A Testing Based Empirical Study of Dynamic Software Update Safety Restrictions

Technical Report: CS-TR-4949
December 18, 2009

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

University of Maryland, College Park
{hayden,hardisty,mwh,jfoster}@cs.umd.edu

ABSTRACT

Recent years have seen significant advances in dynamic software updating (DSU) systems, which allow programs to be patched on the fly. Most DSU systems employ automatic safety checks to avoid applying a patch if doing so may lead to incorrect behavior. This paper presents what we believe is the first comprehensive empirical evaluation of the two most significant DSU safety checks: *activeness safety* (AS), which disallows patches that modify functions on the stack, and *con-freeness safety* (CFS), which allows modifications to active functions, but only when doing so will be type safe.

To measure the checks' effectiveness, we tested them against three years of updates to OpenSSH and vsftpd. We performed this testing using a novel DSU testing methodology that systematically applies updates throughout the execution of a test suite. After testing updates to both applications in this way, we tracked how often the safety checks allow updates and which updates result in test failures. We found that updating without safety checks produced many failures, and that both AS and CFS dramatically reduced, but did not fully eliminate, these failures. CFS yielded more failures than AS, but AS was more restrictive than CFS, disallowing far more successful updates. Our results suggest that neither AS nor CFS is likely suitable for general-purpose DSU on its own. Indeed, we found that selecting update points manually could avoid all failures while still permitting sufficient updates. Our results present a challenge and important insights for future work: to discover safe and sufficient update points fully automatically.

1. 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 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 [12, 9]. Unsanity's Application Enhancer [24] 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 [20, 7, 13, 23].

```
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

Figure 1: Two versions of a program

However, 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, in Figure 1 the two versions of bar 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 [20, 7, 23, 11], 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 [14, 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 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 [25], K42 [13], OPUS [1], Ksplice [4], and Jvolve [23].

Con-freeness safety (CFS). Stoyle et al. [22] 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 [20] uses CFS to ensure update type safety.

While 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.

In this paper, we present what we believe to be the first comprehensive 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 testing framework for Ginseng [20], a freely available DSU system for C programs. Ginseng's updating semantics are similar to many existing DSU systems, so we believe our results will generalize to other implementations. We used our framework to test AS's and CFS's effectiveness on three-years' worth of updates to vsftpd and OpenSSH, two popular open-source server programs that have been extensively studied in the DSU literature [20, 7, 16].

Our framework systematically applies a dynamic patch at every point during a program execution that could produce a different result, using a novel reduction algorithm to avoid redundant tests. We used this framework to execute a suite of system tests tracking whether, when executing a given test and applying a patch at a given 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.

After some preliminary experience with our framework, we found we needed to slightly refactor the test programs by extracting some code blocks into separate functions [20], to avoid AS precluding all possible updates (details are presented in the next section). After making these changes and running our tests, we found a number of interesting results. A multitude of updates fail if we use no safety checks—in total, 1.48M of the total 9.8M 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%).

We also measured how many of the passing update points AS and CFS allow. We found that roughly 60% are allowed by both.

In total, CFS permitted 76% of the passing update points, while AS permitted 62% of them, a difference of about 1.12M update tests. Thus we can see that AS's lower failure rates come at the cost of higher restrictiveness, compared to CFS.

In short, though (as theorized by Gupta) neither check eliminates all failures and permits all successes, both are highly effective in practice, eliminating nearly all failures while still allowing a majority of successes.

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 most 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. Moreover, we found that if we restrict updates to just a few manually specified points in the loops, then there are no test failures. This last finding suggests that user input should be a component of DSU safety checking.

In summary, our study sheds new light on the practical effectiveness of popular DSU safety checks. Based on our results, we believe that a combination of manually specified update points, automatic safety checks, and an additional assurance argument, such as the testing strategy we used for our experiments, will yield a practical, safe solution for dynamic software updating.

2. 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.

2.1 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 [23] and K42 [13] take a similar approach. In some systems, including POLUS [7], DLpop [11], 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 changed references to their 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 user-provided *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.

2.2 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.

UpStare [16], a recently developed DSU system, avoids this problem by allowing a running function to transition to its new version immediately after a patch is applied, without requiring that it exit first. An UpStare dynamic patch can specify 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 our example, it would be straightforward to match up equivalent PC values in the old and new versions, since all existing event processing code is the same and only new processing code is added. Note that, because the loop startup code of the new version will not have a chance to execute, the developer may need to specify state transformation code to be run at update time to mimic the effect of the new startup code.

Using Ginseng and other systems, we can achieve a similar effect by refactoring 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 [20]. 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. While efficacious, the drawback of using code extraction is that developers must anticipate which code to extract before deploying the program. In contrast, UpStare has no such requirement. Nevertheless, prior work has shown it to be largely straightforward to identify the loops and cleanup code that should be extracted to ensure proper semantics [20, 18, 7]. 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.

2.3 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 may 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, 13, 1, 7].

Activeness Safety usefully avoids several potential problems. First, it 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.

Activeness Safety also prevents some *version consistency errors*, which result from the execution of related code at different program versions, despite being type-correct [19]. For example, consider the following example program:

```
int *g = NULL;
void foo() {
  bar();
}
void bar() {
  g = malloc(...);
  *g++;
}
```

(a) Old version

```
int *g = NULL;
void foo() {
  g = malloc(...);
  bar();
}
void bar() {
  *g++;
}
```

(b) New version

Here, the "related code" is the initialization of g and its first use—initially, both occur in bar, but in the new version, the initialization is moved to foo. If we were allow the update to take effect at the start of (the old) foo, then the call to bar will go to the new version, which no longer initializes g. Thus this version-inconsistent execution would result in a crash. The AS check avoids this problem by preventing the update while foo is running.

On the other hand, AS will not eliminate all version consistency problems. Consider the following variation of the above example:

```
int *g = NULL;
void foo() {
   init ();
   bar();
}
void init () {
}
void bar() {
   g = malloc (...);
   *g++;
}
```

(a) Old version

```
int *g = NULL;
void foo() {
   init ();
   bar();
}
void init () {
   g = malloc (...);
}
void bar() {
   *g++;
}
```

(b) New version

An update just before the call to bar has the same negative effect as the earlier example, but in this case it will be allowed by AS, because foo has not changed between the versions. We found examples like this in our experiments (Section 6).

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 2.2 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. [22] 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 polls the run-time system to see if a dynamic update is available by invoking the function DSU_update(). 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, 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 creates opportunities for version consistency errors. Both of the examples of version consistency problems given above are possible with CFS, while only the second is possible with AS. Moreover, CFS may also introduce version consistency errors because it allows old code to still execute after an update.

The question we aim to address in this paper is: how often in practice do these problems arise? In other words, how often are AS and CFS too permissive, and/or too restrictive, when applying updates to practical programs? To answer this question we developed a methodology for systematically testing dynamic updates, which we describe next.

3. TESTING DYNAMIC UPDATES

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 purposes of our experiments, we do so using testing: if we update the old version in the middle of running a system test and the test still passes then we deem the update as correct; otherwise it is incorrect. While testing is an incomplete measure of correctness, system tests cover the most important program behaviors, and provide an easy-to-measure, practical assessment of whether an updated execution makes sense.

3.1 Testing Procedure

We can state the dynamic update testing problem 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. The challenge is determining what tests to execute, when to apply π, and how to check the result.

In what follows, we presume we can specifically enumerate those points at which a particular patch can be applied during a program's execution. In DSU systems like DLpop [11] and Ginseng [20], programmers can provide a *whitelist* of program locations (e.g., line numbers) that are valid for an update, while Lee [14], Gupta et al. [10], Chen et al. [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 if those that satisfy Ginseng's CFS check.

Our approach to update testing is to start with the system test suites for P_0 and P_1 and from them generate *update tests* for a patch π. How we choose tests from the test suites will be described shortly, and for now we assume P_0's execution is deterministic for the chosen tests; we discuss non-determinism in Section 3.3.

For a deterministic test t, we can unambiguously enumerate the update points that arise during the test's execution. We define t^i_π to be the update test that executes P_0 on t, applying π at the i^{th} update point. To run such 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^i_π for a fixed π.

Now we describe how we generate update tests from system tests. 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^i_π 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^i_π 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^i_π 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' as above. At the tester's discretion, a hybrid test could also specify behaviors that as 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.

3.2 Test suite minimization

The procedure described in Section 3 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.

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

Figure 2: Syntax of programs and event traces

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.

Formal language. 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 evaluate s and then e, returning the result of the latter. Statements consist of assignment $x := e$, sequencing $s_1; s_2$, branching if e then s_1 else s_2 (which executes s_1 if e evaluates to a non-zero integer, and s_2 otherwise), 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 3.2.1.

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 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, just like the heap. When a patch is applied, its bindings add to or overwrite 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 express the operational semantics of this language 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 ν is an *event trace*, described below. We also need a sibling judgment $(H, e) \longrightarrow^{\nu} (H', c)$ for expressions, where c is the result of computing the expression e.

The label ν describes an *event trace* 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, if plus is a function that returns the sum of its arguments, then

$$((\mathsf{x} \mapsto 4, \mathsf{plus} \mapsto ...), \mathsf{x} := \mathsf{plus}(\mathsf{x}, \mathsf{5})) \longrightarrow^{\nu} (\mathsf{x} \mapsto 9, \mathsf{plus} \mapsto ...)$$

where ν is

$$\mathit{read}(\mathsf{x}, 4); \mathit{call}(\mathsf{plus}(4, 5)); ...; \mathit{ret}(\mathsf{plus}(4, 5)); \mathit{write}(\mathsf{x}, 9)$$

That is, starting out with a heap that maps x to 4 and plus to an appropriate function, executing $\mathsf{x} := \mathsf{plus}(\mathsf{x}, \mathsf{5})$ 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.

Figure 3 gives the three most interesting operational semantics rules for our language. The other rules are straightforward, and are omitted due to lack of space.

The FUN-CALL rule describes the semantics of expression $f(e_1, ..., e_n)$. First, we look up f's definition in H. Next, we evaluate arguments e_1 through e_n. 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 composes the traces produced by evaluating each of the arguments along with the trace produced by evaluating the function's body, delimited by *call*(f(...)) and *ret*(f(...)) events.

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 reduced 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 that does not affect our formal reasoning about this system; in practice, we choose π and the position in the trace at which to apply UPD-TAKEN before we execute the test.

We can now formally define trace equivalence.

DEFINITION 3.1. *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.

Finding equivalent update points. 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 $t^1_\pi \ldots t^n_\pi$. Our goal is to determine which of these update tests produce equivalent traces for a given patch π. Then we can run a single representative test from each equivalence class while retaining full update coverage.

We compute equivalent update points by applying the gentests function in Figure 4 to the original trace ν_t. The

FUN-CALL

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

UPD-SKIP

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

UPD-TAKEN

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

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

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

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

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

SKIP

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

Figure 3: Operational rules

gentests function invokes conflict(π, ν), 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 conflict(π, ν) is defined at the top of Figure 4. 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*, and there are no conflicts with *ret*(...) because in our updating model an active function is not immediately changed by an update—its next call will be to the new version, but the current code will continue executing as-is. 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 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 4 defines gentests(π, N, U, ν), which uses conflict() to compute a minimal set of update points for ν. Here π is the patch, N is the index of the last-seen

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

—

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

Figure 4: conflict **and** gentests **functions**

noupdate event, U is the set of indexes of update points to test, and ν is the trace (which should contain no *update*(...) events). The 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

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

The set U defines the 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 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

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

corresponding to the execution of the example from the beginning of Section 3.2. If we run $\text{gentests}(\pi, 0, \emptyset, (\nu; \mathit{noupdate}))$ where $\mathit{dom}(\pi) = \{f\}$, 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 $\text{conflict}(f, \pi) = \mathit{true}$. As such, we add $N = 1$ to U. Subsequent occurrences of *noupdate* increment N, but no further elements are added to U because neither $\mathit{call}(g())$ nor $\mathit{call}(h())$ conflict with π. On the other hand, if $\mathit{dom}(\pi) = \{f, g, h\}$, then all three calls would conflict with π, and thus N would be added to U in each case, resulting in $U = \{1, 2, 3\}$.

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^i_π denote the trace produced by induced update test t^i_π.

THEOREM 3.2 (CORRECTNESS). *If* $(H, s) \longrightarrow^\nu H'$ *and* $\text{gentests}(\pi, 0, \emptyset, (\nu; \mathit{noupdate})) = (N, U)$, *then for all* $i \notin U$, *there exists* $j \in U$ *such that* $\nu^i_\pi = \nu^j_\pi$.

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 3.3. *Let* H, s, e, ν, π *be such that* $\neg\text{conflict}(\nu, \pi)$ *and either* $(H, s) \longrightarrow^\nu H'$ *or* $(H, e) \longrightarrow^\nu (H', c)$. *Let* $H_0 = H[x \mapsto \pi(x)]$ *for all* $x \in \mathit{dom}(\pi)$. *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* $x \in \mathit{dom}(\pi)$.

The proof is by induction on evaluation derivations.

3.2.1 Application to full DSU systems

The gentests() algorithm can accommodate a variety of dynamic updating systems. The proof of Theorem 3.2 never refers directly to the definition of the judgment $(H, s) \longrightarrow^\nu H'$, relying entirely on simple properties of traces and Lemma 3.3. Thus, to apply gentests() to a particular DSU system, we need only define conflict() appropriately and then prove that Lemma 3.3 holds.

The semantics above models DSU systems like Ginseng [20] and DLpop [11], 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 3). This semantics also effectively captures the behavior of systems that permit only updates to inactive functions, such as Ksplice [4], OPUS [1], and K42 [13].

We can extend 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 [23] and POLUS [7] provide similar support and would require similar changes.

Immediate updates to active functions. Systems like UpStare [15] 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 reduction for such a system, we add trace events to be emitted at labeled statements. For example, we would emit an event $\mathit{label}_f(\text{L})$ when we begin executing a block in the function f labeled with L and this event would conflict with a patch π when $f \in \mathit{dom}(\pi)$.

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 conflict() as is; i.e., explicitly versioned calls never conflict with a patch, 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 and identify appropriate update tests.

3.2.2 Reduction Effectiveness

Figure 5 illustrates the effectiveness of our reduction algorithm for our actual implementation (described next) when testing our benchmark servers. 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 and 86% points from vsftpd could be eliminated. This is significant because the initial number of tests was very large: over 8M for OpenSSH and over 1.7M for vsftpd. For every patch to both applications, the reduction was over 90% with the exception of 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.

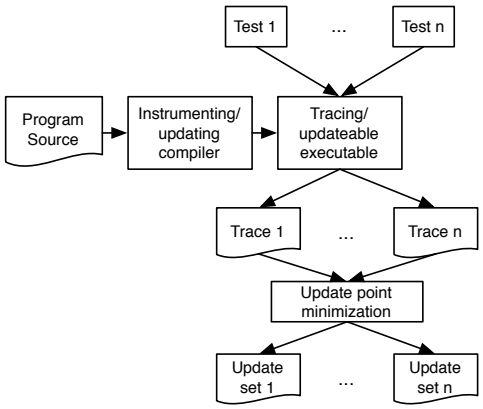
In practice, it is only necessary to test update points that are allowed by the safety check in use. The *CFS* and *AS* columns show the number of points allowed under these models and the further reduction achieved by our algorithm. The combination yields a significant reduction: the maximum number of tested points to achieve full update coverage a patch to either application was 5,467 for patch 8→9 of OpenSSH under CFS and 4,141 for patch 7→8 of OpenSSH under AS.

The manually introduced update points are a small fraction of those in *All*, and we found the reduction strategy 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 of manual points and no vsftpd patch would require over 82 for full update coverage.

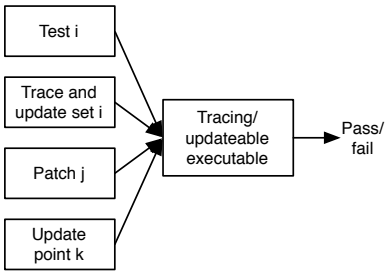
Our reduction algorithm was critical in enabling us to perform our experiments. As we note in Section 4, testing of these reduced points for OpenSSH still required approximately 600 CPU hours to complete.

3.3 DSUTest Implementation

We extended Ginseng to implement our testing framework. Our extended implementation, called DSUTest, works in two phases, illustrated in Figure 6(a) and (b), respectively. In the first phase, the DSUTest compiler instruments



(a) Instrumentation and trace gathering



(b) Running a test case

Figure 6: DSU testing framework architecture

the program to log relevant events to a trace file, and then processes each file to find the reduced 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 reduction, 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 globally unique among related processes. We log the parent and child at each fork, and when we reduce a child process's trace, we may equate some of its initial update points with the parent's update point before the fork, if there were no 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

	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%)**

Figure 5: Update point reduction

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 ignore regions to skip stutter steps, signal handlers, and similar events. 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 [17, 21] to handle multi-threading.

4. EXPERIMENTAL SETUP

	#	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)			

Figure 7: Version and patch information

Using our testing framework, we set out to empirically measure the permissiveness and restrictiveness of the AS and CFS checks. 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.

Test Applications. We tested updates to two long-running server applications: OpenSSH, a widely used SSH server, and vsftpd, a popular FTP server. Figure 7 summarizes the versions of OpenSSH and vsftpd that we consider. We largely re-use the dynamic patches and program versions used by Neamtiu et al. in their Ginseng work [20], with some changes that we describe in the next section. Our OpenSSH releases range from Oct. 2002 to Sept. 2005, and

vsftpd releases range from July. 2004 to Feb. 2008. To make it easy to refer to the versions in the subsequent discussion, we number them starting from 0. For each version, Figure 7 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.

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 [20]. 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.

Program and patch modifications. The program code and patches we received from Neamtiu et al. [20] had been prepared by extracting the connection loop and its cleanup code, as described in Section 2.2, 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 and vsftpd, 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
  rest_f ();   // (extracted) continuation code
}
```

In many cases updates change the "startup" code in the functions in this chain (i.e., the code before the call to g() in the schematic), and thus AS would prevent those updates from being applied during the command loop. However, upon manual inspection we found that the startup code usually bore no tight dependence on the loop code, nor was it ever reexecuted. Therefore, we felt it reasonable, for the purposes of obtaining interesting results, to relax the AS check by also extracting the startup code, so that it is no longer on the stack when the loop executes. 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.

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 command loop as well. 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.

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 testing in parallel.

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.

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 reduction algorithm (Section 3.2) 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 then used the traces produced during testing along with information

	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**

Figure 8: Test failures/points allowed

about the patch contents to retroactively determine which points would be allowed under each safety check.

We ran our experiments for OpenSSH in the "cloud" using 60 Amazon EC2 basic instances [8], each of which provides the equivalent of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor, 1.7 GB memory, and 160GB storage. All OpenSSH update tests for all program versions were executed in under 600 instance-hours. We tested the smaller suite of vsftpd tests on one core of a 2.66GHZ Intel Xeon with 4GB of memory in under 200 hours.

5. 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.

Test Failures/Points Allowed. Figure 8 summarizes the number of update points allowed by each safety check for each patch to OpenSSH and vsftpd, and how many of those points resulted in a failing test.

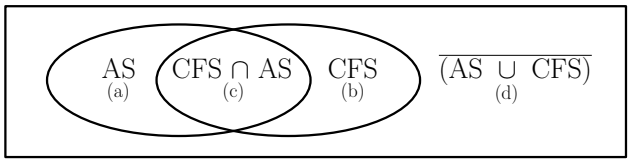
The *All Pts* column of Figure 8 lists over 1.4M failing update points out of 8M total (17.8%) for OpenSSH and over 48K failing runs out of 1.7M total (2.7%) for vsftpd. This is clear evidence that applying updates indiscriminately is extremely risky. Comparing program versions, we see that updates containing few changes typically result in a correspondingly small number of failures. One particularly striking observation is that patches containing no type or function signature changes (OpenSSH patches 1→2 and 3→4 and vsftpd patches 0→1, 2→3, and 3→4) 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 8 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 applications, 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. On the other hand CFS is more permissive, allowing far more update points compared to AS. *Manual* exhibited no test failures, but has many fewer allowed update points.

Comparing safety checks. The top half of Figure 9 classifies each failing update point based on which safety checks would have prevented the failure, while the bottom 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 at the top of the figure.

For both vsftpd and OpenSSH, 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 6.

Turning to the bottom half of Figure 9, we see that more than half 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).



	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%

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%

Successes Allowed

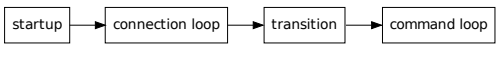
Figure 9: Breakdown of results by safety check

These results suggest CFS provides more update availability than AS.

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:

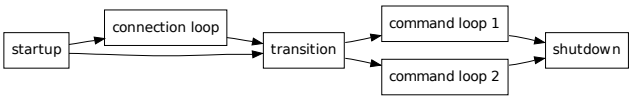


Figure 10 summarizes test failures by program phase and patch (the full tables from which this figure is derived can be found in Figures 13 and 14 in the appendix). Black boxes indicate that all tests pass and grey boxes indicate one or more failures. White boxes indicate that no allowable update points 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. We can also clearly see that 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, which should ensure updates are reasonably available. 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 we have just discussed, updates within the loops are most important, while updates within the startup or transition phases are far less important, since these phases are finite and presumably short.

Threats to Validity. There are several potential threats to the validity of our study. First, the test suites we used for OpenSSH and vsftpd 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 two applications. As such, our results may not generalize, but we believe we have explored enough tests and mature enough applications to strongly suggest the general trend. Likewise, the fact that we changed the applications slightly to make the safety checks more permissive also challenges the generality of our results. But, 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.

6. DISCUSSION

Next, we look in detail at failures that were observed during our experiments, and discuss the limitations of CFS and AS. We also discuss future research directions suggested by our results.

Failures allowed by CFS. The property that distinguishes CFS is that it allows changed code to be executed at the old version following an update, provided this execution will not violate type safety. However, as we discussed in Section 2.3,

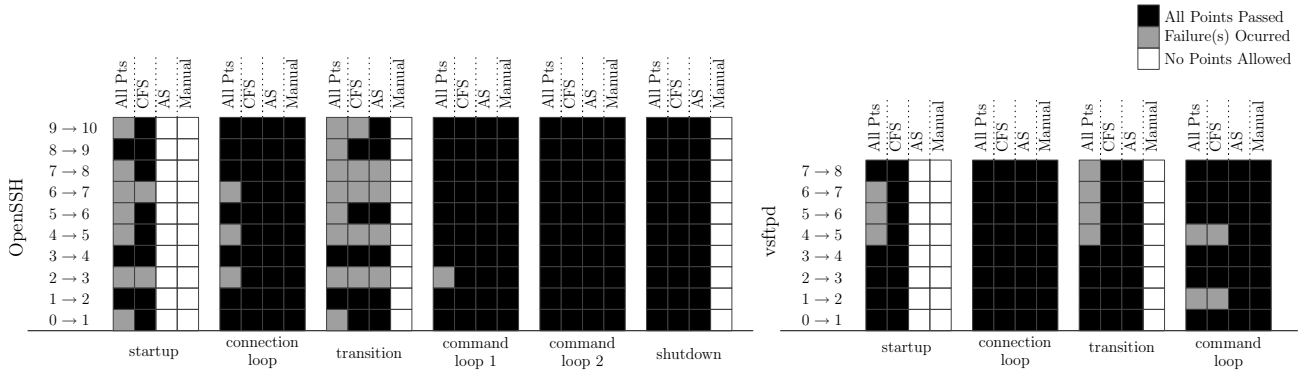


Figure 10: Updatability across program phases

```
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

Figure 11: Skipped return code

even though such executions will be type safe, they may result in failures, and indeed we observed cases of this. One example occurred while testing upload operations against the 1→2 patch to vsftpd. Figure 11 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. After some time, this causes the transfer to timeout and fail. Update points exhibiting this failure are allowed by CFS because, although the code executed following the update in handle_upload_common is changed in the update, executing at the old version is type safe. AS prevents these failures because handle_upload_common is changed and active when the problematic update points are reached.

Failures allowed by CFS and AS. While AS prevents the failure we just saw, as discussed in Section 2.3, 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 base, but was introduced via a code extraction step that is needed to permit many other, safe updates to occur.

Figures 12(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 base, the call to serverloop2 is further down the call chain.)

```
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

Figure 12: Skipped initialization error

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 12(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 4 that we extracted the "startup" code in all functions leading up to the command loops in our subject programs. Consider Figures 12(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, we believe such cases are possible. This may occur in cases where Ginseng's implementation of CFS prevents failures essentially by coincidence, due to conservatism in its static analysis' approximation of a function's calling context.

Future directions. The results of our empirical study, along with observations we made while preparing programs for updating, suggest several potential future directions for more practical and safe DSU systems.

While automatic safety checks are not perfect, we believe they are still very much worthwhile, as our results show that allowing arbitrary update points with no checks results in many failures. There is, however, room for improvement in both AS and CFS, and there may also be other approaches that work better in practice.

The major issue we found with AS is update availability, since functions on the stack can never be updated. One idea to address this issue is to relax AS slightly: We could imagine allowing updates that change functions on the stack, as long as after we return to such a function the code that is executed was not changed by the update—and hence executing the old version of that code is the same as executing the new version. Notice this variant of AS is still more restrictive than CFS, since CFS only requires type safe updates, rather than code equivalence.

The chief benefit of CFS over AS is that it is more permissive without sacrificing type safety. However, sometimes it introduces too much update availability, and so it may be worthwhile to explore ways to reduce permitted updates, such as the transactional version consistency proposed by Neamtiu et al [19].

While improvements to AS and CFS will be beneficial, our experience suggests that increased updatability beyond a certain threshold often provides little benefit while introducing additional points of failure. For example, in OpenSSH and vsftpd, it is important to allow updating during the long-running loops but less important to allow updates elsewhere. Thus, it seems sensible for programmers to prescribe updates only at a small number of carefully selected points. Further work remains in trying to streamline the choice of such update points. For example, even if an updating system makes use of a permissive safety check such as CFS, an offline analysis might help the developer identify update points that will eliminate any version consistency errors. The kind of testing we used in our experiments should also prove useful in choosing safe update points.

We believe that ultimately, a combination of manually chosen update points, automatic safety checks to prevent particular types of failure, and additional assurance arguments based on developer reasoning and update testing, will yield the most practical and safe DSU systems.

7. 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 2.3, 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 [20], programmers can provide a *whitelist* of program locations (e.g., line numbers) that are valid for an update; DLpop [11] has similar behavior. Conversely, Lee [14], 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 [17] and MultithreadedTC [21], which test multi-threaded programs by intelligently enumerating a program's potential thread schedules. At a high level, our technique for test reduction 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 reduction 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 [19]. 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 either the old or new version), while we apply it to the test case reduction.

8. 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 exhaustively testing updates to vsftpd and OpenSSH. We found that updating without the use of safety checks resulted in a large number of failures, and that both checks were able to eliminate the vast majority of these failures. AS was the more restrictive model as it prevented more failures but also more successes than CFS. Significantly, neither check prevented all failures. Ultimately, we believe that a combination of manually specified update points, automatic safety checks, and an additional assurance argument, such as the testing strategy we used for our experiments, will yield a practical, safe solution for dynamic software updating.

9. 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. To appear.

[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] Amazon elastic compute cloud. `http://aws.amazon.com/ec2`.

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

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

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

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

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

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

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

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

[19] 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.

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

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

[22] 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.

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

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

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

APPENDIX

A. 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 8 and 10.

	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	

Figure 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**

Figure 14: Test success and failure (vsftpd Full)