ABSTRACT

Title of dissertation: CLEAR, CORRECT, AND EFFICIENT DYNAMIC SOFTWARE UPDATES

Christopher M. Hayden
Doctor of Philosophy, 2012

Dissertation directed by: Professor Michael Hicks and
Professor Jeffrey S. Foster
Department of Computer Science

Dynamic software updating (DSU) allows programs to be updated as they execute, enabling important changes (e.g., security fixes) to take effect immediately without losing active program state. Most DSU systems aim to add runtime updating support *transparently* to programs—that is, all updating behavior is orchestrated by the DSU system, while avoiding program code modifications. This philosophy of transparency also extends to existing notions of DSU correctness, which emphasize generic correctness properties that apply to all runtime updates, such as type safety.

We claim that runtime updating support should be treated as a *program feature*, both for implementation and for establishing correctness. For implementing DSU, this means that the core updating behavior is made manifest in the program's code, exposing the programmer to the application-specific details they need to understand, while relying on the DSU system for everything else. We argue that this approach can provide several benefits: simplified developer reasoning about up-

date behavior, modest effort to implement, support for arbitrary program changes, lightweight tool support, and negligible runtime overhead. For establishing correctness, treating updating support as a program feature means that developers should specify and check the specific behaviors that an updated program will exhibit as they do for other program features, rather than relying on overly general notions of correctness. We argue that developers can write DSU specifications with little work—usually by adapting single-version specifications—and check them using standard methods: testing and verification.

To support this thesis, we present three pieces of work. First, we describe an empirical study of the techniques used by existing DSU systems to determine when an update can take place. We find that automatic techniques are unable to prevent erroneous behavior and conclude that placing *update points* in developer-chosen main loops is most effective. Next, we present an approach to specifying and checking the correctness of program features under DSU. We propose a specification strategy that can adapt single-version specifications to describe DSU behavior and a new tool that allows reasoning about DSU specifications using standard checking tools. We have implemented our approach for C, and applied it to updates to the Redis key-value store and several synthetic programs. Finally, we present Kitsune, a new DSU system for C programs, that supports the developer in implementing runtime updating as a program feature. We have used Kitsune to update five popular, open-source, single- and multi-threaded programs, and find that few program changes are required to use Kitsune, and that it incurs essentially no performance overhead.

CLEAR, CORRECT, AND EFFICIENT DYNAMIC SOFTWARE UPDATES

by

Christopher Michael Hayden

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2012

Advisory Committee:
Professor Michael Hicks, Co-chair/Advisor
Professor Jeffrey S. Foster, Co-chair/Advisor
Professor Shuvra S. Bhattacharyya, Dean's Representative
Professor Amol Deshpande
Professor Peter J. Keleher

© 2012
Christopher Michael Hayden

Dedicated to Kyle, my supportive and loving wife.

Acknowledgments

I owe a debt of gratitude to many people who helped make this dissertation possible. I could not have performed the research that this dissertation encompasses without the guidance of my advisors, Mike Hicks and Jeff Foster. I started graduate school with only a vague understanding of how research is done. Over the seven years since, Mike and Jeff have taught me how to do thorough research, helped improve my writing, worked late nights with me prior to paper deadlines, and showed me how to treat setbacks as valuable opportunities to adjust course. I believe these lessons and experiences are the most valuable product of my time in graduate school and will stick with me throughout my future endeavors.

Over the course of my Ph.D. program, I have been fortunate to collaborate with Iulian Neamtiu, Ted Smith, Eric Hardisty, Stephen Magill, Nate Foster, Yudi Turpie, and Karla Saur. My collaborators have contributed significantly to this dissertation and my time working with them and the other members of the UMD PL research group has enriched my time as a graduate student.

I could not have started or completed my Ph.D. without the patience and dedication of my wife Kyle. In many ways, this process has been a huge personal indulgence—I am amazed at Kyle's willingness to support and even marry me during my graduate career! I am excited to continue pursuing our mutual and personal goals together in the future.

I must also thank my parents, Jo and Paul. All through my life, they have striven to provide me with the opportunities that would enable me to pursue a fulfilling life. On countless occasions, they managed to nudge me in the right direction

without nudging so hard as to turn me off learning and education. As with all of my choices and accomplishments, my completion of this dissertation reflects the values that my parents instilled in me.

Table of Contents

List of Figures viii

List of Abbreviations x

1 Clear, Correct, and Efficient Dynamic Software Updates 1
- 1.1 Dynamic Software Updating . . . 2
- 1.2 DSU as a Program Feature . . . 8
  - 1.2.1 Motivation . . . 9
- 1.3 Overview . . . 14
  - 1.3.1 Empirical Study . . . 15
  - 1.3.2 Specifying DSU correctness . . . 17
  - 1.3.3 Verification of DSU CO-specs . . . 18
  - 1.3.4 Updating with Kitsune . . . 20

2 Evaluation of DSU Timing Restrictions 23
- 2.1 Empirical assessment of timing effectiveness . . . 27
  - 2.1.1 Results . . . 28
- 2.2 Dynamic software updating background . . . 32
  - 2.2.1 Ginseng implementation basics . . . 32
  - 2.2.2 Updating active code in Ginseng . . . 33
  - 2.2.3 Controlling timing in Ginseng . . . 34
  - 2.2.4 Timing controls in other DSU systems . . . 35
- 2.3 Testing dynamic updates . . . 37
  - 2.3.1 Testing procedure . . . 38
  - 2.3.2 Update test suite minimization . . . 40
  - 2.3.3 Implementation . . . 44
- 2.4 Experimental setup . . . 47
  - 2.4.1 Test applications . . . 47
  - 2.4.2 Test suites . . . 50
  - 2.4.3 Running tests and tabulating results . . . 52
- 2.5 Experimental results . . . 53
  - 2.5.1 Usability . . . 54
  - 2.5.2 Update Safety . . . 57
  - 2.5.3 Update Availability . . . 59
  - 2.5.4 Failure examples . . . 61
- 2.6 Limitations . . . 65
- 2.7 Related work . . . 67
- 2.8 Conclusions . . . 69

3 Specifying DSU Correctness 71
3.1 Prior work on update correctness 73
3.2 Client-oriented specifications 75
3.2.1 Backward compatible CO-specs 76
3.2.2 Post-update CO-specs 76
3.2.3 Conformable CO-specs 78
3.2.4 Relation to prior notions of correctness 80
3.3 Applications of CO-specs 81

4 Verifying DSU CO-specs 83
4.1 Verification via program merging 85
4.1.1 Syntax 86
4.1.2 Semantics 86
4.1.3 Program merging transformation 88
4.1.4 Equivalence 91
4.2 Experiments 93
4.2.1 Programs 94
4.3 Related work 97

5 Kitsune: Efficient, General-purpose DSU for C 98
5.1 Kitsune 102
5.1.1 Data and Control Migration 104
5.1.2 Multi-threading 109
5.2 xfgen 111
5.2.1 Transformer generation 115
5.3 Experiments 120
5.3.1 Programmer effort 121
5.3.2 Performance 126
5.4 Experience using Kitsune 131
5.5 Related Work 136
5.6 Conclusions 142

6 A Study of DSU Quiescence for Multithreaded Programs 143
6.1 Achieving full quiescence 146
6.1.1 Basic approach 146
6.1.2 Avoiding blocking 147
6.1.2.1 Blocking on I/O 148
6.1.2.2 Blocking on condition variables 149
6.2 Results 151
6.2.1 Experimental setup 151
6.2.2 Quiescence times 155
6.2.3 Threats to validity 156
6.3 Prior work 157
6.4 Conclusions 159

7 Future Work 160
7.1 Checking Tools for Kitsune . . . 160
7.2 Kitsune Extensions . . . 161
7.3 DSU as a Feature for Other Languages . . . 164

8 Conclusion 166

A Comparing failures allowed by different timing mechanisms 168
A.1 Program Phases . . . 168
A.2 Minimization Effectiveness . . . 172

B Merging equivalence proof 177
B.1 Overview . . . 177
B.2 Soundness Lemmas . . . 183
B.3 Completeness Lemmas . . . 187
B.4 Auxiliary Lemmas . . . 189

Bibliography 191

List of Figures

1.1 Network-enabled thermostat server 5
1.2 Compiled-in indirection 5
1.3 State transformation, °F→°C 6
1.4 v0→v1 patch 7
1.5 v2 code modifications 7
1.6 AS timing error 10
1.7 Modifications to active code 11
1.8 Refactorings to update active code 11
1.9 Modified loop structure 12
1.10 Added update point 15
1.11 DSU merger output 19
1.12 Kitsune modifications 21

2.1 Two versions of a program 24
2.2 DSU testing framework architecture 44
2.3 Version, patch, and test information 48
2.4 Points allowed/test failures 57
2.5 Skipped return code 62
2.6 Skipped initialization error 63

3.1 Sample C specifications for key-value store. 75
3.2 Transforming new-version specifications 77

4.1 Syntax and semantics. 85
4.2 Merging transformation (partial). 89

4.3 Synthetic examples. 95

5.1 Kitsune build chain 102

5.2 Example; Kitsune additions highlighted 105

5.3 xfgen specification language and type annotations 112

5.4 State size vs. update time 129

A.1 Updatability across program phases 171

A.2 Test success and failure (OpenSSH Full) 173

A.3 Test success and failure (vsftpd Full) 174

A.4 Test success and failure (ngIRCd Full) 175

B.1 Merging old version code. 178

B.2 Merging new version code. 179

B.3 Merging combined version code. 180

List of Abbreviations

AS	Activeness Safety
CLOS	Common Lisp Object System
CO-spec	Client-Oriented Specification
CPU	Central Processing Unit
DSU	Dynamic Software Updating
FTP	File Transfer Protocol
GC	Garbage Collection
HTTP	Hypertext Transfer Protocol
IRC	Internet Relay Chat
I/O	Input/Output
JVM	Java Virtual Machine
KLOC	Thousands of Lines of Code
LOC	Lines Of Code
PIC	Position Independent Code
PC	Program Counter
RHEL	Redhat Enterprise Linux
OS	Operating System
SIQR	Semi-Interquartile Range
SSH	Secure SHell
VM	Virtual Machine
VMM	Virtual Machine Manager

Chapter 1

Clear, Correct, and Efficient Dynamic Software Updates

Software services have become ubiquitous and essential to the lives of users. More and more communication takes place through web-based email, on-line chat, voice-over-IP, and social networking services. Documents and other data are increasingly stored, shared, and even produced on-line. Users are informed and entertained by services that provide on-line news, movie streaming, and music. On-line retailers sell hundreds of billions of dollars of goods each year [18]. All of this online activity is evidence of users' increasing reliance on the availability of software services. When these services become unavailable, it disrupts the users who depend on them and exacts a high cost on the service operators [53, 12, 13].

One source of service downtime is the need to update the software that underlies a service. Software updates allow developers to fix critical security, correctness, and performance problems and introduce valuable new functionality. Hence, the ability to update software is vital. As evidence, when the NASDAQ stock exchange was hacked in 2010, investigators attributed much of its vulnerability to out-of-date software that lacked critical security patches [39]. Similarly, a recent `stopbadware.org` survey of administrators of hacked websites—often used to mount phishing and other attacks—indicated that 54% of respondents who knew how their site was hacked attributed the vulnerability to "out-of-date or insecure

software" [62]. In these cases, failure to update costs the website owner in reputation and computing resources, and is also harmful to targeted Internet users.

The most common approaches to updating software rely on stopping and restarting it, which disrupts active users and reduces availability to new users. This forces administrators to make a painful trade-off between availability and security/features. Dynamic software updating, where software is updated as it runs, is a promising technique for effectively balancing these concerns.

1.1 Dynamic Software Updating

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, allowing software patching without disruption. Support for dynamic software updating (DSU) takes many forms. "Fix-and-continue" development, in which one incrementally develops and tests an application as it runs, has long been common for Smalltalk and CLOS, and Sun's HotSwap VM [37, 21] and Microsoft's .NET Visual Studio for C# and C++ [24] both include special support for it. The Erlang programming language [7], designed by Ericsson for building phone switches and other event-driven software, provides DSU primitives that are regularly used to hot-patch fielded systems. Research DSU systems for other languages such as C, C++, and Java [5, 17, 35, 36, 45, 46, 49, 51, 64] have been able to dynamically update server programs, tracking changes according to those applications' release histories. Ksplice [8] can apply security patches to the Linux kernel at run-time, allowing both

service providers and end-users to benefit from recent OS fixes without disruption.

The strength of DSU is its ability to preserve program state during an update. For certain types of programs that are relatively stateless (e.g., because their state is stored externally in a database) and whose connections are short-lived (e.g., some web applications), rolling stop-and-restart upgrades can exploit redundancy to permit updates with little disruption [3]. In this approach, new client connections reach newly started, upgraded server instances, and instances still running at the old version are shut down as they become inactive.

However, many server programs are a poor match for rolling-upgrade–based techniques because the running instances maintain critical state. For example, this dissertation considers applying DSU to several programs that maintain connections for an unbounded length of time, including OpenSSH (an SSH server), vsftpd (an FTP server), Icecast (a streaming audio server), Tor (a privacy-preserving routing server), and ngIRCd (an IRC server). DSU allows those active connections to immediately benefit from important program updates, whereas rolling upgrades would not. This dissertation also considers caching/key-value servers like Memcached and Redis, which can manage massive amounts of in-memory data. DSU techniques will maintain this in-memory state across the update, while traditional upgrade techniques might lose the active state (for Memcached) or rely on an expensive disk reload that would reduce availability (Redis).

Motivating Example

In this section, we walk through the use of a DSU system for a simple example program to show how runtime updates are supported. We first use the example to show the workings of a DSU system and how it supports a program's evolution. We will later use this example to illustrate several problems that affect most current approaches to DSU. Our running example is a simple server program implementing a network-enabled thermostat (shown in Figure 1.1). In this example, the program loops forever, accepting new connections (with get_conn) and then handling them (with handle_conn). This server maintains one item of global state, temp, holding the thermostat's target temperature in °F. The handle_conn function processes requests to SET or READ the target temperature of the room.

We describe a DSU implementation approach based on indirection for this example, similar to the internal operation of the Ginseng updating system [51]. In particular, we introduce indirection by compiling the program specially so that all function calls are made through function pointers. Figure 1.2 shows how the program might look with function-pointer indirection compiled in by the Ginseng compiler. New code can be installed by redirecting the function pointers at runtime to newly loaded code.

The developer may also provide *state transformation* code to update the program state to be compatible with the new version. As an example, consider a new version of this program that receives and stores the temperature in °C, for which the only code change occurs in handle_conn (cf., Line 9 of Figure 1.4). In addition

```
1  float temp; /* temperature (°F) */
2  void get_conn() { /* v0 impl */ ... }
3  void handle_conn(conn *c) {
4    /* v0 impl */
5    if (c→op == READ) { respond("TEMP: %f", temp); }
6    else if (c→op == SET) {
7      temp = c→val;
8      /* heat room */
9      heater_set_temp((temp - 32.0f) / 1.8f); /* use Celsius */
10     respond("OK");
11   } else { respond("ERROR"); }
12 }
13 int main() {
14   while (1) {
15     conn *c = get_conn();
16     handle_conn(c);
17   }
18 }
```

Figure 1.1: Network-enabled thermostat server

```
1  float temp;
2  void get_conn_v0() { /* v0 impl */ ... }
3  void handle_conn_v0(conn *c) { /* v0 impl */ ... }
4  void (*get_conn_newest)() = &get_conn;
5  void (*handle_conn_newest)(conn *c) = &handle_conn;
6  int (*main_newest)() = &main_v0;
7  int main_v0() {
8    while (1) {
9      conn *c = get_conn_newest();
10     handle_conn_newest(c);
11   }
12 }
13 int main() { return main_newest(); }
```

Figure 1.2: Compiled-in indirection

```
1 extern float temp;
2 void user_xform() {
3     temp = (temp - 32.0f) / 1.8f; /* Convert to Celsius */
4 }
```

Figure 1.3: State transformation, °F→°C

to the code change, the developer provides the user_xform function in Figure 1.3 to implement the transformation of temp.

Given the modified program and state transformer, the DSU system's tools will detect the changes and construct the patch in Figure 1.4. To apply the patch, the developer should compile it to a shared library and signal (e.g., via a Unix signal or other external interface) the DSU runtime system to load it (e.g., with dlopen). The runtime system will then invoke the apply_patch function, which will redirect the handle_conn_newest function pointer and execute the user_xform state transformer. Once apply_patch has been run, all subsequent calls to handle_conn will reach the updated version. Then, the program can resume running and benefit from the modifications made in the patch.

This approach, wherein all function calls following an update invoke the most recent version of the function, makes DSU behavior *transparent*—the DSU system orchestrates execution during the update, requiring no developer modification of the original program. One important concern with the transparent approach is update safety. Consider what would happen if a patch that adds an additional mode argument to handle_conn (see Figure 1.5) were applied just prior to its invocation. The resulting execution calls handle_conn with the wrong number of arguments since

```
1  extern float temp;
2  void user_xform() { temp = (temp - 32.0f) / 1.8f; }
3  void handle_conn_v1(conn *c) {
4    /* v1 impl */ ...
5    if (c→op == READ) { respond("TEMP: %f", temp); }
6    else if (c→op == SET) {
7      temp = c→val;
8      /* heat room */
9      heater_set_temp(temp);
10     respond("OK");
11   } else { respond("ERROR"); }
12 }
13 extern void (*handle_conn_newest)(conn *c);
14 void apply_patch() {
15   handle_conn_newest = &handle_conn_v1;
16   user_xform();
17 }
```

Figure 1.4: v0→v1 patch

execution continues in the old version of main following the update. This execution fails to respect the program's types and would likely corrupt the program's state or produce other erroneous behavior.

A common technique to prevent this problem is to employ an automatic *timing restriction*. One common restriction, *activeness safety* (AS), will not apply a patch

```
1  void handle_conn(conn *c, int mode) { /* v2 impl */ ... }
2  int main() {
3    while (1) {
4      conn *c = get_conn_newest();
5      handle_conn(c, HEAT);
6    }
7  }
```

Figure 1.5: v2 code modifications

if any of the functions that it modifies are active on the stack. This restriction would disallow our erroneous patch since main is active and was modified to pass the new argument to handle_conn. By preventing a large class of patches from being applied, AS trades off significant flexibility for safety. However, as we will show in Chapter 2, timing restrictions like AS do not ensure correctness.

1.2 DSU as a Program Feature

Our thesis is that dynamic updating support should be treated as a *program feature*. Specifically, this means that developers need the ability to orchestrate and reason about the program's behavior as an update happens and also reason that the program behaves correctly following the update. Unlike (mostly) transparent technologies like garbage collection, DSU affects the internal semantics of a program and often its external behavior. Therefore, we argue that developers need the same control and reasoning ability over DSU that they require for other program features.

Existing notions of update safety and correctness focus on general properties (e.g., type safety) rather than the specifics of a patch's code and data changes. We argue that developers should specify and check the specific behaviors that an updating program will exhibit just as they do for other program features. We will show that developers can write DSU specifications with little work—usually by adapting single-version specifications—and check them using standard methods: testing and verification.

Implementing DSU as a feature means that the most important parts of an

update's behavior are made manifest in the program's code, exposing the programmer to the details they need to understand, while relying on the DSU system for everything else. We argue that this approach can provide several benefits: simplified developer reasoning about update behavior, modest effort to implement, support for arbitrary program changes, lightweight tool support, and negligible runtime overhead.

1.2.1 Motivation

Our thesis that DSU should be treated as a program feature was initially motivated by an empirical evaluation of *DSU timing restrictions* that we performed. Timing restrictions are the checks that DSU systems use to ensure that updates are not applied at times that might result in erroneous behavior (e.g., the AS check described earlier). Our study compared AS to con-freeness safety (CFS), a related check that attempts to prevent type-unsafe updates more precisely, and manual update selection. We exhaustively tested the times that an update could occur during a system test suite execution for three well-known open source server programs, and tabulated the number of failing tests that would be allowed under each safety check. We summarize this study further in the next section and describe it in detail in Chapter 2. Here, we discuss several problems with transparent updating systems we discovered in the study.

Type safety is not sufficient. The AS and CFS checks only guarantee that an updating execution will not use instances of program types in an inconsistent way

```
1 int *data = NULL;
2 void get_conn() { data = malloc(sizeof(int )); ... }
3 void handle_conn(conn *c, int mode) { .... = *data; ... }
```

Figure 1.6: AS timing error

following an update. We found that, for the particular tests we considered, neither check could prevent all erroneous executions.

To understand why, consider the patch in Figure 1.6 which adds a global variable data to our running example. data is initialized by get_conn and used again by handle_conn. If an update occurs between the calls those functions (between lines 15 and 16 in Figure 1.1), then data will be NULL when handle_conn uses it and the program will crash. By design, AS and CFS will both permit this erroneous update since the resulting execution does not violate type safety. We observed several instances of failures like this during our experiments.

Programs must often be refactored. One of the main arguments for transparency and the use of automatic safety checks like AS or CFS is that they support updates without significant program modification. However, in our experiments, programs often required refactoring to ensure that typical program changes do not prevent updates indefinitely. Consider the modifications to main shown in Figure 1.7.

Even ignoring timing restrictions, these code modifications would never take effect in the running program because the old version of the main function would always remain on the stack—the changes could only be reached if main were invoked again following the update. To support those changes, developers would need to have

```
1 int main() {
2   while (1) {
3     conn *c = get_conn();
4     if (!c) break;
5     handle_conn(c);
6   }
7   cleanup();
8 }
```

Figure 1.7: Modifications to active code

```
1 void loop_body() {
2   conn *c = get_conn();
3   if (!c) break;
4   handle_conn(c);
5 }
6 void loop_return () {
7   cleanup();
8 }
9 int main() {
10   while (1) {
11     loop_body();
12   }
13   loop_return ();
14 }
```

Figure 1.8: Refactorings to update active code

anticipated them and modified their program as shown in Figure 1.8.

In this modified version of the program, a loop and a block of code were pulled out into new functions. The effect is that each iteration of the loop will invoke the extracted function (loop_body in this case) and so the newest version will be reached during each iteration. Likewise the updated version of loop_return is reached upon loop termination. Ginseng calls these changes *loop extractions* [51] and we needed to use them heavily to support useful updates in our experiments. Interestingly, we

```
1  void main_loop() {
2    while (1) {
3      conn *c = get_conn();
4      handle_conn(c);
5    }
6    cleanup();
7  }
8  int main() {
9    main_loop();
10 }
```

Figure 1.9: Modified loop structure

found that one such extraction caused AS to permit a timing error that it otherwise would have been prevented.

Program evolution is restricted. Loop extraction can not always enable new code to be reached. Consider a patch that modifies the loop structure of the original thermostat server as shown in Figure 1.9, where the event-handling loop has been moved into a new function. AS would prevent this update, since main has been modified. However, even if it were allowed, transparent DSU systems cannot "inject" the call to main_loop onto the stack. During our study, we observed that the 0.4.3→0.5.0 patch to ngIRCd contained a similar change that we could not support. We believe developers should be encouraged to improve the overall design of programs, and that doing so should not prevent dynamic updating.

Manual developer reasoning about correctness may be hard. The developer must consider all possible times that an update might occur and reason about whether each combination of old and new code is correct. If a timing restriction is

in use, the developer must additionally reason about which updates it will allow. For instance, the problematic update given in Figure 1.6 will crash for some update timings but not for others. For larger programs, the vast number of places an update could occur would make manual reasoning prohibitive.

We now describe two additional concerns that our empirical study revealed that, while not specifically problems with current DSU mechanisms, make it difficult for developers to ensure that their dynamic updates are correct.

Need to define correctness of changing semantics. To perform the empirical study, we used suites of tests targeting functionality whose external behavior was not changed by the patch. We did so because existing approaches to DSU correctness did not provide a framework for defining the correctness of updates that change behavior. Although we argue in Section 2.6 that the tests we used were sufficient for our study, it is clear that developers using DSU in practice need ways to define correctness for updates that change program semantics.

Need ways to check update correctness. The testing methodology that we used for this empirical study was a novel step towards checking DSU correctness. However, there are many additional tools (e.g., static analysis) that are useful for checking the correctness of program features, but currently provide no support for DSU.

1.3 Overview

In this dissertation, we present three major contributions that provide evidence that DSU should be treated as a program feature and that doing so is effective.

First, we describe our empirical study of the techniques used by existing DSU systems to determine when an update can take place. We find that automatic techniques are unable to prevent erroneous behavior and conclude that placing *update points* in developer-chosen main loops is most effective. We also observed and report several problems with current DSU mechanisms.

Next, we present an approach to specifying the correctness of program features under DSU. We propose a specification strategy that can adapt single-version specifications to describe DSU behavior. We also develop a new tool to support DSU verification that allows reasoning about DSU specifications using standard checking tools. We have implemented our approach for C, and applied it to updates to the Redis key-value store and several synthetic programs.

Finally, we present Kitsune, a new DSU system for C programs that supports the developer in implementing runtime updating as a program feature. We have used Kitsune to update five popular, open-source, single- and multi-threaded programs, and find that few program changes are required to use Kitsune, and that it incurs essentially no performance overhead.

```
1 int main() {
2   while (1) {
3     dsu_update();
4     conn *c = get_conn();
5     handle_conn(c);
6   }
7 }
```

Figure 1.10: Added update point

1.3.1 Empirical Study

Researchers and practitioners have developed several strategies for determining when an updating can be applied to a running program. Three that are in use in DSU systems are AS, CFS, and placing explicit update points in developer-chosen main loops. This last strategy would have the developer add an update point to thermostat example as shown in Figure 1.10. Under this strategy, the update point is placed such that it is reached in between event-handling operations when there is less in-flight state (c represents in-flight state if an update occurs after get_conn). This approach to update-point placement makes update timing (a key facet of update behavior) explicit, and so is part of our strategy to treat DSU as a feature.

To determine which of the strategies is most effective, we performed an empirical study, described in Chapter 2, that considered several real-world updates to three widely used, open-source server programs, vsftpd, OpenSSH, and ngIRCd. We used the standard OpenSSH test suite and developed custom tests for vsftpd and ngIRCd, and exhaustively tested updating at all possible times during their execution.

To perform this experiment, we developed DSUTest, a testing methodology

for DSU that logs all of the possible update points reached during the execution of a system test and then exhaustively tests each of those points. However, for checks like AS and CFS, thousands of update timings may be possible during the execution of a system test. We observed that many distinct updates would yield provably identical behavior, so we developed an analysis that operates over program traces, dividing update tests into classes of equivalent tests. DSUTest only runs one test from each equivalence class. This allows us to test all the distinct behavior that could result from an update during a test, while running the minimum number of update tests.

The results of this study (described in detail in Section 2.5) showed that all three timing restrictions prevented the majority of failures that might have occurred without any restriction. While AS and CFS both prevented most failures, only manual point selection prevented all of the failing updates. Further, for both AS and CFS, a large number of distinct update timings were possible, suggesting that the reasoning burden for developers to ensure update correctness might be prohibitive. Complicating matters, even if developers were to identify a bug through manual reasoning, systems using AS and CFS often do not provide effective ways for developers to use this knowledge to avoid erroneous timings. In summary, we conclude that manual selection of long-running loops is the best approach.

1.3.2 Specifying DSU correctness

Chapter 3 presents our approach for specifying the correctness of dynamic updates. Our approach allow developers to reason about correct DSU behavior as a program feature, instead of relying on generic notions of correctness.

We present *client-oriented specifications* (or *CO-specs* for short) as a way to specify execution properties from clients' points of view, to show that a dynamic update does not disrupt active sessions. Recall that our thermostat server allows a client to issue requests that READ or SET a stored temperature. The developer may expect that a READ that follows a SET will return the set temperature. CO-specs permit developers to specify these types of external behavior. Here, we provide an example CO-spec that specifies that, for any temperature set, the same temperature will be read.

```
1 float written = ?, read;
2 SET(temp);
3 read = READ();
4 assert(written == read);
```

Note that a specification for this property may hold for most patches, but it would not hold for the patch that transformed the target temperature to Celsius (if that update occurs between the SET and READ). Our approach would allow the developer to detect this inconsistency, which they could address by modifying the patch or adapting the specification.

We have identified several categories of CO-specs that capture most properties of interest and can be derived from single-version specifications. However the

strategy is flexible enough to express behaviors that do not match either individual version, and we have encountered program changes for which this ability is useful.

1.3.3 Verification of DSU CO-specs

Chapter 4 presents the first system for automatically verifying dynamic-software-update (DSU) correctness, which we express using CO-specs. Rather than propose a new verification algorithm that accounts for the semantics of updating, we developed a novel program transformation that produces a program suitable for verification with off-the-shelf tools. Our transformation *merges* an old program and an update into a program that simulates running the program and applying the update at any allowable point. Figure 1.11 shows the merged program for the first patch to our running example that switched `temp` to Celsius.

This merged program models all possible updating executions. The `updated` variable indicates whether the program has updated. Calls to `dsu_update` nondeterminstically determine whether to apply a patch. (The condition on ? on line 3 denotes nondeterministic choice). Calls to each function reach a wrapped version that consults the `updated` flag and calls the appropriate version.

We have implemented our merging transformation for C programs and used it in combination with two types of checking tools, verification and symbolic execution, to check actual updates. We have used this approach to check properties of Redis and several synthetic benchmarks inspired by real changes.

```
1  int updated = 0;
2  void dsu_update() {
3    if (!updated && ?) { updated = 1; user_xform() }
4  }
5  float temp;
6  void user_xform() {
7    temp = (temp - 32.0f) / 1.8f;
8  }
9  void handle_conn_old(conn *c) { /* v0 impl */ ... }
10 void handle_conn_new(conn *c) { /* v1 impl */ ... }
11 void handle_conn(conn *c) {
12   if (updated) hand_conn_new(c);
13   else handle_conn_old(c);
14 }
15 int main() {
16   while (1) {
17     dsu_update();
18     conn *c = get_conn();
19     handle_conn(c);
20   }
21 }
```

Figure 1.11: DSU merger output

1.3.4 Updating with Kitsune

Chapter 5 presents Kitsune, a DSU system for C programs that developers can use to implement DSU as a feature. To do so, the developer modifies the initial version of the program to make updating behavior explicit. Kitsune addresses each of the problems with transparent DSU systems that we identified earlier, and operates in harmony with the main reasons developers use C (e.g., performance and low-level control), due to three key design and implementation choices.

1. Kitsune gives the programmer explicit control over the following important facets of DSU behavior: when an update happens, what program state is migrated to the new version, and where in the program execution resumes when the update completes. The code the developer writes to exert this control makes DSU support a first-class program feature, which the developer can easily reason about or modify.

2. Kitsune performs whole-program updating. Each version of the program is compiled as a shared library. Kitsune provides a driver program that starts execution by loading the initial-version library, and each subsequent update `longjmp`s back to the driver, which loads the new version library and invokes its `main` function. This implementation strategy places no restrictions on the structure of the program, its data, or how either may be modified by an update.

3. Kitsune supports state transformation by providing a tool called xfgen that allows the developer to express the interesting parts of state transformation as simple specifications, and xfgen generates code to do the rest. The generated

```
1  float temp;
2  void get_conn() { /* v0 impl */ ... }
3  void handle_conn(conn *c) { /* v0 impl */ ... }
4  int main() {
5    kitsune_do_automigrate ();
6    while (1) {
7      kitsune_update("main");
8      conn *c = get_conn();
9      handle_conn(c);
10   }
11 }
```

Figure 1.12: Kitsune modifications

code can traverse the heap to perform type transformation where needed.

Figure 1.12 shows our networked thermostat server modified to work with Kitsune. Examples in Chapter 5 give a more complete impression of Kitsune's use, but this example demonstrates the approach at a high level.

Line 7 contains an explicit update point. When an update is requested and this line is reached, kitsune_update will save certain program state, including the name of the update point taken ("main" in this example), and longjmp back to the driver. The driver loads the new version and calls its main function. Because we enter the new version of main we know that we will never return to old-version code so there is no risk of type-unsafe execution. The call to kitsune_do_automigrate will pull the values of old-version global variables forward to the new version, performing transformation where needed. When execution reaches the kitsune_update call on line 7, which has the same name as the update point taken in the previous version, the update is complete and the program resumes handling requests.

For our example patch that converts to Celsius (Figure 1.3), the developer would provide the following spec to xfgen:

```
temp → temp: { $out = ($in - 32.0f) / 1.8f; }
```

This code specifies that, to initialize the new-version variable temp (the symbol on the right-hand size of the arrow), run the provided code block with $out bound to temp at the new version and $in bound to the old version. In general xfgen is much more powerful than this: It produces C code that performs transformations for types wherever they are needed in the heap (see Section 5.2 for examples). The code that xfgen generates to perform this tranformation would be automatically called by kitsune_do_automigrate.

We have implemented Kitsune and used it to update three single-threaded programs—vsftpd, Redis, and Tor—and two multi-threaded programs—memcached and icecast. For each application, we considered from three months' to three years' worth of updates. We found that we could support updates with a small number of modifications (on par for a typical program feature), virtually no steady-state overhead, and short delays at update time.

Chapter 6 describes an additional experiment we performed to measure loss of availability due to Kitsune's handling of multi-threaded updates. This study measured time spent waiting for all threads to update for six programs. We describe implementation strategies for minimizing the updating delay for multi-threaded programs, and report our findings which show that all threads reached update points quickly for each of our benchmark programs.

Chapter 2

Evaluation of DSU Timing Restrictions

While DSU can significantly improve application availability, it is not without risk. Even if the new version of an application runs correctly when started from scratch, the application could behave incorrectly when patched on the fly, depending on *when* the update takes effect. To see why, consider the example in Figure 2.1, which shows two versions of a simplified HTTP server. There are two semantics-preserving changes in the new version. First, the escape function used to take a single argument, but now has been changed to take two arguments (and the call to it from parse is updated accordingly). Second, the global cnt, which counts the number commands processed, is now updated in get_file prior to logging, rather than in parse.

Suppose the old program is running and a *dynamic patch* (a patch to be applied at runtime) based on the new program is ready to take effect as control enters parse, at the point marked */**1**/*. In many DSU systems, functions running at the time of an update continue executing the old code, while subsequent function calls invoke the new version [7, 52, 16, 65, 36]. Thus, we would have a type error: the old parse would call the new escape with a single parameter, instead of two parameters as expected, which could lead to surprising behavior.

To avoid these and other problems, most DSU systems support mechanisms

```
1  main() { ...
2    while (1) { /**2**/
3      byte* packet = network_read();
4      struct event *e = parse(packet); /**3**/
5      switch (e→kind) {
6      case GET: get_file (e→gete.fname); break;
7      case PUT: put_file (e→pute.data); break;
8    }
9  }
10 struct event* parse(byte* pkt) { /**1**/
11    pkt = escape(pkt);
12    ... cnt++;
13 }
14 void get_file (char* name) {
15   log (..., cnt ,...);   ...
16 }
17 char* escape(char* buf) { ... }
```

(a) Old version

```
1  main() { ...
2    /*
3
4
5        as before
6
7
8    */
9  }
10 struct event *parse(byte* pkt) {
11   pkt = escape(pkt,METHOD_1);
12    ...
13 }
14 void get_file (char* name) {
15   cnt++; log (..., cnt ,...);   ...
16 }
17 char* escape(char* buf, int mode) { ... }
```

(b) New version

Figure 2.1: Two versions of a program

that constrain when a dynamic patch may be applied. In this work, we evaluate the *effectiveness* of the most well-studied approaches to controlling update timing. We characterize effectiveness as having three facets. The primary criterion is *safety*: an effective approach to controlling DSU timing should rule out incorrect behavior, such as the type error described above. The flip side is *availability*: timing cannot be restricted so much as to preclude a dynamic update for an extended period. Finally, there is *usability*: an effective approach will not require a developer to perform difficult reasoning or work to add DSU support to her program.

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

Activeness safety (AS) In this approach, an update may be performed only if

those functions changed by the update are not *active*, i.e., if changed functions are not on the activation stack of a running thread. AS prevents the update at location /**1**/ in our example by forbidding the update from taking effect in `parse` since it has changed. AS is probably the most popular approach, used by the commercial DSU system Ksplice [8]; the research systems Dynamic ML [66], K42 [38], OPUS [5], and Jvolve [65]; and advocated by Bracha [11] for web-based end-user apps.

Con-freeness safety (CFS) Stoyle et al. [63] observed that AS may be overly restrictive, and proposed a condition called *con-freeness* that allows updates to active code if the old code that executes after the update will never access data or call a function whose type signature has changed. As such, it would rule also out the problematic update point /**1**/ in the example, since `escape`'s type signature has changed and `escape` would be called after the update takes place in `parse`. Unlike AS, however, CFS would allow an update *after* the call to `escape` since subsequent actions in `parse` do not involve code or data whose type has changed, e.g., `cnt` is still a variable of type **`int`**. In general, it has been proved that AS and CFS both guarantee that no updating execution will exhibit a type error [63]. CFS is used by Ginseng, a research system that has successfully supported dozens of dynamic updates to realistic programs [52].

Manually identified update points Several DSU systems, including Erlang [7], UpStare [45], POLUS [16], DLpop [36], DYMOS [41], and Ekiden [35] impose no automatic timing restrictions. Instead, these systems rely on the programmer to identify legal update points, and thus put her firmly in the driver's seat to balance

safety and availability. A common approach, e.g., advocated by Armstrong for Erlang [7], which we call *manual identification*, is to permit updates only at the start or end of event processing loops (e.g., at position /**2**/ in the example). In fact, doing so would help avoid a problem that both AS and CFS allow. Consider if the example update were performed at /**3**/, which is permitted by AS and CFS because main is the only active function, and is unchanged. By this point the program has called the old version of parse, which runs the statement cnt++. After the update, the program will call the new version of get_file , which contains the statement cnt++ where the old version did not. Thus the execution has increased cnt one too many times, resulting in the call to log being incorrect. The manually chosen point at /**2**/ avoids this problem by ensuring calls to functions will always go to the same code version when processing a single command. Identification of the event-processing loops for adding update points is a straightforward, almost mechanical process. The developer just finds the event loops that correspond to places where updates should be supported and adds an update point.

Author Contributions. The testing strategy that we use for this work was published in HotSWUp '09 [30], and the empirical evaluation was published in TSE [34]. I was lead author on both papers, and was the lead contributor to all facets of this work. My collaborators, Eric Hardisty and Edward K. Smith, contributed by writing application tests and helping modify the subject programs to support updating. The implementation was built on the Ginseng DSU system created by Neamtiu et al [51].

2.1 Empirical assessment of timing effectiveness

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

To perform our study, we considered dynamic updates to three mature, open-source applications: vsftpd, a popular FTP server, OpenSSH daemon, a secure shell server, and ngIRCd, an IRC server. The first two applications have already been studied by several DSU systems [52, 16, 45], while the third is new to this study. Each of these programs is single-threaded, although both OpenSSH and vsftpd use multiple processes. For each application we selected a streak of releases, and for each release (after the first) we constructed a dynamic patch using Ginseng, adjusting it as needed for the timing approach under study. For OpenSSH we chose eleven straight releases over a three year period; for vsftpd we chose nine releases over three years; and for ngIRCd we chose eight releases over eight months. Though we use Ginseng for this study, we argue (in Section 2.2.4) that our results generalize to many other DSU systems, since most adopt similar models and mechanisms.

For each release and patch, we executed a suite of system tests (either provided with the application or written by us) and observed whether a test passed when a dynamic update was applied during the test's execution. Each system test induces many *update tests*, with one update test for each distinct moment during the test's

execution at which the update could be applied. By exhaustively running all update tests we can directly assess effectiveness. In particular, we can assess whether an approach to controlling timing would permit a particular update test (availability), and if so, whether that test passes (safety). We can also look holistically at the allowed update points to assess whether they occur often enough at execution time to provide availability. We observe that even a few syntactic update points may be sufficient in practice as long as they are executed suitably often.

Running a test for every possible update point would be prohibitively expensive. Fortunately, many update tests are provably redundant: Suppose a dynamic patch does not change the code of function `f`. Then a test with an update point just before a call to `f` will behave identically to the same test with an update point just after that call to `f`. Therefore, we only need to run one of these two possible update tests, rather than both possible tests. In the implementation of our systematic testing framework, we built on this intuition to produce a *test minimization* algorithm that dramatically reduces the number of tests we have to run while retaining the same coverage [30]. For our experiments in particular, we found that 95% of the update tests from OpenSSH, 96% of points from vsftpd, and 87% of points from ngIRCd could be eliminated.

2.1.1 Results

The results of our study convince us that the approach of manual update point identification is the most effective. In particular, this approach eliminates all

failures, provides sufficient availability, and is relatively easy to use. The two automatic mechanisms we considered do not preclude all failures and required substantial manual effort.

Assessing *usability*, all three methods required some manual effort to *extract* certain blocks of code into separate functions. In particular, each server program contained a potentially infinite main loop that processed client requests. Since updates to functions do not take effect until the next time the function is called, any updates to the loop-containing function would never be realized. To remedy this problem, we extract the body of the loop into a separate function, so that changes to the loop itself effectively take effect on the next loop iteration [52].

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

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

around them by refactoring the code in various ways.

The manual approach required the least additional work. Following the direction of Armstrong mentioned above [7], we simply prescribed that an update may take place at the beginning of each event processing loop (prior to calling the extracted body). Indeed, we needed to identify such positions to extract loop bodies, so adding the manual update point required no additional work.

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

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

While the more allowed update points the better, in general we only need updates to occur reasonably often. We measured the potential delay to updating that would be introduced by updating only at manual points using our test suite and benchmark programs. We found that while AS or CFS may allow an update to occur

more quickly than manual points (since they permit more potential update points), this delay is typically quite short, usually less than 1ms for our tests (although the delay can be longer for certain requests, e.g., large file downloads). In cases where a developer decides that manual update points are reached too infrequently, she can allow faster updates by adding a manual point to the loops that cause the delay. We also categorized the update points in each program by the program phase they occur in—startup, connection loop, transition, command loop, or shutdown—and found that a significant number of the failures occur in the startup and transition phases, providing further support that update points in loops seem the most reliable.

In summary, this work is the first substantial study of several proposed DSU timing restrictions. While others have argued for [63, 52, 49, 8, 66, 38, 5, 65, 11] and against [45, 7, 16, 36, 41, 35] these approaches, these arguments have previously been qualitative. This work is the first to empirically consider the safety, availability, and usability of these approaches when applied to realistic applications. Our in-depth analysis of the data—including a characterization of the failures allowed and disallowed by the checks and where those failures tend to occur—provides a valuable source of information for judging and motivating ongoing DSU research. In particular, our DSU verification work and the Kitsune updating system were both heavily motivated by the findings of this empirical study.

2.2 Dynamic software updating background

We used Ginseng for our empirical study because it has proven to be quite effective; e.g., published work describes how Ginseng has been used to update six open-source server programs, where updates correspond to actual releases taken from several years' worth of development [52, 49]—upwards of 60 dynamic updates in all. Moreover, Ginseng's updating semantics are quite similar to the semantics of many other DSU systems. As such, we believe that results for Ginseng have broad applicability. Finally, we had ready access to Ginseng expertise since it was developed, in part, by one of the author's collaborators on this work.

The next three subsections describe how Ginseng works, first considering its basic mechanisms, then discussing how it handles updates to active code via extraction, and finally considering how it implements timing controls. The updating approach that we described for the example in the Introduction (c.f., Section 1.1) is quite similar in operation to Ginseng. Here we describe the differences and provide some additional details. We close this section by considering how our results could be interpreted with respect to other DSU systems.

2.2.1 Ginseng implementation basics

We noted that Ginseng supports *type transformation* functions to effect type-level conversions, but did not describe how they work. Under Ginseng, if the old program contained definition `struct entry { int key; void *value; }` and the new version modified this definition to be `struct entry { int key; int priority ; void *value; }`,

then the developer must provide a function that can initialize a value of the new version's type given a value of the old version's type, e.g., by copying the values from unchanged fields key and value, and initializing the new field priority. The program is compiled so that type transformers are invoked on demand: each access to data is prefaced by a check of whether the data is up-to-date, and if not, the representation is converted. The Ginseng compiler inserts padding in updateable values so that their representation can grow over time. A patch that modifies a data type to be larger than the running version's padded representation cannot be applied. See Neamtiu et al. for more details [52].

2.2.2 Updating active code in Ginseng

As we noted in the Introduction (Section 1.2.1), functions that are active during an update will complete execution at the same version at which they were initially invoked, and we sometimes need to refactor the program to allow active code to be updated more readily. Ginseng provides annotations that can be used to automatically extract blocks of code within a long-running function into new functions whose arguments include a **struct** containing all the local variables mentioned in the extracted block. A drawback of using code extraction is that developers must anticipate which code to extract before deploying the program.

2.2.3 Controlling timing in Ginseng

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

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

Ginseng implements CFS as a static analysis. The Ginseng compiler analyzes the program source code to determine, for each update point,[1] those definitions that could be used *concretely* beyond that point (function calls, dereferences of global variables, field accesses of structured types, etc.) by the current function or any function that could be on the stack. Then it stores the set of names of those definitions in a data structure at that point. At run-time when control reaches that point

[1]Update points could be inserted manually or automatically

and an update is available, the patch will be compared against the set: if definitions changed by the update appear in the set but have not changed their type signature then the update is permitted, and otherwise it is delayed. In effect, this check allows updates to active functions, but only if Ginseng can prove those functions will not subsequently call functions or access any data whose type signatures have changed.

2.2.4 Timing controls in other DSU systems

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

It is easy to argue that our results generalize to the approaches used by Ksplice [8], Jvolve [65], K42 [38], DLpop [36], Dynamic ML [66] and Bracha [11]. In terms of updating semantics, the main difference between these systems and Ginseng is that Ginseng applies type transformations lazily rather than all at update-time, and so timing-related errors could manifest in Ginseng that would not manifest in the other systems. However, for our experiments all type transformers are pure functions, so the effects of type transformation would be the same if they were applied at update-time.

POLUS [16] and Erlang [7] employ a slightly different updating model than Ginseng: after an update in these systems, the programmer can partially control

whether a function call should reach the newest version or the contemporaneous one. If the programmer were to specify that all calls are to the most recent version, the results would be the same as those for Ginseng given here. We note that the use of versioned function calls can encode the manual approach. For example, the programmer could avoid the problems that occur due to updates at /**1**/ and /**3**/ in Figure 2.1 by specifying all calls but those to the extracted loop body to be contemporaneous calls; this is essentially the approach recommended by Erlang. Our results confirm the effectiveness of this approach.

UpStare [45] is strictly more expressive than Ginseng in that it permits a dynamic patch to transform the execution state (i.e., the PC and stack) of the program. An UpStare patch developer provides a mapping between PC locations in each changed function's old and new versions and writes a function to initialize the stack of the new version based on the stack of the running version. At update time, if a changed function is active at a PC specified in the mapping, the transformation function is used to initialize the stack, and then execution proceeds at the new version's corresponding PC. UpStare's execution state transformations are akin to code extractions for Ginseng and similar systems: they are used to ensure that the correct new code is reached following an update. Thus the failures due to timing that we observe in our Ginseng-based experiments would correspond to failures using UpStare assuming the developer wrote the stack mapping in a way that corresponded with our loop extraction. Although UpStare supports changes that Ginseng cannot (e.g., changing the ordering of functions on the stack), the patches in our experiments did not require its extra expressiveness and so we believe that UpStare mappings

would aim to achieve the same results for these programs as our code extractions. As a result, developers using UpStare could use the manual identification strategy we evaluate here by limiting the mapped points to those at event loops. Any additional points allowed by the developer's mappings may or may not correspond to the AS, CFS, or unrestricted approaches that we evaluate, depending on the developer's choices. Many of the failures we observed, particularly those allowed by AS, could also occur under UpStare, and reflect the hazards of constructing mappings for it.

UpStare also provides some support for AS-like timing restrictions [44]. Patch developers can specify *update constraints* that preclude updates when particular changed functions are active. The UpStare manual indicates that these constraints are useful to reduce the effort in mapping program states between versions. We believe that our findings apply directly to the use of these constraints.

2.3 Testing dynamic updates

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

We begin by outlining the basic testing procedure. Next we present the intuition behind our minimization algorithm, which eliminates tests of update timings

whose outcome is provably equal to the outcome of other tests. Finally, we present details of our testing framework's implementation.

2.3.1 Testing procedure

Our approach to update testing is as follows. Let P_0 and P_1 be two program versions, and let π be a patch that updates P_0 to P_1. To dynamically test π, we must run P_0, apply π at the allowable update points, and then decide whether the ensuing behavior is acceptable. We do this by deriving *update tests*, one per allowable update point, from selected test t in the system test suites of P_0 and P_1. Here, we use the term *update point* in a dynamic sense: each time the same call to DSU_update() is reached during execution we consider it a separate update point. Assuming we have a deterministic, single-threaded program, the update points for an execution can be numbered unambiguously. Thus, we define t^i_π to be the update test that executes P_0 on t and applies π at the ith update point; if the test passes, then we deem π to be correct for point i. Since t should terminate, there will be a finite number of induced update tests t^i_π for a fixed π. To run update tests, we modify the Ginseng runtime to delay patch application to the ith update point reached. Our implementation handles some forms of non-determinism and multi-process (but not multi-threaded) programs, which we describe in Section 2.3.3.

We select the system tests t from which to derive update tests from the test suites of the old and new program version. Let T_i be a suite of system tests for P_i, for $i \in \{0, 1\}$. We use all $t \in (T_0 \cap T_1)$: since they should pass for both P_0 and P_1,

we expect all t^i_π for all i should pass no matter when the update happens during the test execution.

On the other hand, we cannot generally use tests $t \in (T_1 - T_0)$, which consider functionality relevant only to the new version, or tests in $t \in (T_0 - T_1)$, which likely consider deprecated functionality. For these tests, not all update points will necessarily make sense. For example, suppose P_0 is an FTP server, P_1 adds support for a new command qux, and t tests the proper functioning of qux, by logging into the server and then performing the command. For update tests t^i_π where update point i occurs prior to the login procedure finishing, then we can imagine the test will pass. This is because the login procedure has not changed between the two versions and should work identically in both. On the other hand, for update points j that occur after that point, applying the update will be too late: the old version, which does not support qux, will by that point have rejected the command and terminated the test. In general, tests in $(T_1 - T_0)$ may have some preamble during which a dynamic update is legitimate. Likewise, tests in $(T_0 - T_1)$ may have some legitimate post-amble during which an update could occur. Either way, we cannot identify this preamble automatically, so for simplicity we simply do not consider these tests. In Chapter 4, we develop approaches for DSU specification and checking that can cope with changing program semantics.

2.3.2 Update test suite minimization

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

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

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

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

However, suppose dynamic patch π_2 modifies f, g, and h. In this case, none of the update points are equivalent. If we update at line 1, we will call the new

versions of all three functions. If we update at line 3, we will call the old version of f and the new versions of g and h. If the update happens at line 5, we will call the old f and g and the new h. All of these executions may produce reasonable behavior, but we have to test them to find out.

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

For each event in the trace, we determine whether it *conflicts* with patch π. In particular, if the event is a call, read, or write to F (where F is a function, global variable, or a value of named type) then the event conflicts with π if and only if F is changed by the patch. If F is a function, any change to its text constitutes a change to F. (Note that a change to a function called by F does not render F itself changed, by this definition). If F is a global variable, then either a change to its nominal type

or a modification of its contents during state transformation constitutes a change. Finally, if F is a named type, then a modification of F's definition (e.g., changing a struct's set of fields or their nominal types, or changing the nominal type that underlies a typedef) constitutes a change. If there is no conflict, then the update π could be applied before or after the event and the semantics of the overall program trace would be the same. This makes intuitive sense: if we call a function G, but the patch does not change G, then whether we apply the update before or after calling G makes no difference; we will execute the same code for G. [2] On the other hand, if we did update G, then applying the update before the call will result in calling the new G, whereas applying the update after the call will result in calling the old G.

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

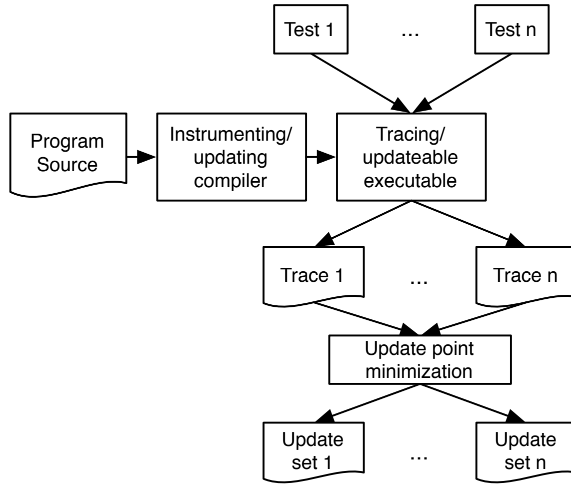
[2] Note that if G itself called some function that would be affected by the update, then this event will also appear in the trace subsequent to the call for G, and it would serve as the source of a conflict.

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

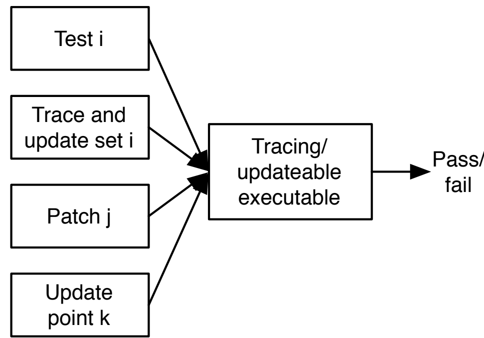
$$\nu = \mathit{update}_1; \mathit{call}(f); \mathit{update}_2; \mathit{call}(g); \mathit{update}_3; \mathit{call}(h)$$

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

We have formalized this minimization algorithm and proven it correct [30, 31]. In practice, the reductions for the three benchmark programs we assess in Section 2.5 were substantial: 95% of update points from OpenSSH, 96% of points from vsftpd, and 87% of points from ngIRCd could be eliminated as redundant. The absolute reduction in update tests was also significant: the initial number of update tests was very large, with over 8M for OpenSSH, 3.9M for vsftpd, and 2.2M for ngIRCd. Running the reduced test suite was time consuming, and would have been prohibitive without reduction. For example, testing OpenSSH with the minimized test suite still required approximately 600 CPU hours to complete. Extensive experimental results assessing the effectiveness of update test minimization for our benchmark programs are given in Appendix Section A.2.



(a) Instrumentation and trace gathering



(b) Running a test case

Figure 2.2: DSU testing framework architecture

2.3.3 Implementation

We extended Ginseng to implement our testing framework. Our extended implementation, called DSUTest, works in two phases, illustrated in Figure 2.2(a) and (b), respectively. In the first phase, the DSUTest compiler instruments the program to log relevant events to a trace file, and then processes each file to find the minimal set of update points to test. In the second phase, the instrumented program replays a given test once per update point identified during the test's minimization,

and tabulates the results.

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

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

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

Non-determinism Our basic methodology presumes that tests are deterministic. However, most programs, including our benchmark servers, exhibit some non-determinism, and thus different runs of the same test may produce slightly different traces. We have encountered non-determinism arising from three main causes. The first is I/O handling by the OS. The main connection loops of our servers block

until they receive a command on a socket, carry out the appropriate behavior, and then continue with the loop. Sometimes the server can wake unpredictably though no I/O is available. In this case, the server "stutter steps" back to the top of the loop, but in doing so may call functions or access data, affecting the trace. Second, the exact timing of any signal handlers can vary between runs. Thus, trace events that occur within a signal handler could be spliced into a trace at different positions in different runs. Finally, some common functionality depends on the environment, such as the current system time, random numbers, and (for vsftpd) process IDs and memory addresses used as hash keys.

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

For the programs in our experiments, we found that it was usually straightforward to designate the code to include in ignore regions. The process entailed comparing several traces produced by executions of a system test. We found that the traces would largely match, except in a few places, as mentioned above. We would then look at the source code that produced the non-determinism and decide whether enclosing the code in an ignore region might mask interesting update be-

havior. In some cases we would add an ignore region; in others, we elected to leave in the non-determinism and rely on match-checking/replay to produce consistent executions. In some cases, several rounds of experimentation were required to get the ignore regions right. To be sure that our experiments are meaningful, we took pains to minimize the size and use of these regions.

Note that we currently limit our focus to single-threaded programs, making no attempt to account for non-determinism that would arise from thread scheduling. In future work, we may explore integrating our framework with techniques for systematically testing under different thread schedules [47, 54] to handle multi-threading.

2.4 Experimental setup

This section describes our experimental setup: which applications we considered, which test suites we used, and how we ran the tests and gathered the data.

2.4.1 Test applications

We tested updates to three long-running server applications: OpenSSH, a widely used SSH server; vsftpd, a popular FTP server; and ngIRCd, an IRC server. Figure 2.3 summarizes the versions of each application that we consider. We largely re-use the OpenSSH and vsftpd dynamic patches used by Neamtiu et al. in their Ginseng work [52], with some changes that we describe in the next section. The OpenSSH releases range from Oct. 2002 to Sept. 2005, and the vsftpd releases range from July 2004 to Feb. 2008. We also developed patches for seven ngIRCd releases

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

Figure 2.3: Version, patch, and test information

that range from Sept. 2002 to May 2003.

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

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

Likewise, the patches to ngIRCd added new features, such as support for IRC commands, *TIME* to display the server time (0.6.0) and *HELP* to list available

commands (0.7.0), as well as new configuration options, e.g., a configurable limit to the number of active connections (0.6.0). These patches also fixed bugs, including buffer overflows (0.5.2), format string errors (0.5.2), and attempts to write on a closed socket (0.5.1). The dynamic patches that we constructed performed transformations like adjusting the lengths of buffers (0.5.2) and modifying the C representation used to hold information about active connections (0.6.0). We also note that we begin our streak of ngIRCd patches at version 0.5.0 because the 0.4.3→0.5.0 patch modified the loop structure of the program in a way that we could not support with Ginseng (or any system with Ginseng-like semantics).

To make it easy to refer to the versions in the subsequent discussion, we number them starting from 0. For each version, Figure 2.3 lists the total lines of code (measured with SLOCCount [67]), the number of update tests (described below), and the number of function signature changes, function body changes, and named type changes (structs, unions and typedefs), that are required to update to the next version. We provide the latter data as it is useful to help explain some of the failures we found, described in the next section.

2.4.2 Test suites

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

features and were developed independently of our evaluation.

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

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

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

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

apply to all tested versions of ngIRCd.

Figure 2.3 shows the single-version line and function coverage information for each tested patch. Line coverage was in the mid-40% range for OpenSSH, and at around 60% for most versions of vsftpd and ngIRCd. Function coverage was in the low-60% range for OpenSSH, in the mid-70% range for vsftpd, and varied from the low-70% range to the low-80% range for ngIRCd. While these figures indicate that some functionality was not tested (e.g., the SSL capabilities of vsftpd, server-to-server connections in ngIRCd, and error handling code generally), our test suites exercise a large number of distinct operations. Even if our test suite does not exhibit every possible DSU timing error for these patches, each set of update tests induced by a system test provides a large set of common update points with which we compare the three timing restrictions. In total, across a variety of real-world patches and tests, we believe our results provide an extensive and realistic corpus of update points for comparison (over 14M in all). Since our goal is to uncover any errors that occur, the test scripts are written to check the correctness of as much of the external behavior as possible, including return codes, response messages, and other effects like downloaded files. Of course, more "blunt" criteria were also used, like ensuring that the program did not crash.

2.4.3 Running tests and tabulating results

As mentioned earlier, updates can take effect at calls to DSU_update(), where these calls can be inserted manually or automatically. For our tests, we directed

DSUTest to automatically insert a call to DSU_update() prior to each function call, and systematically tested the outcome of performing an update at each of these points, with all safety checks disabled. We refer to this set of dynamic update points as *All Pts*. We used our test minimization algorithm (Section 2.3.2) to determine which update tests should actually be performed and then scaled the results back up to the full set of points. For each test execution, we recorded whether the test passed or failed. We marked a run as failing if either the system test itself reports a failure, if the server unexpectedly terminates during the test, or if the test times out. We set the timeout for each run as the time required to gather the initial trace plus 10 seconds.

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

2.5 Experimental results

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

2.5.1 Usability

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

For both AS and CFS, no programmer effort is required to select update points; these are inserted automatically. For the manual approach, we followed the recommended pattern of placing them at the outset of long-running event processing loops [7, 52, 36]. Note that, while we have referred to such update-point placement as "manually identified," it may be possible to automate parts of this relatively systematic procedure. Nevertheless, human judgment is probably necessary to distinguish event handling loops from other loops and to account for the program's update availability requirements. When preparing vsftpd and OpenSSH to support updating, Neamtiu et al. chose to place a single DSU_update() at the beginning of the loop that accepts new connections [52]. We placed an additional update point in each per-session command loop of the applications—some patches we consider add new command handling, and we wanted to allow those to be updated during an active session. OpenSSH provides two distinct command loops to handle different ssh protocol versions, while vsftpd uses only one; so vsftpd contained a total of two calls to DSU_update(), while OpenSSH had three. For ngIRCd there is only one event processing loop, so we placed a single point at its beginning. Following this pattern was quite straightforward: the only work involved was identifying the main loops.

As mentioned in Section 2.2.2 we must manually extract each connection/-

command loop and its cleanup code into separate functions so that each connection loop iteration executes the most recent code and cleans up the server state appropriately when it exits. This task is required for all three timing mechanisms, since in Ginseng (and indeed in nearly all other DSU systems) updates take effect at function calls. (UpStare would not require this effort at the outset, but as discussed in Section 2.2.4, the programmer would have to do something similar when writing her dynamic patch so as to properly map between versions' execution contexts.)

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

```
1 void f() {
2    ...          // startup code
3    g();         // call next function, ultimately reaching
4                 //    the function containing the main loop
5 }
```

In many cases updates change the "startup" code in the functions in this chain (i.e., the code before the call to g() in the schematic), and thus AS would prevent those updates from being applied during the command loop. However, patches can be written to contain state transformation code to execute relevant changes

to the startup code that would have been executed if the program were started from scratch. Therefore, we can (and did) reasonably relax the AS check by also extracting the startup code, so that it is no longer on the stack when the loop executes.[3]

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

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

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

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

Figure 2.4: Points allowed/test failures

2.5.2 Update Safety

Figure 2.4 summarizes the number of update points allowed under each timing restriction for each patch to OpenSSH, vsftpd, and ngIRCd, and how many of those points resulted in a failing test.

The *All Pts* column of Figure 2.4 lists over 1.4M failing update points out of 8M total (17.8%) for OpenSSH, over 128K failing runs out of 3.9M total (3.2%) for vsftpd, and over 308K failing runs out of nearly 2.2M total (13.9%) for ngIRCd. This is clear evidence that applying updates indiscriminately is extremely risky, and thus

timing restrictions are necessary.

The *CFS*, *AS*, and *Manual* columns of Figure 2.4 illustrate that all three timing restrictions disallow the vast majority of failing updates; however both automatic safety checks permit some unsafe updates. For all three programs, CFS allows the most failures, but manages to reduce the total number of failures from 1.4M to 46.8K (96.7% reduction) for OpenSSH, 128K to 2.2K (98.3% reduction) for vsftpd, and 308K to 98 (over 99.9% reduction) for ngIRCd. AS performed even better, allowing only 495 failures (well over 99.9% reduction) for OpenSSH and no failures for vsftpdand ngIRCd. Significantly, only *Manual* identification of update points exhibited no test failures.

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

in the most vsftpd failures.

We investigate the causes of the failures that AS and CFS allow in Section 2.5.4. Appendix A tabulates the relationship between failures (and successes) allowed by both checks.

2.5.3 Update Availability

The most straightforward way to assess update availability to is measure which timing restrictions permit the most update points. Returning to Figure 2.4 we see that both AS and CFS allow many update points, though CFS is more permissive than AS. Both CFS and AS allow several orders of magnitude more update points than are allowed under Manual update point identification. When we consider only the passing update points, as shown in the right half of Figure A.1, the trend continues: In total, CFS permitted 68% of the passing update points, while AS permitted 59% of them, a difference of about 2.1M update tests; roughly 55% of passing update points are allowed by both. The manual approach admitted the least number of update points: about 17.7K, or 0.14% of the passing update points. Thus, across these three approaches to timing restriction, we observe that the lower failure rates of manual point identification (and to a lesser extent AS) come at the cost of fewer correct update points allowed.

Generally speaking, while allowing more correct update points is better than fewer, it also matters where those update points occur during program execution. In particular, since the majority of each server's execution takes place within one

of a few long-running loops, it is crucial that a safe update point is reached on almost every iteration of these loops. Otherwise, we may be unable to update a program in a timely fashion. Assuming loops complete reasonably quickly, and the time transitioning between loops is also quick, just updating in loops may well be sufficient.

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

loops that cause the delay.

2.5.4 Failure examples

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

Failures allowed by CFS The property that distinguishes CFS is that it will execute code that is active at the time of update at the old version, provided this execution will not violate type safety. However, as we mentioned in Section sec:empirical:intro, type-safe executions may nevertheless fail, and indeed we observed cases of this. We have found that sometimes executing a function (or part of it) at the old version and then executing a related function at the new version may induce a failure when the relationship between these two functions changes between versions. We generically refer to these problems as *version consistency errors* [50], since they involve executing the old version of some function and then executing the new version of another where there is a relationship between the two.

One example occurred while testing upload operations against the 1→2 patch to vsftpd. Figure 2.5 shows a simplified version of the relevant code. In this patch, the code that sends the FTP return code 226 indicating a successful transfer was moved from do_file_recv to handle_upload_common. If an update occurs after entering handle_upload_common, but before calling do_file_recv , then the new version of do_file_recv executes and then returns to the old version of handle_upload_common—and thus the server will never write the return code. Eventually this causes the

```
1  void
2  handle_upload_common() {
3    DSU_update();
4    ret = do_file_recv ();
5  }
6  void do_file_recv () {
7    ... // receive file
8    if (ret == SUCCESS)
9      write(226, "OK.");
10   return ret;
11 }
```

(a) Version 1

```
1  void
2  handle_upload_common() {
3    DSU_update();
4    ret = do_file_recv ();
5    if (ret == SUCCESS)
6      write(226, "OK.");
7  }
8  void do_file_recv () {
9    ... // receive file
10   return ret;
11 }
```

(b) Version 2

Figure 2.5: Skipped return code

transfer to time out and fail. Though the code executed following the update in handle_upload_common is changed by the update, the execution is allowed by CFS as the function signatures have not changed. On the other hand, AS precludes the update (and thus, its failure) because handle_upload_common is active.

Failures allowed by CFS and AS While AS prevents the version-consistency failure we just saw, it does not prevent such problems entirely. A particularly interesting example occurs in the 4→5 patch of OpenSSH. This example involves a problem that was not present in the original code, but was introduced via a code extraction step that is needed to permit many other, safe updates to occur.

Figures 2.6(a) and (b) show a highly simplified version of the relevant code for both versions. In version 4, a global pointer is initialized in the serverloop2 function, prior to entry into the command loop. Version 5 moves this initialization earlier into maincont (a function we added during code extraction), prior to calling serverloop2. (In the actual code, the call to serverloop2 is further down the call chain.)

CFS will always allow this update to be applied, because it involves no type

```
1  void maincont() {
2    DSU_update();
3    serverloop2 ();
4  }
5  void serverloop2 () {
6    global_ptr = init ;
7    tmp = (*global_ptr ). pw;
8  }
```

(a) Version 4

```
1  void maincont() {
2    global_ptr = init ;
3    DSU_update();
4    serverloop2 ();
5  }
6  void serverloop2 () {
7    tmp = (*global_ptr ). pw;
8  }
```

(b) Version 5

```
1  void maincont() {
2    extracted ();
3    DSU_update();
4    serverloop2 ();
5  }
6  void extracted () {
7  }
8  void serverloop2 () {
9    global_ptr = init ;
10   tmp = (*global_ptr ). pw;
11 }
```

(c) Ver. 4, after extraction

```
1  void maincont() {
2    extracted ();
3    DSU_update();
4    serverloop2 ();
5  }
6  void extracted () {
7    global_ptr = init ;
8  }
9  void serverloop2 () {
10   tmp = (*global_ptr ). pw;
11 }
```

(d) Ver. 5, after extraction

Figure 2.6: Skipped initialization error

changes, and hence is type-safe. However, if the update indicated in Figure 2.6(a) is taken, then global_ptr will be uninitialized when dereferenced, leading to a crash. On the other hand, AS should prevent this update, because maincont is changed by the update and is active at the update point.

However, recall from Section 2.4 that we extracted the "startup" code in all functions leading up to the command loops in our subject programs. Consider Figures 2.6(c) and (d), which show the two versions of the program after code extraction. Notice that the initialization of global_ptr is moved from serverloop2 to extracted. Thus, the update no longer changes maincont, and when the indicated update point is triggered in our experiments, AS actually allows the update. This example illustrates the tension between update availability and safety when applying AS, and cases like these show the fragility of automatic update safety checks.

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

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

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

2.6 Limitations

Our study found that manual update point identification maximizes update safety and requires the least developer effort while providing sufficient update availability. We now discuss several potential threats to the validity of the study.

- The test suites we used for OpenSSH, vsftpd, and ngIRCd do not exercise all features of the applications, so we may be undercounting how many patches introduce failures into the programs. However, we did endeavor to choose tests that cover the core features of each application, and since we are interested in what the application is doing when it might be updated, we think these tests are representative. For this reason, we did not test cases where the program goes wrong independently of DSU (e.g., error-handling code that only runs

prior to shutting the application).

- As we discussed in Section 2.3.1, our experiments used tests that exercise behavior that persists across the update (although the implementation of that behavior may have changed). This introduces the risk that tests for added/removed/changed behavior might have produced different results. However, defining correct behavior in such cases is not straightforward whereas correctness for our tests was obvious.

- Update points within ignore regions are not tested, so failures due to such points may be missed. We have checked for this possibility by minimizing the size and use of these regions and inspecting their effects. This threat could be completely mitigated by continuing to prevent updates within ignore regions after the application is deployed.

- Our empirical study is limited to three applications and a hand-picked set of updates to them, so the results may not generalize to other applications or updates. This is always a danger with benchmarks. However, we have striven to consider a lengthy streak of updates, and have chosen applications that fit the general mold of single-threaded and/or multi-process server applications written in C. Our results do not directly speak to multi-threaded applications, but for these we note that the qualitative case to be made for manual update points is much stronger than for single-threaded applications, since there are many more application states the programmer must be concerned with [49].

- Our results may be specific to Ginseng, and may not generalize to other updating systems. We think this threat is unlikely, as argued in Section 2.2.4.

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

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

2.7 Related work

While there is much prior work in developing DSU systems (much of which is cited and/or described in Sections 2.2 and 5.5), this work represents the first empirical study of the effectiveness of DSU controls to timing. Most prior work has focused on evaluating different implementation mechanisms (e.g., based on compi-

lation or binary patching), and relatively little focus has been given to assessing the effectiveness of timing mechanisms, particularly for ensuring that updates are safe.

Stoyle et al. [63] proved that CFS, and a flavor of AS, prevent type incorrect executions, but did not evaluate whether the allowed executions may be behaviorally incorrect, as was done in our study. As described in Section 2.2.4, most practical DSU implementations use the AS check but do not evaluate its efficacy, or do so only cursorily. Some systems, such as DyMOS [41] and POLUS [16], permit fine-grained timing controls, but no means to evaluate their proper use is given. Our study is the first to provide empirical data on the effectiveness of common timing controls in a practical setting.

Our approach to generating update tests is related to Chess [47] and MultithreadedTC [54], which test multi-threaded programs by intelligently enumerating a program's potential thread schedules. At a high level, our technique for test minimization is like partial order reduction in model checking [6], 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 [50]. Neamtiu et al. applied this observation to a static analysis for implementing *update transactions* whose execution is version consistent (i.e., consisting of behavior entirely attributable to only one version), while we apply it to test case minimization. Our testing experiments use developer annotations to identify sources of non-determinism in code and compares program traces to be sure they match. Many other techniques have been proposed to provide

deterministic replay including approaches based on libraries [28, 59, 26] and virtual machines [68, 9, 23].

The failure examples in Section 2.5.4 represent the first careful analysis of failures allowed by common DSU systems. The notion of version consistency was identified previously [50], but the relative frequency of version consistency errors was never studied empirically. Indeed, many DSU systems make an implicit assumption that version consistency errors are not a problem [38, 8, 5, 11].

2.8 Conclusions

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

which mechanism, and whether tests of updates at these points succeeded or failed. We also assessed the programmer effort involved to use these mechanisms.

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

Chapter 3

Specifying DSU Correctness

In this chapter, we present a new way of specifying correct DSU behavior in terms of a program's interactions with its clients. The aim of this work is to allow developers to reason about program behavior under DSU as richly as they reason about other program features.

In the previous chapter, we described an empirical study that used testing to ascertain the correctness of dynamic updates. System tests are well-suited to this task because they express execution properties from *clients' points of view*, allowing them to show that a dynamic update does not disrupt active sessions. For example, suppose we wish to update a key-value store such as Redis [56] so that it uses a different internal data structure. To ensure that this update's transformation code is correct, we could test that values inserted into the store by the client are still present after it is dynamically updated.

However, one problem we faced in that work was the lack of a framework for defining correctness for updates that modify program behavior. For this reason, we limited our study to tests for external behavior that did not change between versions. However, the ability to alter program semantics (e.g., fix bugs, modify behavior, and add features) is an essential feature of DSU, and developers need to evaluate the effects of such changes.

Another shortcoming of using tests as specifications is generality. Tests are useful for checking that particular program behaviors are correct for specific input values, but the tests themselves are not specifications—they are instantiations of specifications.

We propose *client-oriented specifications* (or *CO-specs* for short) for specifying correct behavior of event-driven programs and dynamic updates to those programs. CO-specs specify correct interactions between an event-driven program and its clients in a more general way than tests (e.g., by requiring that behavior hold for any input value or for any valid incoming program state). CO-specs may also refer directly to the version that the server is running to describe correct behavior under particular update timings.

We have identified three categories of DSU CO-specs that capture most properties of interest: *backward-compatible* CO-specs describe properties that are identical in the old and new versions; *post-update* CO-specs describe properties that hold after new features are added or bugs are fixed by an update; and *conformable* CO-specs describe properties that are identical in the old and new versions, modulo uniform changes to the external interface. CO-specs in these categories can often be mechanically constructed from CO-specs written for either the old or new program alone. Thus, if a programmer is inclined to specify the behavior of each program version using CO-specs, there is little additional work to specify a dynamic update between the two.

CO-specs are more flexible than generic notions of correctness because they can define an update's correctness as a *collection* of CO-specs from these categories

as well as others for behavior that does not match that of a single program version. We describe prior approaches to DSU correctness in the next section and argue that our approach captures their intent in Section 3.2.4.

In summary, this chapter proposes *client-oriented specifications* as a means to specify general DSU correctness properties, and shows how single-version specifications can be adapted (often mechanically) to specify DSU behavior.

Author Contributions. We presented a paper about our specification approach at VSTTE '12 [32]. I was lead author on the paper and contributed, along with my collaborators Stephen Magill, Michael Hicks, Nate Foster, and Jeffrey S. Foster, to the development of CO-specs and our approach to applying them to DSU.

3.1 Prior work on update correctness

In this section, we review previously proposed notions of dynamic update correctness and argue why they are insufficient for our purposes.

Kramer and Magee [40] proposed that updates are correct if they are *observationally equivalent*—i.e., if the updated program preserves all observable behaviors of the old program. Bloom and Day [10] observed that, while intuitive, this is too restrictive: an update may fix bugs or add new features.

To address the limitations of strict observational equivalence, Gupta et al. [29] proposed *reachability.* This condition classifies an update as correct if, after the update is applied, the program eventually *reaches* some state of the new program. Reachability thus admits bugfixes, where the new state consists of the corrected

code and data, as well as feature additions, where the new state is the old data plus the new code and any new data. Unfortunately, reachability is both too permissive and too restrictive, as shown by the following example. Version 1.1.2 of the vsftpd FTP server introduced a feature that limits the number of connections from a single host. If we update a running vsftpd server, we would expect it to preserve any active connections. But doing so violates reachability. If the number of connections from a particular host exceeds the limit and these connections remain open indefinitely, the server will never enter a reachable state of the new program. On the other hand, reachability would allow an update that terminates all existing connections. This is almost certainly not what we want—if we were willing to drop existing connections we could just restart the server!

We believe that the flaw in all of these approaches is that they attempt to define correctness in a completely general way. We think it makes more sense for programmers to specify the behavior they expect as a collection of properties, just as they might do for program features in the absence of DSU. Some properties will apply to multiple versions of the program while other properties will change as the program evolves. Because the goal of a dynamic update is to preserve active processing and state, the properties should express the expected continuity that a dynamic update is meant to provide to active clients. We therefore introduce *client-oriented specifications* (CO-specs) to specify update properties that satisfy these requirements.

```
1  int get(int k, int *v);
2  void set(int k, int v);
3
4  void arbitrary (int k1) {
5    int k2 = ?, v = ?;
6    if (k1 == k2 || ?)
7      get(k2,&v);
8    else set (k2,v);
9  }
```

(a) interface, helper

```
10 void back_compat_spec() {
11   int k = ?, v_in = ?;
12   int v_out, found;
13   set(k, v_in );
14   while(?) arbitrary (k);
15   found = get(k,&v_out);
16   assert (found &&
17           v_out == v_in);
18 }
```

(b) backward-compat. spec

```
19 void post_update_spec() {
20   int k = ?;
21   int v_out, found;
22   while(?) arbitrary (?);
23   assume(is_updated);
24   delete (k);
25   found = get(k,&v_out);
26   assert (!found);
27 }
```

(c) post-update spec

Figure 3.1: Sample C specifications for key-value store.

3.2 Client-oriented specifications

We can think of a CO-spec as a kind of client program that opens connections, sends messages, and asserts that the output received is correct. CO-specs resemble tests, but certain elements of the test code are left abstract for generality (cf. Figure 3.1). For example, consider again reasoning about updates to a key-value store such as Redis. A CO-spec might model a client that inserts a key-value pair into the store and then looks up the key, checking that it maps to the correct value (even if a dynamic update has occurred in the meantime). We can make such a CO-spec general by leaving certain elements like the particular keys or values used unconstrained. Similarly, we can allow arbitrary actions to be interleaved between the insert and lookup. Such specifications capture essentially arbitrary client interactions with the server.

In our experience writing CO-specs for updates, we have found that they often fall into one of the three categories that we describe now.

3.2.1 Backward compatible CO-specs

Most programs satisfy many of the same properties before and after a dynamic update—e.g., most of a server's behavior that the client observes is unchanged between versions. For instance, in Chapter 2, we observed that OpenSSH's test suite only grew between versions—all of the old tests continued to hold as time went on. This makes intuitive sense: many updates simply add new features, leaving the old features (and properties about them) unchanged, or refactor the program to improve non-functional aspects such as performance.

A *backward-compatible CO-spec* ϕ is one that holds for both the old and new versions independently. Such CO-specs are immediately usable. For example, the CO-spec in Figure 3.1(b) might apply to the old and new program version, and thus it immediately applies to an updating execution; assuming the update could take place during calls to `get` or `set`, we would verify that the update does not drop mappings from the store.

3.2.2 Post-update CO-specs

Another common category of properties consists of those that apply to the new version but not the old version. For example, suppose we added a `delete` feature to the key-value store. Then the CO-spec in Figure 3.1(c) verifies that, after the update, the feature is working properly. The CO-spec employs the flag `is_updated`, which is true after an update has taken place, to ensure that we are testing the new or changed functionality after the update.

$$
\begin{array}{ll}
\mathcal{P}[\![\phi]\!] = & \textsf{while ? do} \\
& \quad \textsf{assume (running } p_0\textsf{);} \\
& \quad \textsf{if ? then } f_0(?) \textsf{ else if ? then } f_1(?) \textsf{ else } \ldots \\
& ; \\
& \textsf{assume (running } p_1\textsf{);} \\
& \phi
\end{array}
$$

(a) Post-update function $\mathcal{P}[\![\cdot]\!]$

$$
\begin{array}{lll}
\mathcal{C}[\![f(v)]\!] & = \textsf{if (running } p_0\textsf{) then } \mathcal{F}[\![f(v)]\!] \textsf{ else } f(v) & \\
& \qquad\qquad\qquad \textit{if } \mathcal{F}[\![f(v)]\!] \textit{ defined} & \\
\mathcal{C}[\![f(v)]\!] & = \textsf{assume (running } p_1\textsf{)}; f(v) & \\
& \qquad\qquad\qquad \textit{if } \mathcal{F}[\![f(v)]\!] \textit{ undefined} & \\
\mathcal{C}[\![\textsf{let } x = e \textsf{ in } e']\!] & = \textsf{let } x = \mathcal{C}[\![e]\!] \textsf{ in } \mathcal{C}[\![e']\!] & \\
\mathcal{C}[\![\textsf{while } e \textsf{ do } e']\!] & = \textsf{while } \mathcal{C}[\![e]\!] \textsf{ do } \mathcal{C}[\![e']\!] & \\
\mathcal{C}[\![\phi']\!] & = \phi' & \textit{for all other } \phi'
\end{array}
$$

(b) Conformance function $\mathcal{C}[\![\cdot]\!]$

ϕ	$\mathcal{C}[\![\phi]\!]$
let $k = ?$ in	let $k =?$ in
let $x = ?$ in	let $x =?$ in
$set(d, k, x)$;	if (running p_0) then $set(k, v)$
	else $set(d, k, v)$;
$del(d, k)$;	assume (running p_1); $del(d, k)$;
let $x'' = get(d, k)$ in	let $x'' =$ (if (running p_0) then $get(k)$
	else $get(d, k)$) in
assert ($x'' = error$)	assert ($x'' = error$)

(c) Conforming new-version spec ϕ using $\mathcal{C}[\![\cdot]\!]$ defined in (b)

Figure 3.2: Transforming new-version specifications

Given a new-version CO-spec ϕ, we can mechanically transform it into a *post-update* CO-spec ϕ', as follows. We can prefix ϕ with an arbitrary sequence of calls into the old program version, ending with the assumption assume (running p_1) to ensure the new version p_1 is running when ϕ is checked. Here, and in what follows, ? denotes a non-deterministically chosen (integer) value, assume discards executions for which its condition does not hold, assume marks executions where its condition does not hold as failing, and running p is true if the program is currently running the code from program version p; in this case p_1 is the new version. Figure 3.2(a) formally defines this transformation as the post-update CO-spec $\mathcal{P}[\![\phi]\!]$, where p_0 defines the functions $f_0, f_1, \ldots$. Thus, $\mathcal{P}[\![\phi]\!]$ can now be checked against an update from p_0 to p_1.

Post-update CO-specs often make sense for updates that add features or fix bugs. However, in general only CO-specs that assume the server could be in an arbitrary initial state are suitable for the post-update transformation. As a trivial example, the CO-spec assert $(get(?) = error)$ explicitly checks that our key-value store starts empty, and may not hold immediately after an update.

3.2.3 Conformable CO-specs

In some cases, updates change the behavior of existing features in a systematic way. For example, the Cassandra distributed database [14] added namespaces to its key-value store when moving from version 0.3 and 0.4. Thus, the new set of server functions now take a namespace identifier as an initial parameter, i.e., set(d,k,v)

associates key k to value v in namespace d, and likewise get(d,k) retrieves the value associated with k in namespace d. After making this change, the developer adapts the existing single-version specifications for the old version to be compatible with the new version. For example, the specification in Figure 3.1(b) would be adjusted so that calls to get and set are made using some default namespace identifier.

To perform this update dynamically, the developer must write a patch whose state transformation function adjusts the key-value store to be compatible with the new code—e.g., any existing key-value pairs already in the server heap could be placed in a default namespace. A reasonable choice is to add a default namespace to each existing key-value pair. To test that this update provides reasonable continuity, we can take a new-version specification ϕ that uses this default namespace and adapt it so that it starts by using the old versions of the changed functions, and then changes to the new version midstream.

We can mechanize this process as follows. We assume we are given a new specification ϕ, as well as a meta-function $\mathcal{F}[\![f(v)]\!]$ that takes a call to a new-version function and transforms it to an appropriate call to an old-version function. As this may not always be possible, $\mathcal{F}[\![\cdot]\!]$ may be partial. Then we can define the meta-function $\mathcal{C}[\![\phi]\!]$ that *conforms* ϕ as shown in Figure 3.2(b). For our example, the developer would define $\mathcal{F}[\![get(d,k)]\!] = get(k)$ and $\mathcal{F}[\![set(d,k,v)]\!] = set(k,v)$. Note that $\mathcal{F}[\![\cdot]\!]$ bears some resemblance to Ajmani et al.'s *future simulation objects* [4], which are bits of code added to old-version servers whose aim is to convert calls from new clients to work with the old code. We are not deploying these conformance functions on-line, but rather are using them to adjust existing specifications to check

proper continuity following an update.

Now suppose that the new version also adds a new function that permits a client to delete an entry: $del(d, k)$ removes any association with k from namespace d. Since there is no analogue to *del* defined in the old version, there is no backward translation for calls $del(d, k)$ that could appear in new-version specifications. To see how $\mathcal{C}[\![\cdot]\!]$ works in this case, consider the example given in Figure 3.2(c), which shows ϕ and $\mathcal{C}[\![\phi]\!]$ side by side. Here, $\mathcal{C}[\![\phi]\!]$ permits updates to happen up until the *del* call, at which point we assume the update has taken place. (This means that the $\mathsf{running}\ p_0$ check that follows it will always be false.)

3.2.4 Relation to prior notions of correctness

These categories encompass prior notions of correctness. Backward compatible specifications capture the spirit of Kramer and Magee's condition, but apply to individual, not all, behaviors. The combination of backward-compatible and post-update specifications capture Bloom and Day's notions of "future-only implementations" and "invisible extensions"—parts of a program whose semantics change but not in a way that affects existing clients [10]. The combination of backward-compatible and conformable specifications match ideas proposed by Ajmani et al. [4], who studied dynamic updates for distributed systems and proposed mechanisms to maintain continuity for clients of a particular version.

CO-specs can also be used to express the constraints intended by Gupta's *reachability* while side-stepping the problem that reachability can leave behavior

under-constrained. For example, for the vsftpd update mentioned above, the programmer can directly write a CO-spec that expresses what should happen to existing client connections, e.g., whether all, some, or none should be preserved. In any of these cases, the specified behavior does not exactly match either individual version and does not fall into one of the categories above.

In summary, DSU CO-specs improve on these prior notions by allowing the various program behaviors of a single dynamic update to be specified using different classes of specs or specs for arbitrary behavior that does not fit into one of these classes. The common cases of backward-compatible and post-update properties can be handled easily, but the developer retains the power to specify other behaviors when necessary. These advantages demonstrate the utility of a full specification language over "one size fits all" notions of update correctness.

3.3 Applications of CO-specs

We expect CO-specs to be useful to DSU developers in a variety of ways and their intended use will determine how they should be expressed. If developers only use CO-specs for communicating DSU requirements, then they could be effectively expressed using natural language. Likewise, if they are to be checked using testing, then they could be written in any programming language capable of connecting to the updated program[1]. For testing, what we have written as function calls to represent external requests to the updated program might, in fact, be calls

[1] The updated program would additionally need to externalize its updating status (e.g., to a log) that the test for a CO-spec could reference.

to the program's client API (or even raw network calls reaching the program). Nondeterministically chosen values within a spec might translate to calls to `random()` or uses of fixed, arbitrary values. Special library support could be used to hide some of these details and support CO-specs as we have written here.

In the experiments we report in the Chapter 4, we aim to verify CO-specs using off-the-shelf tools by applying the program *merging* transformation that we define in Section 4.1. The checking tools that we apply in our experiments only verify single programs in isolation, so we cannot literally write CO-specs as client programs that communicate with a server being updated. To verify a CO-spec for a client-server program using these tools we write the CO-spec in the same programming language as the updated program, replace the server's `main` function with the CO-spec, and have the CO-spec call the relevant server functions directly. In doing so, we are checking the server's core functionality, but not its main loop or any networking code. For example, suppose our key-value store implements functions `get` and `set` to read and write mappings from the store, and the server's main loop would normally dispatch to these functions. We write our CO-specs to call the functions directly as shown in Figure 3.1. If updates are permitted while executing either `get` or `set`, verifying Figure 3.1(b) will establish that the assertions at the end of the specification hold no matter when the update takes place.

Chapter 4

Verifying DSU CO-specs

This chapter present a methodology for verifying that an dynamic update conforms to its CO-specs. Rather than propose a new verification algorithm that accounts for the semantics of updating, we develop a novel program transformation that produces a program suitable for verification with off-the-shelf tools. Our transformation *merges* an old program and an update into a program that simulates running the program and applying the update at any allowable point. We formalize our transformation and prove that it is correct (Section 4.1).

We have implemented our merging transformation for C programs and used it in combination with two existing tools to verify CO-specs for several dynamic updates (Section 4.2). We chose the symbolic executor Otter [57] and the verification tool Thor [42] as they represent two ends of the design space: symbolic execution is easy to use and scales reasonably well but is incomplete, while verification scales less well but provides greater assurance. We wrote two synthetic benchmarks, a key-value store and a multiset implementation, and designed dynamic patches for them based on realistic changes (e.g., one change was inspired by an update to the storage server Cassandra [14]). We also wrote dynamic patches for six releases of Redis [56], a popular, open-source key-value store. We used the Redis code as-is, and wrote the state transformation code ourselves. We wrote CO-specs for each of

these updates, similar to the ones described in the last chapter.

We checked all the benchmark programs with Otter and verified several properties of the synthetic updates using Thor. Both tools successfully uncovered bugs that were intentionally and unintentionally introduced in the state transformation code. The running time for verification of merged programs was roughly four times slower than single-version checking. This slowdown was due to the additional branching introduced by update points and the need to analyze the state transformer code. As tools become faster and more effective, our approach will scale with them. This work presents the first automated technique for verifying the behavioral correctness of dynamic updates. It shows the effectiveness of merging-based verification on practical examples, including Redis [56], a widely deployed server program.

Author Contributions. We presented a paper about our verification approach at VSTTE '12 [32]. I was lead author, implemented the merging strategy that we applied, wrote the patches to Redis, and applied checking using Otter. My collaborators, Stephen Magill, Michael Hicks, Nate Foster, Jeffrey S. Foster and I designed our overall approach. The equivalence proof was performed by Michael Hicks and Nate Foster and has been included in Appendix B for completeness. Stephen Magill additionally applied our approach using the Thor verifier. We used some code from the Ginseng DSU system created by Neamtiu et al. [51] to build the merger.

Prog.	p	$::=$	$p, (g, \lambda x.e) \mid \cdot$	*Variables*	x, y, z		
Exprs.	e	$::=$	$v \mid v_1\ op\ v_2 \mid v_1(v_2) \mid ? \mid !v \mid \mathsf{ref}\ v \mid$	*Globals*	f, g		
			$v_1 := v_2 \mid \mathsf{if}\ v\ e_1\ e_2 \mid \mathsf{update} \mid$	*Operators*	op		
			$\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid \mathsf{assume}\ v \mid$	*Integers*	i, j		
			$\mathsf{while}\ e_1\ \mathsf{do}\ e_2 \mid \mathsf{assert}\ v \mid$	*Addresses*	a		
			$\mathsf{running}\ p \mid \mathsf{error}$	*Heaps*	σ	$\in$	*Locs* $\rightharpoonup$ *Values*
Values	v	$::=$	$x \mid l \mid i \mid (v_1, v_2) \mid ()$	*Patch*	π	$::=$	(p, e)
Locs.	l	$::=$	$a \mid g$	*Labels*	ν	$::=$	$\pi \mid \epsilon$

$\langle p; \sigma; v_1\ op\ v_2 \rangle$	$\rightsquigarrow \langle p; \sigma; v' \rangle$	$v' = [\![op]\!](v_1, v_2)$
$\langle p; \sigma; \mathsf{ref}\ v \rangle$	$\rightsquigarrow \langle p; \sigma[a \mapsto v]; a \rangle$	$a \notin dom(\sigma)$
$\langle p; \sigma; !l \rangle$	$\rightsquigarrow \langle p; \sigma; v \rangle$	$\sigma(l) = v$ and $l \notin dom(p)$
$\langle p; \sigma; a := v \rangle$	$\rightsquigarrow \langle p; \sigma[a \mapsto v]; v \rangle$	$a \in dom(\sigma)$
$\langle p; \sigma; g := v \rangle$	$\rightsquigarrow \langle p; \sigma[g \mapsto v]; v \rangle$	$g \notin dom(p)$
$\langle p; \sigma; ? \rangle$	$\rightsquigarrow \langle p; \sigma; i \rangle$	for some i
$\langle p; \sigma; \mathsf{let}\ x = v\ \mathsf{in}\ e \rangle$	$\rightsquigarrow \langle p; \sigma; e[v/x] \rangle$	
$\langle p; \sigma; f(v) \rangle$	$\rightsquigarrow \langle p; \sigma; e[v/x] \rangle$	$p(f) = \lambda x.e$
$\langle p; \sigma; \mathsf{if}\ 0\ e_1\ e_2 \rangle$	$\rightsquigarrow \langle p; \sigma; e_2 \rangle$	
$\langle p; \sigma; \mathsf{if}\ v\ e_1\ e_2 \rangle$	$\rightsquigarrow \langle p; \sigma; e_1 \rangle$	$v \neq 0$
$\langle p; \sigma; \mathsf{while}\ e_1\ \mathsf{do}\ e_2 \rangle$	$\rightsquigarrow \langle p; \sigma; \mathsf{let}\ x = e_1\ \mathsf{in}$ $\mathsf{if}\ x\ (e_2; \mathsf{while}\ e_1\ \mathsf{do}\ e_2)\ 0 \rangle$	$x \notin fv(e_1, e_2)$
$\langle p; \sigma; \mathsf{update} \rangle$	$\rightsquigarrow \langle p; \sigma; 0 \rangle$	
$\langle p; \sigma; \mathsf{update} \rangle$	$\overset{\pi}{\rightsquigarrow} \langle p_\pi; \sigma; (e_\pi; 1) \rangle$	$\pi = (p_\pi, e_\pi)$
$\langle p; \sigma; \mathsf{running}\ p \rangle$	$\rightsquigarrow \langle p; \sigma; 1 \rangle$	
$\langle p; \sigma; \mathsf{running}\ p' \rangle$	$\rightsquigarrow \langle p; \sigma; 0 \rangle$	$p' \neq p$
$\langle p; \sigma; \mathsf{assume}\ v \rangle$	$\rightsquigarrow \langle p; \sigma; v \rangle$	$v \neq 0$
$\langle p; \sigma; \mathsf{assert}\ v \rangle$	$\rightsquigarrow \langle p; \sigma; v \rangle$	$v \neq 0$
$\langle p; \sigma; \mathsf{assert}\ 0 \rangle$	$\rightsquigarrow \langle p; \sigma; \mathsf{error} \rangle$	
$\langle p; \sigma; \mathsf{let}\ x = \mathsf{error}\ \mathsf{in}\ e \rangle$	$\rightsquigarrow \langle p; \sigma; \mathsf{error} \rangle$	

$$\frac{\langle p; \sigma; e_1 \rangle \overset{\nu}{\rightsquigarrow} \langle p'; \sigma'; e_1' \rangle}{\langle p; \sigma; \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \rangle \overset{\nu}{\rightsquigarrow} \langle p'; \sigma'; \mathsf{let}\ x = e_1'\ \mathsf{in}\ e_2 \rangle}$$

Figure 4.1: Syntax and semantics.

4.1 Verification via program merging

We verify CO-specs by *merging* an existing program version with its update, so that the semantics of the merged program is equivalent to the updating program. This section formalizes a semantics for dynamic updates to single-threaded programs, then defines the merging transformation and proves it correct with respect to the semantics. Many server programs for which dynamic updating is useful are single-threaded [36, 52, 34]. However, an important next step for this work would be to adapt it to support updates to multi-threaded (and distributed) programs.

4.1.1 Syntax

The top of Figure 4.1 defines the syntax of a simple programming language supporting dynamic updates. It is based on the Proteus dynamic update calculus [63], and closely models the semantics of common DSU systems, including Ginseng [52] (which is the foundation of our implementation), Ksplice [8], Jvolve [64], K42 [38], DLpop [36], Dynamic ML [66] and Bracha's DSU system [11].

A *program* p is a mapping from function names g to functions $\lambda x.e$. A function body e is defined by a mostly standard core language with a few extensions for updating. Our language contains a construct update, which indicates a position where a dynamic update may take effect. To support writing specifications, the language includes an expression ?, which represents a random integer, and expressions assume v, assert v, and running p, all of whose semantics are discussed below. Expressions are in administrative normal (A-normal) form [25] to keep the semantics simple—e.g., instead of $e_1 + e_2$, we write let $x = e_1$ in let $y = e_2$ in $x + y$. We write $e_1; e_2$ as shorthand for let $x = e_1$ in e_2, where x is fresh for e_2.

4.1.2 Semantics

The semantics, given in the latter half of Figure 4.1, is written as a series of small-step rewriting rules between *configurations* of the form $\langle p; \sigma; e \rangle$, which contain the program p, its current heap σ, and the current expression e being evaluated. A heap is a partial function from locations l to values v, and a location l is either a (dynamically allocated) address a or a (static) global name g. Note that while the

language does not include closures, global names g are values, and so the language does support C-style function pointers.[1]

Most of the operational semantics rules are straightforward. We write $e[x/v]$ for the capture-avoiding substitution of x with v in e. We assume that the semantics of primitive operations op is defined by some mathematical function $[\![op]\!]$; e.g., $[\![+]\!]$ is the integer addition function. Loops are rewritten to conditionals, where in both cases a non-zero guard is treated as true and zero is treated as false. Addresses a for dynamically allocated memory must be allocated prior to assigning to them, whereas a global variable g is created when it is first assigned to. This semantics allows state transformation functions, described below, to define new global variables that are accessible to an updated program.

The update command identifies a position in the program at which a dynamic update may take place. Semantically, update non-deterministically transitions either to 0, indicating that an update did not occur, or to 1 (eventually), indicating that a dynamic update was available and was applied.[2] In the case where an update occurs, the transition arrow is labeled with the patch π; all other (unadorned) transitions implicitly have label ϵ. A patch π is a pair (p_π, e_π) consisting of the new program code (including unmodified functions) p_π and an expression e_π that transforms the current heap as necessary, e.g., to update an existing data structure or add a new one for compatibility with the new program p_π. In practice, e_π will be a call to a

[1] Variables names x are values so that we can use a simple grammar to enforce A-normal form. The downside is that syntactically well-formed programs could pass around unbound variables and store them in the heap. The ability to express such programs is immaterial to our modeling of DSU, and could be easily ruled out with a simple static type system.

[2] In practice, update would be implemented by having the run-time system check for an update and apply it if one is available [36].

function defined in p_π. The transformer expression e_π is placed in redex position and is evaluated immediately; to avoid capture, non-global variables may not appear free in e_π. Notice that an update that changes function f has no effect on running instances of f since evaluation of their code began prior to the update taking place.

The placement of the update command has a strong influence on the semantics of updates. Placing update pervasively throughout the code essentially models asynchronous updates. Or, as prior work recommends [40, 4, 52, 34], we could insert update selectively, e.g., at the end of each request-handling function or within the request-handling loop, to make an update easier to reason about

The constructs running p, assume v, and assert v allow us to write specifications. The expression running p returns 1 if p is the program currently running and 0 otherwise; i.e., we encode a program version as the program text itself. The expression assert v returns v if it is non-zero, and error otherwise, which by the rule for let propagates to the top level. Finally, the expression assume v returns v if v is non-zero, and otherwise is stuck.

4.1.3 Program merging transformation

We now present our program merging transformation, which takes an old program configuration $\langle p, \sigma, e \rangle$ and a patch π and yields a single *merged program* configuration, written $\langle p, \sigma, e \rangle \rhd \pi$. We present the transformation formally and then prove that the merged program is equivalent to the original program with the patch applied dynamically.

$$\llbracket p', (g, \lambda y.e) \rrbracket^{p,\pi} \triangleq \llbracket p' \rrbracket^{p,\pi},\ (g, \lambda y.\llbracket e \rrbracket^{p,\pi}),\ (g_{ptr}, \lambda y.\mathsf{let}\ z = \mathit{isupd}()\ \mathsf{in}\ \mathsf{if}\ z\ g'(y)\ g(y))$$

$$\llbracket \cdot \rrbracket^{p,\pi} \triangleq (\cdot, (\mathit{isupd}, \lambda y.\mathsf{let}\ z = !\mathit{uflag}\ \mathsf{in}\ z > 0))$$

(a) Old version programs

$$\{\!| p', (g, \lambda y.e) |\!\}^{p} \triangleq \{\!| p' |\!\}^{p}, (g', \lambda y.\{\!| e |\!\}^{p})$$

$$\{\!| \cdot |\!\}^{p} \triangleq \cdot$$

(b) New version programs

$$\llbracket g \rrbracket^{p,\pi} \triangleq \begin{cases} g_{ptr} & \text{if } p(g) = \lambda x.e \\ g & \text{otherwise} \end{cases}$$

$$\llbracket \mathsf{running}\ p'' \rrbracket^{p,(p_\pi,e_\pi)} \triangleq \begin{cases} \mathsf{let}\ z = \mathit{isupd}()\ \mathsf{in}\ z = 0 & \text{if } p = p'' \\ \mathit{isupd}() & \text{if } p_\pi = p'' \\ 0 & \text{otherwise} \end{cases}$$

$$\llbracket \mathsf{update} \rrbracket^{p,(p_\pi,e_\pi)} \triangleq \mathsf{let}\ z = \mathit{isupd}()\ \mathsf{in}\ \mathsf{if}\ z\ 0\ (\mathit{uflag} := ?;\ \mathsf{let}\ z = \mathit{isupd}()\ \mathsf{in}\ \mathsf{if}\ z\ (\{\!| e |\!\}^{p_\pi}; 1)\ 0)$$

(c) Old version expressions

$$\{\!| g |\!\}^{p} \triangleq \begin{cases} g' & \text{if } p(g) = \lambda x.e \\ g & \text{otherwise} \end{cases}$$

$$\{\!| \mathsf{running}\ p'' |\!\}^{p} \triangleq \begin{cases} 1 & \text{if } p = p'' \\ 0 & \text{otherwise} \end{cases}$$

$$\{\!| \mathsf{update} |\!\}^{p} \triangleq 0$$

(d) New version expressions

$$\langle p; \sigma; e \rangle \rhd \pi \triangleq \langle \overline{p}, \overline{\sigma}[\mathit{uflag} \mapsto i], \overline{e} \rangle$$

$$\text{where } (p_\pi, e_\pi) = \pi \quad \overline{p} = \{\!| p_\pi |\!\}^{p_\pi}, \llbracket p \rrbracket^{p,\pi} \quad \overline{e} = \llbracket e \rrbracket^{p,\pi}$$

$$i \le 0 \quad \overline{\sigma} = \{ l \mapsto \llbracket v \rrbracket^{p,\pi} \mid \sigma(l) = v \}$$

(e) Merging a configuration and a patch

Figure 4.2: Merging transformation (partial).

The definition of $\langle p, \sigma, e \rangle \rhd \pi$ is given in Figure 4.2(e). It makes use of functions $\llbracket \cdot \rrbracket^{\cdot}$ and $\{\!| \cdot |\!\}^{\cdot}$, defined in Figure 4.2(a)–(d). We present the interesting cases; the remaining cases are translated structurally in the natural way. For simplicity, the transformation assumes the updated program p_π does not delete any functions in p. Deletion of function f can be modeled by a new version of f with the same signature as the original and the body $\mathsf{assert}\,(0)$.

The merging transformation renames each new-version function from g to g', and changes all new-version code to call g' instead of g (the first rewrite rules in

Figure 4.2(b) and (d), respectively). For each old-version function g, it generates a new function g_{ptr} whose body conditionally calls the old or new version of g, depending on whether an update has occurred (Figure 4.2(a)). The transformation introduces a global variable *uflag* (Figure 4.2(e)) and a function *isupd* to keep track of whether the update has taken place (bottom of Figure 4.2(a)). All calls to g in the old version are rewritten to call g_{ptr} instead (top of Figure 4.2(c)).

The transformation rewrites occurrences of `update` in old-version code into expressions that check whether *uflag* is positive (bottom of Figure 4.2(c)). If it is, then the update has already taken place, so there is nothing to do. Otherwise, the transformation sets *uflag* to ?, which simulates a non-deterministic choice of whether to apply the update. If *uflag* now has a positive value, the update path was chosen, so the transformation executes the developer-provided state transformation e, which must also be transformed according to $\{\!|\cdot|\!\}^{\cdot}$ to properly reference functions in the new program. While this transformation results in multiple occurrences of the expression e, in practice e is a call to a state transformation function defined in the new version and so does not significantly increase code size.

Version tests `running` p are translated into calls to *isupd* in the old version, and to appropriate constants in the new code (since we know the update has occurred if new code is running).

While we focus on merging a program with a single update, the merging strategy can be readily generalized to multiple updates. To see the basic idea, consider a process $\langle p, \sigma, e\rangle$ and a sequence of two updates π_1 and π_2. We would first merge $\langle p, \sigma, e\rangle$ and π_1, producing $\langle p, \sigma, e\rangle \rhd \pi_1$. Then we would merge the result with π_2,

essentially producing $(\langle p, \sigma, e\rangle \rhd \pi_1) \rhd \pi_2$. To do this properly requires some small changes to the transformation. First, we need additional bookkeeping information to be passed between iterations of the transformation, e.g., instead of just g and g' as the old and new function names, we would have g^0, g^1, g^2, etc., and likewise *uflag* becomes $uflag^1$, $uflag^2$, etc. (Interestingly, no changes are needed to translations of $\mathsf{running}\ p$, essentially since functions that have not changed are redundantly included in the patch and distinguished by the transformation.) Second, we must change $\{\!|\mathsf{update}|\!\}^p$ to be the identity, i.e., to leave the new version's update keyword in place, so that it can be used to update to the next version to be merged.

With a more general merging transformation we can prove properties about multiple updates. For backward-compatible CO-specs there is no additional work since they are the same across all versions. For post-update specifications, we could generalize the transformation in Figure 3.2(a) so that the assumption in the loop is $\mathsf{assume}\ (\mathsf{running}\ p_0 \vee \cdots \vee \mathsf{running}\ p_{n-1})$ and the assumption after the loop is $\mathsf{assume}\ (\mathsf{running}\ p_n)$, where the post-update CO-spec spans versions p_0 through p_n. We can make a similar generalization of the transformation in Figure 3.2(c) (composing multiple conformance functions together).

4.1.4 Equivalence

We can now prove that an update to an old-program configuration is correct if and only if the result of merging that configuration and the update is correct. This result lets us use stock verification tools to check properties of dynamic updates

using the merged program, which simulates updating, instead of having to develop new tools or extend existing ones.

We say that a program and a sequence of updates are *correct* if evaluation never reaches error (i.e., if there are no assertion failures). More formally:

Definition 1 (Correctness)**.** *A configuration $\langle p; \sigma; e\rangle$ and an update π are* correct*, written $\models \langle p; \sigma; e\rangle, \pi$, if and only if for all p', σ', e' it is the case that $\langle p; \sigma; e\rangle \stackrel{\pi}{\leadsto}^* \langle p'; \sigma'; e'\rangle$ implies e' is not* error*.*

The expression e at startup could be a call to an entry-point function (i.e., main). A correct program need not apply π, though no other update may occur. When no update is permitted we write $\models \langle p; \sigma; e\rangle$.

Theorem 1 (Equivalence)**.** *For all p, σ, e, π such that $dom(p_\pi) \supseteq dom(p)$ we have that* $\models \langle p; \sigma; e\rangle, \pi$ if and only if $\models (\langle p, \sigma, e\rangle \rhd \pi)$.

The proof is by bisimulation and is given in Appendix B along with proof sketches of key supporting lemmas.

Observe that type errors result in stuck programs, e.g., !1 does not reduce, while the above theorem speaks only about reductions to error. We have chosen not to consider type safety in the formal system to keep things simple; adding types, we could appeal to standard techniques [63, 64, 66, 22]. Our implementation catches type errors that could arise due to a dynamic update by transforming them into assertion violations. In particular, we rename functions and global variables whose type has changed prior to merging, essentially modeling the change as a deletion of one variable and the addition of another. Deleted functions are modeled

as mentioned above, and deleted global variables are essentially assigned the error expression. Thus, any old code that accesses a stale definition post-update (including one with a changed type) fails with an assertion violation.

4.2 Experiments

To evaluate our approach to verification, we have implemented the merging transformation for C programs, with the additional work to handle C being largely routine. We merged several programs and dynamic updates and then checked the merged programs against a range of CO-specs. We analyzed the merged programs using two different tools: the symbolic executor Otter, developed by Ma et al. [57], and the verification tool Thor, developed by Magill et al. [43]. The tools represent a tradeoff: Otter is easier to use and more scalable but provides incomplete assurance, while Thor can guarantee correctness but is less scalable and requires more manual effort. Overall, both tools proved useful. Otter successfully checked all the COs-specs we tried, generally in less than one minute. Thor was able to fully verify several updates, though running times were longer. Both tools found bugs in updates, including mistakes we introduced inadvertently. On average, verification of merged code took four times longer than verification of a single version. Since our approach is independent of the verification tool used, its performance and effectiveness will improve as advances are made in verification technology.

4.2.1 Programs

We ran Otter and Thor on updates to three target programs. The first two are small, synthetic examples: a multiset server, which maintains a multiset of integer values, and a key-value store. For each program, we also developed a number of updates inspired by common program changes such as memory and performance optimizations and semantic changes observed in real-world systems such as Cassandra [14]. The third program we considered is Redis [56], a widely used open-source key-value server. At roughly 12k lines of C code, Redis is significantly larger that our synthetic examples, and is currently not tractable for Thor. We developed six dynamic patches for Redis that update between each pair of consecutive versions from 1.3.6 through 1.3.12, and we also wrote a set of CO-specs that describe basic correctness properties of the updates.

As we mention in Section 3.3, CO-specs are joined with the server code and the main function invokes the CO-spec after it initializes server data structures. The new-version source code includes the state transformation code, which is identified by a distinguished function name recognized by the merger.

Synthetic Examples. Figure 4.3 lists the synthetic benchmarks we constructed for our multiset and key-value store programs. Each grouping of rows shows a dynamic update and a list of CO-specs we wrote for that update. The multiset program has routines to add and delete elements and to test membership. The updates both change to a set semantics, where duplicate elements are disallowed. The first (correct) state transformer removes all duplicates from a linked list that maintains

Program – *change* CO-specs	**Thor time (s)**			**Otter time (s)**		
	old	**new**	**mrg**	**old**	**new**	**mrg**
Multiset – *disallow duplicates* (correct)						
mem-mem^b	90.11	121.27	1003.22	6.29	9.72	49.37
add-mem^b	64.17	89.71	537.01	3.26	10.48	50.84
add-add-del-set^g			–			4.04
Multiset – *disallow duplicates* (broken)						
mem-mem^b	25.33	57.78	133.68	6.28	9.77	42.5
add-mem^b	15.68	33.50	80.07	3.25	9.94	33.53
add-add-del-set-fails^g			122.71			5.49
Key-value store – *bug fix*						
put-get^b	27.01	26.13	41.62	3.28	2.54	18.42
new-def-shadows^g			–			4.19
new-def-shadows-bc-fails^b	38.97	41.52	117.56	3.88	2.06	19.03
Key-value store – *added namespaces*						
new-def-shadows-post^p		–	–		1.02	2.99
put-get^p		–	–		18.32	228.69
new-def-shadows-conf^c	–	–	–	1.19	1.93	7.53
put-get-conf^c	–	–	–	4.23	7.09	61.41
Key-value store – *optimization* (broken)						
put-get-back^b	42.133	–	–	2.08	11.01	56.44
new-def-shadows-back^b	15.344	–	–	2.14	11.33	56.03
Key-value store – *optimization* (correct)						
put-get-back^b	41.87	–	–	2.07	10.87	69.31
new-def-shadows-back^b	15.72	–	–	2.14	10.96	68.95

b – backward compatible p – post update c – conformable g – general
A dash indicates that the example could not be verified.

Figure 4.3: Synthetic examples.

the current multiset. The second update has a broken state transformer that fails to remove duplicates.

The key-value store program also implements its store with a linked list. The updates are inspired by code changes we have seen in practice and include a bug fix (bindings could not be overwritten), a feature addition (adding namespaces), and an optimization (removing overwritten bindings), where for this last update the state transformer was broken at first.

The properties span all the categories of CO-specs that we outlined in Section 3.2. Backward compatible specs, such as add-mem, check core functionality that does not change between versions (`add` actually adds elements, `delete` removes elements, etc.). Post-update and general CO-specs are used to check that functionality *does* change, but only in expected ways. For example, new-def-shadows in the

bug-fix update checks that, following the update, new key-value bindings properly overwrite old bindings (which was not true in the old version).

We wrote specifications to be as general as possible. For example, add-mem, on the second line of the table in Figure 4.3, checks that after an element is added, it is reported as present after an arbitrary sequence of function calls that does not include `delete()`. The code for our synthetic examples and their associated CO-specs is available on-line.

Thor. We ran Thor on a 2.8GHz Intel Core 2 Duo with 4GB of memory. The average slowdown was 3.9 times, and ranged from 1.5 times to 8.3 times. Much of the slowdown derived from per-update-point analysis of the state transformation function; tools that compute procedure summaries or otherwise support modular verification would likely do better. Thor could not verify all our examples, owing to complex state transformation code and CO-specs that specify very precise properties. For example, for the multiset-to-set example, Thor was able to prove that the state transformer preserves list membership (used to verify *mem-mem*), but not that it leaves at most one copy of any element in the list (needed for *add-add-del-set*).

The CO-specs we considered lie at the boundary of what is possible for current verification technology. To verify all our examples requires a robust treatment of pointer manipulation, integer arithmetic, and reasoning about collections. We are not aware of any tools that currently offer such a combination. However, we hope that the demonstrated utility of such specifications will help inspire further research in this area.

4.3 Related work

This work represents the first approach for automatically verifying the correctness of dynamic software updates. As mentioned in the introduction, prior automated analyses focus on safety properties like type safety [63], rather than correctness. As described in Section 3.2.4, our notion of client-oriented specifications captures and extends prior notions of update correctness.

Our verification methodology generalizes our prior work [30, 34] on systematically *testing* dynamic software updates that we used for the empirical study in Chapter 2. Given tests that pass for both the old and new versions, the tool tests every possible updating execution. This approach only supported backward-compatible properties and does not extend to general properties (e.g., with non-deterministically chosen operations or values).

The merging transformation proposed here was inspired by KISS [55], a tool that transforms multi-threaded programs into single-threaded programs that fix the timing of context switches. This allows them to be analyzed by non–thread-aware tools, just as our merging transformation makes dynamic patches palatable to analysis tools that are not DSU-aware.

An alternative technique for verifying dynamic updates, explored by Charlton et al. [15], uses a Hoare logic to prove that programs and updates satisfy specifications expressed as pre/post-conditions. We find CO-specs preferable to pre/post-conditions because they require less manual effort to verify, and because they naturally express rich properties that span multiple server commands.

Chapter 5

Kitsune: Efficient, General-purpose DSU for C

In Chapter 2, we performed an empirical evaluation of the timing restrictions used by transparent DSU systems. Over the course of that work, we identified a set of problems associated with supporting DSU transparently: automatic timing restrictions are insufficient to ensure correctness; programs must often be refactored in anticipation of certain changes; program evolution is restricted; and manual reasoning about update behavior may be difficult. An additional, crucial problem is that DSU systems for C make trade-offs that are incompatible with two main reasons developers choose to use C: performance and control over low-level data representations.

In this chapter, we present Kitsune, a DSU system for C programs that solves each of these problem by having the developer implement DSU as a program feature. Implementation as a program feature means that the developer modifies the program in simple ways to indicate what should happen during a dynamic update. Kitsune is the first DSU system to solve all of these problems (we compare against related systems in Section 5.5.)

Kitsune is able to solve the problems we have identified due to three key design and implementation choices. First, Kitsune gives the programmer explicit control over the updating process, which is reflected as three kinds of additions to the orig-

inal program: (1) a handful of calls to kitsune_update(...), placed at the start of one or more of the program's long-running loops, to specify *update points* at which dynamic updates may take effect; (2) code to initiate *data migration*, which is the transformation of old global state to be compatible with the new program version; and (3) code to perform *control migration*, which redirects execution to the corresponding update point in the new version. In our experience, these code additions are small (see below) and fairly easy to write because of Kitsune's simple semantics. This approach makes developer reasoning easier because there are fewer update points to reason about and update behavior is manifest in the code. (Section 5.1 explains Kitsune's use in detail.)

Second, Kitsune uses entirely standard compilation. After a translation pass to add some boilerplate calls to the Kitsune runtime, a Kitsune program is compiled and linked to form a shared object file (via a simple Makefile change). A Kitsune program is launched using a driver program that loads the first version's shared object file and transfers control to it. When a dynamic update becomes available (only at specific program points, as discussed shortly), the program longjmps back to the driver routine, which loads the new application version and calls the new version's main function. Thus, application code is updated all at once, and as a consequence, Kitsune places no restrictions on coding idioms or data representations; it allows the application's internal structure to be changed arbitrarily from one version to another; and it does not inhibit any compiler optimizations.

Finally, Kitsune includes a novel tool called xfgen that makes it easy to write code to migrate and transform old program state to be compatible with a new

program version. The input to xfgen is a series of *type and variable transformation specifications*, one per changed type or variable, that describe in intuitive notation how to translate data from the old to new format. The output of xfgen is C code that performs the transformations wherever they are needed: at a high level, the generated transformers operate analogously to a tracing garbage collector, traversing the heap starting at global variables and locals marked by the programmer. When the traversal reaches data requiring transformation, it allocates new memory cells and initializes them according to the actions in the transformers, taking care to maintain the shape of the data structures. The old version's copies of any migrated data structures are freed once the update is complete. Kitsune's approach is easy to use, relative to other DSU systems; it adds no overhead during the non-updating portion of execution, and it does not change data layout. (Section 5.2 describes xfgen.)

We have implemented Kitsune and used it to update three single-threaded programs—vsftpd, Redis, and Tor—and two multi-threaded programs—memcached and icecast. For each application, we considered from three months' to three years' worth of updates. We found that the number of code changes we needed to make for Kitsune was generally small, between 53 and 159 LoC total, across all versions of a program. The change count is basically stable, and not generally related to the application size, e.g., 134 LoC for 16 KLoC icecast vs. 159 LoC for 76 KLoC Tor. xfgen was also very effective, allowing us to write state transformers with similarly small specifications consisting of between 27 and 200 lines in total; the size here depends on the number of data structure changes across an application's streak.

We tested that all programs behaved correctly under our updates.

We measured Kitsune's performance overhead, and found it ranged from -2.2% to +1.8%, which is in the noise on modern environments [48]. We also found that the time required to perform an update was typically less than 40ms; icecast's longer, ∼1s update time is due to internal timing constraints and does not adversely affect the application. (See Section 5.3 for full details.)

Considered as a whole, we think that Kitsune's design meshes well with C without limiting the form of dynamic updates, and without imposing an undue burden on DSU programmers. In short, we find Kitsune to be the most flexible, efficient, and easy to use DSU system for C developed to date.

Author Contributions. We presented a paper about a prior system, Ekiden, that influenced Kitsune's design at HotSWUp '11 [35], and a paper about Kitsune is currently under review. The author of this dissertation is lead author on both papers, was the main designer and implementer of Kitsune (and Ekiden), modified several of the benchmark programs to work under Kitsune, and performed all benchmarking. Edward K. Smith contributed to the implementation of Kitsune and implemented Kitsune support for Tor. Michail Denchev adapted our Ekiden vsftpd patches to work with Kitune and Jonathan Turpie helped modify Memcached to support Kitsune. Karla Saur helped implement performance benchmarks.

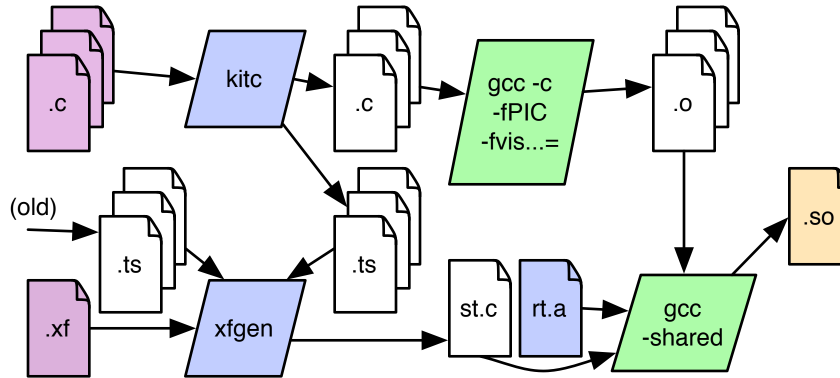


Figure 5.1: Kitsune build chain

5.1 Kitsune

The process of building a Kitsune application is illustrated in Figure 5.1. There are two inputs provided by the programmer: the main application's `.c` source files (upper left) and an xfgen `.xf` specification file for transforming the running state during an update (not needed for the initial version). The source files are processed by the Kitsune compiler kitc to add some boilerplate calls derived from programmer annotations. Rather than compile and link the resulting `.c` files to a standalone executable, these files are compiled to be position independent (using gcc's -fPIC flag) and linked, along with the Kitsune runtime system `rt.a`, into a shared object library app.so. (For the best performance we also use gcc's -fvisibility=hidden option to prevent application symbols from being exported, since exported symbols incur heavy overhead when called.) When building an updating version of the program, the `.xf` file is compiled by xfgen to C code and linked in as well. Processing the `.xf` requires `.ts` *type summary files* produced by kitc for the old and current versions (described in detail in Section 5.2.1).

The first version of a program is started by executing kitsune app.so *args...*,

where *args...* are the program's usual command-line arguments. The kitsune executable is Kitsune's application-independent *driver routine*, which dynamically loads the shared library and then performs some initialization. Among other things, the driver installs a signal handler for SIGUSR2 [1] which is later used to signal that an update is available. The driver also calls setjmp, and then transfers control to the (globally visible) kitsune_init function defined in rt.a; this function performs some setup and calls the application's (non-exported) main function. The kitsune driver is only 110 lines of C code and is the only part of a program that cannot be dynamically updated.

When SIGUSR2 is received, the handler sets a global flag. As discussed in more detail below, the running program is expected to call the function kitsune_update at points at which an update is permitted to take effect; such calls are dubbed *update points* [36]. The kitsune_update function will notice the flag has been set and call longjmp to return to the driver, which then dynamically loads the new program version's shared object library. Since the longjmp call will reset the stack, the kitsune_update function copies any local variables marked for migration to the heap before jumping back to the driver. Thus, just after an update, the old version's full state (e.g., its heap, open files and connections, process/parent id, etc.) is still available. At this point, kitsune_init is invoked to start the new version.

The new program version now must do two things: (1) migrate and transform the old version's data, and (2) direct control to a point in the new version that is

[1] The exact method for signaling that an update is available is left to the discretion of the programmer. Kitsune provides support for SIGUSR2 by default, which worked well in most cases. In our experiments, we only used a different mechanism for Tor, for which we opted to extend its existing control framework to initiate updates.

equivalent to the point at which the update took place in the old version. We call these activities *data migration* and *control migration*, respectively. The programmer directs the control flow and the timing of state transformation using a few judicious calls into the Kitsune runtime system, and defines state transformation code itself using xfgen.

We next illustrate basic data and control migration using an example and consider xfgen in Section 5.2.

5.1.1 Data and Control Migration

The C program in Figure 5.2 implements a simple key-value server. Clients connect to the server and send either `get` i to get the integer value associated with index i, or `set` i n to associate index i with value n. In the figure we have highlighted the extra code we needed to perform data and control migration. Let us ignore the highlighted code for the moment so that we can discuss the program's core operation. Execution of the program begins at main() on line 32. After defining some local variables, we call load_config () (code not shown) to initialize the three global configuration variables defined on line 2 and then allocate an empty mapping. Then we call setup_connection() (code also not shown) to begin listening on main_sock, and enter the main loop on lines 43–47. Here we simply wait for a connection and then call client_loop () to handle that connection.

The client_loop () function repeatedly reads a command from the socket; finds the handler (a function pointer) for that command in dispatch_tab (created on

```
1  /* config variables set by load_config() (code not shown) */
2  int config_foo, config_bar, config_size; /* automigrated */
3
4  typedef int data;
5  data *mapping; /* automigrated */
6
7  int op_count=0; /* automigrated */
8  struct dispatch_item
9    { char *key; dispatch_fn *fun; } dispatch_tab
10   __attribute__ ((kitsune_no_automigrate))
11   = { {"get", &handle_get }, {"set", &handle_set } };
12
13 void handle_set(int sock) {
14   key = recv_int(sock);
15   val = recv_int(sock);
16   mapping[key] = val;
17   send_response("%d> ok", op_count);
18 }
19 void handle_get(int sock) {
20   key = recv_int(sock);
21   send_response("%d> %d=%d", op_count, key, mapping[key]);
22 }
23 void client_loop(int sock) {
24   while (1) {
25     kitsune_update("client");
26     char *cmd = read_from_socket(sock);
27     if (!cmd) break;
28     dispatch_fn *cmd_handler = lookup(dispatch_tab, cmd);
29     op_count++;
30     cmd_handler(sock); }
31 }
32 int main() __attribute__ (( kitsune_note_locals )) {
33   int main_sock, client_sock;
34   kitsune_do_automigrate();
35   if (!kitsune_is_updating()) {
36     load_config();
37     mapping = malloc(config_size * sizeof(data)); }
38   if (!MIGRATE_LOCAL(main_sock))
39     main_sock = setup_connection();
40   if (kitsune_is_updating_from("client")) {
41     MIGRATE_LOCAL(client_sock);
42     client_loop(client_sock); }
43   while (1) {
44     kitsune_update("main");
45     client_sock = get_connection(main_sock);
46     client_loop(client_sock); }
47 }
```

Figure 5.2: Example; Kitsune additions highlighted

lines 9–11); increments a global counter op_count that tracks the number of requests; and then dispatches to handle_set or handle_get. If there was no command received from the socket, then we exit the loop on line 27.

While this code is very simple, many server programs share this same general structure—a main loop that listens for connections; a client loop that dispatches different commands; and handler functions that implement those commands. Now consider the highlighted code, which implements Kitsune control and data migration.[2]

Migrating control. A dynamic update is initiated when the program calls kitsune_update(*name*), where *name* identifies the update point, which can be queried when the new program version is launched. In Figure 5.2 we have added update points on lines 25 and 44, i.e., we have one update point to start each long-running loop. These are good choices for update points because the program is *quiescent*, i.e., in between events, when there is less in-flight state [51, 34].

The kitsune driver will load the new version and call its main function, so the programmer must write code to direct execution back to the equivalent spot in the new program. This code will likely include calls to kitsune_is_updating (), which returns true if the program is being run as a dynamic update (or its variant kitsune_is_updating_from (*name*) for updates triggered at that named update point), to distinguish update resumption from normal startup.

In Figure 5.2, the conditional on line 35 prevents the configuration from being reloaded and mapping from being reallocated when run as an update, since in this case we will migrate that state from the old program version instead (discussed below). If the update was initiated from the client loop, then on line 40 we migrate

[2]We should emphasize that because this example is tiny, the amount of highlighted code is disproportionately large (see Section 5.3).

client_sock from the previous version and then go straight to that loop. Notice that when we return from this call, we will enter the beginning of the main loop, just as if we had returned from the call on line 46. Also notice we do not specifically test for an update from the "main" update point, as in that case the control flow of the program naturally falls through to that update point.

Migrating state. When a Kitsune program starts as an update, critical state from the previous version of the program remains available in memory so it can migrate to the new program version. The programmer is responsible for identifying what state must be migrated, and specifying how that migration is to take place.

The first step is to identify the global and local variables that should be migrated. All global variables are migrated by default (that is, "automigrated"), and the programmer can identify any exceptions. For our example, migration occurs for the configuration variables on line 2 and for mapping on line 5. We use the kitsune_no_automigrate attribute on line 10 to prevent dispatch_tab from being automigrated, so that it is initialized normally—with pointers to new version functions—rather than overwritten with old version data. Local variables are *not* automigrated—the programmer must annotate a function with kitsune_note_locals (c.f. main()) to support migration of its local variables.

To facilitate data migration, kitc generates a per-file do_registration () function that registers the names and addresses of all global variables, including **static**s, and records for each one whether it is automigratable. The do_registration () function is marked as a constructor so it is called automatically by dlopen. Similarly, kitc introduces code in each of the functions annotated with kitsune_note_locals to reg-

ister (on function entry) and deregister (on function exit) the names and addresses of local variables (in thread-local storage).

The second step is to indicate *when* data should be migrated after the new version starts. Calling kitsune_do_automigrate() (line 34) starts migration of global state, calling a *state transformation function* for each registered variable that is automigratable. These functions implement data transformation (versus just copying), and are produced by xfgen from programmer specifications. Each function follows a particular naming convention, and the runtime finds them in the new program version using dlsym(). If no state transformation function is found, the data is copied. xfgen-generated transformers traverse the heap starting from global and (programmer-designated) local variables.

Within a function annotated with kitsune_note_locals , the user calls MIGRATE_LOCAL(var) to migrate (via the appropriate state transformer) the old version of var to the new version, e.g., as used on line 41 to migrate client_sock . MIGRATE_LOCAL() returns 1 if the program was started as a dynamic update; on line 38 we test this result to decide whether to initialize main_sock.

Our overall design for state migration reflects our experience that we typically need to migrate all, or nearly all, global variables, whereas we need only migrate a few local variables—only locals up to the relevant update point are needed, and of these, most contain transient state. We also assume that all state that might be transformed is reachable from the application's local and global variables. In our experiments, this assumption was only violated in memcached, in which the only pointers to some application data were stored in a library. This problem is addressed

by caching such pointers in the main application; see Section 5.3.1.

Cleaning up after an update. After updating, Kitsune reclaims space taken up by the old program version. Since control and data migration are under programmer control in Kitsune, we need to specify the point at which the update is "complete." That point is when the new program version reaches the same update point at which the update occurred (c.f. the branch on line 42 of Figure 5.2, which then reaches the update point on line 25). Kitsune then unloads the code and stack data from the previous program version; to be safe, the programmer must ensure there are no stale pointers to these locations. For example, programmers must ensure any strings in the data segment that need to migrate are copied to the heap (which can be done in state transformers, or with `strdup` in the program text). Kitsune also frees any heap memory that xfgen-generated transformers have marked as freeable. Finally, control returns to the new version.

5.1.2 Multi-threading

Updating a multi-threaded program is more challenging since the programmer must migrate control and data for every thread. We could require the programmer to write this code manually, but we have observed that when the set of threads before and after the update is the same, a little additional support can make it easier to migrate those threads automatically.

To make a pthreads program Kitsune-enabled, the programmer must modify all thread creation sites to use a wrapper for `pthread_create` called `kitsune_pthread_-`

create. A thread created with kitsune_pthread_create(tid, f, arg) has its thread id tid, the name of thread function f, and the value of f's argument arg are (atomically) added to a global list kitsune_threads of live threads. When a thread exits normally, it removes its entry from kitsune_threads.

Once an update becomes available, each non-main thread stops itself when it reaches an update point, recording the name of the update point in its kitsune_threads entry. When all threads have reached their update points, the main thread starts updating as described in Section 5.1.1, and continues until it finally reaches its own update point in the new version. Then the run-time system iterates through kitsune_threads and relaunches each thread, calling the new version of the recorded thread function with its recorded argument. If needed, the developer can provide a special transformation function to modify the set of threads or transform a thread's entry function and argument. Each of those threads then executes, performing whatever initialization and data migration is needed. Each thread pauses when it reaches the update point where it was stopped. Once all threads have paused, the Kitsune runtime cleans up the old program version, releasing its code and data as usual, and resumes the main thread and all paused threads.

For Kitsune's approach to work, the program must follow several conventions. First, each long-running thread must periodically reach an update point. Typically this means a thread needs an update point in any long running loop and should avoid blocking I/O and similar operations. Second, threads should not hold resources, such as locks, at update points, since the thread could be killed and restarted at that point. This requirement is in keeping with the general criterion for choosing update

points, which stipulates that little or no state should be in-flight. Third, the program should be insensitive to the order in which the threads are restarted in the new version. We expect this holds because the main thread will likely migrate any shared state, which would otherwise be the main source of contention between threads. Finally, recreating threads changes their thread IDs, and so the program should not store those IDs in memory. (We could extend Kitsune to relax this requirement.) The programs we considered in our experiments satisfy these requirements, and we conjecture adapting non-compliant programs to them should be easy.

5.2 xfgen

As mentioned briefly in Section 5.1.1, Kitsune's runtime invokes state transformer functions for each automigrating variable, following a naming convention to locate the appropriate transformer function. In the general case, the developer can construct such functions manually. Kitsune also includes xfgen, a tool that produces state transformation functions from simple specifications. The design of xfgen aims to make common kinds of state transformers easy to write while maintaining the flexibility to implement arbitrary transformations.

Figures 5.3(a) and (b) summarize xfgen's transformer specification language. Each transformer has one of the forms shown in part (a). The INIT transformers describe how instances of new variables or types should be initialized, and the $\rightarrow$ transformers describe how to transform variables or types that have changed and/or been renamed. Here $\{new,old\}_var$ is either a local or global variable name

INIT *new_var*: {*action*}
INIT *new_type*: {*action*}
old_var → *new_var*: {*action*}
old_type → *new_type*: {*action*}
old_var → *new_var*
old_type → *new_type*

(a) transformers

$in, $out – old/new type or var
$old/newsym(x) – x in old/new prog.
$old/newtype(t) – t in old/new prog.
$base – containing **struct**
$xform(old, new) – xformer ref.

(b) special variables

KS_PTRARRAY(S) – size of ptd-to array
KS_ARRAY(S) – size of array
KS_OPAQUE – non-traversed pointer
KS_FORALL(@t) – polymorphism intro.
KS_VAR(@t) – refer to type var
KS_INST(typ) – instantiate poly. type

(c) type annotations

Figure 5.3: xfgen specification language and type annotations

and {*new,old*}*_type* is either a regular C type name or a **struct** type field (we will see an example below). The transformer *action* consists of arbitrary C code that may reference the special xfgen variables shown in Figure 5.3(b), which refer to entities from the old or new program version. A → transformation without an action identifies a variable/type renaming.

We next illustrate the specification language through a series of examples, and then discuss how transformer functions are generated from specifications.

```
struct list {int key; data val; struct list *next;} *mapping;
```

Then we can specify the following transformer:

```
1  mapping → mapping: {
2   int key;
3   $out = NULL;
4   for(key = 0; key < $oldsym(config_size); key++) {
5     if ($in[key] != 0) {
6       $newtype(struct list) *cur =
7         malloc(sizeof($newtype(struct list)));
8       cur→key = key;
9       cur→val = $in[key];
10      cur→next = $out;
11      $out = cur;
12 } } }
```

Here mapping → mapping indicates this is a transformer for the old version of mapping (the occurrence to the left of the arrow, referred to as $in within the transformer) to the new version of mapping (referred to as $out). The body of the transformer loops over the old mapping array (whose length is stored in old version's config_size), allocating and initializing linked list cells appropriately. In the call to malloc, we use $newtype(**struct** list) to refer to the list type in the new program version.

Example 3. Finally, suppose the programmer wants to change type data from **int** to **long**, and at the same time extend mapping with field **int** cid to note which client established a particular mapping:

```
typedef long data;
struct list {
  int key; data val; int cid; struct list *next;} *mapping;
```

The programmer can specify that val should be simply copied over and cid should be initialized to 1:

```
typedef data → typedef data: { $out = (long) $in; }
INIT struct list.cid { $out = 1 }
```

Because the type of mapping changed, xfgen will use these specifications to generate

a function that traverses the mapping data structure, initializing the new version of mapping along the way. This is possible because there is a structural relationship between elements in the old list and elements in the new list, and because by default xfgen-created state transformers stop traversal at NULL, the list terminator. (We could not use this approach for the previous array-to-list change because the data elements were not related structurally.)

Other special variables. In the examples so far, we have seen uses of all but the last two special variables in Figure 5.3(b). The variable $base refers to the struct whose field is being updated. For example, in

```
INIT struct s.x: { $out = $base.y }
```

new field x of **struct** s is initialized to field y in the same **struct**. Variable $xform refers to a particular type transformer. For example, suppose we merged Examples 2 and 3 into a single update that migrated mapping to a list and changed data's type to **long**. Then we could use the transformer from Example 2, changing line 9 to

```
XF_INVOKE($xform(data, data), &$in[key], &cur→val);
```

$xform is passed an old and a new type name (here, both are data) and it looks up (or forces the creation of) the transformer between those types. This transformer is returned as a closure that takes pointers to the old and new object versions and can be called using XF_INVOKE.

5.2.1 Transformer generation

xfgen generates code to perform migration and transformation from the .xf file and the type summary files (.ts files in Figure 5.1) of the old and new versions. A type summary file contains all of the type definitions (e.g., **struct**, **typedef**) and global and local variable declarations from its corresponding .c source file, noting which are eligible for migration (according to the rules given in Section 5.1.1). xfgen uses type information to generate code that can inspect and manipulate program data, and it uses migration information to make sure .xf files are complete: an .xf file is rejected if it fails to define a transformer for a migratable variable or type that has changed between versions.

Type annotations. xfgen sometimes needs type information beyond what is available in C. For example, suppose we write a transformer for a variable foo *x. Then xfgen needs to know how to traverse the memory pointed to by x, e.g., whether x is a pointer to a single foo instance or an array. In Kitsune, this extra information is provided by the programmer as annotations, shown in Figure 5.3(c). kitc recognizes these annotations and adds the information supplied by them to the .ts files.

The annotations, inspired by Deputy [19], are fairly straightforward. KS_PTRARRAY(S) provides a size S (an integer or variable) for a pointed-to array. KS_ARRAY(S) provides a size S for array fields at the end of a **struct** (which can be left unsized in C). KS_OPAQUE annotates pointers that should be copied as values, rather than recursed inside during traversals. By default, xfgen assumes that t* values for all types t are annotated with KS_PTRARRAY(1); explicit annotations

override this default.

Finally, xfgen includes annotations to handle some idiomatic uses of **void**∗ to encode parametric polymorphism (a.k.a. generics). For example, the following definition introduces a **struct** list type that is parameterized by type variable @t, which is the type of its contents:

```
struct list {
  void KS_VAR(@t) *val;
  struct list KS_INST(@t) *next;
} KS_FORALL(@t);
```

KS_FORALL(@t) introduces polymorphism, KS_VAR(@t) refers to type variable @t, and KS_INST(@t) instantiates a polymorphic type with type @t. With the above declaration, we could write **struct** list KS_INST(**int**) ∗x to declare that x is a list of **int**s.

Variable transformers. For each migrated variable listed in the new version's .ts file, if that variable is named explicitly in an old_var → new_var transformer, then xfgen generates C code from the given action, substituting references appropriately. For example, $in and $out in the action are replaced by values returned from kitsune_lookup_old and _new, respectively, which return a pointer to a symbol in the old or new program version, respectively, or null if no such symbol exists. For each remaining migrated variable x, xfgen will consult the y → x renaming rule if one exists to determine the source symbol y; otherwise it assumes x's name is unchanged. xfgen then generates C code that migrates the variable by calling the type transformation from the type of the old-version symbol to x's type (as described next).

As an example, xfgen will produce the following C code from Example 1, above:

```
void _kitsune_transform_get_count () {
 int *o_op_count = (int*) kitsune_lookup_old ("op_count");
 int *n_get_count = (int*)kitsune_lookup_new("get_count");
 *n_get_count = *o_op_count;
}
void _kitsune_transform_set_count () { /* as above */ }
```

Type transformers. Generating C code for manually specified type transformers is analogous to what is done for manually specified variable transformers. We also generate transformation functions for all (unchanged) types `t` of migratable data, i.e., for the following cases: (1) when a migrated variable has type `t` but no manual transformer (as with `struct list` in Example 3, above); (2) when traversed data references values of type `t` (e.g., if an unchanged global variable had type `struct s` where `s`'s definition includes a value of type `t`); and (3) when a transformer for `t` is referenced directly in a manual transformer (e.g., as with `$xform(data, data)` mentioned above). For these cases, the functions merely recursively invoke transformation functions on the immediate children of the type in question (skipping `NULL` values); no values of that type are copied, but pointers may be redirected to values that are.

Whenever a type transformer migrates a pointer, it performs several steps. First, if the pointer is `NULL`, it does nothing. Otherwise, it checks a global map to see if the pointer has been migrated before; if so it returns the old target. Doing this maintains the shape of the heap and avoids infinite loops during traversal of cycles. If neither of these two conditions apply, it calls the appropriate transformer for the

Program	**#**	**Vers**	**LoC**	**Upd**	**Ctrl**	**Data**	**KS_***	**Oth**	**Σ**	**v→v**	**t→t**	**Σ**	**xf LoC**
vsftpd	14	(1.1.0–2.0.6)	12,202	6	26	17+8	6+14	28+8	83+30	9	21	30	101
Redis	5	(2.0.0–2.0.4)	13,387	1	2	3	43	4	53	0	4	4	37
Tor	13	(0.2.1.18–0.2.1.30)	76,090	1	39	37+6	19	57	153+6	16	15	31	189
*Memcached**	3	(1.2.2–1.2.4)	4,181	4	9	13	20	66	112	12	10	22	27
*Icecast**	5	(2.2.0–2.3.1)	15,759	11+1	22+3	14+9	32+3	39	118+16	25	50	75	200

*Multi-threaded

Table 5.1: Kitsune benchmark programs, and modifications to support updating

pointer's target. If the pointer is to a global or local variable, it stores the result in the corresponding new-version variable's space. If the pointer target's type has truly changed (and so must have a manual transformer with an action), it mallocs space and stores the result there, remembering the result's address in the global map. It also stores the old-version pointer on a list of addresses to be reclaimed once the update is complete.

Following the above procedure, xfgen will generate transformer/traversal code that will deeply explore the heap and ensure that all pointers to the data and stack segment are transformed to work once the old program is unloaded. If the programmer knows that a particular data structure contains only pointers into the heap (and not to global or local variables) and no pointed-to objects require transformation, she can create transformers that truncate the traversal to reduce update time (we used this trick for redis-mod in Figure 5.4).

The transformers generated by xfgen assume there are no pointers into the middle of transformed objects. To help check this assumption, we provide an execution mode in which the created transformers use an interval tree to record the start and end of each object they transform. A transformer reports an error if it is ever asked to migrate an object that overlaps with, but does not exactly match the bounds of, a previously migrated object.

5.3 Experiments

To evaluate Kitsune, we used it to develop updates for five widely deployed server programs. We found that few code changes are required to support updating: between 53 and 159 LOC, the lion's share of which were made to the initial version. Likewise, xfgen specifications were generally small, at around 3-4 lines per changed variable/type. These numbers are comparable to, or better than, prior work. Kitsune performance is uniformly better, with essentially no steady-state overhead. We found that the time required to apply an update ranges from 2ms up to 1s, depending on the program; in all cases, however, the times seem acceptable for typical use.

Benchmarks. We chose a suite of benchmark programs that maintain in-process state that would be beneficial to preserve during an update. *Vsftpd* is a popular open-source FTP server [3]. *Redis* is a key-value database used by several high-traffic services, including guardian.co.uk and digg.com. *Tor* is a popular onion-router that provides anonymous Internet access [4]. *Memcached* is a widely used, high-performance data caching system employed by sites such as Twitter and Wikipedia. *Icecast* is a popular music streaming server. All of these programs maintain persistent network connections that an offline update would interrupt. Redis and memcached also maintain potentially large volumes of in-memory data that would either be would be lost (memcached) or expensive to restore (redis) following an update. Vsftpd also serves as a useful benchmark because several other DSU systems have

[3]The updates to vsftpd were adapted from Ekiden [35] updates by one of the author's collaborators.

[4]The updates to Tor were written by one of the author's collaborators.

used it for evaluation [51, 45, 17, 35].

The left portion Table 5.1 lists for each program the length of the version streak we looked at (for n versions, there are $n-1$ updates), which versions we considered, and the number of source lines of the last version as computed by sloccount. We consider at least three months of releases per program; for Tor we cover two years and for vsftpd we cover three.

5.3.1 Programmer effort

Here we describe the manual effort that was required to prepare these programs for updating with Kitsune, and to craft updates corresponding to releases of these programs. In all cases, the versions we updated behaved correctly before, during, and after updates were applied.

The center portion of Table 5.1 summarizes the Kitsune-related changes we made to these programs, tabulating the number of update points added; the number of lines of code needed for control migration, e.g., kitsune_is_updating (), and data migration, e.g., kitsune_do_automigrate (); the number of type annotations for xfgen, e.g., KS_PTRARRAY; the number of lines changed for other reasons; and their sum. Each column shows the number of changes in the first version, followed by $+n$ where n is the sum of changes in all subsequent versions; if this is omitted, no further changes were needed.

One striking trend in the table is that most required changes occurred in the first version. Control flow migration and update points were particularly stable, es-

sentially because the top-level control flow structure of the programs rarely changed. We also found control flow migration code relatively easy to write. Data migration code and annotations were occasionally added along with new data structures. Another interesting trend is that the magnitude of the changes required is not directly proportional to either the code size or number of versions considered, e.g., 134 LoC for 16 KLoC icecast vs. 159 LoC for 76 KLoC Tor. On reflection, this trend makes sense. Changes to support control migration depend on the number and location of update points, and data annotations depend on the type and number of data structures; none of these characteristics scales directly with code size. Together, these numbers show that with Kitsune, DSU is a stable program feature that is straightforward to add and easy to maintain.

The rightmost columns of the table describe the xfgen specifications we wrote for each program's updates. In particular, we list the number of variable transformers (v→v) and type transformers (t→t), across all versions, and their sum. We also list the total number of lines of transformer code we wrote, across all versions. We can see that, on average, 3–4 lines of xfgen code were needed for each transformer. Most state either required no (or very simple) transformation. One of the largest transformations was a 19-line redis rule to choose the right transformation for a **void**∗ field based on an integer key indicating the field's type. When transformation annotations were necessary, they were either obvious (specifying generic types or bounded arrays) or prompted by xfgen.

Now we consider the particulars of each program, and when possible measure the magnitude of these changes against those required by prior systems. In general

we find the number of required changes to be comparable.

Vsftpd. Many of the changes we made to vsftpd were typical across our benchmarks: we added type information for generics and inserted control flow changes to avoid overwriting OS state when updated. We added one update point for each of the five long-running loops in the program, e.g., the connection listener, login processor, command processor, etc.

The most interesting change we made to vsftpd was to handle I/O. Vsftpd replaces calls to `recv` with calls to a wrapper that restarts the actual read if it is interrupted, e.g., by the receipt of a signal. We inserted one update point in the wrapper so that interruption can initiate an update. To simplify the control-flow changes needed, the update point need not be given its own name, but can reuse the name of the update point in the loop that initiated the wrapped call; this is because this loop will reinitiate the call when the update completes.

Other DSU systems. Neamtiu et al. [51] applied Ginseng, another DSU system, to vsftpd. They updated a subset of the version streak we did (finishing at version 2.0.3). Even though their changes support just one update point (versus our six, which permit updating in many more situations), the effort was comparable: They report 50 LOC changed and 162 lines for state transformation, compared to 127 LOC changed and 101 lines of state transformation for Kitsune.

Makris and Bazzi [45] also updated vsftpd using UpStare for a shorter streak. They say that "some manual initialization of new variables and struct fields" was required, along with "11 user-defined continuation mappings," but provide no detail as to their overall size.

Redis. Redis required few modifications to support updating. We placed a single update point in its main event loop and added one check to avoid some reinitialization. The vast majority of Redis's state is stored in a single global variable, `server`, so few variables needed migration. Redis makes extensive use of linked lists and hash tables, and we used xfgen's generics annotations to model their types precisely. The version streak we considered included only code modifications, but we still needed xfgen to migrate data structures that contain global variable addresses (which change with updates). Finally, redis uses custom memory management functions that xfgen does not support, so we modified these functions to directly call the standard `malloc` and `free`. We leave support of custom allocators to future work.

We are unaware of prior work applying DSU to redis.

Tor. Tor is the largest of our benchmark programs, at ~76 KLoC. Adding DSU support required one update point in Tor's main loop, and a small number of control flow modifications to prevent reinitialization of updated state. The small size of the latter is particularly surprising given the very large amount of state in Tor. We did need to add code to migrate one object (representing the network consensus) manually, because it cannot be refreshed correctly until the rest of the state has been migrated. The bulk of Tor's changes served to expose DSU functionality in Tor's "control port" interface, e.g., so that updates could be triggered using standard tools.

One challenge was that Tor uses libevent for event processing, and that library stores function pointers inside event **struct**s. Tor maintains a list of those **struct**s, and we wrote state transformers that update those pointers when a new version

is loaded. These transformers, along with similar transformers updating function pointers used by either Tor or the OpenSSL library, comprised the majority of xfgen rules.

We are unaware of prior work applying DSU to Tor.

Memcached. Memcached is a multi-threaded server that uses libevent, like Tor. As with Tor, we needed to make some minor changes to memcached so the updating signal properly reaches the Kitsune library to initiate the update process. We also needed to reinstall new function pointers in libevent after an update. More interestingly, we needed to add code to memcached to maintain a list of active connections, so that xfgen properly generates code to transform these connections at update-time; in the ordinary implementation of memcached, connection objects are not otherwise reachable from global and local variables once they are passed to libevent.

Other DSU systems. Neamtiu and Hicks [49] updated memcached using Ginseng. They needed 26 lines of program changes and 12 lines for state transformation. Kitsune required more changes in part because we did not change libevent itself, which in Neamtiu and Hicks' setup was merged into the main program (and thus was updatable). Their changes also created a problem with reaching update points suitably often due to intervening blocking calls; placing the update point outside libevent avoided this issue.

Icecast. Icecast is another multi-threaded program, with separate threads for connection acceptance, connection handling, file serving, receiving a stream from another server, sending statistics, and more. Thus, it needed more than the usual number of update points. We added annotations to migrate local variables or skip

Program	Orig (siqr)	Kitsune	Ginseng
64-bit, 4×2.4Ghz E7450 (6 core), 24GB mem, RHEL 5.7			
vsftpd 2.0.6	6.55s (0.04s)	+0.75%	–
memcached 1.2.4	59.30s (3.25s)	+0.51%	–
redis 2.0.4	46.83s (0.40s)	-0.31%	–
icecast 2.3.1	10.11s (2.27s)	-2.18%	–
32-bit, 1×3.6Ghz Pentium D (2 core), 2GB mem, Ubuntu 10.10			
vsftpd 2.0.3	5.71s (0.01s)	+1.79%	+8.05%
memcached 1.2.4	101.40s (0.35s)	-0.49%	+18.44%
redis 2.0.4	43.88s (0.16s)	-1.21%	–
icecast 2.3.1	35.71s (0.68s)	+1.18%	-0.28%

Table 5.2: Steady-state performance overhead

initialization within the entry functions of each thread, as needed. The most complex icecast patch added a new thread to handle authentication (requiring an added update point) and reduced the number of connection threads. Kitsune provides programmatic access to the set of threads during transformation to support these changes.

Other DSU systems. Neamtiu and Hicks [49] also considered updates to the same streak of icecast versions (plus one earlier version). They changed 154 LOC and wrote 80 lines of state transformation code. For Kitsune we changed 134 lines of the main program, and wrote 200 lines of xfgen specifications.

5.3.2 Performance

Steady-state performance overhead. We measured the steady-state overhead of Kitsune on all programs except Tor, discussed separately below. For comparison, we also measured the overhead of Ginseng for the three programs (vsftpd, memcached, and icecast) for which Ginseng updates were available.

We used the following workloads: For memcached, we ran memslap (2.5M operations using memslap's default workload). For redis, we used redis-benchmark (1M GET and 1M SET operations), and for a fair comparison, we modified the non-updating version of redis to use the standard memory allocation functions, as we had done to support xfgen. For vsftpd, we measured the time to perform the following interaction 2K times: connect to the server, change directories, and retrieve a directory listing. For icecast, we used a benchmark originally developed for Ginseng [49] that measures the time taken for 16 simultaneous clients to download 7 music files, each roughly 2MB in size. For all programs, we ran the client and server on the same machine.

Table 5.2 reports the results. We ran each benchmark 21 times and report the median time for the unmodified programs along with the semi-interquartile range (SIQR), and the slowdowns for Kitsune and Ginseng (the median Kitsune or Ginseng time compared to the median original time). The top of the table gives results on a 24 core, 64-bit machine, and the bottom gives results on a 2 core, 32-bit machine; Ginseng only works in 32-bit mode.

From this data, we can see that Kitsune has essentially no steady-state overhead: the performance differences range from -2.18% to 1.79%, which is well within the noise on modern environments [48]. In contrast, for two of the three programs (vsftpd and memcached), the Ginseng overhead is more significant. While we have not ourselves benchmarked UpStare, the authors of that system report overhead of 16.0% for vsftpd (version 2.0.5) [45].

Tor. While we did not measure the overhead of Kitsune on Tor directly, we

Program	Med. (siqr)	Min	Max
64-bit, 4×2.4Ghz E7450 (6 core), 24GB mem, RHEL 5.7			
vsftpd →2.0.6	2.99ms (0.04ms)	2.62	3.09
memcached →1.2.4	2.50ms (0.05ms)	2.27	2.68
redis →2.0.4	39.70ms (0.98ms)	36.14	82.66
icecast →2.3.1	990.89ms (0.95ms)	451.73	992.71
icecast-nsp →2.3.1	187.89ms (1.77ms)	87.14	191.32
tor →0.2.1.30	11.81ms (0.12ms)	11.65	13.83
32-bit, 1×3.6Ghz Pentium D (2 core), 2GB mem, Ubuntu 10.10			
vsftpd →2.0.3	2.62ms (0.03ms)	2.52	2.71
memcached →1.2.4	2.44ms (0.08ms)	2.27	3.12
redis →2.0.4	38.83ms (0.64ms)	37.69	41.80
icecast →2.3.1	885.39ms (7.47ms)	859.00	908.87
tor →0.2.1.30	10.43ms (0.46ms)	10.08	12.98

Table 5.3: Kitsune update times

did test it by running a Tor relay in the wild. We dynamically updated this relay from version 0.2.1.18 to version 0.2.1.28 as it was carrying traffic for Tor clients. We initiated several dynamic updates during periods of load, when as many as four thousand connections carrying up to 11Mb/s of traffic (up and down) were live. No client connections were disrupted (which would have been indicated by broken or renegotiated TLS sessions). Over the course of this experiment, our relay carried 7TB of traffic.

Time required for an update. We also measured the time it takes to deploy an update, i.e., the elapsed time from when an update is signaled as available to when the update has completed. Table 5.3 summarizes the results for the last update in each streak, giving the median, SIQR, minimum, and maximum update times. For each program, we picked a suitable workload during which we did the update. For vsftpd, we updated after an FTP client had connected to and interacted

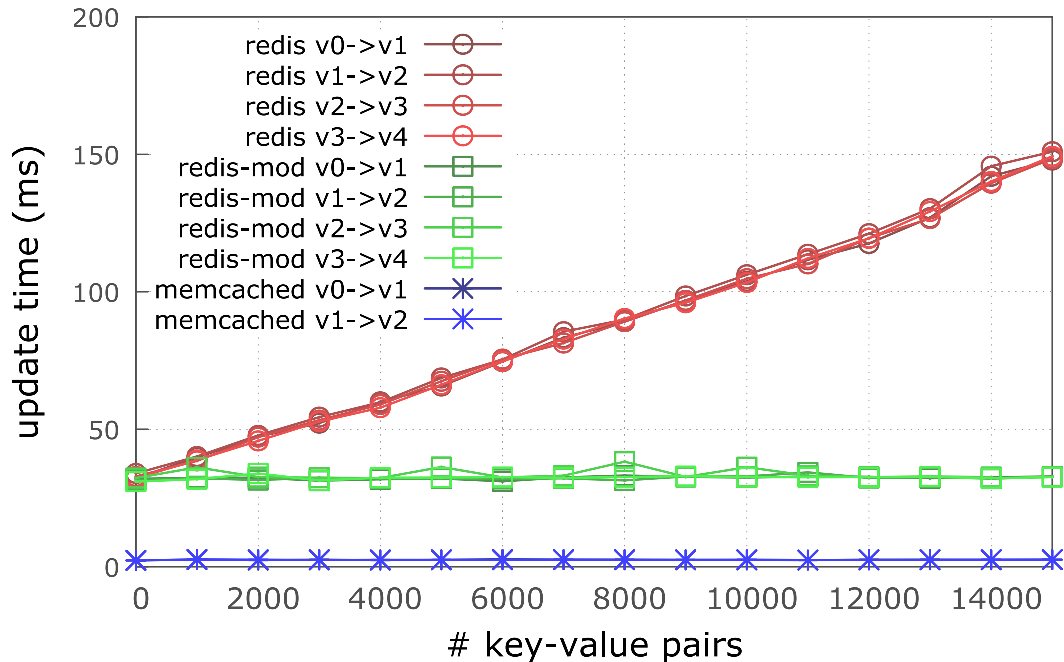


Figure 5.4: State size vs. update time

with the server; for redis and Memcached, we inserted 1K and 15K key-value pairs, respectively, prior to update; and for icecast, we established one connection to a music source and 10 clients receiving that stream prior to updating. For Tor, we fully bootstrapped as a client, establishing multiple circuits through the network and communicating with directory servers, and then applied the update.

For all programs except icecast, the update times are quite small. For icecast, most of the nearly 1 second delay occurs while the Kitsune runtime waits for each thread to reach an update point. This time was lengthened by one-second sleeps sprinkled throughout several of these threads. The line in the table labeled *icecast-nsp* measures the update time when these sleeps are removed, and shows the resulting time is much shorter. Because the sleeps are there, we conjecture icecast can tolerate the pause for updates; we did not observe a noticeable stop in the

streamed music during the update. In recent work [33], we have developed techniques to support faster update times, showing significant improvements for icecast in particular. We plan to port these ideas to Kitsune in the near future.

We also recorded the time taken for the unmodified versions of the benchmark programs to reach their main loops after entering `main`. Restarting the program in this way fails to maintain important in-process state, so we provide this comparison only as a point of reference for the Kitsune update times that we measured. For our 64-bit benchmarking environment (the 32-bit environment showed the same trend), we observed median startup times that were typically a few milliseconds (with the exception of Tor's), ranging from 0.56ms for `vsftpd` to 2.35ms for Icecast. While these startup times were often close to the Kitsune update times, Icecast took much longer (990.89ms) to update. Conversely, Tor took 566.05ms to start up, which was significantly longer than the 11.81ms time taken to update. Note that the startup times described here do not include code loading (which is reflected in the Kitsune update times), so they slightly underestimate the true time.

Recall from Section 5.2.1 that xfgen-generated transformers may traverse significant portions of the heap, and thus for some updates the update time may vary with the size of the program state. Among our programs, the most likely to exhibit this issue are redis and memcached, as they may accumulate significant state. Figure 5.4 graphs the update time for these two programs versus the number of key-value pairs stored. For redis, the update time grows linearly because we traverse each of the data items on the heap, since some contain pointers to global variables that must be redirected to the new version's data segment. On the other hand,

memcached's update times remain relatively constant because it stores its data in arrays that we treat opaquely, removing the need to traverse each instance.

Examining redis more closely, we observed that the pointers that force us to traverse the heap in fact point to a small, finite set of static locations. Thus, we created a modified (42 LOC changed) version of redis, labeled redis-mod, that stores integer indices into a table in place of those pointers. This obviates the need for a full heap traversal for all the updates in our streak, allowing updates times to remain constant for the tested heap sizes. Programs that use Kitsune may benefit from a similar transformation if they maintain a large amount of state containing static pointers.

5.4 Experience using Kitsune

In this section, we describe our experience performing the manual work required to modify a program to be updateable with Kitsune and the process we used to debug errors.

Debugging. To develop and test the modifications that we make to a program to support updating, we found it useful to start out by developing a *self update* that updates to the same version. This technique is useful because the modifications made to the program to support updating with Kitsune are typically independent of the patch being applied. Once control flow, data migration, and update points are in place, those changes will also apply when the program is updated from other versions. Likewise, we found that, because changes for Kitsune were quite stable,

the modifications to support a self update only needed to be tweaked slightly across later versions of the program in response to its evolution.

Control flow modifications. We found that modifying control flow to reach the correct loop after startup was straightforward for every program. Making these changes for the main thread required the most work, but the process was not prone to errors. One exception was that testing (e.g., self-update testing) sometimes revealed the need to run particular parts of the startup code both during normal startup and during an update. For example, we needed to reinstall signal-handling functions following each update to use the new-version function address. Alternatively, we could have handled such problems using state transformation, e.g., by either automatically updating signal-handler callbacks or having the developer do so in a state transformer function. We often chose to implement them as control-flow changes because the same code change would apply for every patch.

We also needed to make control flow modifications for the startup of each program thread. However, we found that these modifications were far simpler than the ones for the main thread since thread-entry functions typically perform less work before entering long-running loops.

Local variable annotations. While Kitsune will automatically migrate global variables, it requires the developer to add code to migrate locals. Some local variables contain important state (e.g., listen sockets) that must be migrated while others contain the results of intermediate computations that are no longer needed. We found that, after taking the time to understand the work performed by a program

during startup, it was clear which local state required migration.

Kitsune also requires the developer to use the kitsune_note_locals annotation to identify the functions containing local state that the next version can access. In our experience, these annotations were always added to the functions containing calls to MIGRATE_LOCAL. While the function annotation ensures that every local variable from an annotated function is available to new-version transformers, we have not needed this flexibility thus far—our practice of beginning each updating streak with a self update has enabled us to find all the needed locals from the start. If this observation continues to hold, we may consider only supporting migration of locals used in calls to MIGRATE_LOCAL to eliminate the need for the kitsune_note_locals annotation.

Placement of update points. We found that placement of update points was straightforward and almost entirely dictated by the structure of the program for most of our benchmark programs. As we noted earlier, vsftpd required some special handling because it wraps calls to I/O functions inside new functions. This wrapping required us to add an additional update point to one of these functions to support updating while the server is idle.

Memcached differs from the other benchmark programs in that its main loop is implemented inside the libevent library; this presented a challenge because we wanted to permit updates within that loop without modifying libevent. To achieve this, we changed memcached to handle SIGUSR2 (the update signal) using libevent. The main thread's libevent callback for this signal notifies each child thread to

return to application code from libevent and then does so itself. Interestingly, this approach required us to "sandwich" the invocation of libevent's loop between two update points:

```
kitsune_update("upd");   /* complete any active update */
event_base_loop(libevent_base, 0); /* pass control to libevent */
kitsune_update("upd");   /* a new update upon return */
```

When an update is signaled, the event_base_loop call returns and initiates the update. Then, when the program restarts, execution will reach the update point just prior to the event_base_loop call in the new code; since this update point is given the same label as the one initiating the call, the update completes.

xfgen experience. Overall, we found that xfgen provided an elegant way to express transformation and avoid writing boilerplate traversals. However, we did encounter some challenges applying it, which we discuss next to motivate future improvements.

As we have described, the developer may annotate data structures with type information to help xfgen generate traversal code for transformation. This process requires the developer to ensure that all pointers contained in migrated data types are correctly annotated. In some cases, for example when data contains void pointers, xfgen will produce error messages directing the developer to the places in the code requiring annotation. However, if the developer forgets to annotate that a field points to an array rather than a single instance, xfgen will silently generate incomplete traversal code. We encountered this problem at some points during our experiments and it was often tricky to diagnose. The visualization tool that we propose in Chapter 7 is intended to identify such errors more readily.

There are some code constructs, including tagged void pointers and **union**s, for which annotations are insufficient for xfgen to generate traversal code. In those cases, the developer must write a manual transformation rule to show how to traverse that field, which they must include in every patch. Redis required one of these rules and implementing it was fairly straightforward.

The most common source of problems when modifying existing programs to work with Kitsune were pointers in the heap to static addresses, e.g., addresses of global variables, functions, and string literals. Once the old-version code is unloaded following an update, these pointers (if they have not been updated to point to new-version addresses) become invalid and dereferencing them will crash the program. xfgen generates code to transform pointers to global variables or functions when they are encountered during traversal. The generated code was usually sufficient, except when handling pointers to static C strings (which xfgen does not currently transform to new-version addresses) or when static pointers are maintained inside of library code. We encountered problems with static C strings in Tor; to fix them, we realized that the data structure containing them was initialized at startup, but never written to afterwards, so we annotated it to not be migrated during an update. Both Tor and Memcached required some manual code to ensure that function pointers inside the libevent library are reinitialized at the new version. We discuss potential solutions for supporting state in libraries in Chapter 7.

5.5 Related Work

Table 5.4 characterizes the mechanisms used to implement Kitsune and other recent C/C++ DSU systems; most of these systems target applications, while Ksplice, K42, and DynaMOS support (or are) OS kernels. We discuss tradeoffs resulting from these mechanism choices, and argue that Kitsune provides the greatest flexibility and best performance with modest programmer effort. The footnotes in the table summarize the discussion below.

Code updates. Most systems effect code updates at the level of individual functions (or objects). As noted in the first column, Ksplice [8], OPUS [5], DynaMOS [46], and POLUS[5] [17] insert, at run-time, a trampoline in the old function to jump to the function's new version. As noted in the second column, Ginseng [51] and K42 use indirection: Ginseng compiles direct function calls into calls via function pointers, while the K42 OS's object handles are indirected via a hand-coded *object translation table* (OTT); updates take effect by redirecting indirection targets to the new versions.

There are several drawbacks to using these mechanisms. Trampolines require a writable code segment, which makes the application more vulnerable to code injection attacks. Trampoline-based updating may break programs optimized using inlining, since it presumes to know where the start of a function is, so POLUS and OPUS both forbid inlining. (Ksplice is able to account for the compiler's inlining decisions.) Using indirect calls adds overhead to normal ("steady state") execution

[5]LUCOS is from the same research group that produced POLUS, and is essentially a version of POLUS that uses VMMs to effect changes in operating systems. All comments we make about POLUS apply equally to LUCOS, so we do not mention it further in this section.

	Code upd			Data upd			Timing	
	tramp	ind	prog	repl	shdw	wrap	nonactv	upd pts
DynaMOS[4]	×				×			
Ekiden			×	×				×
Ginseng[1,2,3,4,5]		×				×		×
K42[2,5]		×		×			×	
Kitsune			×	×				×
Ksplice	×				×		×	
OPUS[2]	×			–	–	–	×	
POLUS[2,4,5]	×			×				
UpStare[2,3]			×	×				(×)

[1]needs deep analysis
[2]inhibits optimizations
[3]pervasive instrumentation
[4]mixes old and new code
[5]relaxed thread sync.

Table 5.4: Comparing DSU systems for C/C++

and also inhibits inlining. Most onerously, neither trampolines nor indirections support updating functions that never exit, such as `main`, which changes relatively frequently [34], or the scheduling loop in the OS. In the best case, programmers must refactor the program so that long-running loop bodies are in separate functions [51].

The remaining three systems, UpStare [45], Ekiden [35],[6] and Kitsune support more general changes by updating at the granularity of the whole program rather than individual functions. UpStare loads in code for the new program and then performs *stack reconstruction*: the running program *unwinds* the current stack one function at a time back to `main`, and then *rewinds* the stack to a new-version program point specified by the programmer. In contrast, Kitsune relies on the programmer to migrate control to the equivalent new-version program point.

[6]Ekiden is the precursor of Kitsune and works by transferring state to an updated process; the programmer API is roughly the same, but Ekiden induces slower update times and requires more memory.

Kitsune's manual approach pays dividends in both better performance and simpler semantics. To allow updates to happen at any program point, UpStare's compiler adds unwinding/rewinding code to all functions; while convenient, this imposes performance overhead on normal execution. Moreover, to exploit UpStare's flexibility, a developer must carefully define how to map from all possible old-version thread stacks to new-version equivalents. UpStare reduces this burden allowing the programmer to limit updates to fewer program points, just as Kitsune does. But then the value of general-purpose stack reconstruction is less clear. Kitsune allows all compiler optimizations, and code to support control migration imposes no overhead during normal execution since such code only appears on program paths leading to update points, and these paths tend not to intersect with normal execution paths. Moreover, expressing control migration in the code rather than in a specification to the side is arguably advantageous: with only a few update points there is very little code to write, and its presence in the program makes the update semantics explicit and easier to understand.

Data updates. Returning to the table, we can see that most systems handle changes to data structure representations employing *object replacement*, in which the programmer can allocate replacement objects and initialize them using data from the old version. Ksplice and DynaMOS leave the old objects alone but allocate *shadow data structures* that contain only the new fields. Ginseng uses an approach called *type wrapping*: programs are compiled to use mediator functions to access updatable objects, and these functions initiate transformation of objects that are not up to date.

Shadow data structures have the benefit that fewer functions are changed by an update: if we add a new field to a **struct**, then only code that uses that field is affected, rather than all code that uses the **struct**. But programmers must write additional code to deal with shadow fields and manage their lifetimes, which imposes run-time overhead and clutters the software over time. Type wrapping has the benefit that there is no need to find objects in order to update them; rather, object transformation will occur lazily as the new version executes. But type wrapping has several limitations: (1) mediator functions slow normal execution; (2) objects must be compiled to have extra "slop" space for future growth which hurts performance (e.g., cache locality) and may prove insufficient for some changes; (3) the change in representation forbids certain coding idioms (e.g., involving typecasts to/from **void**∗), which Ginseng identifies using a whole-program analysis that has trouble scaling.

Object replacement offers the best steady state performance, but there must be a way to find all instances of changed objects (e.g., by chasing pointers from global variables) and redirect these pointers to newly allocated, transformed objects. K42's coding style makes this easy—the system can just traverse the OTT—but most applications are not written this way. Kitsune's xfgen tool is able to generate traversal code given relatively small specifications and some type annotations; in other systems, the programmer burden is higher. Note that DSU for type-safe languages can avoid xfgen's traversal generation: the garbage collector can be used to automatically find and initiate transformation of changed objects [64, 27] without need of further type annotations.

Timing. Returning to the table, we consider how systems determine when an update may take effect. Ksplice, K42, and OPUS only permit an update when no thread is running code that will be changed. While this restriction reduces post-update errors, it does not eliminate them [34], and moreover imposes strong restrictions on the form of an update and how quickly it can be applied.

For increased flexibility, other systems allow updates to active code. Kitsune and UpStare updates take place when all threads reach a programmer-designated *update point* (for UpStare, such points may be system-determined). We have found this simple approach works quite well in practice. In contrast, Ginseng allows an update to take effect so long as it *appears* as though it occurred when all threads were at update points [49]. This approach does accelerate update times, but the static analysis that underlies it scales poorly and is conservative, requiring awkward code restructurings. POLUS allows threads to update immediately, and thus because POLUS updates take effect at function calls, after an update a program may wind up running bits of old and new code at the same time; a study using Ginseng showed mixing code versions substantially increases the odds of errors [34]. Moreover, POLUS data structures are versioned, with version N of the code accessing version N of the data, so the programmer defines callbacks (invoked via virtual memory page protection support) to keep the copies in sync. We imagine this could be tricky. Our experience with the simple barrier approach suggests these more sophisticated approaches, with higher programmer demands, may be unnecessary.

Checkpointing. Checkpoint-and-restart systems [58] allow programs to be relaunched "in the middle" of execution from a checkpoint. At a high-level this bears

some similarity to DSU, but checkpointing systems do not provide support for changing code or data representations on restart. As mentioned above, in earlier work we developed Ekiden, a system that serializes and transfers state between processes—i.e., Ekiden works like a checkpointing system that does permit code and data modification. However, we found that the cost of transferring state in Ekiden was significant, and hence moved to the Kitsune model, which allows in-process code and data changes.

Dynamic updating in other languages. Compared to C/C++, DSU for higher-level languages is a cleaner affair. In particular, such languages use garbage collection (GC) and can delegate the task of finding and transforming changed objects to the GC [64, 27]. Moreover, these languages are type-safe, so added metadata for managing DSU can be safely hidden from the application (avoiding the problems experienced in Ginseng). For VM-based languages like Java and C#, the JIT avoids any limitation on the use of optimizations. For example, the JIT can track its prior inlining decisions and undo them when an inlined function is updated, redoing them once the update takes effect [64]. Kitsune was designed specifically for C, which lacks such high-level features and places a higher premium on performance and low-level control.

Stewart and Chakravarty implement a whole-program updating system for Haskell that, like Kitsune, makes updating behavior explicit [61]. Haskell is a purely-functional language and so it does not support mutable state scattered throughout the program as C does; as such they provide no special support for transforming program state. They do not report how many lines needed to be changed to add

updating support or how much steady state overhead their approach introduces.

5.6 Conclusions

We have presented Kitsune, a new system for dynamically updating C programs. Kitsune works by updating the entire program at once, thus avoiding the restrictions imposed by other DSU systems on data representations, programming idioms, and compiler optimizations. Kitsune's design allows program changes for updatability to be simple and informative, and xfgen makes writing state transformers much easier. Our results applying Kitsune to single- and multi-threaded benchmarks show that Kitsune has essentially no performance overhead, and code changes required to use Kitsune are comparable to, or only slightly more than, prior systems. We believe that Kitsune's careful balancing of flexibility, efficiency, and ease-of-use makes it a major step forward in practical dynamic software updating for C.

Chapter 6

A Study of DSU Quiescence for Multithreaded Programs

This chapter describes an experiment we performed to evaluate the effectiveness of Kitsune's handling of multithreaded updates. This study measured the period of reduced availability due to time spent waiting for all threads to update for six programs. Here, we report our findings, and describe the implementation strategies we used to reduce the delay.

Finding ways to apply multithreaded updates with minimal delay has been a fruitful topic for the research community, and a variety of solutions have been proposed. Most of this research has followed a common theme: updates should be supported at as many points during program execution as possible. However, this goal has a serious drawback: developers must reason about the correctness of all possible update timings, and that task becomes harder to do the more update points and threads there are. Moreover, many of the proposed techniques also employ complex program analyses or other mechanisms that are difficult to use, scale poorly, and/or impose run-time overhead.

As we have seen, Kitsune takes the opposite point of view: it only support updates at a few, developer-identified *quiescent points*, i.e., program locations that are reached between iterations of event-processing loops and at which there is typically less in-flight state. For example, the code below shows a typical thread body

to which we have added an update point.

```
1 void *thread_entry(void *arg) {
2   /* thread init code */
3   while (1) {
4     qbench_update();    /* update point */
5     /* loop body: typically handles a single program event */
6   }
7 }
```

We refer to the state in which all program threads have reached an update point as *full quiescence*. While full quiescence is attractive because it reduces the programmer's reasoning burden, one concern is that reaching full quiescence may significantly delay the application of an update, and may degrade the program's performance in the process.

We present a small empirical study that shows that, for many programs, simple modifications allow quiescent points to be reached sufficiently often to support updates with little delay. This result suggests that rather than use mechanisms that are hard to implement and hard to reason about, we can instead ask programmers to modify programs in simple ways to make DSU more effective.

For our study, we added quiescent update points to Apache httpd, icecast, memcached, suricata, iperf, and space tyrant. We chose programs covering a wide range of domains including media streaming, caching, intrusion detection, and gaming. We chose the quiescent points using the manual strategy we evaluated in Chapter 2 and applied to Kitsune. We linked each modified program with a library, QBench, with which we measured the time the program took to reach full quiescence under various workloads.

Reaching quiescence may be delayed by blocking calls, e.g., those that perform I/O. We found that two simple program changes could effectively overcome this delay. First, the DSU runtime can interrupt some blocked calls by sending a signal to the thread; several updating systems do this "for free" since signals are used to alert the program that an update is available. Second, for the remaining blocking calls, we changed the programs to use alternative, interruptible implementations of some system functions (specifically, pthread_cond_wait and sleep) and added code to redirect control flow back to an update point when a blocking call is interrupted. All of these changes, and the implementation of QBench, are described in detail in Section 6.1. On average, programs needed 22 lines of code to be changed including adding update points. With these changes, the median time to quiescence with workload was 0.200ms (0.169ms w/o load), with a worst case of 107.558ms (w/o load) for icecast, and a best case of 0.078ms (w/o load) for space tyrant. These results are described in detail in Section 6.2.

In summary, we found that, for a representative suite of benchmark programs, reaching full quiescence can be done quickly given proper run-time support and a small number of program changes. While more experience is needed to see if this result generalizes, we believe it suggests that simple mechanisms may be sufficient to properly balance the safety and timeliness of dynamic updates.

Author Contributions. This work will appear in HotSWUp '12 [30]. I was lead author, designed QBench, and modified roughly half of the benchmark programs to support benchmarking. My collaborator, Karla Saur, modified the other benchmark

programs and ran the experiments.

6.1 Achieving full quiescence

To test our hypothesis that full quiescence permits sufficiently timely dynamic updates, we modified several multithreaded programs to include appropriate update points, and then measured the time it took to reach full quiescence. We describe our approach here, along with the QBench library we developed to implement quiescence and measure the time required to achieve it.

6.1.1 Basic approach

For each program we must add a handful of calls to qbench_update to identify legal update points. The semantics of qbench_update is simple: if no update has been requested, it is a no-op; otherwise, the calling thread blocks until all other threads have also called qbench_update. In detail, we request an update by sending a program the SIGUSR2 signal. QBench installs a signal handler that sets a flag indicating that an update has been requested. A qbench_update call blocks if the flag is set. Once all threads have blocked, the system has reached *full quiescence*. In an actual DSU system, the update would take effect at this point. For our study, QBench instead reports the *quiescence time*, which is the elapsed time from when the first thread reaches an update point (marking the start of reduced program availability) to when the last thread has. Then it simply unsets the flag and releases all of the threads to continue their execution, so we can verify the program operates

as expected.

Quiescence is achieved when all threads have reached an update point, which we track by maintaining a thread count and ensuring that each thread hits an update point. To determine when all threads have reached qbench_update (and to store other thread-specific metadata described later), we replace calls to pthread_create with calls to QBench's qbench_pthread_create, which tracks the lifetime of each thread. QBench maintains a count of threads and stores the metadata for all threads in a doubly linked list. Each thread also stores a pointer to its own metadata using thread-local storage, which permits constant time access. When a thread dies, a callback is invoked to clean up thread-local data. QBench provides a cleanup function for a thread's metadata that unlinks it from the global list and decrements the global thread count.

6.1.2 Avoiding blocking

Achieving full quiescence may be delayed or thwarted by *blocking calls*. For example, a call to qbench_update may be preceded by a call that reads from a socket. If this call blocks, the thread will not reach its update point until data is available. Worse still, one thread could hold a mutex when it reaches its update point, but then another thread could block on the same mutex prior to reaching its own update point, delaying full quiescence indefinitely.

To avoid these problems, the programmer must ensure that all blocking calls that appear on any path to an update point are *interruptible*. This requirement

immediately rules out the second situation above: the program is not permitted to hold any locks when it reaches an update point, because pthread_mutex_lock is not interruptible (nor would it be sensible to make it so). Fortunately, we found that no quiescent points in the programs we considered ever held a lock.

For the benchmark programs in our study, blocking calls that could inhibit quiescence fell into two categories: blocking I/O calls and calls to pthread_mutex_wait. We found that in both cases, we could interrupt the call and the program would either behave correctly with no changes, or we could make it behave correctly with a few small modifications.

6.1.2.1 Blocking on I/O

Mature server programs are often written to deal with interrupted blocking calls, so adding update points to such programs requires little or no change. Consider the following example.

```
1  void *thread_entry(void *arg) {
2    /* thread init code */
3    while (1) {
4      qbench_update();
5      res = accept(sockfd, addr, addrlen);
6      if (res == 1 && errno == EINTR)
7        continue;
8      /* ... handle connection */
9    }
10 }
```

Under normal circumstances an accept call will block until a connection is accepted. However, if a signal is received the call will be interrupted, returning 1 and setting

the errno to EINTR.[1] In the above code snippet, the programmer has accounted for this possibility by returning control to the start of the loop so as to retry the accept. Because an operator initiates a program update by sending the process a SIGUSR2 signal, adding the update point to line 4 in the example ensures the blocked accept call will be released and will reach the update point quickly when the update is signaled.

Note that signals are normally handled by a program's main thread, so only that thread's blocking calls are interrupted. To interrupt blocking I/O calls in all threads, QBench's main signal handler sends a signal to any other thread that has not already reached its update point and is not waiting on a condition variable; how we handle the latter situation is described next.

6.1.2.2 Blocking on condition variables

We observed that the threads in our benchmark programs often coordinate using condition variables, blocking on calls to pthread_cond_wait. As a matter of good style, programmers guard against spurious wake-ups of such calls by placing them in loops, as the following non-highlighted code on lines 6–7 shows:

```
1 void *thread_entry(void *arg) {
2   /* thread init code */
3   while (1) {
4     qbench_update();
5     pthread_mutex_lock(&mutex);
6     while (! input_is_ready () && !qbench_update_requested()) {
7       qbench_pthread_cond_wait(&cond, &mutex);
```

[1] POSIX supports auto-restarting interrupted, "slow" system calls [60] (i.e., without returning EINTR), which would defeat our scheme. We disable that feature by excluding SA_RESTART from the configuration mask used when installing the signal handler.

```
8       }
9       pthread_mutex_unlock(&mutex);
10      if (qbench_update_requested())
11        continue; /* reaches qbench_update */
12      /* ... handle connection */
13    }
14  }
```

To allow an update to interrupt this idiomatic use of condition variables, we first modify the condition to check whether an update has been requested, as shown in the highlighted code on line 6. We also modify the code following the condition variable loop to jump back to the start of the loop if an update is requested (lines 10-11). This changes ensures that an update is reached if pthread_cond_wait wakes. One straightforward way to force the pthread_cond_wait call on line 7 to wake up would be to replace it with pthread_cond_timedwait with a short timeout. But this approach incurs some unnecessary delay and potentially expensive polling overhead. Therefore, we replace the pthread_cond_wait call with a call to qbench_pthread_cond_wait, which (before calling pthread_cond_wait) notes the condition variable argument in the global list of threads so that it can later be signaled by another thread once an update has been requested.

These solutions for waking a thread blocking on I/O or condition variables require that another thread be available to signal the process or condition variable (e.g., since pthread_cond_signal cannot safely be called from a signal handler). This presents a problem if the thread that receives the initial updating signal is blocked on a condition variable and so may not wake up to signal other threads. For this reason, QBench launches one additional thread that sleeps during the vast majority

of execution, but periodically wakes and checks the update-requested flag. If an update was requested, it will attempt to signal any threads that have not yet reached an update point.

While the above two circumstances cover the vast majority of blocking calls, we note that one of our benchmark programs, Suricata, required custom code to be called from a signal handler to unblock one of its threads (as we describe in Section 6.2.1). POSIX requires that the same signal handler function be used for all threads in a process. To compensate, QBench provides a library function, qbench_thread_update_callback that allows the developer to provide a callback function to be executed for the current thread when an update signal is received.

6.2 Results

This section presents the results of a study in which we used QBench to measure the quiescence behavior of six multithreaded programs. We found that the changes required to support full quiescence were small (an average of 22 lines per program), and quiescence could be achieved fairly quickly (in less than 1ms in most cases).

6.2.1 Experimental setup

The first three columns of Table 6.1 describe the size and thread structure of our subject programs. This subsection describes each program briefly, the workload that we used to test it, and how we needed to modify it to achieve full quiescence

Program	LoC Total	# of Threads	Upd Points	Changed LoC (†)	Required Manual Chgs	w/Load (ms)		w/o Load (ms)	
						All Chgs	Upd only	All Chgs	Upd only
httpd-2.2.22	232651	$2+c,\ \ c=3$	5	7 (5)	3 (Cond. Var. Loop)	0.185	0.230	0.123	0.150
icecast-2.3.2	17038	6	12	3 (3)	1 (Thread Sleeps)	105.152	954.32	107.558	986.265
iperf-2.0.5	3996	$3+n^{\circ},\ \ n=1$	5	8 (3)	1 (Cond. Var. Loop)	0.193	DNQ	0.169	DNQ
memcached-1.4.13	9404	$2+c,\ \ c=4$	4	27 (4)	2 (libevent changes)	0.166	DNQ	0.155	DNQ
space-tyrant-0.354	8721	$3+2n^{\circ}, n=5$	6	8 (6)	1 (Thread Sleeps)	0.426	20.583	0.078	20.304
suricata-1.2.1	260344	$8+c,\ \ c=3$	7	11 (6)	1 (libpcap break)	0.503	68.098	0.378	DNQ

*Configurable: c workers °Varies by n connected clients †Calls to QBench excluding update DNQ = Does not quiesce.

Table 6.1: Thread Information

rapidly.

Apache httpd. Apache httpd is a widely used web server. We configured httpd to use thread-based concurrency with 3 worker threads. To achieve full quiescence quickly, we first needed to make the standard changes described in Section 6.1 and summarized in columns 4–6 of Table 6.1. We report the number of update points and lines of code changed for each program and keep a separate count of the changes that include calls to our library. In the remainder of this section, we describe only the changes given in the Manual Changes column, i.e., those that tweak existing program code beyond adding/substituting calls to QBench.

For httpd, the only such change was modifying a loop written to immediately retry an interrupted poll operation to break out of the loop if an update is requested.

For our experiments with httpd, we used a workload of downloading a large file from the server.

Icecast. Icecast is a streaming audio server that is popular for hosting Internet radio stations. In its standard configuration, it runs with 6 threads, all of which quiesce without modification. Several of the threads use sleep operations to reduce polling; it turns out these sleep times are the dominant component of the time to reach full quiescence.

For icecast, we selected a workload that corresponds to receiving an audio stream from an outside source and forwards it to connected clients. We used the Ezstream command-line tool to generate a source mp3 stream, connected 5 mplayer clients, and requested an update mid-stream.

Iperf. Iperf is a program that measures the network performance (e.g., bandwidth, delay jitter, and datagram loss) between two machines. Although the same executable is used for both client and server modes, we only modified the server code to reach update points during execution. Iperf has 3 threads at startup and an additional thread for each connected client. The main thread has a conditional wait in a while loop. We added an additional update-request–flag check to jump back to the update point when needed. We measured iperf quiescence times while a client (running on the same machine) performed a network measurement.

Memcached. Memcached is a high-performance, distributed caching server that uses libevent to drive its main thread and a configurable number of worker threads. We added an additional libevent handler for the main-thread event loop to respond to SIGUSR2 and break out of libevent. Upon return, the main thread sends a byte on the notification sockets for each worker and then reaches an update point. The notifications cause each worker thread to enter its event handler, where it sees that an update was requested ands returns from libevent to reach an update point. Effectively, these update points were placed at quiescent points, although the style is different from the other programs.

Our test requests an update to a Memcached instance under load from the memslap Memcached benchmark.

Space Tyrant. Space Tyrant is a server for a text-based, multiplayer space strategy game. At startup, Space Tyrant has 3 threads and creates 2 more for each connected player. Space Tyrant implements long sleep operations using loops that check for

server-shutdown events between shorter sleeps. We added an additional update-request–flag check to jump back to the update point when needed. Space Tyrant's threads required no additional modification. We updated Space Tyrant with 5 concurrent telnet client connections.

Suricata. Suricata is a network intrusion detector that monitors the packets that pass through a network interface. By default, Suricata is configured to use 11 threads. One thread required special treatment: It calls into libpcap's blocking pcap_dispatch function to process packets. libpcap provides a function pcap_breakloop that can be called from a signal handler to interrupt pcap_dispatch. We install a thread-specific handler function (cf. Section 6.1.2.2) to break out of the loop when an update signal is received.

In our tests we ran Suricata with a default set of 7,946 packet analysis rules. We requested an update as Suricata processed the packets produced by a constant stream of 10 concurrent http requests and one large file download.

6.2.2 Quiescence times

The four rightmost columns of Table 6.1 report the median quiescence times of 11 benchmark runs. All tests were run on a machine with an Intel Core 2 Duo T5550 processor with 2GB of memory. For each program, we measured the time taken to reach full quiescence under two workloads: while the server was idle (i.e., no connected clients) and while performing the (program-dependent) work described in the previous section. The idle workloads were used to reveal problematic cases

where threads block indefinitely waiting for input. We also measured the quiescence times when using only update points and no other QBench calls.

The table shows we were able to reach full quiescence quickly for both workloads when using QBench; limiting ourselves only to update points would fail to quiesce some programs. Without using our library with Suricata, the quiescence time is variable depending on the rate of traffic filling the input buffers. Nearly all programs quiesced in under 1ms (for both workloads); Icecast's longer times are due to sleep operations inserted by the programmers.

6.2.3 Threats to validity

For this study, we did not actually apply dynamic updates to the benchmark programs, so we cannot be sure that the quiescent points inserted are the ones that would be used in practice. However, the choice of quiescent points is largely dictated by the structure of the code (usually at the beginning of each thread's event loop), so it is unlikely that we might find a preferred point that would be reached less often.

In our study, we ensured that blocking operations that occur at the beginning of event handling, when it is safe to immediately jump back to the beginning of the loop, are interruptible. A blocking I/O operation in the middle of event-handling could delay full quiescence if clients are extremely slow or stalled. Our current experiments do not attempt to force this situation to occur, so we do not know whether this is a problem in practice for these programs.

It is also possible that our observations for this particular selection of programs do not generalize to most other programs. To avoid this risk, we have attempted to consider a wide variety of program types. It may also be useful to specifically look for programs that are implemented in such a way that they would not work well in this approach.

6.3 Prior work

Here we summarize prior work on multithreaded program updates, focusing on how that work controls update timing. We find that while some prior work is insufficiently flexible, much prior work is perhaps overly concerned with minimizing update times. The results of our study suggest that such concerns may not be warranted.

Several systems [8, 38, 64] forbid updates to any code that is actively running. Some synchronization is needed to ensure that all threads satisfy this condition. Unfortunately, as we have observed in Chapter 2, this safety condition is insufficient to ensure update safety, and it provides no guarantee that an update is applied in a timely manner. For example, if a program's `main` function is modified by an update, the update will be delayed indefinitely because `main` is always running.

STUMP [49] lifts the restriction against updates of active code: instead, any update may take effect when all threads have reached programmer-identified *update points*. To potentially reduce delay at update time, STUMP implements a relaxed synchronization protocol that permits an update whenever it *appears* as if the up-

date took effect at legal update points. A static analysis determines which program points are equivalent to update points [50] and incorporates this information into the synchronization protocol. Unfortunately, in the worst case, there is no guarantee that meaningful opportunities for updating will be created. Moreover, it may be difficult for a developer to understand the results of the analyses, e.g., to understand why it did not permit more update points. Finally, the static analysis itself is fairly intricate, and may not scale to large programs. The reported update times for STUMP for the same programs used in our study (icecast, space tyrant, and memcached) are higher—1,068ms, 6ms, and 1ms, respectively—though the experimental setup is different.

UpStare [45] supports *immediate* updates, with no synchronization, by allowing threads to update at any point during program execution. To provide this support, UpStare requires the developer to create a mapping between each program point in the old version of a changed function and the corresponding point in its new version; such a mapping could require a significant manual effort, depending on the size and complexity of the change. UpStare prevents blocking library calls from delaying an update by substituting versions that include special handling when an update has been requested; we use a similar, but simpler, approach in this work. The UpStare paper does not report update times for any multithreaded programs.

POLUS [17] supports immediate updates by permitting contemporaneous threads to execute code from different program versions. When a thread accesses a piece of shared state, POLUS uses developer-provided, bidirectional transformation functions to ensure that each thread sees the representation of state that it expects.

With this approach, however, the developer must additionally puzzle out the possible multi-version executions and reason that thread interactions via bi-directional transformations will make sense. POLUS was applied to one multithreaded program, Apache httpd. The authors report (for a different hardware configuration) update times on the order of 15ms, but these also include time to transform any in-flight state.

6.4 Conclusions

In this study, we found that, for a diverse set of benchmark programs, explicit update points at quiescent program points were able to support multithreaded updates quickly and with little implementation complexity. This finding suggests that DSU systems like Kitsune that do not rely on complex program transformation or analysis (i.e., those most likely to see real-world adoption) may be sufficient to bring runtime updates to a non-trivial set of multithreaded programs.

Chapter 7

Future Work

This dissertation has presented several contributions to the state of the art in DSU correctness and implementation. Over the course of these projects, we have identified several new research directions and potential improvements to our current approaches.

7.1 Checking Tools for Kitsune

Our current implementations of DSUTest and the program merger target updates for Ginseng or an updating system with similar semantics. To expand the usefulness of those techniques, we could adapt them to work with Kitsune.

DSUTest. For DSUTest, we believe it should be straightforward to create a modified testing driver program that can serve the two functions required by DSUTest: to output the number of times each update point was reached during a non-updating execution, and to control the timing of the dynamic update based on a command-line argument. This would simplify the testing process, since the same compiled program could be used in production or for DSU testing. This approach would not attempt to compensate for non-determinism, but we believe that concern is less significant in practice than it was for our empirical study, where we took pains to ensure the validity of our results. Although this implementation would not support

update-test minimization, our experiments showed that minimization was unneeded (and unhelpful) when Kitsune-style manual update points are used.

DSU Merger. For the DSU merger, the control-flow portions of the merging process could potentially be quite simple. The old- and new-version symbols could be renamed to be distinct and merged into the same file. A modified `kitsune_update` would be included in the merged file that would, upon update, invoke the new-version `main` function and then terminate execution after `main` returns. One potentially limiting factor would the need for a checking tool that could reason precisely enough about which update point was taken when analyzing the new-version start-up code. It may be possible to reduce the burden on analysis tools by generating separate start-up code for each possible update point. This would in essence be the result of partially evaluating the start-up control-flow conditions (i.e., `kitsune_is_updating_from ("point")`) with respect to name of the update point taken. Another challenge would be to ensure that the xfgen specifications are compiled to code that can be effectively analyzed by the target verification tools. For example, the current xfgen compiler generates code that looks up symbol addresses in a lookup table before performing transformation. A symbolic executor like Otter should be able to support this code, but many other static-analysis tools cannot.

7.2 Kitsune Extensions

Over the course of our work on Kisune, we have identified several directions for future work.

Automation. Although we have seen good evidence that making update behavior explicit in the program is advantageous, there is still room for tools to help the developer make those modifications. For example, it may be valuable for a tool to analyze a program to detect its long-running event-processing loops. The tool could apply heuristics based on the program's loop nesting structure, the particular operations performed in the loop body, and/or analysis of loop termination via reachability [20] to suggest which loops should contain update points. Once update points have been added to the program (either manually or through suggestions), a tool could suggest modifications to the program's start-up control flow to ensure that the program returns to the correct loop when started up following an update. Alternatively, a tool could identify what state might be overwritten when starting up following an update. This information could be used to detect errors in a developer's modifications to start-up control flow.

Visualization. To help developers write state transformers, we could develop a tool to visualize the old-version state at the time of the update. To facilitate this tool support, the program could be compiled to expose data size information at runtime. This tool could allow developers to iteratively construct their transformation rules and see their effects immediately. Such a tool could also aid developers in implementing and debugging their type annotations.

Library State. Currently, Kitsune instruments the program itself for state migration, but does not have direct support for migrating the state maintained within libraries. The programs that we have updated so far either use libraries that do not

contain state that is invalidated by the update (e.g., function pointers) or they have APIs for reinitializing such state. If Kitsune-style updates become more popular, important libraries could be modified to export cleaner interfaces for performing migration. To support this, future work could experiment with API designs that facilitate library state migration in a modular way (e.g., by having the runtime system pass a mapping function into the library that it could use to update function addresses). Likewise, specific requirements for library "reset" APIs could be developed. DSU-compatible libraries could elect to either directly support migration or else they could implement a reset interface.

Lazy Transformation. Currently Kitsune transforms all program state during the update process. This avoids the costly compiled-in version checks and type-wrapping that Ginseng [52] uses to implement laziness. Nonetheless, if a program with a large data set requires extensive transformation, the resulting delay may be too disruptive to the program's clients. In future work, we plan to look at ways that virtual-memory protection could be used to implement lazy transformation with Kitsune. For example, the Kitsune runtime could protect pages containing new version state that has not yet been migrated forward and delay that migration until a fault is handled. This will produce some steady-state performance overhead due to transformations performed mid-request and protection faults that result from false sharing. Experimentation is needed to assess the potential costs and benefits of this approach.

7.3 DSU as a Feature for Other Languages

Kitsune implements whole-program updating and explicit updating semantics, aiming to support DSU in harmony with the C language. Future work could analyze what it means for DSU to be in harmony with other languages/environments, and determine whether an approach based on Kitsune would be appropriate (or how it would need to be modified).

Java. One promising target for this research is the Java language. Unlike C, the Java runtime system provides precise type information. This means that none of Kitsune's type annotations for C (e.g., E_PTRARRAY) would be needed. Also, it should be possible to write code to walk the Java heap and perform transformation using reflection. Another option would be to use an instrumented JVM to allow transformation to occur during a garbage collection (GC) pass (or perhaps lazily if concurrent GC were used). Finally, the xfgen specification language is not currently equipped to define transformation involving class hierarchies, and it may need to be extended in interesting ways to support complex changes (e.g., like modifying a class' parent).

Dynamic Languages. Since dynamic languages like Ruby [2] and Python [1] are generally less amenable to static analysis than statically typed languages like C, Kitsune-style updating, where the new version begins execution from the start, may be an excellent match since it requires no complex analysis. However, xfgen does use C program types to generate transformation. An interesting research question would be to consider what tool support would aid developers in writing state trans-

formation code for dynamic languages. One possibility would be to build a tool like the visualization tool proposed earlier in this chapter for dynamic languages. This approach would fit well with the test-centric philosophy preferred by many users of dynamic languages.

Chapter 8

Conclusion

In this dissertation, we have argued that dynamic software updating is best treated as a program feature.

- For purposes of DSU correctness, this means that developers need the ability to reason about the correctness of specific program behaviors under update, just as they would for any other program feature. To support this claim, we presented an empirical study that showed that the most common automatic safety checks do not ensure update correctness and identified a critical set of problems with reasoning about and using transparent DSU systems. We developed a new approach to specifying the correct behavior of program features under DSU, and showed that it can often be applied with little additional work over single-version specification. Finally, we developed a novel approach to verifying DSU correctness using off-the-shelf tools that are not DSU-aware.

- For purposes of DSU implementation, we claim that programs should be written to make the important parts of DSU behavior (e.g., when updates can happen and what happens during and after an update) explicit in the program code. We tested this claim by implementing Kitsune, a new DSU system for C that avoids the restrictions imposed by other DSU systems on data representations, programming idioms, and compiler optimizations. Kitsune addresses

the problems that our empirical study identified with transparent updating systems and operates in harmony with C. We applied Kitsune to a large set of updates to widely used programs and found that using it required little programmer effort and imposed virtually no steady-state overhead on program execution. We find Kitsune to be the most flexible, efficient, and easy to use DSU system for C developed to date.

Appendix A

Comparing failures allowed by different timing mechanisms

Figure 2.4 shows the total count of update points for each patch and how many test failures we permitted by each timing restriction. Although this was sufficient to evaluate the efficacy of the checks, we were also curious about the exact relationship between AS and CFS in terms of which particular points they allow.

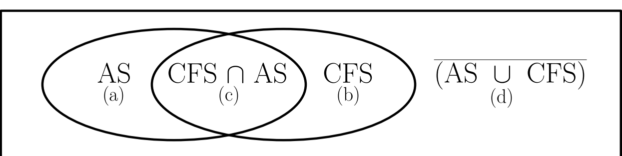
The left half of Figure A.1 classifies each failing update point based on which safety checks would have prevented the failure. We break down the failures into four basic categories, one per row of the table, visualized in the Venn diagram above it.

For vsftpd, OpenSSH, and ngIRCd, we see that well over 97% of the failing points are disallowed by both safety checks (row (c)). 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)).[1] Lastly, well fewer than 1% of the failures for each application are allowed under both safety checks (row (d)).

A.1 Program Phases

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

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



	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	2,194	2%
	(b) Only Prevented by CFS	0	0%
	(c) Prevented by AS and CFS	126,130	98%
	(d) Prevented by Neither	0	0%
	(a+b+c+d) Total Failures	128,324	100%
ngIRCd	(a) Only Prevented by AS	98	$< 1\%$
	(b) Only Prevented by CFS	0	0%
	(c) Prevented by AS and CFS	308,364	$\sim$100%
	(d) Prevented by Neither	0	0%
	(a+b+c+d) Total Failures	308,462	100%

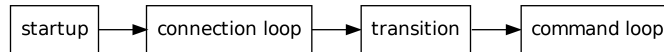
Failures Prevented

	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	337,036	9%
	(b) Only Allowed by CFS	1,136,488	30%
	(c) Allowed by AS and CFS	1,736,079	46%
	(d) Allowed by Neither	594,715	16%
	(a+b+c+d) Total Passing	3,804,318	100%
ngIRCd	(a) Only Allowed by AS	0	0%
	(b) Only Allowed by CFS	643,044	34%
	(c) Allowed by AS and CFS	941,565	49%
	(d) Allowed by Neither	333,580	17%
	(a+b+c+d) Total Passing	1,918,189	100%

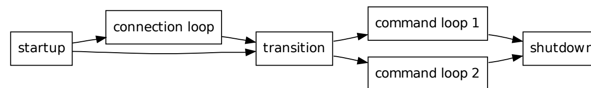
Successes Allowed

Table A.1: Breakdown of results by safety check

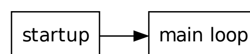
corresponding to their long-running loops and the transitions between them. The execution of vsftpd consists of a connection loop that accepts session requests and forks child processes to handle them, and a command loop in each child process that receives, processes, and responds to requests from the 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 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 for OpenSSH occur as follows:



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



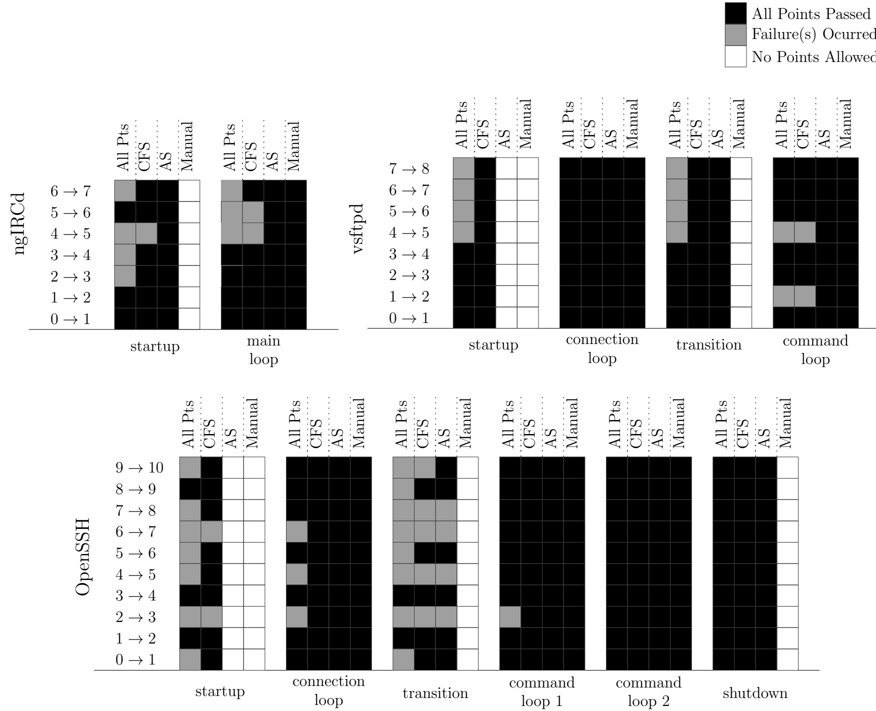


Figure A.1: Updatability across program phases

Figure A.1 summarizes test failures by program phase and patch (the full tables from which this figure is derived can be found in Figures A.2, A.3, and A.4). Black boxes indicate that all tests pass and grey boxes indicate one or more failures. White boxes indicate that no allowable update points were reached during execution of the particular phase.

We can see that CFS allows at least one update point in each program phase, while AS precludes updates during the startup phases for OpenSSH and vsftpd. There are no manually placed update points in the startup and transition phases. In all cases, updates are permitted within the command and connection loop phases, ensuring reasonable availability to updates. Moreover, for Manual and AS, no failures occur at update points within the loops, while for CFS, the only loop-phase failures occur in the vsftpd command loop.

Our observation that updates within the loops are most important, while updates within the startup or transition phases are much less so (since these phases are finite and presumably short) suggests that update availability is best evaluated using a simpler criteria: are updates supported during each long running loop of the program? Each of the three timing restrictions that we have evaluated satisfy this criteria. Manually identified update points were able to do so while not permitting any failing update executions in our experiments.

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 2.4 and A.1.

A.2 Minimization Effectiveness

Figure A.2 illustrates the effectiveness of our minimization algorithm as it was applied in our empirical study. 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 tests from OpenSSH, 96% from vsftpd, and 86% from ngIRCd could be eliminated. This is significant because the initial number of tests was very large: over 14.2M in total for all three programs.

If our testing approach were used in practice, rather than for this empirical study, it would only necessary to test update points that are allowed by the safety

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

Figure A.2: Test success and failure (OpenSSH Full)

	Update	All Pts		CFS		AS		Manual	
		Total	Failed	Total	Failed	Total	Failed	Total	Failed
vsftpd init	0→1	214,035	0	214,035	0	5,751	0	no pts	
	1→2	214,047	0	143,248	0	5,751	0	no pts	
	2→3	220,163	0	220,163	0	5,751	0	no pts	
	3→4	271,574	0	271,574	0	3,294	0	no pts	
	4→5	218,721	4,427	2,518	0	27	0	no pts	
	5→6	223,198	785	5,840	0	27	0	no pts	
	6→7	223,900	1,189	2,519	0	5,751	0	no pts	
	7→8	224,366	2,815	2,492	0	5,751	0	no pts	
	Total	**1,810,004**	**9,216**	**862,389**	**0**	**32,103**	**0**	no pts	
vsftpd mainloop	0→1	1,674	0	1,674	0	1,674	0	54	0
	1→2	1,674	0	1,539	0	1,674	0	54	0
	2→3	1,674	0	1,674	0	1,512	0	54	0
	3→4	1,971	0	1,971	0	1,485	0	54	0
	4→5	1,674	0	1,350	0	108	0	54	0
	5→6	1,674	0	1,350	0	1,620	0	54	0
	6→7	1,674	0	1,350	0	1,674	0	54	0
	7→8	1,674	0	1,350	0	108	0	54	0
	Total	**13,689**	**0**	**12,258**	**0**	**9,855**	**0**	**432**	**0**
vsftpd transition	0→1	85,318	0	85,318	0	67,367	0	no pts	
	1→2	85,349	0	17,437	0	67,609	0	no pts	
	2→3	87,925	0	87,925	0	49,135	0	no pts	
	3→4	88,222	0	88,222	0	23,249	0	no pts	
	4→5	87,431	32,377	1,152	0	6,034	0	no pts	
	5→6	87,589	108	16,216	0	46,834	0	no pts	
	6→7	87,698	81	1,152	0	19,399	0	no pts	
	7→8	87,913	431	966	0	8,825	0	no pts	
	Total	**697,445**	**32,997**	**298,388**	**0**	**288,452**	**0**	no pts	
vsftpd clientloop	0→1	136,883	0	136,883	0	134,649	0	100	0
	1→2	138,913	2,993	36,053	726	111,735	0	100	0
	2→3	160,732	0	160,732	0	123,328	0	101	0
	3→4	145,304	0	145,304	0	63,965	0	103	0
	4→5	179,101	83,118	14,277	1,468	196	0	101	0
	5→6	198,571	0	42,593	0	167,076	0	101	0
	6→7	216,573	0	24,318	0	196	0	101	0
	7→8	235,427	0	202	0	196	0	101	0
	Total	**1,411,504**	**86,111**	**560,362**	**2,194**	**601,341**	**0**	**808**	**0**

Figure A.3: Test success and failure (vsftpd Full)

check in use. The *CFS* and *AS* columns show the number of points allowed under these checks 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 for a patch to any application under a safety check was vsftpd patch 3→4 which required 42,431 tests (all other patches required far fewer). This

	Update	All Pts		CFS		AS		Manual	
		Total	Failed	Total	Failed	Total	Failed	Total	Failed
ngIRCd init	0→1	137,326	0	137,326	0	134,504	0	no pts	
	1→2	137,326	0	137,326	0	132,362	0	no pts	
	2→3	137,326	204	1,632	0	68	0	no pts	
	3→4	137,360	1,086	1,632	0	68	0	no pts	
	4→5	138,278	2,215	1,666	34	68	0	no pts	
	5→6	194,038	0	194,038	0	2,108	0	no pts	
	6→7	194,038	158,916	476	0	68	0	no pts	
	Total	**1,075,692**	**162,421**	**474,096**	**34**	**269,246**	**0**	no pts	
ngIRCd ngircdloop	0→1	154,005	0	154,005	0	18,326	0	372	0
	1→2	152,232	0	148,984	0	35,010	0	370	0
	2→3	152,324	0	375	0	375	0	375	0
	3→4	152,540	0	376	0	376	0	376	0
	4→5	143,406	135,890	321	61	260	0	260	0
	5→6	198,181	3	198,181	3	9,603	0	384	0
	6→7	198,271	10,148	384	0	384	0	384	0
	Total	**1,150,959**	**146,041**	**502,626**	**64**	**64,334**	**0**	**2,521**	**0**

Figure A.4: Test success and failure (ngIRCd Full)

particular patch did not modify any types signatures, so CFS permitted all possible timings.

The manually introduced update points are a small fraction of those in *All*, and we found the minimization 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 minimization as no patch to any program would require more than 870 update tests.

Our minimization algorithm was critical in enabling us to perform our experiments. As we note in Section 2.4, testing of these reduced points for OpenSSH still required approximately 600 CPU hours to complete.

	Update	All Pts	CFS	AS	Manual
OpenSSH	$0 \rightarrow 1$	580,871 $\rightarrow$ 31,791 (95%)	68,044 $\rightarrow$ 3,687 (95%)	35,314 $\rightarrow$ 3,027 (91%)	566 $\rightarrow$ 566 (0%)
	$1 \rightarrow 2$	705,322 $\rightarrow$ 1,795 (~100%)	705,322 $\rightarrow$ 1,795 (~100%)	587,578 $\rightarrow$ 1,717 (~100%)	630 $\rightarrow$ 592 (6%)
	$2 \rightarrow 3$	638,720 $\rightarrow$ 63,011 (90%)	75,307 $\rightarrow$ 5,454 (93%)	20,902 $\rightarrow$ 2,353 (89%)	568 $\rightarrow$ 568 (0%)
	$3 \rightarrow 4$	772,198 $\rightarrow$ 4,324 (99%)	772,198 $\rightarrow$ 4,324 (99%)	638,803 $\rightarrow$ 3,775 (99%)	783 $\rightarrow$ 770 (2%)
	$4 \rightarrow 5$	773,086 $\rightarrow$ 27,399 (96%)	110,633 $\rightarrow$ 4,592 (96%)	21,343 $\rightarrow$ 1,564 (93%)	782 $\rightarrow$ 782 (0%)
	$5 \rightarrow 6$	878,235 $\rightarrow$ 17,398 (98%)	130,000 $\rightarrow$ 1,292 (99%)	111,950 $\rightarrow$ 1,723 (98%)	860 $\rightarrow$ 841 (2%)
	$6 \rightarrow 7$	879,668 $\rightarrow$ 47,092 (95%)	96,183 $\rightarrow$ 4,568 (95%)	44,278 $\rightarrow$ 2,139 (95%)	859 $\rightarrow$ 859 (0%)
	$7 \rightarrow 8$	918,717 $\rightarrow$ 89,601 (90%)	80,070 $\rightarrow$ 3,925 (95%)	100,854 $\rightarrow$ 4,141 (96%)	850 $\rightarrow$ 850 (0%)
	$8 \rightarrow 9$	973,364 $\rightarrow$ 34,293 (96%)	261,885 $\rightarrow$ 5,467 (98%)	61,724 $\rightarrow$ 2,070 (97%)	868 $\rightarrow$ 823 (5%)
	$9 \rightarrow 10$	933,514 $\rightarrow$ 52,356 (94%)	121,337 $\rightarrow$ 3,424 (97%)	61,051 $\rightarrow$ 2,891 (95%)	833 $\rightarrow$ 833 (0%)
	Total	**8,053,695 $\rightarrow$ 369,060 (95%)**	**2,420,979 $\rightarrow$ 38,528 (98%)**	**1,683,797 $\rightarrow$ 25,400 (98%)**	**7,599 $\rightarrow$ 7,484 (2%)**
vsftpd	$0 \rightarrow 1$	437,910 $\rightarrow$ 83 (~100%)	437,910 $\rightarrow$ 83 (~100%)	209,441 $\rightarrow$ 83 (~100%)	154 $\rightarrow$ 28 (82%)
	$1 \rightarrow 2$	439,983 $\rightarrow$ 1,239 (~100%)	198,277 $\rightarrow$ 1,235 (99%)	186,769 $\rightarrow$ 345 (~100%)	154 $\rightarrow$ 127 (18%)
	$2 \rightarrow 3$	470,494 $\rightarrow$ 1,155 (~100%)	470,494 $\rightarrow$ 1,155 (~100%)	179,726 $\rightarrow$ 820 (~100%)	155 $\rightarrow$ 127 (18%)
	$3 \rightarrow 4$	507,071 $\rightarrow$ 42,421 (92%)	507,071 $\rightarrow$ 42,421 (92%)	91,993 $\rightarrow$ 7,030 (92%)	157 $\rightarrow$ 130 (17%)
	$4 \rightarrow 5$	486,927 $\rightarrow$ 41,589 (91%)	19,297 $\rightarrow$ 4,542 (76%)	6,365 $\rightarrow$ 545 (91%)	155 $\rightarrow$ 155 (0%)
	$5 \rightarrow 6$	511,032 $\rightarrow$ 39,478 (92%)	65,999 $\rightarrow$ 9,530 (86%)	215,557 $\rightarrow$ 12,717 (94%)	155 $\rightarrow$ 127 (18%)
	$6 \rightarrow 7$	529,845 $\rightarrow$ 11,619 (98%)	29,339 $\rightarrow$ 1,263 (96%)	27,020 $\rightarrow$ 831 (97%)	155 $\rightarrow$ 128 (17%)
	$7 \rightarrow 8$	549,380 $\rightarrow$ 21,101 (96%)	5,010 $\rightarrow$ 717 (86%)	14,880 $\rightarrow$ 581 (96%)	155 $\rightarrow$ 155 (0%)
	Total	**3,932,642 $\rightarrow$ 158,685 (96%)**	**1,733,397 $\rightarrow$ 60,946 (96%)**	**931,751 $\rightarrow$ 22,952 (98%)**	**1,240 $\rightarrow$ 977 (21%)**
ngIRCd	$0 \rightarrow 1$	291,331 $\rightarrow$ 11,733 (96%)	291,331 $\rightarrow$ 11,733 (96%)	152,830 $\rightarrow$ 2,971 (98%)	372 $\rightarrow$ 337 (9%)
	$1 \rightarrow 2$	289,558 $\rightarrow$ 10,539 (96%)	286,310 $\rightarrow$ 10,539 (96%)	167,372 $\rightarrow$ 2,137 (99%)	370 $\rightarrow$ 335 (9%)
	$2 \rightarrow 3$	289,650 $\rightarrow$ 6,913 (98%)	2,007 $\rightarrow$ 477 (76%)	443 $\rightarrow$ 443 (0%)	375 $\rightarrow$ 375 (0%)
	$3 \rightarrow 4$	289,900 $\rightarrow$ 7,039 (98%)	2,008 $\rightarrow$ 478 (76%)	444 $\rightarrow$ 444 (0%)	376 $\rightarrow$ 376 (0%)
	$4 \rightarrow 5$	281,684 $\rightarrow$ 110,125 (61%)	1,987 $\rightarrow$ 729 (63%)	328 $\rightarrow$ 328 (0%)	260 $\rightarrow$ 260 (0%)
	$5 \rightarrow 6$	392,219 $\rightarrow$ 12,812 (97%)	392,219 $\rightarrow$ 12,812 (97%)	11,711 $\rightarrow$ 1,773 (85%)	384 $\rightarrow$ 384 (0%)
	$6 \rightarrow 7$	392,309 $\rightarrow$ 119,505 (70%)	860 $\rightarrow$ 452 (47%)	452 $\rightarrow$ 452 (0%)	384 $\rightarrow$ 384 (0%)
	Total	**2,226,651 $\rightarrow$ 278,666 (87%)**	**976,722 $\rightarrow$ 37,220 (96%)**	**333,580 $\rightarrow$ 8,548 (97%)**	**2,521 $\rightarrow$ 2,451 (3%)**

Table A.2: Update point minimization

Appendix B

Merging equivalence proof

This appendix presents a formal proof of Theorem 1 from Section 4.1, which states that a configuration $\langle p, \sigma, e \rangle$ updated by patch π is correct if and only if the merged configuration $\langle p, \sigma, e \rangle \rhd \pi$ is correct. Note that the definition of correctness given in the paper specifies only a single patch π to be applied. For our proof, we generalize to correctness over sequences of patches $\vec{\pi}$.

B.1 Overview

The proof is structured in three parts: First, we prove a soundness lemma showing that the merged program simulates every step of execution in the old and new programs, as well as the updating step from old to new. Second, we prove a completeness lemma showing that every execution in the merged program corresponds to an execution in the original program, the new program, or the updated program. Finally, we use these results to prove the main equivalence result.

Before we present these lemmas, we need a little notation. We define three merging transformations—for old-version, new-version, and combined-version code. The first two complete the presentation of merging given in Figure 4.2. Figure B.1 fully defines the transformation $[\![\cdot]\!]^{p,\pi}$ which applies to old-version code, and Figure B.2 fully defines $\{\!|\cdot|\!\}^{p}$, which is used with new-version code. The highlights

$$
\begin{array}{rcl}
[\![x]\!]^{p,\pi} & \triangleq & x \\
[\![a]\!]^{p,\pi} & \triangleq & a \\
[\![g]\!]^{p,\pi} & \triangleq & \begin{cases} g_{ptr} & \text{if } p(g) = \lambda x.e \\ g & \text{otherwise} \end{cases} \\
[\![i]\!]^{p,\pi} & \triangleq & i \\
[\![(v_1, v_2)]\!]^{p,\pi} & \triangleq & ([\![v_1]\!]^{p,\pi}, [\![v_2]\!]^{p,\pi}) \\
[\![()]\!]^{p,\pi} & \triangleq & () \\
[\![v_1 \ op \ v_2]\!]^{p,\pi} & \triangleq & [\![v_1]\!]^{p,\pi} \ op \ [\![v_2]\!]^{p,\pi} \\
[\![v_1(v_2)]\!]^{p,\pi} & \triangleq & [\![v_1]\!]^{p,\pi}([\![v_2]\!]^{p,\pi}) \\
[\![?]\!]^{p,\pi} & \triangleq & ? \\
[\![v_1 := v_2]\!]^{p,\pi} & \triangleq & [\![v_1]\!]^{p,\pi} := [\![v_2]\!]^{p,\pi} \\
[\![!v]\!]^{p,\pi} & \triangleq & ![\![v]\!]^{p,\pi} \\
[\![\mathsf{ref}\ v]\!]^{p,\pi} & \triangleq & \mathsf{ref}\ [\![v]\!]^{p,\pi} \\
[\![\mathsf{if}\ v\ e_1\ e_2]\!]^{p,\pi} & \triangleq & \mathsf{if}\ [\![v]\!]^{p,\pi}\ [\![e_1]\!]^{p,\pi}\ [\![e_2]\!]^{p,\pi} \\
[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!]^{p,\pi} & \triangleq & \mathsf{let}\ x = [\![e_2]\!]^{p,\pi}\ \mathsf{in}\ [\![e_2]\!]^{p,\pi} \\
[\![\mathsf{while}\ e_1\ \mathsf{do}\ e_2]\!]^{p,\pi} & \triangleq & \mathsf{while}\ [\![e_1]\!]^{p,\pi}\ \mathsf{do}\ [\![e_2]\!]^{p,\pi} \\
[\![\mathsf{update}]\!]^{p,(p_\pi, e_\pi)} & \triangleq & \mathsf{let}\ z = isupd()\ \mathsf{in} \\
 & & \mathsf{if}\ z\ 0\ (uflag := ?; \mathsf{let}\ z = isupd()\ \mathsf{in}\ \mathsf{if}\ z\ (\{\!|e_\pi|\!\}^{p'}; 1)\ 0) \\
[\![\mathsf{assume}\ v]\!]^{p,\pi} & \triangleq & \mathsf{assume}\ [\![v]\!]^{p,\pi} \\
[\![\mathsf{assert}\ v]\!]^{p,\pi} & \triangleq & \mathsf{assert}\ [\![v]\!]^{p,\pi} \\
[\![\mathsf{running}\ p'']\!]^{p,(p_\pi, e_\pi)} & \triangleq & \begin{cases} \mathsf{let}\ z = isupd()\ \mathsf{in}\ z = 0 & \text{if } p'' = p \\ \mathsf{let}\ z = isupd()\ \mathsf{in}\ z \neq 0 & \text{if } p'' = p_\pi \\ \mathsf{let}\ z = 0\ \mathsf{in}\ z & \text{otherwise} \end{cases} \\
[\![\mathsf{error}]\!]^{p,\pi} & \triangleq & \mathsf{error}
\end{array}
$$

$$
\begin{array}{rcl}
[\![p, (g, \lambda y.e)]\!]^{p,\pi} & \triangleq & [\![p]\!]^{p,\pi}, \\
 & & (g, \lambda y.[\![e]\!]^{p,\pi}), \\
 & & (g_{ptr}, \lambda y.\mathsf{let}\ z = isupd()\ \mathsf{in}\ \mathsf{if}\ z\ g'(y)\ g(y)) \\
[\![\cdot]\!]^{p,\pi} & \triangleq & (\cdot, (isupd, \lambda y.\mathsf{let}\ z = !uflag\ \mathsf{in}\ z > 0))
\end{array}
$$

Figure B.1: Merging old version code.

$$
\begin{array}{rcl}
\{\!|x|\!\}^p & \triangleq & x \\
\{\!|a|\!\}^p & \triangleq & a \\
\{\!|g|\!\}^p & \triangleq & \begin{cases} g' & \text{if } p(g) = \lambda x.e \\ g & \text{otherwise} \end{cases} \\
\{\!|i|\!\}^p & \triangleq & i \\
\{\!|(v_1, v_2)|\!\}^p & \triangleq & (\{\!|v_1|\!\}^p, \{\!|v_2|\!\}^p) \\
\{\!|()|\!\}^p & \triangleq & () \\
\{\!|v_1 \; op \; v_2|\!\}^p & \triangleq & \{\!|v_1|\!\}^p \; op \; \{\!|v_2|\!\}^p \\
\{\!|v_1(v_2)|\!\}^p & \triangleq & \{\!|v_1|\!\}^p(\{\!|v_2|\!\}^p) \\
\{\!|?|\!\}^p & \triangleq & ? \\
\{\!|v_1 := v_2|\!\}^p & \triangleq & \{\!|v_1|\!\}^p := \{\!|v_2|\!\}^p \\
\{\!|!v|\!\}^p & \triangleq & !\{\!|v|\!\}^p \\
\{\!|\mathsf{ref}\ v|\!\}^p & \triangleq & \mathsf{ref}\ \{\!|v|\!\}^p \\
\{\!|\mathsf{if}\ v\ e_1\ e_2|\!\}^p & \triangleq & \mathsf{if}\ \{\!|v|\!\}^p\ \{\!|e_1|\!\}^p\ \{\!|e_2|\!\}^p \\
\{\!|\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2|\!\}^p & \triangleq & \mathsf{let}\ x = \{\!|e_1|\!\}^p\ \mathsf{in}\ \{\!|e_2|\!\}^p \\
\{\!|\mathsf{while}\ e_1\ \mathsf{do}\ e_2|\!\}^p & \triangleq & \mathsf{while}\ \{\!|e_1|\!\}^p\ \mathsf{do}\ \{\!|e_2|\!\}^p \\
\{\!|\mathsf{update}|\!\}^p & \triangleq & \mathsf{let}\ z = 0\ \mathsf{in}\ z \\
\{\!|\mathsf{assume}\ v|\!\}^p & \triangleq & \mathsf{assume}\ \{\!|v|\!\}^p \\
\{\!|\mathsf{assert}\ v|\!\}^p & \triangleq & \mathsf{assert}\ \{\!|v|\!\}^p \\
\{\!|\mathsf{running}\ p''|\!\}^p & \triangleq & \begin{cases} \mathsf{let}\ z = 1\ \mathsf{in}\ z & \text{if } p = p'' \\ \mathsf{let}\ z = 0\ \mathsf{in}\ z & \text{otherwise} \end{cases} \\
\{\!|\mathsf{error}|\!\}^p & \triangleq & \mathsf{error}
\end{array}
$$

$$
\begin{array}{rcl}
\{\!|p, (g, \lambda y.e)|\!\}^p & \triangleq & \{\!|p|\!\}^p, (g', \lambda y.\{\!|e|\!\}^p) \\
\{\!| \cdot |\!\}^p & \triangleq & \cdot
\end{array}
$$

Figure B.2: Merging new version code.

of these transformations were explained in Section 4.1.3. For technical reasons, we modify the transformations slightly so that non-values (e.g., update) map to non-values (e.g., let $z = 0$ in z instead of 0). The third transformation $(\!| \cdot |\!)^{p,p_\pi}$ combines $[\![\cdot]\!]^{p,\pi}$ and $\{\!|\cdot|\!\}^p$ and returns a *set* of expressions as a result. For example, it translates

$$
\begin{array}{rcl}
(\!|x|\!)^{p,p_\pi} & \triangleq & \{x\} \\
(\!|a|\!)^{p,p_\pi} & \triangleq & \{a\} \\
(\!|g|\!)^{p,p_\pi} & \triangleq & \begin{cases} \{g', g_{ptr}\} & \text{if } p(g) = \lambda x.e \\ \{g\} & \text{otherwise} \end{cases} \\
(\!|i|\!)^{p,p_\pi} & \triangleq & \{i\} \\
(\!|(v_1, v_2)|\!)^{p,p_\pi} & \triangleq & \{(v_1', v_2') \mid v_1' \in (\!|v_1|\!)^{p,p_\pi} \wedge v_2' \in (\!|v_2|\!)^{p,p_\pi}\} \\
(\!|()|\!)^{p,p_\pi} & \triangleq & \{()\} \\
(\!|v_1 \ op \ v_2|\!)^{p,p_\pi} & \triangleq & \{v_1' \ op \ v_2' \mid v_1' \in (\!|v_1|\!)^{p,p_\pi} \wedge v_2' \in (\!|v_2|\!)^{p,p_\pi}\} \\
(\!|v_1(v_2)|\!)^{p,p_\pi} & \triangleq & \{v_1'(v_2') \mid v_1' \in (\!|v_1|\!)^{p,p_\pi} \wedge v_2' \in (\!|v_2|\!)^{p,p_\pi}\} \\
(\!|?|\!)^{p,p_\pi} & \triangleq & \{?\} \\
(\!|v_1 := v_2|\!)^{p,p_\pi} & \triangleq & \{v_1' := v_2' \mid v_1' \in (\!|v_1|\!)^{p,p_\pi} \wedge v_2' \in (\!|v_2|\!)^{p,p_\pi}\} \\
(\!|!v|\!)^{p,p_\pi} & \triangleq & \{!v' \mid v' \in (\!|v|\!)^{p,p_\pi}\} \\
(\!|\mathsf{ref}\ v|\!)^{p,p_\pi} & \triangleq & \{\mathsf{ref}\ v' \mid v' \in (\!|v|\!)^{p,p_\pi}\} \\
(\!|\mathsf{if}\ v\ e_1\ e_2|\!)^{p,p_\pi} & \triangleq & \{\mathsf{if}\ v'\ e_1'\ e_2' \mid v \in (\!|v|\!)^{p,p_\pi} \wedge e_1' \in (\!|e_1|\!)^{p,p_\pi} \wedge e_2' \in (\!|e_2|\!)^{p,p_\pi}\} \\
(\!|\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2|\!)^{p,p_\pi} & \triangleq & \{\mathsf{let}\ x = e_1'\ \mathsf{in}\ e_2' \mid e_1' \in (\!|e_1|\!)^{p,p_\pi} \wedge e_2' \in (\!|e_2|\!)^{p,p_\pi}\} \\
(\!|\mathsf{while}\ e_1\ \mathsf{do}\ e_2|\!)^{p,p_\pi} & \triangleq & \{\mathsf{while}\ e_1'\ \mathsf{do}\ e_2' \mid e_1' \in (\!|e_1|\!)^{p,p_\pi} \wedge e_2' \in (\!|e_2|\!)^{p,p_\pi}\} \\
(\!|\mathsf{update}|\!)^{p,p_\pi} & \triangleq & \{\mathsf{let}\ z = 0\ \mathsf{in}\ z\} \\
(\!|\mathsf{assume}\ v|\!)^{p,p_\pi} & \triangleq & \{\mathsf{assume}\ v' \mid v' \in (\!|v|\!)^{p,p_\pi}\} \\
(\!|\mathsf{assert}\ v|\!)^{p,p_\pi} & \triangleq & \{\mathsf{assert}\ v' \mid v' \in (\!|v|\!)^{p,p_\pi}\} \\
(\!|\mathsf{running}\ p''|\!)^{p,p_\pi} & \triangleq & \begin{cases} \{\mathsf{let}\ z = 0\ \mathsf{in}\ z, \mathsf{let}\ z = isupd()\ \mathsf{in}\ z = 0\} & \text{if } p'' = p \\ \{\mathsf{let}\ z = 1\ \mathsf{in}\ z, isupd()\} & \text{if } p'' = p_\pi \\ \{\mathsf{let}\ z = 0\ \mathsf{in}\ z\} & \text{otherwise} \end{cases} \\
(\!|\mathsf{error}|\!)^{p,p_\pi} & \triangleq & \{\mathsf{error}\}
\end{array}
$$

Figure B.3: Merging combined version code.

function pointers g to $\{g', g_{ptr}\}$. The $(\!| \cdot |\!)^{\cdot,\cdot}$ transformation is needed because after the simulated update takes place, function pointers f may either bind to old/new versions f_{ptr} or to new versions f'.

Next, using these transformations on expressions, we define a transformation

on configurations:

$$\langle p;\sigma;e\rangle \ \triangleright\ \pi \ \triangleq\ \langle \overline{p}, \overline{\sigma}[\mathit{uflag} \mapsto i], \overline{e}\rangle$$

$$\langle p;\sigma;e\rangle \ [\triangleright]\ \pi \ \triangleq\ \langle \overline{p}, \hat{\sigma}[\mathit{uflag} \mapsto j], \hat{e}\rangle$$

$$\text{where} \quad \overline{p} = \{\!|p_\pi|\!\}^{p_\pi}, [\![p]\!]^{p,\pi} \qquad \pi = (p_\pi, e_\pi) \qquad i \leq 0$$

$$\overline{e} = [\![e]\!]^{p,\pi} \qquad \hat{e} = (\!|e|\!)^{p,p_\pi} \qquad j > 0$$

$$\overline{\sigma} = \{l \mapsto [\![v]\!]^{p,\pi} \mid \sigma(l) = v\}$$

$$\hat{\sigma} = \{\sigma' \mid dom(\sigma') = dom(\sigma) \wedge \forall l \in dom(\sigma).\ \sigma'(l) \in (\!|\sigma(l)|\!)^{p,p_\pi}\}$$

The $(\cdot \triangleright \cdot)$ and $(\cdot\ [\triangleright]\ \cdot)$ transformations simulate the behavior of the program before and after the update occurs respectively. Note that both transformations describe sets of configurations: $(\cdot \triangleright \cdot)$ contains all configurations of the specified form where *uflag* is bound in the heap to an integer $i \leq 0$ while $(\cdot\ [\triangleright]\ \cdot)$ is a set due to the use of $(\!| \cdot |\!)^{p,p_\pi}$. These sets are needed to set up the simulations between executions in the old, new, and transformed programs. To streamline the presentation we will occasionally abuse notation slightly, lifting various notions from elements to sets in the obvious way. For example, we write $\langle p;\sigma;e\rangle \triangleright \pi \rightsquigarrow \langle p;\sigma';e'\rangle\ [\triangleright]\ \pi$ to indicate that every configuration in $\langle p;\sigma;e\rangle \triangleright \pi$ steps to a configuration in $\langle p;\sigma';e'\rangle\ [\triangleright]\ \pi$.

The first lemma states that any execution in an untransformed program is matched by an execution in the transformed program.

Lemma 1 (Soundness). *For all* $p, p', \sigma, \sigma', e, e', \vec{\nu}, \pi$ *with* $\pi = (p_\pi, e_\pi)$ *we have* $\langle p;\sigma;e\rangle \overset{\vec{\nu}}{\rightsquigarrow}{}^{*} \langle p';\sigma';e'\rangle$ *implies*

1. *if* $\vec{\nu} = \epsilon$ *then* $p' = p$ *and* $\langle p; \sigma; e \rangle \rhd \pi \leadsto^* \langle p; \sigma'; e' \rangle \rhd \pi$

2. *if* $\vec{\nu} = \pi$ *then* $p' = p_\pi$ *and* $\langle p; \sigma; e \rangle \rhd \pi \leadsto^* \langle p; \sigma'; e' \rangle [\rhd] \pi$*.*

The second lemma states that for any execution trace of the transformed program, there is a corresponding trace of the untransformed program. However, the transformed program may need to execute a little more to match up with an untransformed state.

Lemma 2 (Completeness)**.** *For all* $p, p', \sigma, \sigma', e, e', \pi$ *such that* $\pi = (p_\pi, e_\pi)$*, if* $\langle p; \sigma; e \rangle \rhd \pi \leadsto^* \langle p'; \sigma'; e' \rangle$ *then there exist* σ'' *and* e'' *such that*

- $\langle p'; \sigma'; e' \rangle \leadsto^* \langle p; \sigma''; e'' \rangle \rhd \pi$ *and* $\langle p; \sigma; e \rangle \leadsto^* \langle p; \sigma'', e'' \rangle$ *; or*
- $\langle p'; \sigma'; e' \rangle \leadsto^* \langle p; \sigma''; e'' \rangle [\rhd] \pi$ *and* $\langle p; \sigma; e \rangle \overset{\pi}{\leadsto}^* \langle p_\pi; \sigma'', e'' \rangle$ *; or*

Using these lemmas, we prove the main result:

of Theorem 1. Recall the statement of the theorem:

For all p, σ, e, π with $\pi = (p_\pi, e_\pi)$ and $dom(p_\pi) \supseteq dom(p)$ we have $\models \langle p; \sigma; e \rangle, \pi$ *if and only if* $\models \langle p, \sigma, e \rangle \rhd \pi$.

We prove each direction separately.

($\Leftarrow$) Let $\langle p; \sigma; e \rangle \rhd \pi \leadsto^* \langle p'; \sigma'; e' \rangle$ be an execution of the transformed program.

By Lemma 2 there exists a σ'' and e'' such that either:

- $\langle p'; \sigma'; e' \rangle \leadsto^* \langle p; \sigma''; e'' \rangle \rhd \pi$ and $\langle p; \sigma; e \rangle \leadsto^* \langle p; \sigma''; e'' \rangle$. By assumption, we have $\models \langle p; \sigma; e \rangle, \pi$ and hence e'' is not error. Using Lemma 8 we also have that e' is not error.

- $\langle p'; \sigma'; e' \rangle \rightsquigarrow^* \langle p; \sigma''; e'' \rangle\ [\triangleright]\ \pi$ and $\langle p; \sigma; e \rangle \overset{\pi}{\rightsquigarrow}{}^* \langle p'; \sigma''; e'' \rangle$. The result follows by a similar argument as the previous case.

($\Rightarrow$) Let $\langle p; \sigma; e \rangle \overset{\vec{\nu}}{\rightsquigarrow}{}^* \langle p'; \sigma'; e' \rangle$ be an execution. By Lemma 1 we have:

- $\vec{\nu} = \epsilon$ implies $p' = p$ and $\langle p; \sigma; e \rangle \triangleright \pi \rightsquigarrow^* \langle p; \sigma'; e' \rangle \triangleright \pi$. By assumption, we have $\models \langle p; \sigma; e \rangle, \pi$ and hence $[\![e']\!]^{p,\pi}$ is not error. Using Lemma 8 we also have that e' is not error.

- $\vec{\nu} = \pi$ implies $p' = p_\pi$ and $\langle p; \sigma; e \rangle \triangleright \pi \rightsquigarrow^* \langle p; \sigma'; e' \rangle\ [\triangleright]\ \pi$. The result follows by a similar argument as the previous case. $\square$

$\square$

B.2 Soundness Lemmas

The main soundness lemma follows from the three lemmas proved in this section. The first shows that the simulation between the original and transformed programs holds before to an update when taking a single step.

Lemma 3. *For all $p, \sigma, \sigma', e, e', \pi$, if $\langle p; \sigma; e \rangle \rightsquigarrow \langle p; \sigma'; e' \rangle$ then $\langle p; \sigma; e \rangle \triangleright \pi \rightsquigarrow^+ \langle p; \sigma'; e' \rangle \triangleright \pi$.*

Proof. Let $(p_\pi, e_\pi) = \pi$ and define $\overline{p}$, $\overline{\sigma}$, and $\overline{\sigma}'$ as follows:

$$\overline{p} = \{\![p_\pi]\!\}^p, [\![p]\!]^{p,\pi}$$

$$\overline{\sigma} = \{l \mapsto [\![v]\!]^{p,\pi} \mid \sigma(l) = v\}[\mathit{uflag} \mapsto i]$$

$$\overline{\sigma}' = \{l \mapsto [\![v]\!]^{p,\pi} \mid \sigma'(l) = v\}[\mathit{uflag} \mapsto i']$$

where $i \leq 0$ and $i' \leq 0$.

The proof is by induction on $\langle p; \sigma; e\rangle \rightsquigarrow \langle p; \sigma'; e'\rangle$. Most cases are straightforward calculations using the definition of the transformation. We show just a few of the most interesting cases.

Case $\langle p; \sigma; v_1 \; op \; v_2\rangle \rightsquigarrow \langle p; \sigma; v'\rangle$ where $v' = [\![op]\!](v_1, v_2)$:

For this case, we must assume that $[\![op]\!](v_1, v_2) = v'$ implies $[\![op]\!]([\![v_1]\!]^{p,\pi}, [\![v_2]\!]^{p,\pi}) = [\![v']\!]^{p,\pi}$. This rules out operators such as $<$ on function pointers (which makes intuitive sense, because relative ordering on pointers will not be preseved by the transformation in general). We calculate as follows,

$$
\begin{array}{rcll}
\langle p; \sigma; v_1 \; op \; v_2\rangle \rhd \pi & = & \langle \overline{p}; \overline{\sigma}; [\![v_1 \; op \; v_2]\!]^{p,\pi}\rangle & \\
 & = & \langle \overline{p}; \overline{\sigma}; [\![v_1]\!]^{p,\pi} \; op \; [\![v_2]\!]^{p,\pi}\rangle & \\
 & \rightsquigarrow & \langle \overline{p}; \overline{\sigma}; [\![op]\!]([\![v_1]\!]^{p,\pi}, [\![v_2]\!]^{p,\pi})\rangle & \\
 & = & \langle p; \sigma; [\![v']\!]^{p,\pi}\rangle \rhd \pi & \text{by assumption} \\
 & = & \langle p; \sigma; v'\rangle \rhd \pi &
\end{array}
$$

and obtain the required result.

Case: $\langle p; \sigma; \mathsf{update}\rangle \rightsquigarrow \langle p; \sigma; 0\rangle$

We calculate as follows,

$$
\begin{array}{lll}
& \langle p;\sigma;\mathsf{update}\rangle \rhd \pi & \\
= & \langle\overline{p};\overline{\sigma};[\![\mathsf{update}]\!]^{p,\pi}\rangle & \\
= & \langle\overline{p};\overline{\sigma};\mathsf{let}\ z = isupd()\ \mathsf{in} & \\
& \quad \mathsf{if}\ z\ 0\ (uflag := ?; \mathsf{let}\ z = isupd()\ \mathsf{in}\ \mathsf{if}\ z\ (\{\!|e_\pi|\!\}^p;1)\ 0)\rangle & \\
\rightsquigarrow & \langle\overline{p};\overline{\sigma};\mathsf{let}\ z = \mathsf{let}\ z = !uflag\ \mathsf{in}\ z > 0\ \mathsf{in} & \\
& \quad \mathsf{if}\ z\ 0\ (uflag := ?; \mathsf{let}\ z = isupd()\ \mathsf{in}\ \mathsf{if}\ z\ (\{\!|e_\pi|\!\}^p;1)\ 0)\rangle & \\
\rightsquigarrow^+ & \langle\overline{p};\overline{\sigma};(uflag := ?; \mathsf{let}\ z = isupd()\ \mathsf{in}\ \mathsf{if}\ z\ (\{\!|e_\pi|\!\}^p;1)\ 0)\rangle & \text{as } \overline{\sigma}(uflag) = i \leq 0 \\
\rightsquigarrow^+ & \langle\overline{p};\overline{\sigma}[uflag \mapsto i'];0\rangle & \text{where } i' \leq 0 \\
= & \langle\overline{p};\overline{\sigma}[uflag \mapsto i'];[\![0]\!]^{p,\pi}\rangle & \\
= & \langle p;\sigma;0\rangle \rhd \pi &
\end{array}
$$

and obtain the required result.

Case: $\langle p;\sigma;\mathsf{update}\rangle \overset{\pi}{\rightsquigarrow} \langle p';\sigma;(e;1)\rangle$ where $\pi = (p', e)$

Can't happen, as $\vec{\nu} = \epsilon$ by assumption.

Case: $\langle p;\sigma;\mathsf{running}\ p\rangle \rightsquigarrow \langle p;\sigma;1\rangle$

We calculate as follows,

$$
\begin{array}{rcll}
\langle p; \sigma; \mathsf{running}\ p\rangle \ \rhd\ \pi & = & \langle \overline{p}; \overline{\sigma}; [\![\mathsf{running}\ p]\!]^{p,\pi}\rangle & \\
& = & \langle \overline{p}; \overline{\sigma}; \mathsf{let}\ z = isupd()\ \mathsf{in}\ z = 0\rangle & \\
& \rightsquigarrow & \langle \overline{p}; \overline{\sigma}; \mathsf{let}\ z = \mathsf{let}\ z = !uflag\ \mathsf{in}\ z > 0\ \mathsf{in}\ z = 0\rangle & \\
& \rightsquigarrow^{+} & \langle \overline{p}; \overline{\sigma}; 1\rangle & \text{as } \overline{\sigma}(uflag) = i \leq 0 \\
& = & \langle \overline{p}; \overline{\sigma}; [\![1]\!]^{p,\pi}\rangle & \\
& = & \langle p; \sigma; 1\rangle \ \rhd\ \pi &
\end{array}
$$

and obtain the required result.

□

The next lemma proves the simulation is also preserved by updates.

Lemma 4. *For all* $p, p', \sigma, \sigma', e, e', \pi$ *with* $\pi = (p_\pi, e_\pi)$ *we have* $\langle p; \sigma; e\rangle \overset{\pi}{\rightsquigarrow} \langle p'; \sigma'; e'\rangle$ *implies* $p' = p_\pi$ *and* $\langle p; \sigma; e\rangle \ \rhd\ \pi \rightsquigarrow^{+} \langle p; \sigma'; e'\rangle\ [\rhd]\ \pi$.

Proof. Let $(p_\pi, e_\pi) = \pi$ and define $\overline{p}$, $\overline{\sigma}$, and $\hat{\sigma}'$ as follows.

$$
\begin{array}{rcl}
\overline{p} & = & \{\!|p_\pi|\!\}^{p}, [\![p]\!]^{p,\pi} \\
\overline{\sigma} & = & \{l \mapsto [\![v]\!]^{p,\pi} \mid \sigma(l) = v\}[uflag \mapsto i] \\
\hat{\sigma}' & = & \{\sigma'[uflag \mapsto j] \mid dom(\sigma') = dom(\sigma) \wedge \forall l \in dom(\sigma).\ \sigma'(l) \in (\!|\sigma(l)|\!)^{p,p_\pi}\}
\end{array}
$$

where $i \leq 0$ and $j > 0$.

The proof is by induction on $\langle p; \sigma; e\rangle \overset{\pi}{\rightsquigarrow} \langle p'; \sigma'; e'\rangle$. We show just one case:

Case: $\langle p; \sigma; \mathsf{update}\rangle \overset{\pi}{\rightsquigarrow} \langle p_\pi; \sigma; (e_\pi; 1)\rangle$

We calculate as follows,

$$
\begin{array}{ll}
& \langle p; \sigma; \mathsf{update} \rangle \ \triangleright\ \pi \\
= & \langle \overline{p}; \overline{\sigma}; \mathsf{let}\ z = \mathit{isupd}()\ \mathsf{in} \\
& \quad \mathsf{if}\ z\ 0\ (\mathit{uflag} := ?; \mathsf{let}\ z = \mathit{isupd}()\ \mathsf{in}\ \mathsf{if}\ z\ (\{\!|e_\pi|\!\}^p; 1)\ 0)\rangle \\
\rightsquigarrow^+ & \langle \overline{p}; \overline{\sigma}; \mathsf{let}\ z = 0\ \mathsf{in} \qquad \text{as } \overline{\sigma}(\mathit{uflag}) = i \leq 0 \\
& \quad \mathsf{if}\ z\ 0\ (\mathit{uflag} := ?; \mathsf{let}\ z = \mathit{isupd}()\ \mathsf{in}\ \mathsf{if}\ z\ (\{\!|e_\pi|\!\}^p; 1)\ 0)\rangle \\
\rightsquigarrow^+ & \langle \overline{p}; \overline{\sigma}[\mathit{uflag} \mapsto j]; \mathsf{let}\ z = \mathit{isupd}()\ \mathsf{in}\ \mathsf{if}\ z\ (\{\!|e_\pi|\!\}^p; 1)\ 0\rangle \qquad \text{where } j > 0 \\
\rightsquigarrow^+ & \langle \overline{p}; \overline{\sigma}[\mathit{uflag} \mapsto j]; (\{\!|e_\pi|\!\}^p; 1)\rangle \\
\in & \langle \overline{p}; \hat{\sigma}[\mathit{uflag} \mapsto j]; (\!|e_\pi; 1|\!)^{p,p_\pi}\rangle \\
\subseteq & \langle p; \sigma; (e_\pi; 1)\rangle\ [\triangleright]\ \pi
\end{array}
$$

and obtain the required result.

□

The third lemma proves that the simulation is preserved following an update.

Lemma 5. *For all $p, \sigma, \sigma', e, e', \pi$ with $\pi = (p_\pi, e_\pi)$ we have $\langle p_\pi; \sigma; e\rangle \rightsquigarrow \langle p_\pi; \sigma'; e'\rangle$ implies $\langle p; \sigma; e\rangle\ [\triangleright]\ \pi \rightsquigarrow^+ \langle p; \sigma'; e'\rangle\ [\triangleright]\ \pi$.*

Proof. Similar to the previous soundness lemmas. □

B.3 Completeness Lemmas

The main completeness lemma follows from repeated applications of the next two lemmas.

Lemma 6. *For all* $p, p', \sigma, \sigma', e, e', \pi$ *where* $\pi = (p_\pi, e_\pi)$, *if*

$$\langle p; \sigma; e\rangle \rhd \pi \ \leadsto^+ \ \langle p'; \sigma'; e'\rangle$$

and there does not exist σ_0 *and* e_0 *such that either*

$$\langle p; \sigma; e\rangle \leadsto \langle p; \sigma_0; e_0\rangle \quad and \quad \langle p; \sigma; e\rangle \rhd \pi \leadsto^+ \langle p; \sigma_0; e_0\rangle \rhd \pi \leadsto^+ \langle p'; \sigma'; e'\rangle$$

or

$$\langle p; \sigma; e\rangle \overset{\pi}{\leadsto} \langle p'; \sigma_0; e_0\rangle \quad and \quad \langle p; \sigma; e\rangle \rhd \pi \leadsto^+ \langle p; \sigma_0; e_0\rangle \ [\rhd] \ \pi \leadsto^+ \langle p'; \sigma'; e'\rangle$$

then there exists σ'' *and* e'' *such that either*

- $\langle p'; \sigma''; e''\rangle = \langle p; \sigma'; e'\rangle \rhd \pi$ *and* $\langle p; \sigma; e\rangle \leadsto \langle p; \sigma''; e''\rangle$ *; or*
- $\langle p'; \sigma''; e''\rangle \in \langle p; \sigma'; e'\rangle \ [\rhd] \ \pi$ *and* $\langle p; \sigma; e\rangle \overset{\pi}{\leadsto} \langle p_\pi; \sigma''; e''\rangle$*; or*
- $\langle p'; \sigma'; e'\rangle \overset{\nu}{\leadsto} \langle p'; \sigma''; e''\rangle$ *for some* ν*.*

Intuitively, this lemma states that if the transformed program can take some number of steps, then either that state corresponds to a reachable untransformed state, or can take another step, eventually reaching a corresponding state.

The second lemma is similar, but considers post-update states:

Lemma 7. *For all* $p, p', \sigma, \sigma', e, e', \pi$ *where* $\pi = (p_\pi, e_\pi)$, *if*

$$\langle p; \sigma; e\rangle \ [\rhd] \ \pi \ \leadsto^+ \ \langle p'; \sigma'; e'\rangle$$

and there do not exist σ_0 *and* e_0 *such that*

$$\langle p_\pi; \sigma; e\rangle \rightsquigarrow \langle p_\pi; \sigma_0; e_0\rangle \quad \textit{and} \quad \langle p; \sigma; e\rangle\ [\triangleright]\ \pi \rightsquigarrow^+ \langle p; \sigma_0; e_0\rangle\ [\triangleright]\ \pi \rightsquigarrow^+ \langle p'; \sigma'; e'\rangle$$

Then there exist σ'' *and* e'' *such that either*

- $\langle p'; \sigma'; e'\rangle \in \langle p; \sigma''; e''\rangle\ [\triangleright]\ \pi$ *and* $\langle p_\pi; \sigma; e\rangle \rightsquigarrow \langle p_\pi; \sigma''; e''\rangle$ *; or*
- $\langle p'; \sigma'; e'\rangle \rightsquigarrow \langle p'; \sigma''; e''\rangle$.

B.4 Auxiliary Lemmas

Lemma 8 (Error)**.** *For all* p, π, e*, we have* $e \neq \mathsf{error}$ *if and only if:*

- $[\![e]\!]^{p,\pi} \neq \mathsf{error}$*;*
- $\{\!|e|\!\}^{p} \neq \mathsf{error}$*; and*
- $(\!|e|\!)^{p,\pi} \not\ni \mathsf{error}$*.*

Lemma 9 (Non-Zero)**.** *For all* p, π, v*, we have* $v \neq 0$ *if and only if:*

1. $[\![v]\!]^{p,\pi} \neq 0$*;*
2. $\{\!|v|\!\}^{p} \neq 0$*; and*
3. $(\!|v|\!)^{p,p_\pi} \not\ni 0$*.*

Lemma 10 (Substitution)**.** *For all* p*,* p'*,* π*,* x*,* v*, and* e *we have the following:*

- $[\![e[v/x]]\!]^{p,\pi} = [\![e]\!]^{p,\pi}[[\![v]\!]^{p,\pi}/x]$*;*

- $\{\!|e[v/x]|\!\}^p = \{\!|e|\!\}^p[\{\!|v|\!\}^p/x]$; *and*

- $(\!|e[v/x]|\!)^{p,p_\pi} = (\!|e|\!)^{p,p_\pi}[(\!|v|\!)^{p,p_\pi}/x]$.

Bibliography

[1] The python language. `http://www.python.org/`.

[2] The ruby language. `http://www.ruby-lang.org/en/`.

[3] Upgrading weblogic application environments. `http://docs.oracle.com/cd/E12840_01/wls/docs103/pdf/upgrade.pdf`, July 2008.

[4] Sameer Ajmani, Barbara Liskov, and Liuba Shrira. Modular software upgrades for distributed systems. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, July 2006.

[5] Gautam Altekar, Ilya Bagrak, Paul Burstein, and Andrew Schultz. OPUS: Online patches and updates for security. In *USENIX Security Symposium*, 2005.

[6] Rajeev Alur, Robert K. Brayton, Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. Partial-order reduction in symbolic state space exploration. In *Proceedings of the International Conference on Computer Aided Verification*, 1997.

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

[8] Jeff Arnold and M. Frans Kaashoek. Ksplice: automatic rebootless kernel updates. In *Proceedings of the European Conference on Computer Systems (EuroSys*, 2009.

[9] Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the ACM SIGPLAN International Conference on Virtual Execution Environments (VEE)*, 2006.

[10] T. Bloom and M. Day. Reconfiguration and module replacement in Argus: theory and practice. *Software Engineering Journal*, 8(2):102–108, 1993.

[11] Gilad Bracha. Objects as software services. `http://bracha.org/objectsAsSoftwareServices.pdf`, August 2006.

[12] Computer Associates (CA). The avoidable cost of downtime (phase 1). `http://www.arcserve.com/us/%20~/media/files/supportingpieces/arcserve/avoidable-cost-of-downtime-summary.pdf`, November 2010.

[13] Computer Associates (CA). The avoidable cost of downtime (phase 2). `http://www.arcserve.com/us/lpg/%20~/media/Files/SupportingPieces/ARCserve/avoidable-cost-of-downtime-summary-phase-2.pdf`, May 2011.

[14] Cassandra API overview. `http://wiki.apache.org/cassandra/API`.

[15] Nathaniel Charlton, Ben Horsfall, and Bernhard Reus. Formal reasoning about runtime code update. In *Hot Topics in Software Upgrades (HotSWUp)*, 2011.

[16] Haibo Chen, Jie Yu, Rong Chen, Binyu Zang, and Pen-Chung Yew. Polus: A powerful live updating system. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 271–281, 2007.

[17] Haibo Chen, Jie Yu, Chengqun Hang, Binyu Zang, and Pen-Chung Yew. Dynamic software updating using a relaxed consistency model. *Proceedings of the IEEE Transactions on Software Engineering*, 37(5), September 2011.

[18] The 2009 U.S. digital year in review. `http://www.comscore.com/Press_Events/Presentations_Whitepapers/2010/The_2009_U.S._Digital_Year_in_Review`, May 2010.

[19] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. Dependent types for low-level programming. In *Proceedings of the European Symposium on Programming*, pages 520–535, 2007.

[20] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[21] M. Dmitriev. Towards flexible and safe technology for runtime evolution of java language applications. In *Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution, in association with OOPSLA 2001*, October 2001.

[22] Dominic Duggan. Type-based hot swapping of running modules. In *Proceedings of the The ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2001.

[23] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *OSDI*, 2002.

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

[25] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 1993.

[26] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay debugging for distributed applications. In *Proceedings of the USENIX Annual Technical Conference*, 2006.

[27] S. Gilmore, D. Kirli, and C. Walton. Dynamic ML without dynamic types. Technical Report ECS-LFCS-97-378, LFCS, University of Edinburgh, 1997.

[28] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans, and Kaashoek Zheng Zhang. R2: An application-level kernel for record and replay. In *OSDI*, 2008.

[29] Deepak Gupta, Pankaj Jalote, and Gautam Barua. A formal framework for on-line software version change. *IEEE TSE*, 22(2), 1996.

[30] Christopher M. Hayden, Eric A. Hardisty, Michael Hicks, and Jeffrey S. Foster. Efficient Systematic Testing for Dynamically Updatable Software. In *Proceedings of the Hot Topics in Software Upgrades (HotSWUp)*, 2009.

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

[32] Christopher M. Hayden, Stephen Magill, Michael Hicks, Nate Foster, and Jeffrey S. Foster. Specifying and verifying the correctness of dynamic software updates. In *Proceedings of the Verified Software: Theories, Tools and Experiments*, January 2012.

[33] Christopher M. Hayden, Karla Saur, Michael Hicks, and Jeffrey S. Foster. A study of dynamic software update quiescence in multi-threaded programs, 2012.

[34] Christopher M. Hayden, Edward K. Smith, Eric A. Hardisty, Michael Hicks, and Jeffrey S. Foster. Evaluating dynamic software update safety using efficient systematic testing. *Proceedings of the IEEE Transactions on Software Engineering*, 99(PrePrints), September 2011.

[35] Christopher M. Hayden, Edward K. Smith, Michael Hicks, and Jeffrey S. Foster. State Transfer for Clear and Efficient Runtime Updates. In *Proceedings of the Hot Topics in Software Upgrades (HotSWUp)*, 2011.

[36] Michael Hicks and Scott Nettles. Dynamic software updating. *Proceedings of the ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(6), 2005.

[37] Java platform debugger architecture. This supports class replacement. See `http://java.sun.com/j2se/1.4.2/docs/guide/jpda/`.

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

[39] Leo King. Nasdaq out of date software helped hackers - report. *Computerworld UK*, November 2011.

[40] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. *Proceedings of the IEEE Transactions on Software Engineering*, 16(11), 1990.

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

[42] S. Magill, M.-H. Tsai, P. Lee, and Y.-K. Tsay. THOR: A tool for reasoning about shape and arithmetic. In *CAV*, LNCS 5123, pages 428–432. Springer, 2008.

[43] Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. Automatic numeric abstractions for heap-manipulating programs. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, 2010.

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

[45] Kristis Makris and Rida Bazzi. Immediate Multi-Threaded Dynamic Software Updates Using Stack Reconstruction. In *USENIX Annual Technical Conference*, 2009.

[46] Kristis Makris and Kyung Dong Ryu. Dynamic and Adaptive Updates of Non-Quiescent Subsystems in Commodity Operating System Kernels. In *European Conference on Computer Systems (EuroSys*, 2007.

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

[48] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceedings of the ACM Conference on Architectural support for programming languages and operating systems*, pages 265–276, 2009.

[49] Iulian Neamtiu and Michael Hicks. Safe and timely dynamic updates for multi-threaded programs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[50] Iulian Neamtiu, Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Contextual effects for version-consistent dynamic software updating and safe con-

current programming. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, 2008.

[51] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical dynamic software updating for C. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[52] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical dynamic software updating for C. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[53] Dave Patterson. A simple way to estimate the cost of downtime. `http://roc.cs.berkeley.edu/talks/pdf/LISA.pdf`, November 2002.

[54] William Pugh and Nathaniel Ayewah. Unit testing concurrent software. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2007.

[55] Shaz Qadeer and Dinghao Wu. KISS: Leep it simple and sequential. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2004.

[56] Redis - project hosting on Google Code. `http://code.google.com/p/redis/`.

[57] Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S. Foster, and Adam Porter. Using symbolic evaluation to understand behavior in configurable software systems. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2010.

[58] Eric Roman. A survey of checkpoint/restart implementations. Technical report, Lawrence Berkeley National Laboratory, Tech, 2002.

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

[60] Richard W. Stevens and Stephen A. Rago. *Advanced Programming in the UNIX(R) Environment (2nd Edition)*. Addison-Wesley Professional, 2005.

[61] Don Stewart and Manuel M. T. Chakravarty. Dynamic applications from the ground up. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 27–38, New York, NY, USA, 2005. ACM Press.

[62] stopbadware.org. Compromised websites: an owner's perspective. `http://www.stopbadware.org/pdfs/compromised-websites-an-owners-perspective.pdf`, February 2012.

[63] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. *Mutatis Mutandis*: Safe and flexible dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 29(4), 2007.

[64] Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. Dynamic Software Updates: A VM-centric Approach. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[65] Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. Dynamic software updates for Java: A VM-centric approach. In *Proceedings of the ACM*

Conference on Programming Language Design and Implementation (PLDI), 2009.

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

[67] David A. Wheeler. Sloccount. `http://www.dwheeler.com/sloccount/`.

[68] Min Xu, Vyacheslav Malyugin, Jeffrey Sheldon, Ganesh Venkitachalam, Boris Weissman, and Vmware Inc. Retrace: Collecting execution trace with virtual machine deterministic replay. In *MoBS*, 2007.