conflicts with *skip* or *noupdate*. Given a different update with patch π', there is a conflict if and only if π' and π affect overlapping functions or variables (each update test will perform one update per run, making

$$\begin{array}{l}
\text{conflict}(\pi, \textit{skip}) = \textit{false} \\
\text{conflict}(\pi, \textit{call}(x(\ldots))) = (x \in \text{dom}(\pi)) \\
\text{conflict}(\pi, \textit{read}(x, \ldots)) = (x \in \text{dom}(\pi)) \\
\text{conflict}(\pi, \textit{write}(x, \ldots)) = (x \in \text{dom}(\pi)) \\
\text{conflict}(\pi, \textit{noupdate}) = \textit{false} \\
\text{conflict}(\pi, \textit{update}(\pi')) = (dom(\pi) \cap dom(\pi') \neq \emptyset) \\
\text{conflict}(\pi, \nu_1; \nu_2) = (\text{conflict}(\pi, \nu_1) \vee \text{conflict}(\pi, \nu_2))
\end{array}$$

—

$$\begin{array}{l}
\text{gentests}(\pi, N, U, \nu) = (N, U) \\
\quad \text{where } \nu \neq (\nu_1; \nu_2) \wedge \nu \neq \textit{noupdate} \wedge \neg\text{conflict}(\pi, \nu) \\
\text{gentests}(\pi, N, U, \nu) = (N, U \cup \{N\}) \\
\quad \text{where } \nu \neq (\nu_1; \nu_2) \wedge \nu \neq \textit{noupdate} \wedge \text{conflict}(\pi, \nu) \\
\text{gentests}(\pi, N, U, \textit{noupdate}) = (N + 1, U) \\
\text{gentests}(\pi, N, U, \nu_1; \nu_2) = \\
\quad \text{let } (N', U') = \text{gentests}(\pi, N, U, \nu_1) \text{ in } \text{gentests}(\pi, N', U', \nu_2)
\end{array}$$

Fig. 3. conflict and gentests functions

this case academic). Finally, a patch π conflicts with trace $\nu_1; \nu_2$ if it conflicts with either ν_1 or ν_2.

The bottom of Figure 3 defines $\text{gentests}(\pi, N, U, \nu)$, which uses $\text{conflict}()$ to compute a minimal set of update tests for ν. Here π is the patch, N is the index of the last-seen *noupdate* event, U is the set of indexes of update points to test, and ν is the trace (which should contain no *update*$(\ldots)$ events). The $\text{gentests}()$ function returns a pair (N', U') with the new index N' of the most recently seen update point and new set U' of update point indexes to test. Thus, given a complete trace ν_t from a system test t, we compute

$$(N, U) = \text{gentests}(\pi, 0, \emptyset, (\nu_t; \textit{noupdate}))$$

The set U defines a minimal set of update points that achieve 100% update coverage; we ignore $i = 0$, if it happens to be in U, since 0 represents the beginning of the trace and not a proper update point.

In the definition of $\text{gentests}()$ the first clause handles the case when ν is not a sequence, is not an update point, and does not conflict with π. In this case, the output sets N and U are the same as the inputs. The second clause is similar, but handles the case when the event *does* conflict with π. In this case, if the update π had been applied before the event ν took place, its outcome might be different. As such, we add the index N of the most recent update point to our set U. The third clause increments the counter N when it sees a *noupdate* event. Finally, the last clause simply processes the two subtraces ν_1 and ν_2 in sequence.

As an example, consider the trace

$$\nu = \textit{noupdate}; \textit{call}(f()); \textit{noupdate}; \textit{call}(g()); \textit{noupdate}; \textit{call}(h())$$

corresponding to the execution of the example from the beginning of Section IV. If we run $\mathrm{gentests}(\pi, 0, \emptyset, (\nu; \textit{noupdate}))$ where $dom(\pi) = \{f\}$, our outcome will be $U = \{1\}$, as follows. When we see the first *noupdate*, we increment $N = 0$ to $N = 1$. Then we see the call to f, where $\mathrm{conflict}(f, \pi) = true$. As such, we add $N = 1$ to U. Subsequent occurrences of *noupdate* increment N, but no further elements are added to U because neither $\textit{call}(g())$ nor $\textit{call}(h())$ conflict with π. On the other hand, if $dom(\pi) = \{f, g, h\}$, then all three calls would conflict with π, and thus N would be added to U in each case, resulting in $U = \{1, 2, 3\}$.

C. Correctness

We have proven our algorithm correct. Given a system test $t = (H, s)$, let ν denote the trace produced by executing t with no updates. Also let ν_π^i denote the trace produced by induced update test t_π^i.

Theorem 4.2 (Correctness): If $(H, s) \longrightarrow^\nu H'$ and $\mathrm{gentests}(\pi, 0, \emptyset, (\nu; \textit{noupdate})) = (N, U)$, then for all $i \notin U$, there exists $j \in U$ such that ν_π^i and ν_π^j are π-equivalent for (H, s).

The proof of this proposition depends crucially on the proof of the following lemma, which shows that if a patch does not conflict with a trace, then applying the patch does not affect the generated trace.

Lemma 4.3: Let H, s, e, ν, π be such that $\neg\mathrm{conflict}(\nu, \pi)$ and either $(H, s) \longrightarrow^\nu H'$ or $(H, e) \longrightarrow^\nu (H', c)$. Let $H_0 = H[x \mapsto \pi(x)]$ for all $x \in \mathrm{dom}(\pi)$. Then we have $(H_0, s) \longrightarrow^\nu H_0'$ or $(H_0, e) \longrightarrow^\nu (H_0', c)$, respectively, with $H_0' = H'[x \mapsto \pi(x)]$ for all $x \in \mathrm{dom}(\pi)$.

The proof is by induction on evaluation derivations.

D. Application to full DSU systems

The $\mathrm{gentests}()$ algorithm can accommodate a variety of dynamic updating systems. The proof of Theorem 4.2 never refers directly to the definition of the judgment $(H, s) \longrightarrow^\nu H'$, relying entirely on simple properties of traces and Lemma 4.3. Thus, to apply $\mathrm{gentests}()$ to a particular DSU system, we need only define $\mathrm{conflict}()$ appropriately and then prove that Lemma 4.3 holds.

The semantics above models DSU systems like Ginseng [21] and DLpop [12], which allow updates to running functions but delay the effect of those updates until the next time the function is called (this is captured precisely in the FUN-CALL rule in Figure 12). This semantics also effectively captures the behavior of systems that permit only updates to inactive functions, such as Ksplice [4], OPUS [1], and K42 [14].

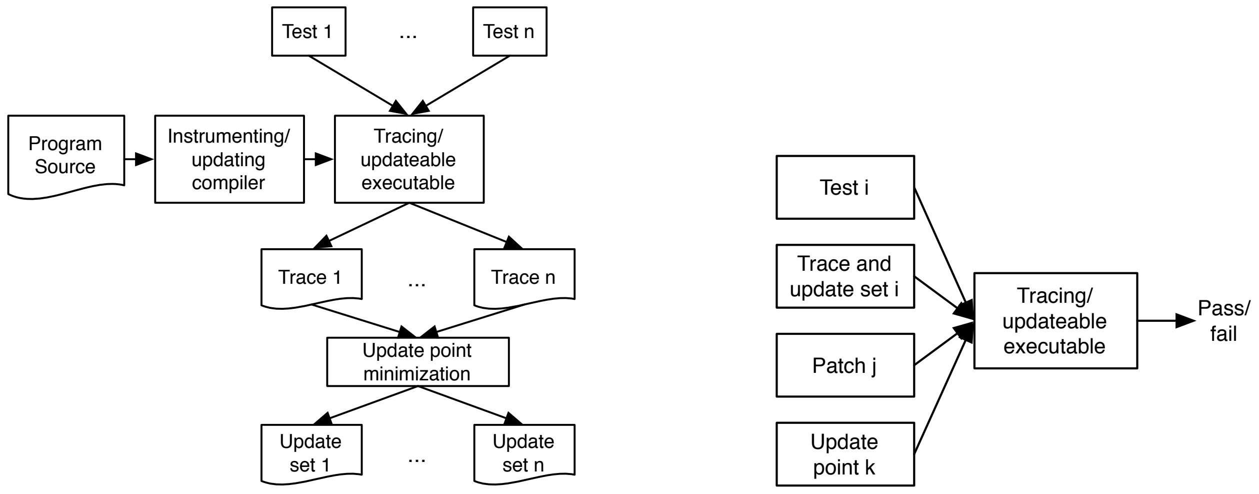
We can extend $\mathrm{gentests}$ to support other updating semantics as well. We consider three possible extensions next.

Updating type definitions: In the full Ginseng, patches can also change type definitions, where accesses to values of updatable type occur via special wrapper functions. When an update occurs, subsequent calls to wrappers first convert the accessed value using a transformer function. Thus we must trace calls to these functions and consider calls conflicting when a patch modifies the respective type definition. Systems like Jvolve [24] and POLUS [7] provide similar support and would require similar changes.

Immediate updates to active functions: Systems like UpStare [16] allow updating some active functions *immediately*, which is to say that if the function is running, it will immediately transition to the new version, without exiting first. Immediate updates can be modeled in our semantics by adding *labels* L to program statements and interpreting a patch π so that if a function f is updated, then when execution reaches label L in f, we begin executing the code starting at L in $\pi(f)$.

To perform test minimization for such a system, we add trace events to be emitted at labeled statements. For example, we would emit an event $label_f$(L) when we begin executing a block in the function f labeled with L and this event would conflict with a patch π when $f \in dom(\pi)$.

Versioned calls: Systems like POLUS [7] and UpgradeJ [5] allow explicitly versioned function calls. For example, we could extend our language with the syntax $[f](e_1, ..., e_n)$ to denote the call to f should be to the *same version* as the code making the call, rather than the most recent version. Explicitly versioned calls, e.g., $[f]^4(e_1, ..., e_n)$ indicating that version 4 of f should be called, are also possible. In this case, we can extend our definition of traces to include explicitly versioned function calls *call*$([f](c_1, ..., c_n))$ (and likewise for returning from a call), while leaving $\mathrm{conflict}()$ as is; i.e., explicitly versioned calls never conflict with a patch,



(a) Instrumentation and trace gathering (b) Running a test case

Fig. 4. DSU testing framework architecture

since the call will always execute an extant code version, and thus be unaffected by the update. Note that if that extant code includes normal calls, which will invoke the newest version, the processing of the event trace will identify these as conflicting.

V. Implementation

We extended Ginseng to implement our testing framework. Our extended implementation, called DSUTest, works in two phases, illustrated in Figure 4(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. We use as few ignore regions as possible to avoid missing test failures due to untested update points within these regions.

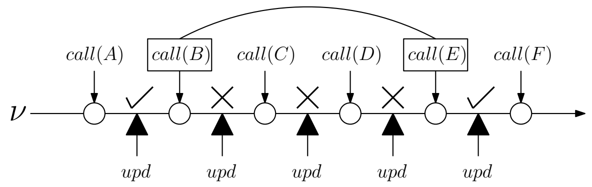
Note that we currently limit our focus to single-threaded programs, making no attempt to account for non-determinism that would arise from thread scheduling. We may explore integrating our framework with techniques for systematically testing under different thread schedules [18],

[22] to handle multi-threading.

Diagnosing Test Failures

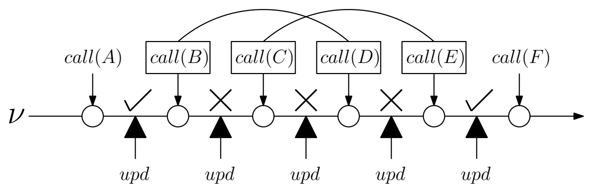
If an update test fails, we must understand why: either the patch itself is incorrect (e.g., it transforms the program's state incorrectly), or the patch is applied at an inappropriate time. Compared to more ad hoc methods of testing, our testing framework affords some advantage in diagnosing failures systematically, as we illustrate in this section.

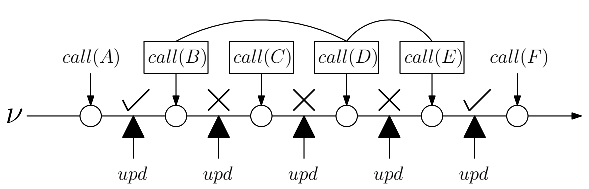
Safety checks such as AS or CFS ensure that all updates are type safe (Section II-C), so any remaining update-specific failures must be version consistency errors, which arise from executing related code at two different versions. We can systematically diagnose the code involved in such errors by applying the following observation. Consider this trace:



This trace of function calls and tested update points exhibits a version consistency error between functions B and E. The three failing update points (marked with X's) occur in executions where B has been executed at the old version and E at the new version. This suggests a useful heuristic to determine the pair of functions involved in a version consistency error: at each end of a sequence of failing update points, consider all function calls that occur until a passing update point is encountered. In this example, there is a single such function (B/E) at each end. While other changed functions that are not revealed by this heuristic may play a role in the error, we have found that knowing these endpoints is always useful and often points directly to the source of the error.

It is also possible that a sequence of failing update tests may result from multiple version consistency errors. The following example traces show two of the many ways this can occur:





In both cases, the version consistency errors (indicated by arcs) are not strictly between function calls at opposite ends of the sequence of failures. However, it is worth noting that a call prior to the first failure and one following the last failure will each be involved in a version consistency error, although not necessarily with each other.

Based on these observations, we developed a tool to help us understand the failures we observed in our experiments. Our tool analyzes the set of update test results, identifies sequences of failures, and presents the developer with the context of the endpoints of these sequences. This simple approach was sufficient for experimental purposes, but employing additional criteria, such as considering only functions that access the same global variables, may further reduce manual inspection effort.

Once we have diagnosed an updating error, then the final step is fixing it. Needless to say, the right fix is entirely dependent on the subject program. One generic approach is to restrict the timing of updates so that the failing update points cannot occur. However, this reduces update availability. Another approach is to use a state transformer to change the program state at update time so that problematic updates can succeed. This providers greater update availability, but may be impractical when it involves state external to the program or depends on when the patch is applied. A full investigation of remediation strategies is an interesting direction of future work.

VI. Experimental Setup

Using our testing framework, we set out to empirically measure the permissiveness and restrictiveness of the AS and CFS checks. Secondarily, we were interested in the practical effectiveness of our minimization algorithm. In this section we describe our experimental setup: which applications we considered, how we modified the applications to make them amenable to update testing, and which test suites we used.

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 5 summarizes the versions of each application that we consider. We largely re-use the dynamic patches and program versions of OpenSSH and vsftpd used by Neamtiu et al. in their Ginseng work [21], with some changes that we describe in the next section. The OpenSSH releases range from Oct. 2002 to

	#	Version	LoC	Tsts	Δ to next ver		
					Sig	Fun	Type
OpenSSH	0	3.5p1	46,735	75	3	98	5
	1	3.6.1p1	48,459	75	0	6	0
	2	3.6.1p2	48,473	76	5	238	11
	3	3.7.1p1	50,448	91	0	18	0
	4	3.7.1p2	50,460	91	13	172	10
	5	3.8p1	51,822	104	0	24	1
	6	3.8.1p1	51,838	104	6	257	10
	7	3.9p1	53,260	104	4	179	12
	8	4.0p1	56,068	105	0	72	3
	9	4.1p1	56,104	104	10	157	7
	10	4.2p1	57,294	(Not patched)			
vsftpd	0	2.0.0	13,048	13	0	6	0
	1	2.0.1	13,059	13	1	12	0
	2	2.0.2pre2	13,114	13	0	21	0
	3	2.0.2pre3	14,293	13	0	76	0
	4	2.0.2	16,970	13	0	10	1
	5	2.0.3	12,977	13	0	25	1
	6	2.0.4	14,427	14	0	100	2
	7	2.0.5	14,482	13	0	93	2
	8	2.0.6	14,785	(Not patched)			
ngircd	0	0.5.0	8,157	10	0	6	0
	1	0.5.1	8,160	10	0	23	1
	2	0.5.2	8,161	10	12	28	2
	3	0.5.3	8,178	10	1	17	2
	4	0.5.4	8,211	10	4	104	8
	5	0.6.0	9,302	10	0	24	0
	6	0.6.1	9,333	10	2	79	4
	7	0.7.0	10,043	(Not patched)			

Fig. 5. Version and patch information

Sept. 2005, and vsftpd releases range from July 2004 to Feb. 2008. We also developed patches for eight versions of ngIRCd released from Sept. 2002 to May 2003. To make it easy to refer to the versions in the subsequent discussion, we number them starting from 0. For each version, Figure 5 lists the total lines of code (measured with sloccount), the number of update tests (drawn from unmodified and hybrid system 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.

B. Update point selection

As mentioned earlier, updates can take effect at calls to DSU_update(), where these calls can be inserted manually or automatically. To consider the effectiveness of AS and CFS, we direct Ginseng to automatically insert a call to DSU_update() prior to each function call, and systematically test the outcome of performing an update at each of these points. We refer to this set of dynamic update points as *All Pts*. We separately consider the subset of these update points that satisfy the AS and CFS safety checks.

As a last point of comparison, we consider the results of update tests with manually selected update points. In particular, when preparing vsftpd and OpenSSH to support updating, Neamtiu et al. chose to place a single DSU_update() at the beginning of the connection loop. They argued that updates that occur at *quiescent points*, i.e., places where there are no in-flight operations, are more likely to succeed than arbitrarily chosen points [21]. We decided to test this claim by seeing whether our tests would pass for these points, and determine whether these points would permit updates often enough. In addition to the points advocated by Neamtiu et al., we added an additional manual update point into 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. In preparing ngIRCd for updating, we similarly inserted a manual update point to be hit in between iterations of the main server loop.

C. Program and patch modifications

The program code and patches developed by Neamtiu et al. had been prepared by extracting the connection loop and its cleanup code, as described in Section II-B, so that each connection loop iteration would execute the most recent code. We additionally extracted the command loops and cleanup code to ensure a similar semantics; Neamtiu et al. did not do this because they did not consider updates during command processing.

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 they all include 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 in the chain;
               //    last one is the 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. Because these programs were prepared for updating at the main loops, the patches 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 felt it reasonable to relax the AS check by also extracting the startup code, so that it is no longer on the stack when the loop executes. Had we not done this, AS would not have supported the updates in our benchmarks.[3]

We also found the CFS check to be unreasonably restrictive in one instance. To implement CFS, Ginseng uses a static analysis. Unfortunately, this analysis over-approximates the set of possible calls through a table of function pointers, and as such spuriously forbids updates within the OpenSSH command loops. Rather than strengthen the analysis to avoid this imprecision, we performed some additional code extractions so that updates within the command loop would pass the CFS check. These extractions have no bearing on the behavior of the updated execution, and serve merely to overcome the conservatism of the analysis.

D. Test Suites

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

[3]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.

As vsftpd is not distributed with any system tests, we constructed 13 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 a significant portion of its functionality.

We also developed a suite of 10 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. All tests in this suite apply to all tested versions of ngIRCd.

E. Running Tests

For each test execution, we record whether the test passed or failed. We mark 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.

To compile complete testing results for a program and a patch, we disabled all safety checks and used DSUTest to gather results for applying the patch at each update point reached in the test suite. We utilized our test minimization algorithm (Section IV) to determine which update tests should actually be performed and then scaled the results back up to the full set of points. Having done this, we used the traces produced during testing along with information about the patch contents to retroactively determine which points would be allowed under each safety check.

VII. Experimental Results

This section presents our experimental results. Our goal was to understand the effectiveness of the safety checks, in terms of permissiveness (failing to prevent incorrect behavior) and restrictiveness (failing to allow correct behavior), compared to using no checks at all, and compared to allowing updates only at manually placed positions. We also look at the effectiveness of our minimization algorithm in reducing the number of update tests that must be executed to achieve full update coverage.

A. Test Failures/Points Allowed

Figure 6 summarizes the number of update points allowed by each safety check for each patch to OpenSSH, vsftpd, and ngIRCd, and how many of those points resulted in a failing test.

	Update	All Pts		CFS		AS		Manual	
		Failed	Total	Failed	Total	Failed	Total	Failed	Total
OpenSSH	0→1	19,715	580,871	0	68,044	0	35,314	0	566
	1→2	0	705,322	0	705,322	0	587,578	0	630
	2→3	306,965	638,720	1,688	75,307	4	20,902	0	568
	3→4	0	772,198	0	772,198	0	638,803	0	783
	4→5	565,681	773,086	609	110,633	380	21,343	0	782
	5→6	10,703	878,235	0	130,000	0	111,950	0	860
	6→7	163,333	879,668	44,461	96,183	110	44,278	0	859
	7→8	11,380	918,717	1	80,070	1	100,854	0	850
	8→9	3	973,364	0	261,885	0	61,724	0	868
	9→10	357,919	933,514	24	121,337	0	61,051	0	833
	Total	**1,435,699**	**8,053,695**	**46,783**	**2,420,979**	**495**	**1,683,797**	**0**	**7,599**
vsftpd	0→1	0	210,142	0	210,142	0	102,307	0	80
	1→2	2,462	210,142	558	90,073	0	69,775	0	80
	2→3	0	215,223	0	215,223	0	55,555	0	80
	3→4	0	220,564	0	220,564	0	37,265	0	80
	4→5	43,233	218,586	546	4,478	0	2,123	0	80
	5→6	58	223,098	0	24,924	0	67,330	0	80
	6→7	2,115	233,199	0	3,737	0	7,437	0	82
	7→8	234	222,296	0	1,993	0	3,098	0	80
	Total	**48,102**	**1,753,250**	**1,104**	**771,134**	**0**	**344,890**	**0**	**642**
ngIRCd	0→1	0	86,090	0	86,090	0	59,160	0	105
	1→2	0	98,603	0	97,526	0	52,149	0	110
	2→3	6,980	98,603	0	690	0	130	0	110
	3→4	348	99,184	0	691	0	131	0	111
	4→5	44,265	86,009	0	576	0	116	0	96
	5→6	0	118,786	0	118,786	0	44,220	0	126
	6→7	49,649	112,283	0	256	0	136	0	116
	Total	**101,242**	**699,558**	**0**	**304,615**	**0**	**156,042**	**0**	**774**

Fig. 6. Test failures/points allowed

The *All Pts* column of Figure 6 lists over 1.4M failing update points out of 8M total (17.8%) for OpenSSH, over 48K failing runs out of 1.7M total (2.7%) for vsftpd, and over 101K failing runs out of nearly 700K total (15.5%) for ngIRCd. This is clear evidence that applying updates indiscriminately is extremely risky. 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 no failures. However, it is worth noting that these patches also contained relatively few overall changes. 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 most vsftpd failures.

The *CFS* and *AS* columns of Figure 6 illustrate that both checks succeed at dramatically reducing (but not eliminating) the total number of failures, while still allowing a significant number of update points. For both vsftpd and OpenSSH, 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 and 61K to 1.1K (97.7% reduction) for vsftpd. AS performed even better, allowing only 495 failures (well over 99.9% reduction) for OpenSSH and no failures for vsftpd. Both safety checks prevented all update failures to ngIRCd. On the other hand CFS is generally more permissive, allowing far more update points compared to AS for all three applications. *Manual* exhibited no test failures, but has many fewer allowed update points.

The left half of Figure 7 classifies each failing update point based on which safety checks would have prevented the failure, while the right half of the figure shows how many update points that pass all test cases are allowed by the checks. We break down the results into four basic categories, one per row in each of the table, visualized in the Venn diagram above it.

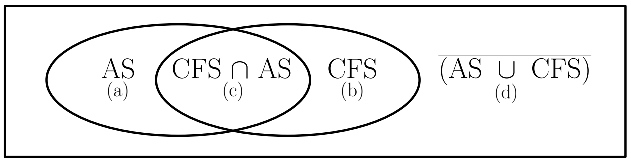
For vsftpd, OpenSSH, and ngIRCd, we see that well over 95% of the failing points would be disallowed by both safety checks (row (c)). We manually examined several of these failures, and found type safety violations to be the most common cause. Indeed, since both AS and CFS ensure updates are type safe, it seems likely that a large portion of the failures are due to type errors. The next largest category of failures are those that are allowed by CFS but disallowed by AS (row (a)), and no failures are prevented only by CFS (row (b)). Lastly, fewer than 1% of the failures for each application are allowed under all safety checks (row (d)). These last three categories of failures are those where mistimed updates are type safe but violate some other program logic. We discuss several examples of these failures in detail in Section VII-B.

Turning to the right half of Figure 7, we see that nearly half or more of the possible update points are allowed by both checks (row (c)), and that of the remaining points CFS permits many more than AS (rows (b) and (a), respectively). These results suggest CFS provides more update availability than AS.

B. Observed Failures

While our experiments show that the AS and CFS safety checks are quite effective, they are not perfect. Therefore we felt it would be useful to examine the remaining failures allowed by these checks, to understand their limitations, and to identify future research directions.

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



	Category	Failures	% Failures
OpenSSH	(a) Only Prevented by AS	46,288	3%
	(b) Only Prevented by CFS	0	0%
	(c) Prevented by AS and CFS	1,388,916	97%
	(d) Prevented by Neither	495	< 1%
	(a+b+c+d) Total Failures	1,435,699	100%
vsftpd	(a) Only Prevented by AS	1,104	2%
	(b) Only Prevented by CFS	0	0%
	(c) Prevented by AS and CFS	46,998	98%
	(d) Prevented by Neither	0	0%
	(a+b+c+d) Total Failures	48,102	100%
ngIRCd	(a) Only Prevented by AS	0	0%
	(b) Only Prevented by CFS	0	0%
	(c) Prevented by AS and CFS	101,242	100%
	(d) Prevented by Neither	0	0%
	(a+b+c+d) Total Failures	101,242	100%

Failures Prevented

	Category	Successes	% Successes
OpenSSH	(a) Only Allowed by AS	102,076	2%
	(b) Only Allowed by CFS	792,970	12%
	(c) Allowed by AS and CFS	4,141,724	63%
	(d) Allowed by Neither	1,581,226	24%
	(a+b+c+d) Total Passing	6,617,996	100%
vsftpd	(a) Only Allowed by AS	106,234	6%
	(b) Only Allowed by CFS	531,374	31%
	(c) Allowed by AS and CFS	828,884	49%
	(d) Allowed by Neither	238,656	14%
	(a+b+c+d) Total Passing	1,705,148	100%
ngIRCd	(a) Only Allowed by AS	0	0%
	(b) Only Allowed by CFS	148,573	25%
	(c) Allowed by AS and CFS	293,701	49%
	(d) Allowed by Neither	156,042	26%
	(a+b+c+d) Total Passing	598,316	100%

Successes Allowed

Fig. 7. Breakdown of results by safety check

safety. However, as we mentioned in Section II-C, type-safe executions may nevertheless fail, and indeed we observed cases of this. One example occurred while testing upload operations against the 1→2 patch to vsftpd. Figure 8 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, 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 failure we just saw, as discussed in Section II-C, it does not prevent such version consistency problems entirely. A particularly interesting example occurs in the 4→5 patch of OpenSSH. This example involves a version consistency violation that was not present in the original code, but was introduced via a code

```
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. 8. 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. 9. Skipped initialization error

extraction step that is needed to permit many other, safe updates to occur.

Figures 9(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 9(a) is taken, then global_ptr will be uninitialized when dereferenced, leading to a segfault. 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 VI that we extracted the "startup" code in all functions leading up to the command loops in our subject programs. Consider Figures 9(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.

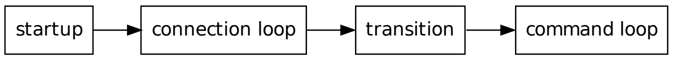
Failures allowed by AS: Although we encountered no instances of failures that are prevented by CFS but allowed by AS in our experiment, such cases are theoretically possible. More specifically, Ginseng's static analysis is conservative, and this conservatism could cause Ginseng to disallow an update point that is allowed by AS. If that update point induced a version consistency error, we would then see a failure prevented by CFS but allowed by AS.

C. Update points per program phase

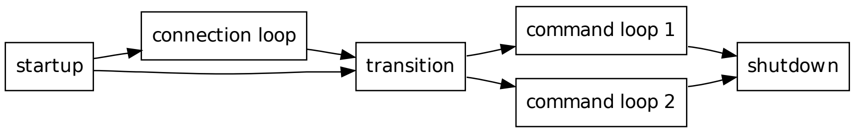
Generally speaking, while allowing more correct update points is better than fewer, it also matters where those update points occur during program execution. In particular, since the majority of each server's execution takes place within one of a few long-running loops, it is crucial that a safe update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

To get a more refined view of where updates are allowed and where they fail, we have broken down the execution of our benchmark programs into phases corresponding to their long-running loops and the transitions between them.

The execution of vsftpd consists of a connection loop that accepts session requests and forks child processes to handle them, and a command loop in each child process that receives, processes, and responds to requests from the client. In addition, vsftpd includes a startup phase that initializes and configures the server state, and a transition phase that performs some per-connection initialization. Transitions between phases occur as follows:



We have identified a similar set of phases for OpenSSH. The key differences are the presence of two command loop phases that handle requests for different protocol versions, a brief shutdown phase to handle cleanup after a client connection ends, and the possibility of skipping the connection loop under certain configurations. The transitions between phases for OpenSSH occur as follows:



Unlike vsftpd and OpenSSH which fork new processes to service client connections independently, ngIRCd employs a simpler program structure where new connections and requests from existing connections are serviced by the main server loop running in a single process. In our test executions, we observed two distinct phases: the main loop and the preceding startup execution.

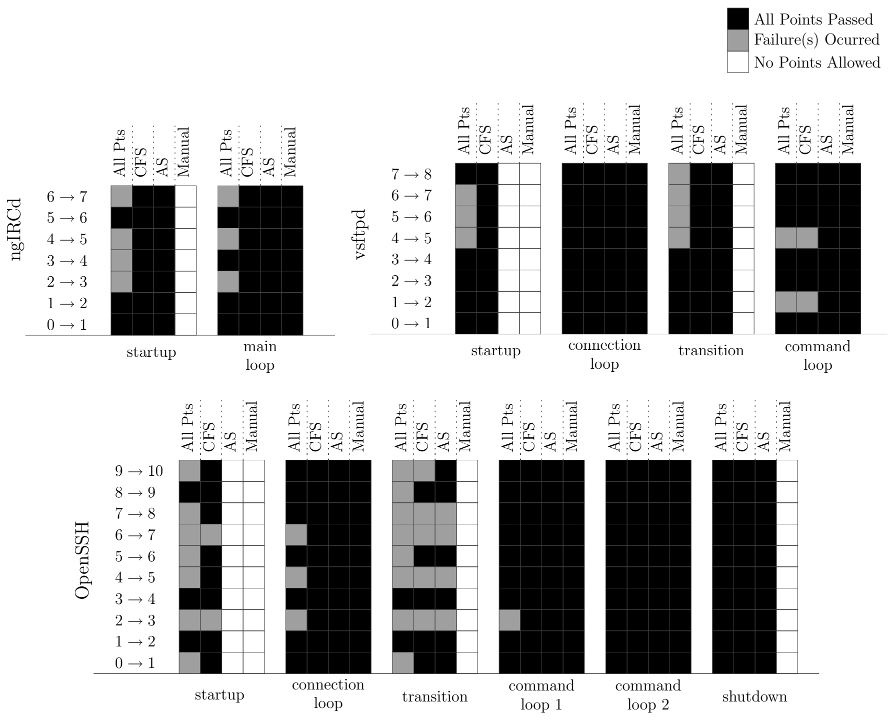


Fig. 10. Updatability across program phases

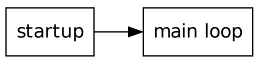


Figure 10 summarizes test failures by program phase and patch (the full tables from which this figure is derived can be found in Figures 13, 14, and 15 in the Appendix). Black boxes indicate that all tests pass and grey boxes indicate one or more failures. White boxes indicate that no allowable update points were reached during execution of the particular phase.

We can see that CFS allows at least one update point in each program phase, while AS precludes updates during the startup phases for OpenSSH and vsftpd. There are no manually placed update points in the startup and transition phases. In all cases, updates are permitted within the command and connection loop phases, ensuring reasonable availability to updates. Moreover, for Manual and AS, no failures occur at update points within the loops, while for CFS, the only loop-phase failures occur in the vsftpd command loop. This is interesting because, as just discussed, updates within the loops are most important, while updates within the startup or transition phases are much less so, since these phases are finite and presumably short.

	Update	All Pts	CFS	AS	Manual
OpenSSH	0 → 1	580,871 → 31,791 (95%)	68,044 → 3,687 (95%)	35,314 → 3,027 (91%)	566 → 566 (0%)
	1 → 2	705,322 → 1,795 (∼100%)	705,322 → 1,795 (∼100%)	587,578 → 1,717 (∼100%)	630 → 592 (6%)
	2 → 3	638,720 → 63,011 (90%)	75,307 → 5,454 (93%)	20,902 → 2,353 (89%)	568 → 568 (0%)
	3 → 4	772,198 → 4,324 (99%)	772,198 → 4,324 (99%)	638,803 → 3,775 (99%)	783 → 770 (2%)
	4 → 5	773,086 → 27,399 (96%)	110,633 → 4,592 (96%)	21,343 → 1,564 (93%)	782 → 782 (0%)
	5 → 6	878,235 → 17,398 (98%)	130,000 → 1,292 (99%)	111,950 → 1,723 (98%)	860 → 841 (2%)
	6 → 7	879,668 → 47,092 (95%)	96,183 → 4,568 (95%)	44,278 → 2,139 (95%)	859 → 859 (0%)
	7 → 8	918,717 → 89,601 (90%)	80,070 → 3,925 (95%)	100,854 → 4,141 (96%)	850 → 850 (0%)
	8 → 9	973,364 → 34,293 (96%)	261,885 → 5,467 (98%)	61,724 → 2,070 (97%)	868 → 823 (5%)
	9 → 10	933,514 → 52,356 (94%)	121,337 → 3,424 (97%)	61,051 → 2,891 (95%)	833 → 833 (0%)
	Total	**8,053,695 → 369,060 (95%)**	**2,420,979 → 38,528 (98%)**	**1,683,797 → 25,400 (98%)**	**7,599 → 7,484 (2%)**
vsftpd	0 → 1	210,142 → 26 (∼100%)	210,142 → 26 (∼100%)	102,307 → 26 (∼100%)	80 → 13 (84%)
	1 → 2	210,142 → 516 (∼100%)	90,073 → 514 (99%)	69,775 → 166 (∼100%)	80 → 67 (16%)
	2 → 3	215,223 → 1,122 (99%)	215,223 → 1,122 (99%)	55,555 → 553 (99%)	80 → 67 (16%)
	3 → 4	220,564 → 3,866 (98%)	220,564 → 3,866 (98%)	37,265 → 1,912 (95%)	80 → 80 (0%)
	4 → 5	218,586 → 19,893 (91%)	4,478 → 1,196 (73%)	2,123 → 301 (86%)	80 → 80 (0%)
	5 → 6	223,098 → 15,910 (93%)	24,924 → 3,485 (86%)	67,330 → 3,567 (95%)	80 → 67 (16%)
	6 → 7	233,199 → 200,653 (14%)	3,737 → 1,433 (62%)	7,437 → 2,742 (63%)	82 → 68 (17%)
	7 → 8	222,296 → 10,371 (95%)	1,993 → 353 (82%)	3,098 → 275 (91%)	80 → 80 (0%)
	Total	**1,753,250 → 252,357 (86%)**	**771,134 → 11,995 (98%)**	**344,890 → 9,542 (97%)**	**642 → 522 (19%)**
ngIRCd	0 → 1	86,090 → 423 (∼100%)	86,090 → 423 (∼100%)	59,160 → 104 (∼100%)	105 → 27 (74%)
	1 → 2	98,603 → 827 (99%)	97,526 → 827 (99%)	52,149 → 206 (∼100%)	110 → 75 (32%)
	2 → 3	98,603 → 1,766 (98%)	690 → 140 (80%)	130 → 130 (0%)	110 → 110 (0%)
	3 → 4	99,184 → 1,578 (98%)	691 → 141 (80%)	131 → 131 (0%)	111 → 111 (0%)
	4 → 5	86,009 → 32,068 (63%)	576 → 136 (76%)	116 → 116 (0%)	96 → 96 (0%)
	5 → 6	118,786 → 119 (∼100%)	118,786 → 119 (∼100%)	44,220 → 79 (∼100%)	126 → 61 (52%)
	6 → 7	112,283 → 30,702 (73%)	256 → 136 (47%)	136 → 136 (0%)	116 → 116 (0%)
	Total	**699,558 → 67,483 (90%)**	**304,615 → 1,922 (99%)**	**156,042 → 902 (99%)**	**774 → 596 (23%)**

Fig. 11. Update point minimization

D. Minimization Effectiveness

Figure 11 illustrates the effectiveness of our minimization algorithm at reducing the number of update tests to run for our benchmarks. In each column, we show the original count to the left of the arrow, the minimized count to the right of it, and the percent reduction in parentheses. The *All Pts* column shows the reduction for the full set of update points that are reached during the execution of each application's test suite. Overall, 95% of update points from OpenSSH, 86% of points from vsftpd, and 90% of points from ngIRCd could be eliminated. This is significant because the initial number of tests was very large: over 8M for OpenSSH, 1.7M for vsftpd, and just under 700K for ngIRCd. For 22 of the 25 patches across all three applications reductions of over 90% were achieved. Patches 4→5 and 6→7 of ngIRCd were the largest ngIRCd patches (by number of changed functions) and reductions of 63% and 73%, respectively, were achieved. The final exception was the 6→7 patch to vsftpd, for which only a 14% reduction was possible. This particular patch included complex state transformation code that accesses a large number of global variables. When a variable may be read or written during state transformation, update points before and after an access to that variable in the program trace cannot be considered equivalent. In general, the amount of reduction is roughly inversely proportional to the size of the patch and the particular tests being run.

In practice, it is only necessary to test update points that are allowed by the safety check in use. The *CFS* and *AS* columns show the number of points allowed under these models and the further reduction achieved by our algorithm. The combination yields a significant reduction: overall 98.5% of the CFS-safe and 98.3% of the AS-safe points could be eliminated leaving only a tiny fraction of the original set of update points. In the worst case, the number of tested points to achieve full update coverage for any single patch to our benchmark applications under CFS was 5,467 for patch 8→9 of OpenSSH and under AS was 4,141 for patch 7→8 of OpenSSH, out of well over 900K original points in each case.

The manually introduced update points are a small fraction of those in *All*, and we found the minimization algorithm to be ineffective at further reducing these points. This is because the manually inserted update points occur once per iteration of the long-running loops of the program and so many function calls may occur between iterations, increasing the chances of a conflict. In effect, we may view manual update point selection as a highly-effective form of reduction in itself as no OpenSSH patch would require more than 870 tests, no vsftpd patch would require over 82 tests, and no ngIRCd patch would require more than 126 update tests for full update coverage.

Our minimization algorithm was critical in enabling us to perform our experiments. As an example, testing of these reduced points for OpenSSH (a 95% reduction of the full test suite) still required approximately 600 CPU hours to complete.

E. Threats to Validity

There are several potential threats to the validity of our 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. Second, our empirical study is currently limited to these three applications. As such, our results may not generalize. We believe we have explored enough tests and mature enough applications to strongly suggest the general trend. Third, the fact that we changed the applications slightly to make the safety checks more permissive also challenges the generality of our results. As discussed earlier, we believe the changes are ones that developers using these safety checks would have reasonably made so as to ensure a proper level of updatability. Lastly, because update points within ignore regions are not tested, bugs in patches may not be found during test. For this reason, we have manually inspected these regions and attempted to minimize their size. This threat could be completely mitigated by continuing to prevent updates within ignore regions after the application is deployed.

VIII. Related work

Gupta et al. [10] 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 safety checks on a program and patch that are sufficient to ensure validity, but only under limited circumstances. For example, Gupta's check only applies when a patch adds new functionality and programs do not use complex data types and pointers. As described in Section II-C, more practical DSU implementations tend to use either the AS and CFS checks. Our study is the first to provide empirical data on the effectiveness of these checks in practical situations.

Many DSU systems allow programmers to further restrict update timing, rather than rely wholly on automatic safety checks. For example, as already mentioned, using Ginseng [21], programmers can provide a *whitelist* of program locations (e.g., line numbers) that are valid for an update; DLpop [12] has similar behavior. Conversely, Lee [15], Gupta et al. [10], Chen et al. [7], and others support a *blacklist* (e.g., by requiring that certain functions must be inactive prior to updating), indicating particular points that are not allowed. We leave more detailed empirical study of these mechanisms to future work.

Our approach to generating update tests is related to Chess [18] and MultithreadedTC [22], 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 [20]. 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 the test case minimization.

IX. Conclusions

We have presented an empirical evaluation of two well-known DSU safety checks, activeness safety and con-freeness safety. Our evaluation is based on systematically testing updates to OpenSSH, vsftpd, and ngIRCd, using a novel DSU testing framework we developed. This framework is noteworthy in that it can systematically consider 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.

Our study found that updating without the use of safety checks resulted in a large number of failures, and that both AS and CFS were able to eliminate the vast majority, but not all, of these failures. AS was the more restrictive check as it prevented more failures but also more successes than CFS. Indeed, we have to manually make slight modifications to the programs in order to accommodate AS, or it would have been so restrictive as to prevent nearly all useful updates.

Our study suggests several lines of future work. One idea is to attempt combine the best features of AS and CFS. For example, a relaxed version of AS could allow updates that change active functions, as long as any on-stack code that can be subsequently executed is the same in both the old and new version. This variant of AS is more restrictive than CFS and would avoid the need for code extractions in some cases, e.g., the preamble of main, that are now required for AS to be of any use.

On the other hand, developing an automatic safety check that eliminates all failures while being sufficiently permissive is probably out of reach in the general case. It would be interesting to explore whether such a check is feasible for certain simple cases, such as for small code changes for the purposes of security fixes [4], [1]. For the general case, we may attempt to limit our consideration to a few possible update points that are reached sufficiently often; our experimental results showed that this approach works well. An interesting research direction is to attempt to find sufficiently many update points to ensure any dynamic update will eventually take effect, e.g., applying techniques similar to those used to prove termination via reachability [8]. Once these points are identified, our novel systematic testing framework should prove useful in helping developers test their dynamic patches under realistic circumstances.

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, R. Virding, C. Wikstrom, and M. Williams. *Concurrent programming in ERLANG (2nd ed.)*. Prentice Hall International Ltd., 1996.

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

[5] G. M. Bierman, M. J. Parkinson, and J. Noble. UpgradeJ: Incremental typechecking for class upgrades. In *ECOOP*, 2008.

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

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

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

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

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

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

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

[16] K. Makris and R. Bazzi. Immediate multi-threaded dynamic software updates using stack reconstruction. Technical Report TR-08-007, Arizona State University, 2008.

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

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

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

[20] 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.

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

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

[23] 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.

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

[25] Unsanity. Application Enhancer – enhance the applications by loading modules. http://www.unsanity.com/haxies/ape.

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

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

Michael Hicks is an Assistant 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 Assistant 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.

APPENDIX A
TRACE SEMANTICS

Figure 12 gives the operational semantics rules for our formal language (given in Figure 2). The semantics are expressed as a relation $(H, s) \longrightarrow^{\nu} H'$, where H and s are the current heap and statement, H' is the heap after s has been completely executed, and an *event trace*, described below. We also need a sibling relation $(H, e) \longrightarrow^{\nu} (H', c)$ for expressions, where c is the result of computing the expression e. Following is a discussion of the three most interesting evaluation rules: FUN-CALL, UPD-SKIP, and UPD-TAKEN.

The FUN-CALL rule describes the semantics of expressions of the form $f(e_1, ..., e_n)$. First, we evaluate arguments e_1 through e_n. Next, we look up f's definition in the resulting H. Then we set e' to be e (the body of f), but with all occurrences of its formal parameters x_n replaced by the actual arguments c_n. We then evaluate e' to produce c, which is returned by the call. The trace ν' computed by the function call is composed of the traces produced by evaluating each of the arguments, the *call*$(f(...))$ event signifying the function call, and the trace produced by evaluating the function's body.

The semantics of update are non-deterministic, allowing us to either skip or take an update. In the former case we apply UPD-SKIP, which treats update like skip but produces a *noupdate* event. In the latter case we apply UPD-TAKEN, which produces a new program H' with bindings in π replacing or adding to those in H. For example, if given (H, s) where s is $\mathsf{f}(2); \mathsf{update}; \mathsf{f}(3)$, then the first call to f would use $H(\mathsf{f})$, and if we executed update using UPD-TAKEN, then the second call to f would use $H'(\mathsf{f})$. Note that we have not specified where π comes from in this rule, as this models the unanticipated nature of dynamic update. Nondeterminism does not affect our formal reasoning about this system. In practice, when testing a patch we choose π and the position in the trace at which to apply UPD-TAKEN before we execute a test.

APPENDIX B
FULL RESULTS

Following are tables showing the complete breakdown of program failures across each application, safety check, and program phase. These results are summarized in the text in Figures 6 and 10.

FUN-CALL
$$\frac{\begin{array}{c}(H, e_1) \longrightarrow^{\nu_1} (H_1, c_1) ... (H_{n-1}, e_n) \longrightarrow^{\nu_n} (H_n, c_n) \qquad H_n(f) = \lambda(x_1, ..., x_n).e \\ e' = e[x_n \mapsto c_n] \text{ for all } n \qquad (H_n, e') \longrightarrow^{\nu} (H', c) \qquad \nu' = \nu_1; ...; \nu_n; \mathit{call}(f(c_1, ..., c_n)); \nu\end{array}}{(H, f(e_1, ..., e_n)) \longrightarrow^{\nu'} (H', c)}$$

UPD-SKIP
$$\frac{}{(H, \mathsf{update}) \longrightarrow^{\mathit{noupdate}} H}$$

UPD-TAKEN
$$\frac{H' = H[x \mapsto \pi(x)] \text{ for all } x \text{ in } dom(\pi)}{(H, \mathsf{update}) \longrightarrow^{\mathit{update}(\pi)} H'}$$

EXPR-SEQ
$$\frac{(H, s) \longrightarrow^{\nu_1} H' \qquad (H', e) \longrightarrow^{\nu_2} (H'', c)}{(H, s; e) \longrightarrow^{\nu_1;\nu_2} (H'', c)}$$

STMT-SEQ
$$\frac{(H, s_1) \longrightarrow^{\nu_1} H' \qquad (H', s_2) \longrightarrow^{\nu_2} H''}{(H, s_1; s_2) \longrightarrow^{\nu_1;\nu_2} H''}$$

ASGN
$$\frac{(H, e) \longrightarrow^{\nu} (H', c) \qquad H'' = H'[x \mapsto c] \qquad \nu' = \nu; \mathit{write}(x, c)}{(H, x := e) \longrightarrow^{\nu'} H''}$$

VAR
$$\frac{H(x) = c}{(H, x) \longrightarrow^{\mathit{read}(x,c)} (H, c)}$$

COND-TRUE
$$\frac{(H, e) \longrightarrow^{\nu_1} (H', c) \qquad c \neq 0 \qquad (H', s1) \longrightarrow^{\nu_2} H''}{(H, \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2) \longrightarrow^{\nu_1;\nu_2} H''}$$

COND-FALSE
$$\frac{(H, e) \longrightarrow^{\nu_1} (H', 0) \qquad (H', s2) \longrightarrow^{\nu_2} H''}{(H, \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2) \longrightarrow^{\nu_1;\nu_2} H''}$$

LOOP-TRUE
$$\frac{\begin{array}{c}(H, e) \longrightarrow^{\nu_1} (H', c) \qquad c \neq 0 \\ (H', s) \longrightarrow^{\nu_2} H'' \qquad (H'', \mathsf{while}\ e\ \mathsf{do}\ s) \longrightarrow^{\nu_3} H'''\end{array}}{(H, \mathsf{while}\ e\ \mathsf{do}\ s) \longrightarrow^{\nu_1;\nu_2;\nu_3} H'''}$$

LOOP-FALSE
$$\frac{(H, e) \longrightarrow^{\nu} (H', 0)}{(H, \mathsf{while}\ e\ \mathsf{do}\ s) \longrightarrow^{\nu} H'}$$

SKIP
$$\frac{}{(H, \mathsf{skip}) \longrightarrow^{\mathit{skip}} H}$$

Fig. 12. Operational semantics

	Update	All Pts		CFS		AS		Manual	
		Failed	Total	Failed	Total	Failed	Total	Failed	Total
OpenSSH init	0→1	7,226	68,141	0	25,455	no pts		no pts	
	1→2	0	90,776	0	90,776	no pts		no pts	
	2→3	10,830	87,569	906	32,703	no pts		no pts	
	3→4	0	103,035	0	103,035	no pts		no pts	
	4→5	9,191	103,035	0	38,010	no pts		no pts	
	5→6	10,596	116,164	0	47,455	no pts		no pts	
	6→7	108,669	116,872	44,351	47,691	no pts		no pts	
	7→8	11,222	138,750	0	1,572	no pts		no pts	
	8→9	0	153,985	0	71,880	no pts		no pts	
	9→10	2	149,279	0	45,477	no pts		no pts	
	Total	**157,736**	**1,127,606**	**45,257**	**504,054**	no pts		no pts	
OpenSSH mainloop	0→1	0	1,151	0	48	0	48	0	24
	1→2	0	1,196	0	1,196	0	1,196	0	27
	2→3	2	1,121	0	44	0	44	0	22
	3→4	0	1,187	0	1,187	0	1,187	0	24
	4→5	46	1,172	0	46	0	46	0	23
	5→6	0	1,636	0	70	0	70	0	35
	6→7	212	1,636	0	70	0	70	0	35
	7→8	0	4,234	0	68	0	68	0	34
	8→9	0	4,694	0	2,254	0	78	0	39
	9→10	0	4,396	0	72	0	72	0	36
	Total	**260**	**22,423**	**0**	**5,055**	**0**	**2,879**	**0**	**299**
OpenSSH transition	0→1	12,489	473,167	0	28,847	0	32,503	no pts	
	1→2	0	572,337	0	572,337	0	550,537	no pts	
	2→3	290,876	510,969	782	29,076	4	8,954	no pts	
	3→4	0	618,569	0	618,569	0	588,380	no pts	
	4→5	556,444	619,472	609	33,954	380	9,363	no pts	
	5→6	107	705,534	0	41,043	0	57,056	no pts	
	6→7	54,452	706,432	110	32,746	110	38,359	no pts	
	7→8	158	721,499	1	45,421	1	67,627	no pts	
	8→9	3	759,782	0	146,387	0	48,551	no pts	
	9→10	357,917	726,649	24	35,608	0	45,554	no pts	
	Total	**1,272,446**	**6,414,410**	**1,526**	**1,583,988**	**495**	**1,446,884**	no pts	
OpenSSH clientloop1	0→1	0	26,542	0	12,443	0	2,490	0	415
	1→2	0	28,593	0	28,593	0	23,425	0	479
	2→3	5,257	26,995	0	4,174	0	836	0	418
	3→4	0	37,537	0	37,537	0	37,537	0	632
	4→5	0	37,537	0	27,431	0	1,264	0	632
	5→6	0	42,904	0	30,215	0	42,904	0	698
	6→7	0	42,858	0	11,144	0	5,576	0	697
	7→8	0	42,640	0	22,128	0	22,128	0	692
	8→9	0	43,216	0	30,353	0	1,408	0	704
	9→10	0	41,075	0	28,937	0	4,020	0	670
	Total	**5,257**	**369,897**	**0**	**232,955**	**0**	**141,588**	**0**	**6,037**
OpenSSH clientloop2	0→1	0	10,759	0	1,232	0	254	0	127
	1→2	0	10,483	0	10,483	0	10,483	0	124
	2→3	0	10,852	0	8,665	0	10,125	0	128
	3→4	0	10,759	0	10,759	0	10,759	0	127
	4→5	0	10,759	0	10,651	0	10,651	0	127
	5→6	0	10,759	0	10,651	0	10,759	0	127
	6→7	0	10,759	0	3,991	0	254	0	127
	7→8	0	10,483	0	10,340	0	9,920	0	124
	8→9	0	10,576	0	10,470	0	10,576	0	125
	9→10	0	10,759	0	10,651	0	10,254	0	127
	Total	**0**	**106,948**	**0**	**87,893**	**0**	**84,035**	**0**	**1,263**
OpenSSH shutdown	0→1	0	1,111	0	19	0	19	no pts	
	1→2	0	1,937	0	1,937	0	1,937	no pts	
	2→3	0	1,214	0	645	0	943	no pts	
	3→4	0	1,111	0	1,111	0	940	no pts	
	4→5	0	1,111	0	541	0	19	no pts	
	5→6	0	1,238	0	566	0	1,161	no pts	
	6→7	0	1,111	0	541	0	19	no pts	
	7→8	0	1,111	0	541	0	1,111	no pts	
	8→9	0	1,111	0	541	0	1,111	no pts	
	9→10	0	1,356	0	592	0	1,151	no pts	
	Total	**0**	**12,411**	**0**	**7,034**	**0**	**8,411**	no pts	

Fig. 13. Test success and failure (OpenSSH Full)

	Update	All Pts		CFS		AS		Manual	
		Failed	Total	Failed	Total	Failed	Total	Failed	Total
vsftpd init	0→1	0	100,672	0	100,672	0	1,222	no pts	
	1→2	0	100,672	0	68,172	0	1,222	no pts	
	2→3	0	103,246	0	103,246	0	1,222	no pts	
	3→4	0	103,259	0	103,259	0	1,053	no pts	
	4→5	2,991	101,543	0	806	0	13	no pts	
	5→6	45	104,689	0	2,405	0	13	no pts	
	6→7	2,111	113,106	0	784	0	952	no pts	
	7→8	0	105,261	0	793	0	884	no pts	
	Total	**5,147**	**832,448**	**0**	**380,137**	**0**	**6,581**	no pts	
vsftpd mainloop	0→1	0	806	0	806	0	806	0	26
	1→2	0	806	0	741	0	806	0	26
	2→3	0	806	0	806	0	728	0	26
	3→4	0	949	0	949	0	715	0	26
	4→5	0	806	0	650	0	741	0	26
	5→6	0	806	0	650	0	780	0	26
	6→7	0	868	0	700	0	868	0	28
	7→8	0	806	0	650	0	52	0	26
	Total	**0**	**6,653**	**0**	**5,952**	**0**	**5,496**	**0**	**210**
vsftpd transition	0→1	0	36,270	0	36,270	0	27,885	no pts	
	1→2	0	36,270	0	7,579	0	27,846	no pts	
	2→3	0	37,505	0	37,505	0	18,603	no pts	
	3→4	0	37,648	0	37,648	0	10,803	no pts	
	4→5	15,503	37,258	0	533	0	1,261	no pts	
	5→6	13	37,336	0	7,618	0	19,513	no pts	
	6→7	4	45,174	0	612	0	5,509	no pts	
	7→8	234	37,492	0	442	0	2,054	no pts	
	Total	**15,754**	**304,953**	**0**	**128,207**	**0**	**113,474**	no pts	
vsftpd clientloop	0→1	0	72,394	0	72,394	0	72,394	0	54
	1→2	2,462	72,394	558	13,581	0	39,901	0	54
	2→3	0	73,666	0	73,666	0	35,002	0	54
	3→4	0	78,708	0	78,708	0	24,694	0	54
	4→5	24,739	78,979	546	2,489	0	108	0	54
	5→6	0	80,267	0	14,251	0	47,024	0	54
	6→7	0	74,051	0	1,641	0	108	0	54
	7→8	0	78,737	0	108	0	108	0	54
	Total	**27,201**	**609,196**	**1,104**	**256,838**	**0**	**219,339**	**0**	**432**

Fig. 14. Test success and failure (vsftpd Full)

	Update	All Pts		CFS		AS		Manual	
		Failed	Total	Failed	Total	Failed	Total	Failed	Total
ngIRCd init	0→1	0	39,810	0	39,810	0	39,220	no pts	
	1→2	0	41,080	0	41,080	0	39,760	no pts	
	2→3	60	41,080	0	580	0	20	no pts	
	3→4	348	41,090	0	580	0	20	no pts	
	4→5	613	40,080	0	480	0	20	no pts	
	5→6	0	56,460	0	56,460	0	620	no pts	
	6→7	46,701	56,460	0	140	0	20	no pts	
	Total	**47,722**	**316,060**	**0**	**139,130**	**0**	**79,680**	no pts	
ngIRCd mainloop	0→1	0	46,280	0	46,280	0	19,940	0	105
	1→2	0	57,523	0	56,446	0	12,389	0	110
	2→3	6,920	57,523	0	110	0	110	0	110
	3→4	0	58,094	0	111	0	111	0	111
	4→5	43,652	45,929	0	96	0	96	0	96
	5→6	0	62,326	0	62,326	0	43,600	0	126
	6→7	2,948	55,823	0	116	0	116	0	116
	Total	**53,520**	**383,498**	**0**	**165,485**	**0**	**76,362**	**0**	**774**

Fig. 15. Test success and failure (ngIRCd Full)