Efficient Systematic Testing for Dynamically Updatable Software

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. However, a significant challenge remains: How can we ensure the act of applying a patch does not itself introduce errors? In this paper, we address this problem by presenting a new systematic testing methodology for updatable programs. Our idea is to transform standard system tests into *update tests* that execute as before, but each transformed test applies a patch at a different *update point* during execution. To mitigate the increase in the number of tests, we developed an algorithm for *test suite minimization* that finds a subset of update points that, if fully tested, yields the equivalent to full update point coverage. We implemented our approach and evaluated it on OpenSSH and vsftpd, two widely used server applications. We found that minimization is highly effective, reducing the number of update tests required for full coverage by 93%.

1. Introduction

Researchers and practitioners have long 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. 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 while it runs. To avoid such problems, several DSU systems use safety checks to control when updates are allowed [2, 4, 7, 11]. However, while these checks are useful, they are not sufficient: Gupta has shown that no automated check can ensure update correctness in the general case [5].

To validate dynamic updates effectively, we propose a methodology to systematically test them (Section 2). Given a suite of existing system tests for some program P, we can produce a suite of *update tests* as follows. For each program point μ reached during execution of system test t at which a dynamic patch π could be applied, we define an update test t^{μ}_{π} that runs t on the old version of P and applies π at μ. If the patch is correct, and t is valid for both the old and new versions of P, then t^{μ}_{π} should succeed. If test t would produce different behaviors for different versions (as could happen for a bug fix or feature addition), we can construct a *hybrid update test* that accepts both the old and the new version's behavior.

HotSWUp'09, October 25, 2009, Orlando, Florida, USA.

Directly testing every update point for every system test would likely result in too many tests to be practical. To address this issue, we developed a novel algorithm for *test suite minimization* that reduces the number of tests without reducing their coverage (Section 3). Out insight is that many points reached during a test's execution will in fact induce equivalent executions when applying a given patch. For example, consider a sequence $\mu_1; \mathsf{f}(); \mu_2$. If a dynamic patch π does not alter f or any code or data used by f, then applying π at either μ_1 or μ_2 will produce the same behavior. Using this insight, our algorithm identifies sets of equivalent update points where only one of them needs to be tested.

We have implemented our approach for testing updates, which we call DSUTest (Section 4), for programs compiled with the Ginseng DSU framework [11]. Section 5 presents our evaluation of DSUTest, considering dynamic updates derived from three years' worth of releases for two widely used servers, vsftpd (9 versions) and OpenSSH (11 versions). Using existing and custom tests for these programs, we found that if we unconditionally allow update points prior to each function call, our suite of system tests applied to all versions induce more than 9.8 million update tests. Applying minimization reduces this to around 620,000 update tests, i.e., we can perform 93% fewer tests and still get full update coverage.

Typical updating systems do not permit updates to be applied unconditionally, but only when certain safety checks are satisfied. *Activeness safety* (AS) is a popular check that disallows changes to active functions. Applying the AS check, 2 million update tests are induced, many fewer than without AS. But minimization remains effective, reducing this number to roughly 35,000 tests, a reduction of 98%. When restricting ourselves to the manually inserted update points prescribed by the original Ginseng work [11], we generate roughly 8,200 update tests, which can be reduced by 2.9% with our algorithm. Manually inserted points yield less reduction because they occur in top-level loops, and so are separated by a large number of function calls, increasing the chances of a call to a changed function between points. All update tests succeeded for the manually chosen points, confirming claims in prior published results. As might be expected, many tests fail when updates are permitted at any point, and some fail when applied at AS points; our technical report [6] includes a detailed analysis of these failures.

To our knowledge, our work is the first to explore how to efficiently and systematically test dynamically updatable software.

2. Dynamic update testing

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. The details of the form of π and how it is implemented at run-time differ between systems, but they are not relevant to our basic methodology. 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.

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 Ginseng [11], programmers can provide a *whitelist* of program locations (e.g., line numbers) that are valid for an update, while Gupta et al. [5], POLUS [4], 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 Ginseng's type safety checks are also satisfied.

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. For a deterministic test t, we can unambiguously enumerate the update points that arise during the test's execution. (We discuss non-determinism in Section 4.) 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 consider how to 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. In this case, we can 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.

Constructing hybrid tests for certain program modifications may not be straightforward. For example, P_1 may fix a bug in P_0 that resulted in a crash or other behavior that could be incorrectly attributed to an update test failure. Likewise, P_1 may deprecate functionality that worked correctly in P_0. See our companion technical report for further discussion of these cases [6].

3. Test suite minimization

The procedure described in Section 2 lets us systematically derive update tests from existing system tests. Unfortunately, we have found this procedure vastly multiples the number of tests to run. For example, our experiments with roughly 90 system tests applied to ten versions of OpenSSH yielded more than 8 million update tests. We mitigate this increase in test suite size with a novel algorithm that eliminates all provably redundant tests.

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

```
1  void main() { DSU_update();     f();
2                DSU_update();     g();
3                DSU_update();     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, 2, or 3, 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 on lines 1–3 form an equivalence class, and we need only test one of the three to cover the whole class.

However, suppose dynamic patch π_2 modifies f, g, and h. In this case, none of the update points are equivalent. If we update on line 1, we will call the new versions of all three functions. If we update on line 2, we will call the old version of f and the new versions of g and h. If the update on line 3 happens, 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 each of them to find out.

3.1 Formal language

We present our minimization algorithm in terms of a small formal language. This language permits dynamic updates to function definitions and global variables such that, following an update, active functions continue to execute the old version while all subsequent function calls invoke the newest version. This semantics is employed by Ginseng, and subsumes the semantics of systems that use the activeness check, such as Ksplice [2], K42 [7], and Jvolve [12]. In our companion technical report [6], we sketch how our approach can be adapted to support the semantics of UpStare [8], which allows an active function to transition immediately to its new version, and other systems like POLUS [4] and UpgradeJ [3], which allow explicitly versioned function calls.

Figure 1 gives the language syntax. Expressions e 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 s 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 corresponds to Ginseng's DSU_update() call, as described above.

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 parameters 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 currently executing 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 an execution. Event traces are defined at the bottom of Figure 1. Each event corresponds to the execution of one program construct, and individual events are concatenated using the ; operator. For example, if plus returns the sum of its arguments, then

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

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 $\mathsf{x} := \mathsf{plus}(\mathsf{x}, 5)$ produces a heap that maps x to 9. The execution also yields an event trace ν indicating x

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

Figure 1. Syntax of programs and event traces

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

$$
\frac{\nu = \mathit{noupdate}}{(H, \mathsf{update}) \longrightarrow^{\nu} H} \text{[UPD-SKIP]}
\qquad
\frac{H' = H[x \mapsto \pi(x)] \quad \forall x \in \mathit{dom}(\pi)}{(H, \mathsf{update}) \longrightarrow^{\mathit{update}(\pi)} H'} \text{[UPD-TAKEN]}
$$

Figure 2. Selected operational rules

was read; plus was called, executed (...), and returned; and then x was written. We discuss *update*(π) and *noupdate* events shortly.

Figure 2 gives the three most interesting operational semantics rules for our language. The other rules are straightforward, and are presented in our technical report [6].

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 specify π and the position in the trace at which to apply UPD-TAKEN before executing each update test.

3.2 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 (with no updates taken), 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. By equivalent, we mean that although they vary in the update point

$$
\begin{array}{l}
\textbf{conflict}(\pi, \mathit{skip}) = \mathit{false} \\
\textbf{conflict}(\pi, \mathit{call}(x(\ldots)) = (x \in \mathrm{dom}(\pi)) \\
\textbf{conflict}(\pi, \mathit{ret}(\ldots)) = \mathit{false} \\
\textbf{conflict}(\pi, \mathit{read}(x, \ldots)) = (x \in \mathrm{dom}(\pi)) \\
\textbf{conflict}(\pi, \mathit{write}(x, \ldots)) = (x \in \mathrm{dom}(\pi)) \\
\textbf{conflict}(\pi, \mathit{noupdate}) = \mathit{false} \\
\textbf{conflict}(\pi, \mathit{update}(\pi')) = (\mathit{dom}(\pi) \cap \mathit{dom}(\pi') \neq \emptyset) \\
\textbf{conflict}(\pi, \nu_1; \nu_2) = (\textbf{conflict}(\pi, \nu_1) \vee \textbf{conflict}(\pi, \nu_2)) \\
\\
\textbf{gentests}(\pi, N, U, \nu) = (N, U) \\
\quad \text{where } \nu \neq (\nu_1; \nu_2) \wedge \nu \neq \mathit{noupdate} \wedge \neg\textbf{conflict}(\pi, \nu) \\
\textbf{gentests}(\pi, N, U, \nu) = (N, U \cup \{N\}) \\
\quad \text{where } \nu \neq (\nu_1; \nu_2) \wedge \nu \neq \mathit{noupdate} \wedge \textbf{conflict}(\pi, \nu) \\
\textbf{gentests}(\pi, N, U, \mathit{noupdate}) = (N + 1, U) \\
\textbf{gentests}(\pi, N, U, \nu_1; \nu_2) = \\
\quad \text{let } (N', U') = \textbf{gentests}(\pi, N, U, \nu_1) \text{ in } \textbf{gentests}(\pi, N', U', \nu_2)
\end{array}
$$

Figure 3. **conflict** and **gentests** functions

taken, they read and write the same values to and from the same variables, call the same functions with the same parameters, etc.—in other words, their behavior is identical except for update timing. Then we can run a single representative test from each equivalence class while retaining full update coverage.

We compute equivalent update points by applying the **gentests** function in Figure 3 to the original trace ν_t. The **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).

Function **conflict**(π, ν) is defined at the top of Figure 3. Given a call, read, or write to x, there is a conflict with π if and only if $x \in \mathrm{dom}(\pi)$. There are no conflicts with *skip*, *noupdate*, or *ret*(...)—the last because currently executing functions continue as-is following an update. Given a different update with patch π', patch π conflicts with it 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 3 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 *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') where N' is the most recently seen update point and U' is the total set of update point indexes to test. Thus, given a complete trace ν_t from system test t, we compute

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

The set U defines the minimal set of update points that achieve 100% update coverage; we ignore $i = 0$, if it is 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 handles the case when the event *does* conflict with π: 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. The last clause 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. If we run **gentests**$(\pi, 0, \emptyset, (\nu; \textit{noupdate}))$ where $dom(\pi) = \{f\}$, our outcome will be $U = \{1\}$, as follows. When we see the first *noupdate*, we increment $N = 0$ to $N = 1$. Then we see the call to f, where **conflict**$(f, \pi) = true$. As such, we add $N = 1$ to U. Subsequent occurrences of *noupdate* increment N, but no further elements are added to U because neither $call(g())$ nor $call(h())$ conflict with π. On the other hand, if $dom(\pi) = \{f, g, h\}$, then all three calls would conflict with π, and thus N would be added to U in each case, resulting in $U = \{1, 2, 3\}$.

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

Informally, the above theorem says that any update point i is either in U, or there is some equivalent update point in U that yields the same trace. The proof of this theorem depends crucially on 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 3.2. *Let* H, s, e, ν, π *be such that* $\neg$**conflict**(ν, π) *and either* $(H, s) \longrightarrow^\nu H'$ *or* $(H, e) \longrightarrow^\nu (H', c)$*. Let* $H_0 = H[x \mapsto \pi(x)]$ *for all* $x \in dom(\pi)$*. 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* $x \in dom(\pi)$.

The proof is by induction on $(H, s) \longrightarrow^\nu H'$ (or $(H, e) \longrightarrow^\nu (H', c)$). To use **gentests** with a different updating semantics would require re-defining **conflict** and re-proving Lemma 3.2 (but not Theorem 3.1).

4. Implementation

We have implemented our testing framework and minimization algorithm for Ginseng, a compiler and run-time system for dynamically updating C programs [11]. Our implementation, DSUTest, extends our formal presentation in three ways. First, it accounts for Ginseng's support for changes to type definitions, where accesses to values of updatable type occur via special wrapper functions. Thus we must trace calls to these functions and consider calls conflicting when a patch modifies the respective type definition. Second, our implementation accommodates server programs that fork independent subprocesses that could themselves be updated. To handle this case, we annotate events with a deterministically chosen process identifier. Finally, our basic methodology presumes that tests are deterministic; our implementation accommodates some forms of nondeterminism, e.g., I/O buffering by the OS and functionality that depends on the environment, such as the current time of day. In particular, to keep update tests consistent with the initial trace, we check that each update test trace matches the original up to the chosen update point, and re-run the test if not. For highly non-deterministic activities, e.g., signal handling, we designate *ignore regions* of code in which the test trace need not match the original. We disallow dynamic updates within such regions, though conflicting events within the regions make update points before and after the region non-equivalent. Our TR presents further details [6].

5. Experiments

We evaluated the practicality of our testing approach using two long-running server applications: OpenSSH, a widely used SSH server, and vsftpd, a popular FTP server. Here we consider the effectiveness of our minimization algorithm; we do not consider the outcomes of the tests themselves for lack of space.

5.1 Methodology

The left half of Figure 4 summarizes the versions of OpenSSH and vsftpd over which we applied our minimization algorithm. We largely reuse the dynamic patches and slightly modified program versions from our earlier Ginseng work [11]. To make it easy to refer to the versions in our discussion, we number them starting from 0. For each version, Figure 4 lists the total lines of code (measured with sloccount), the number of update tests (drawn from unmodified and hybrid system tests), and the number of changes to function signatures, function bodies, and named type definitions (structs, unions and typedefs) required to update to the next version.

Test cases. For OpenSSH we used the suite of system tests 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 based on return codes and command output. We split some large tests with orthogonal components into several smaller tests, so we could run them in parallel.

vsftpd is not distributed with any system tests, so 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. For the 6→7 patch, we wrote an additional hybrid test (Section 2) to exercise a new feature that limits the number of failed login attempts that are allowed.

Update point selection. In Ginseng, the programmer specifies update points by manually inserting DSU_update() calls. In prior work with vsftpd and OpenSSH, we inserted one or two of these calls at *quiescent points*—positions where there are no in-flight operations, hence updates will more likely succeed [11]. For the current experiments, we manually inserted two update points per application: one at the beginning of the connection acceptance loop, and one at the beginning of the command processing loop. In our results, these are referred to as *Manual Pts*. We also considered programmatically inserting an update point prior to each function call throughout the program to approximate the operation of a system that does not use explicit update points; we refer to these points as *All Pts* in our results. Finally, we considered the subset of *All Pts* that would be allowed by the widely used activeness safety check, in which an update is only permitted if the patch changes functions not referenced by the current activation stack [2, 5, 7, 12]; we refer to these points as *AS Pts*.

5.2 Results

The right half of Figure 4 shows the number of update tests we would have to execute—with and without minimization—for the *All Pts* and *AS Pts* sets of update points. For each, we show the original count, the minimized count, and the percent reduction. These results show that minimization eliminates a tremendous number of potential tests. Overall, minimization results in a 95% reduction for OpenSSH and an 86% reduction for vsftpd.

The amount of reduction is roughly inversely proportional to the size of the patch. For example, we can see that the OpenSSH patch from versions 1→2 changed only six functions, and there is a correspondingly large reduction in the number of distinct update points. On the other hand, patch 2→3 changes many more functions, and these changes create conflicts that reduce minimization. The amount of minimization depends on the actual changes and the tests being run. The 6→7 patch to vsftpd resulted in a significantly smaller reduction than all other patches. This patch added

	#	Version	LoC	Tsts	Δ to next ver			Reduction	
					Sig	Fun	Type	All Pts	AS Pts
OpenSSH	0	3.5p1	46,735	75	3	98	5	580,871 → 31,791 (95%)	35,314 → 3,027 (91%)
	1	3.6.1p1	48,459	75	0	6	0	705,322 → 1,795 (∼100%)	587,578 → 1,717 (∼100%)
	2	3.6.1p2	48,473	76	5	238	11	638,720 → 63,011 (90%)	20,902 → 2,353 (89%)
	3	3.7.1p1	50,448	91	0	18	0	772,198 → 4,324 (99%)	638,803 → 3,775 (99%)
	4	3.7.1p2	50,460	91	13	172	10	773,086 → 27,399 (96%)	21,343 → 1,564 (93%)
	5	3.8p1	51,822	104	0	24	1	878,235 → 17,398 (98%)	111,950 → 1,723 (98%)
	6	3.8.1p1	51,838	104	6	257	10	879,668 → 47,092 (95%)	44,278 → 2,139 (95%)
	7	3.9p1	53,260	104	4	179	12	918,717 → 89,601 (90%)	100,854 → 4,141 (96%)
	8	4.0p1	56,068	105	0	72	3	973,364 → 34,293 (96%)	61,724 → 2,070 (97%)
	9	4.1p1	56,104	104	10	157	7	933,514 → 52,356 (94%)	61,051 → 2,891 (95%)
	10	4.2p1	57,294	(Not patched)					
							Total	**8,053,695 → 369,060 (95%)**	**1,683,797 → 25,400 (98%)**
vsftpd	0	2.0.0	13,048	13	0	6	0	210,142 → 26 (∼100%)	102,307 → 26 (∼100%)
	1	2.0.1	13,059	13	1	12	0	210,142 → 516 (∼100%)	69,775 → 166 (∼100%)
	2	2.0.2pre2	13,114	13	0	21	0	215,223 → 1,122 (99%)	55,555 → 553 (99%)
	3	2.0.2pre3	14,293	13	0	76	0	220,564 → 3,866 (98%)	37,265 → 1,912 (95%)
	4	2.0.2	16,970	13	0	10	1	218,586 → 19,893 (91%)	2,123 → 301 (86%)
	5	2.0.3	12,977	13	0	25	1	223,098 → 15,910 (93%)	67,330 → 3,567 (95%)
	6	2.0.4	14,427	14	0	100	2	233,199 → 200,653 (14%)	7,437 → 2,742 (63%)
	7	2.0.5	14,482	13	0	93	2	222,296 → 10,371 (95%)	3,098 → 275 (91%)
	8	2.0.6	14,785	(Not patched)					
							Total	**1,753,250 → 252,357 (86%)**	**344,890 → 9,542 (97%)**

Figure 4. Reduction effectiveness

a new failed login limit feature which could be enabled through a new configuration flag. When applied, this patch reloads the vsftpd configuration, which could update many global variables (to reflect the new configuration), creating conflicts that inhibit reduction.

The *AS Pts* set is 79% smaller, overall, than the full set of points across both applications. Applying minimization results in an additional 98% reduction. With the combination of activeness safety checking and minimization, the maximum number of tests for any update is 4,141, which requires running each system test an average of 40 times. This demonstrates that reduction is useful even with the more limited set of update points allowed by activeness safety.

The manually introduced update points are a small fraction of those in *All Pts*, and the reduction effectiveness for *Manual Pts* is substantially lower. Summing up all runs of both applications, the four manual update points are dynamically executed a total of 8,241 times. Applying the reduction yields a total of 8,006 distinct update points, for an improvement of 2.9%. Manually inserted update points are in the top-level loops of the program, and thus many function calls may occur between iterations, increasing the chances of a conflict. Note, however, that when using manual points no patch would require more than 859 update tests in total, which requires running each system test an average of 8 times.

6. Related work

Gupta et al. [5] 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. Just as software testing is a tractable alternative to full program verification, our framework is a tractable alternative to determining complete update validity—we consider the executions of particular tests rather than all possible executions, and the test oracle decides the validity of the final state of the patched program.

Our approach to generating update tests is related to Chess [9] which tests multi-threaded programs by intelligently enumerating a program's potential thread schedules; we enumerate and test potential update points. At a high level, our technique for test minimization is like partial order reduction in model checking [1], which is used to avoid consideration of distinct program executions that result in the same states. Our minimization algorithm on traces is inspired by Neamtiu et al.'s observation that an update at two program points is equivalent if the activity between those two points is unaffected by the patch [10]. Neamtiu et al. applied this observation to a static analysis for implementing *version-consistent update transactions*, while we apply it to the test case minimization.

References

[1] 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.

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

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

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

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

[6] C. Hayden, E. Hardisty, M. Hicks, and J. Foster. A testing based empirical study of dynamic software update safety restrictions. Technical Report CS-TR-4947, University of Maryland, College Park, 2009.

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

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

[9] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, 2008.

[10] 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.

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

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