A

Kitsune: Efficient, General-purpose Dynamic Software Updating for C

Christopher M. Hayden, University of Maryland, College Park
Karla Saur, University of Maryland, College Park
Edward K. Smith, University of Maryland, College Park
Michael Hicks, University of Maryland, College Park
Jeffrey S. Foster, University of Maryland, College Park

Dynamic software updating (DSU) systems facilitate software updates to running programs, thereby permitting developers to add features and fix bugs without downtime. This paper introduces Kitsune, a DSU system for C. Kitsune's design has three notable features. First, Kitsune updates the whole program, rather than individual functions, using a mechanism that places no restrictions on data representations or allowed compiler optimizations. Second, Kitsune makes the important aspects of updating explicit in the program text, making the program's semantics easy to understand while minimizing programmer effort. Finally, the programmer can write simple specifications to direct Kitsune to generate code that traverses and transforms old-version state for use by new code; such state transformation is often necessary and is significantly more difficult in prior DSU systems. We have used Kitsune to update six popular, open-source, single- and multi-threaded programs, and find that few program changes are required to use Kitsune, that it incurs essentially no performance overhead, and that update times are fast.

Categories and Subject Descriptors: C.4 [**Performance of Systems**]: Reliability, availability, and serviceability

General Terms: Design, Languages

Additional Key Words and Phrases: dynamic software updating

ACM Reference Format:
Christopher M. Hayden, Karla Saur, Edward K. Smith, Michael Hicks, and Jeffrey S. Foster. 2014. Kitsune: Efficient, General-purpose Dynamic Software Updating for C *ACM T Progr Lang Sys* V, N, Article A (January YYYY), 38 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Running software systems without incurring downtime is very important in today's 24/7 world. *Dynamic software updating* (DSU) services can update programs with new code—to fix bugs or add features—without shutting them down. The research community has shown that general-purpose DSU is feasible: systems that support dynamic upgrades to running C, C++, and Java programs have been applied to dozens of realistic applications, tracking changes according to those applications' release histories [Altekar et al. 2005; Chen et al. 2011; Hayden et al. 2011; Hicks and Nettles 2005; Makris and Bazzi 2009; Makris and Ryu 2007; Neamtiu and Hicks 2009; Neamtiu et al. 2006;

This research was supported in part by NSF grant CCF-0910530 and the partnership between UMIACS and the Laboratory for Telecommunication Sciences.
Author's addresses: Dept. of Computer Science University of Maryland A.V. Williams Building College Park, MD 20742
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© YYYY ACM 1539-9087/YYYY/01-ARTA $15.00
DOI:http://dx.doi.org/10.1145/0000000.0000000

Subramanian et al. 2009; Pina and Hicks 2013; Giuffrida et al. 2013; Payer and Gross 2013; Payer et al. 2013]. Concurrently, industry has begun to package DSU support into commercial products [Arnold and Kaashoek 2009; LiveRebel 2013].

The strength of DSU is its ability to preserve program state during an update. For example, servers for databases, media, FTP, SSH, and routing can maintain client connections for unbounded time periods. DSU can allow those active connections to immediately benefit from important program updates (e.g., security fixes), whereas traditional updating strategies like rolling upgrades cannot. Servers may also maintain significant in-memory state; examples include memcached (a caching server) and redis (a key-value server). DSU techniques can maintain this in-memory state across the update, whereas traditional upgrade techniques will lose it (memcached) or must rely on an expensive disk reload that degrades performance (redis). This problem is acute enough that Facebook uses a custom version of memcached that keeps in-memory state in a ramdisk to which it can reconnect on restart [Nishtala et al. 2013]. However, Facebook's approach permits no changes to the memory representation and may be potentially unsafe if there are significant program changes.

We are interested in supporting general-purpose DSU for single- and multi-threaded C applications. While progress made by existing DSU systems is promising, a truly practical system must be in harmony with the main reasons developers use C: control over low-level data representations; explicit resource management; legacy code; and, perhaps above all, performance. In this paper we present Kitsune, a DSU system for C that is the first to satisfy these motivations while supporting general-purpose dynamic updates in a programmer-friendly manner.

Approach. Kitsune operates in harmony with C thanks to three key design and implementation choices. First, 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). When a Kitsune program is launched, the runtime starts a driver routine 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.

Second, Kitsune gives the programmer explicit control over the updating process, which is reflected as three kinds of additions to the original 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*, where old-version data is assigned to new-version variables (possibly after *transformation* to accommodate program changes); 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. (Sections 2 and 3 explain Kitsune's use in detail.)

Finally, Kitsune includes a novel tool called *xfgen* that assists the programmer in writing code to migrate and transform old program state to be compatible with a new program version. The input to xfgen is a series of programmer-provided *transformation specifications* ("transformers" for short), 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 state migration, executing transformation wher-

ever needed. The generated code operates 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 using the transformers, taking care to maintain the shape of the heap. The old version's copies of any transformed data structures are freed once the update is complete. To support generating safe traversal code, the programmer may need to add lightweight, Deputy-style annotations [Condit et al. 2007] to some types, e.g., to indicate the length of an array. 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 4 describes xfgen.)

Results. We have implemented Kitsune and used it to update three single-threaded programs—vsftpd, redis, and Tor—and three multi-threaded programs—memcached, icecast, and Snort. 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 57 and 529 LoC total, across all versions of a program. The change count is generally related to the amount/complexity of heap-resident state and its initialization, rather than to overall code size. For example, 134 LoC were changed for 16 KLoC icecast vs. 529 LoC for 215 KLoC Snort. xfgen was also very effective, allowing us to write transformers with similarly small specifications totaling between 27 and 297 lines; the size here depends on the number of data structure changes across the sequence of updates. We thoroughly tested that all programs behave correctly under our updates.

We measured Kitsune's performance overhead on normal execution and found it ranged from -2.2% to +2.35%, which is in the noise on modern systems [Mytkowicz et al. 2009]. This is substantially better than the overhead that Ginseng [Neamtiu et al. 2006; Neamtiu and Hicks 2009] and UpStare [Makris and Bazzi 2009], two other general-purpose DSU systems for C, impose on some of the same programs: we measured this overhead to be as high as 18.4% for the former and 41.6% for the latter. We also measured the time an application must pause while Kitsune performs an update and found it was typically less than 40ms, and less than 400ms in the worst case. This time is substantially faster than the time to save and restore state from disk; for redis we found this could take as long as 40 *seconds* for a 15 GB heap. For some multi-threaded programs we require threads to synchronize before an update can take place; we found that making some minor program changes could dramatically reduce this synchronization time from an indefinite pause (in the worst case) to tens of milliseconds.

Related work. Kitsune's design adopts the best ideas from existing systems while it eschews their shortcomings; a thorough analysis of the design space is given in Section 7. Notably, Kitsune's design drew significant inspiration from UpStare and Ginseng. From the former, Kitsune adopts the notion of whole-program updates, rather than per-function updates, and from the latter it adopts the idea of updating only at explicitly specified update points, rather than at arbitrary positions. Whole-program updating eliminates the need to refactor programs to support certain updates (e.g., "loop extraction" [Neamtiu et al. 2006] to enable updating long-running loops), while update points simplify the task of testing and otherwise reasoning about an update, since far fewer program states need to be considered.

On the other hand, Kitsune specifically rejects other elements of these systems' designs to better balance competing concerns. For example, both UpStare and Ginseng require nontrivial compilers to enable updating—UpStare compiles the entire program specially to enable *stack reconstruction*, a mechanism that reduces the control migration problem to specifying a *stack mapping* at update-time, while Ginseng's compiler

inserts extra levels of indirection, adds "slop" space to **struct** definitions, inserts read-/write barriers to support on-the-fly control and data migration, and performs a static analysis to ensure that these compilation changes will not break the program (e.g., due to tricky uses of typecasts).

Kitsune's design requires the programmer to write slightly more code in the worst case compared to Ginseng and UpStare, but in general, it confers several advantages, including: (1) significantly better performance, since control migration code is localized to program paths that rarely overlap with normal program execution, while Ginseng's and UpStare's compilation changes are pervasive; (2) simplified update understanding, since the programmer can just read the code without having to mentally apply a stack mapping and/or indirection model to it; (3) a simpler implementation, since no special compiler/analyzer is needed; and (4) greater flexibility and scalability, since Kitsune does not require (conservative, slow) whole-program analysis that prohibits certain programming idioms.

In essence, Kitsune makes DSU a first-class *program feature* that is implemented and maintained by the programmer, a task that is made simpler and more manageable thanks to the careful design of the Kitsune run-time library and tool suite. Considered as a whole, we find Kitsune to be the most flexible, efficient, and easy to use (and deploy) DSU system for C developed to date.[1]

Documentation and the source code of both Kitsune and the programs we retrofitted to use it are freely available at http://kitsune-dsu.com/.

2. KITSUNE

A Kitsune application's execution goes through three phases:

Normal execution. When started for the first time, and while no dynamic update is available, the application executes normally.

Update preparation. Once a dynamic update becomes available, the application thread(s) must reach a state in which the update can be safely applied. The programmer will insert calls to the function kitsune_update at program points at which an update is permitted to take effect; such calls are dubbed *update points* [Hicks and Nettles 2005]. When an update is available, the kitsune_update function starts the update process. If the program is single-threaded, the new program is loaded and the next phase, update execution, begins. If the program is multi-threaded, each thread blocks until all reach an update point, and then update execution begins.

Update execution. The threads running the old code have their stacks unwound, the entire new program is loaded, and its main function is called by the main thread. Since main also executes during normal startup, the programmer adds a few Kitsune API calls to direct it to behave differently during an update. This added code will 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 equivalent to the point at which the update took place in the old version, identified by calls to kitsune_update. We call these two activities *data migration* and *control migration*, respectively. Once all threads have reached

[1]This paper is an expanded version of Hayden et al. [2012c], with changes including: (a) consideration of dynamic updates to Snort, a large and complicated application; (b) better handling of multi-threaded programs, incorporating ideas proposed in a workshop paper [Hayden et al. 2012b] to improve update times; (c) additional performance experiments considering large heaps; (d) comparisons to related work published since the original paper; (e) many improvements to the exposition.

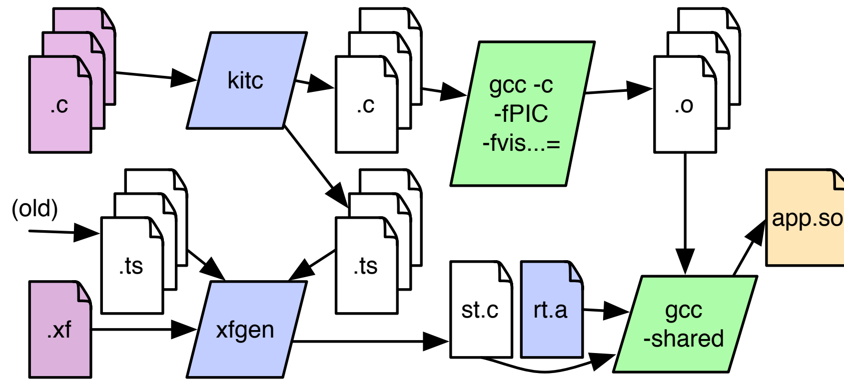

Fig. 1. Kitsune build chain

their update points, the update is complete, resources are freed, and normal execution resumes.

In what follows, we explain how we build programs to implement this semantics (Section 2.1), and how data and control migration are orchestrated by the programmer using the Kitsune API (Section 2.2). For simplicity, we start by assuming we are working with a single-threaded program, and later describe how we handle multi-threaded programs and other language features (Section 3).

2.1. Implementing dynamic updating

The process of building a Kitsune application is illustrated in Figure 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. kitc is built using CIL [Necula et al. 2002]. 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 app.so, a shared object library. (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. During this process, xfgen uses .ts *type summary files* produced by kitc for the old and current versions (described in detail in Section 4.2). We consider xfgen in Section 4.

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,[2] 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; at this point, *normal execution* begins. The

[2]The exact method for signaling that an update is available is left to the discretion of the programmer. However, Kitsune provides a sensible default of SIGUSR2, which works well in most cases. In programs we worked with, only Tor, which had a previously existing control framework, required a different mechanism.

kitsune driver is 120 lines of C code and is the sole part of a program that cannot be dynamically updated.

When SIGUSR2 is received, the handler sets a global flag; this starts the *update preparation* phase. 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 and initialize Kitsune's internal data structures (such as a hashmap used to manage migration), and then the *update execution* phase begins when the driver calls into the program's main function. The programmer will have inserted calls to various Kitsune API functions to perform data and control migration during this phase, which we illustrate in detail next.

2.2. Example

To use Kitsune, the programmer must slightly modify her application to insert update points and add code to perform control and data migration. This subsection uses an example to illustrate what these modifications look like using the Kitsune API.

Consider the C program in Figure 2, which 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 need to perform data and control migration.

Let us ignore the highlighted code for the moment and discuss the program's core operation. Execution begins with main() on line 30. After defining some local variables, the function calls load_config() (code not shown), which initializes config_size defined on line 1 and then allocates an empty mapping. Next, main() calls setup_connection() (code also not shown) to begin listening on main_sock. Finally, main() enters the main loop on lines 41–45. This loop repeatedly waits for a connection and then calls 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 lines 18–19); increments a global counter op_count that tracks the number of requests; and then dispatches to handle_set or handle_get. If the client disconnects, the function exits the loop on line 24 and returns. 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 the developer has added to the program to implement Kitsune control and data migration. This code makes use of several primitive operations that Kitsune provides which we have summarized in Table I and discuss here. We should emphasize that because this example is tiny, the amount of highlighted code is disproportionately large (see Section 5).

Migrating control. A dynamic update is initiated when the program calls kitsune_update(*name*), where *name* distinguishes this *update point* from other update points in the program; we will see why such naming is important shortly. In Figure 2 we have added update points on lines 23 and 42, i.e., we have one update point at the start of each long-running loop. These are good choices for update points because the

```
1  int config_size; /* automigrated */
2  typedef int data;
3  data *mapping; /* automigrated */
4  int op_count=0; /* automigrated */
5
6  void handle_set(int sock) {
7    int key = recv_int(sock);
8    int val = recv_int(sock);
9    mapping[key] = val;
10   send_response(sock, "%d>␣ok", op_count);
11 }
12 void handle_get(int sock) {
13   int key = recv_int(sock);
14   send_response(sock, "%d>␣%d=%d", op_count, key, mapping[key]);
15 }
16 typedef void (*dispatch_fn)(int);
17 struct dispatch_item { char *key; dispatch_fn fun; };
18 struct dispatch_item dispatch_tab[2] __attribute__ ((kitsune_no_automigrate))
19   = { {"get", &handle_get }, {"set", &handle_set } };
20
21 void client_loop (int sock) {
22   while (1) {
23     kitsune_update("client");
24     char *cmd = read_request(sock); if (!cmd) break;
25     dispatch_fn handler = lookup(dispatch_tab, cmd);
26     op_count++;
27     handler(sock); }
28   close(sock);
29 }
30 int main() __attribute__ ((kitsune_note_locals)) {
31   int main_sock, client_sock;
32   kitsune_do_automigrate();
33   if (!kitsune_is_updating()) {
34     load_config(); /* sets config_size */
35     mapping = malloc(config_size * sizeof(data)); }
36   if (!MIGRATE_LOCAL(main_sock))
37     main_sock = setup_connection();
38   if (kitsune_is_updating_from("client")) {
39     MIGRATE_LOCAL(client_sock);
40     client_loop(client_sock); }
41   while (1) {
42     kitsune_update("main");
43     client_sock = get_connection(main_sock);
44     client_loop(client_sock); }
45 }
```

Fig. 2. Example; Kitsune additions highlighted

program is *quiescent*, i.e., in between events, when update-relevant state is not in the middle of being modified [Neamtiu et al. 2006; Hayden et al. 2012d].[3]

The kitsune driver will load the new version and call its main function. During update execution, the program will direct control toward the equivalent update point in the new version, which is to say, one having the same name as the update point that initiated the update in the old version. To do this, it will branch on kitsune_is_updating (), which returns true if the program is being run as a dynamic update, and lets the program distinguish update execution from normal startup. The Kitsune API also includes the variant kitsune_is_updating_from(*name*), which checks whether the update was triggered at the named update point.

In Figure 2, the conditional on line 33 prevents the configuration from being reloaded and mapping from being reallocated when run as an update, since in this case the program will migrate that state from the old version instead (discussed below). If the update was initiated from the client loop, then on line 38 the program migrates client_sock from the previous version and then goes straight to that loop. Notice that when control returns from this call, the program will enter the beginning of the main loop, just as if it had returned from the call on line 44. 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 data. When a program using Kitsune starts update execution, critical data from the previous version of the program remains available in memory. The programmer is responsible for identifying what portion of that data should be migrated to the new version 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 config_size (line 1), mapping (line 3), and op_count (line 4). We use the kitsune_no_automigrate attribute on line 18 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. Only local variables in functions annotated with the kitsune_note_locals attribute are eligible for migration (c.f. main()).

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 register (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 32) starts migration of global state, calling a *migration function* for each registered variable that is automigratable. These functions traverse data structures, transforming them wherever necessary, and are produced by xfgen automatically, when migratable data is unchanged between versions, or else according to programmer specifications. Each function follows a particular naming convention, and the runtime finds it in the new program version using dlsym().

Within a function annotated with kitsune_note_locals, the user calls MIGRATE_LOCAL(var) to migrate (via the appropriate migration function) the

[3]Note that our definition of *quiescent* differs from (and is not comparable to) that of some prior work, which defines it to mean that all updated functions are *inactive*, i.e., not running.

Table I. Kitsune primitives

API call / attribute	**Semantics**	
	Normal Execution	*Update Execution*
kitsune_update(label)	Begins the update process when called, if a dynamic update is available	Marks the completion of an update (so resources can be freed, etc.)
kitsune_is_updating ()	Returns *false*	Returns *true*
kitsune_is_updating_from (label)	Returns *false*	Returns *true* if the update began from an update point with argument label
kitsune_do_automigrate()	Does nothing	Runs migration code to initialize the automigratable global variables
__attribute__ ((kitsune_no_automigrate))	Global variables *without* this attribute are migrated during the call to kitsune_do_automigrate()	
__attribute__ ((kitsune_note_locals))	Local variables in a function with this attribute have their addresses registered when the function is called so they can be migrated should a new update begin before the function returns. The addresses are deregistered when the function returns	
MIGRATE_LOCAL(localvar)	Returns *false*	Invokes migration code to initialize local variable localvar from its old version; returns *true*
MIGRATE_GLOBAL(globalvar)	Returns *false*	Invokes migration code to initialize global variable globalvar with kitsune_no_automigrate attribute; returns *true*
MIGRATE_LOCAL_STATIC (funcname, localvar) MIGRATE_GLOBAL_STATIC (globalvar)	As above, but for static global/local variables	

old version of var to the new version, e.g., as used on line 39 to migrate client_sock. MIGRATE_LOCAL() returns true if the program was started as a dynamic update; on line 36 we test this result to decide whether to initialize main_sock.

Our overall design for data 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 data that should be migrated is reachable from the application's local and global variables. In our experiments, this assumption was true except for memcached, in which pointers to some application data were stored only in a library. We solved this problem by caching such pointers in the main application; see Section 5.2.

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 (cf. the branch on line 40 of Figure 2, which then reaches the update point on line 23). 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 migrations have marked as freeable. Finally, control returns to the new version to begin normal execution.

Checking correctness. To test that code supporting dynamic updates is correct, a programmer can test that the program properly updates to itself (e.g., use the program's normal system test suite, slightly modified, to update the program during a test run). Doing so ensures that all of the control and data migration code works as it should, preserving the state following the update. In our experience, such *self updates* are the most challenging part of retrofitting a program to use Kitsune, and normal program updates take comparatively far less time.

3. ADVANCED FEATURES

This section describes how Kitsune supports programs that make use of multiple threads and dynamically loaded libraries, and discusses Kitsune's current limitations.

3.1. Multi-threading

Updating a multi-threaded program is more challenging since the programmer must migrate control and data for every thread. Using the methodology in the previous section, the programmer could add control migration to the main thread to kill and restart, or reuse, existing threads. However, most of the time the number and role of threads before and after the update is the same (or close to it), and thus Kitsune provides support to ease thread management during an update.

To make a pthreads program Kitsune-enabled, the programmer modifies all thread creation sites to use a wrapper for pthread_create called kitsune_pthread_create.[4] A thread created with kitsune_pthread_create(tid, f, arg) has its thread id tid, thread function f, and f's argument arg atomically added to a global list kitsune_threads of live threads. When a thread exits normally, it removes its entry from kitsune_threads.

When an update to a multi-threaded program is requested, Kitsune sets a flag to indicate the update has been received, as usual. When a thread reaches an update point, if that flag is set then the thread records the name of the update point in its kitsune_threads entry and then blocks. Once all threads have blocked at update points, the system has reached *full quiescence* and is ready to be begin the actual update.

At this point, the main thread starts updating as described in Section 2.2 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 control 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, freeing its code and data as usual, and resumes the main thread and all paused threads.

3.1.1. Requirements for multi-threaded updates.

In order to work with the design just described, Kitsune multi-threaded programs must meet—or be modified to meet—several requirements. First, the program should be insensitive to the order in which the threads are restarted in the new version. We find this often holds because the main thread naturally migrates any shared state, which would otherwise be the main source of contention between threads. Second recreating threads changes their thread IDs, so

[4]We could have automated this step, but wanted to give the programmer the option of orchestrating quiescence manually, e.g., by killing the existing threads and starting new ones, if she so chooses.

the program should not store those IDs in memory. We could extend Kitsune to relax this requirement, but so far have not found it necessary.

Third, to achieve full quiescence quickly, threads should avoid non-interruptible *blocking calls* on paths to update points. For example, a call to kitsune_update may be preceded by a call that reads from a socket. If this call blocks and cannot be interrupted, 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.

3.1.2. Avoiding blocking calls. To ensure full quiescence the programmer must confirm 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 are ever reached by threads holding locks.

For programs we have considered, 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.

Blocking on I/O. Mature server programs, such as the kind we studied, 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     kitsune_update();
5     res = accept(sockfd, addr, addrlen);
6     if (res == -1 && errno == EINTR)
7       continue;
8     /* ... handle connection */
9 } }
```

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.[5] 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 in Kitsune program updates are initiated 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, Kitsune'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.

[5]POSIX supports auto-restarting interrupted, "slow" system calls [Stevens and Rago 2005] (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.

Blocking on Condition Variables. We observe that the threads 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      kitsune_update();
5      pthread_mutex_lock(&mutex);
6      while (! input_is_ready () && !kitsune_update_requested()) {
7        kitsune_pthread_cond_wait(&cond, &mutex);
8      }
9      pthread_mutex_unlock(&mutex);
10     if (kitsune_update_requested())
11       continue; /* reaches kitsune_update */
12     /* ... handle connection */
13 } }
```

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 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 kitsune_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.

Uninterruptible sleep calls are similar to blocking condition variables. To solve this problem, we implemented an interruptible sleep call, kitsune_ms_sleep, that can be easily swapped out for blocking sleep functions. When the program receives the update request signal, the sleep will be interrupted. Similar to the other interruptible calls, programmers should write code to redirect control flow back to an update point.

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, Kitsune launches one additional thread that sleeps during most 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 circumstances cover the vast majority of blocking calls, one of our subject programs, suricata, required custom code to be called from a signal handler to unblock one of its threads (as we describe in Section 6.3). However, POSIX requires that the same signal handler function be used for all threads in a process. Thus, Kitsune provides a library function, kitsune_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. A separate callback function may be set for each thread as needed, or left unset for no callback (in the majority of cases).

3.2. Dynamically loaded shared objects

Recall that to use Kitsune, we must compile the application to a shared library that can be loaded and unloaded. If that application itself loads other shared libraries that change with an update, then we have to update those other shared libraries as well. If the library was statically linked with the host program, it will be reloaded naturally along with the new version of the program; if it was linked dynamically, then the control migration code can be written to call dlclose and dlopen to unload and load shared objects, respectively. If the shared object has state, then it must be compiled with kitc so that state can be registered and updated.

One issue we have encountered with shared objects is that Kitsune uses the names of globals and (registered) locals internal to each shared object. Thus, from Kitsune's perspective, symbol names can conflict across different shared objects (and the main program), even if they are hidden in terms of standard dynamic linking. This situation arose for us when updating Snort, which uses a plug-in architecture for protocol analyzers.

To solve this problem we introduced namespace support in the style of C++ using a directive to the kitc compiler. The programmer can specify a namespace for a file with the file-level command `E_NAMESPACE` *name*, where *name* is a unique string for that namespace. The same namespace is used for all files that build the same shared object. When kitc compiles a file, it prepends the *name* to the front of each symbol in that file as it stores it in the .ts file. This change propagates to how kitc generates and uses symbol registrations, avoiding conflicts.

3.3. Limitations

Kitsune's main limitations at present are a lack of support for custom allocators and pointers to the interior of objects.

Custom allocators. One notable limitation is that Kitsune does not support the use of custom memory allocators. Recall that Kitsune updates the heap by traversing the pointer graph for the program, redirecting pointers to statically-allocated memory, or using the standard malloc and free functions to allocate new objects and free old ones on a per-object basis. However, if an object was allocated by a custom allocator, freeing it with the standard free function will not work, and likewise allocating memory with standard malloc during transformation will fail if the updated program uses a custom free on that memory later.

Supporting custom allocation is certainly possible, e.g., by storing additional metadata to disambiguate pointers. But so far custom allocation has not been an issue for Kitsune, for two reasons. First, in many cases we can simply replace the use of a custom allocator with the standard one, which often improves performance [Berger et al. 2002]. Second, many custom allocators are *type-specific*, i.e., they only allocate objects of some type T. As such, if an update does not change T's type, then none of the memory managed by T's allocator will need to be allocated or freed, and so the update can be performed safely. Snort uses more than a dozen type-specific allocators, but none of the updates we considered changed their types. If such an update occurred, Kitsune could not support it and normal stop/restart would have to be employed.

Interior pointers. Another memory management issue is that Kitsune assumes there are no pointers into the middle of objects. This comes up in two ways. First, our generated code assumes that when it comes across a **struct** foo* that this pointer is to the front of an allocated object, and thus can be freed and reallocated if **struct** foo changed size. Second, pointers into the middle of objects that could be reallocated may dangle, e.g., a **int** *x may point to the field of a particular **struct**, but if that object is freed

INIT *new_var*: {*action*}	$in, $out – old/new type or var
INIT *new_type*: {*action*}	$old/newsym(x) – x in old/new prog.
old_var → *new_var*: {*action*}	$old/newtype(t) – t in old/new prog.
old_type → *new_type*: {*action*}	$base – containing **struct**
old_var → *new_var*	$xform(old, new) – transformer function
old_type → *new_type*	from old to new type/var
(a) transformers	(b) special variables

Fig. 3. xfgen specification language

and reallocated then x will now point to stale memory. We currently address this issue by writing code to manually transform interior pointers. (In all of our test programs across all versions, we only encountered one instance of an interior pointer.)

Once again, we could address these issues by storing more metadata, but metadata management (e.g., in conjunction with calls to malloc/free) may add overhead to normal execution. We do support a debugging mode that maintains an interval tree to keep track of the address ranges of allocated blocks. This allows us Kitsune-generated code to identify pointers into the middle of allocated blocks and issue a warning if this pointer could be affected by a memory management action taken at update time.

4. XFGEN

As mentioned briefly in Section 2.2, Kitsune's runtime invokes migration functions for each automigrating variable, following a naming convention to locate the appropriate migration function. Kitsune includes a tool, xfgen, that produces migration functions automatically for variables and types that have not changed, and generates migration functions for those that have changed according to specifications the programmer expresses in a simple, domain-specific language. We refer to such explicitly specified migration functions as *transformers*, since they are used to transform the data from an old representation to a new one. The design of xfgen is based on our prior experience applying DSU to C [Hayden et al. 2012d; Hayden et al. 2011; Hicks and Nettles 2005; Neamtiu and Hicks 2009; Neamtiu et al. 2006], and aims to make common kinds of state transformers easy to write while maintaining the flexibility to implement arbitrary transformations.

This section presents the xfgen specification language, giving a series of examples, and then describes how xfgen generates migration functions.

4.1. Transformer specifications

Figures 3(a) and (b) summarize xfgen's specification language. Each transformer has one of the forms shown in part (a). The INIT transformers describe how to initialize new variables or values of new types, and the → transformers describe how to transform variables or types that have changed and/or been renamed. Here {*new,old*}*_var* is either a noted local (via kitsune_note_locals from Table I) or global variable name and {*new,old*}*_type* is either a regular C type name or a **struct** field (we will see examples below). The transformer *action* consists of C code that may reference the special xfgen variables shown in Figure 3(b). These variables refer to entities from the old or new program version. A → transformation without an action identifies a variable/type renaming.

Example 1. Suppose we wrote a new version of the program in Figure 2 in which we removed the variable op_count and replaced it with two new variables get_count and set_count that record per-operation counts. These variables will need to be initialized during the update. We do not know exactly how many get and set operations have

occurred, but we do have their sum in op_count, so we might over-approximate each with the sum, as follows:

```
INIT get_count: { $out = $oldsym(op_count); }
INIT set_count: { $out = $oldsym(op_count); }
```

Here we are initializing new variables, so we use an INIT transformer, and the action uses $oldsym(op_count) to refer to the old version's value of op_count and $out to refer to the output of the transformer, i.e., get_count and set_count in the new version. Alternatively, we might wish to preserve the total count, and in lieu of precise information we might assume most calls are gets, and some are sets:

```
INIT get_count: { $out = (int) floor ($oldsym(op_count)*0.9); }
INIT set_count: { $out = (int) ceil ($oldsym(op_count)*0.1); }
```

In general, how a programmer writes a transformer depends on an application's invariants and the desired properties of the updated application's behavior [Hayden et al. 2012a].

Example 2. While transformers are often simple, xfgen is powerful enough to express more complicated changes. For example, suppose we change line 3 in Figure 2 so that, rather than an array, mapping is a linked list:

```
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 new version of mapping (the occurrence to the right of the arrow, referred to as $out within the action) from the old version of mapping (referred to as $in). 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 simply be 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. As we will see shortly, this is possible because we have merely added a (non-pointer) field to the **struct**, so the traversal of the existing data structure accurately recovers the structure of the updated data structure. (We could not use this approach for the previous array-to-list change because the data elements were not related in a simple structural manner.)

Other special variables. In the examples so far, we have seen uses of all but the last two special variables in Figure 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(old,new) names the migration function between types old and new. This variable is useful when defining a transformer for a container data structure, so that an action can recursively call the migration for each contained object. For example, suppose we merged Examples 2 and 3 into a single update that transformed 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 looks up (or forces the creation of) the migration between its argument variables/types. This migration is returned as a closure (a function pointer and an attendant environment) that takes pointers to the old and new object versions. A closure is called using the XF_INVOKE macro.

4.2. Migration generation

xfgen generates code to perform migration. At a high level, the generated code will traverse the heap, starting from the migratable global and local variables in the old version's stack and data segment, and assign the (possibly transformed) data to the corresponding variables in the new version. With xfgen, the programmer is able to focus on defining *what* transformation should be used for data representations that have changed, and is relieved of the tedium of *how* to find all of the old values, preserve the structure of the heap, and manage memory.

xfgen generates migration code based on the contents of the developer-provided .xf specification file and the Kitsune-maintained .ts type summary files for the old and new versions (see Figure 1). 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 2.2). The type summary files of the old and new versions are compared, to identify those data definitions that have changed from the old to the new version. To assist the programmer, xfgen can check that .xf files are complete: an .xf file is

rejected if it fails to define a transformer that applies to migratable data that was added or whose type changed between versions.

xfgen uses type information to generate migration functions for migratable types and variables (or portions thereof) that have not changed—these functions will iterate over the variable/type in question, recursively invoking migration functions on the variable/type's subcomponents. Migratable data includes migrated local and global variables, their types, and types they transitively reference; e.g., **struct** bar is migratable if **struct** foo is migratable and contains a pointer to a **struct** bar. xfgen sometimes needs additional programmer-provided type information to work correctly; e.g., it may need to know the lengths of arrays. We discuss annotations in detail below.

In the remainder of this section, we describe how migration code is generated for variables and types, and how annotations are used to ensure the generated code is complete.

Migrating variables. For each migrated variable listed in the new version's .ts file, if that variable is named explicitly in an $old_var \rightarrow new_var$ specification, then xfgen generates C code from the specified action, substituting references appropriately. For example, $in and $out 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. Thus xfgen will produce the following C code from the overapproximating specifications in Example 1, above:

```
void _kitsune_migrate_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_migrate_set_count() { /* similar to the above */}
```

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. In that case, as we noted previously, the lack of a symbol x in the old version code, or the lack of an explicit transformer if x's type has changed, will cause xfgen to reject the .xf file. Assuming the old version symbol has type *old_type* and the new version has type *new_type*, xfgen's generated function for x will simply call the migration function for $old_type \rightarrow new_type$; if *old_type* and *new_type* have the same definition (and no explicit transformer has been specified) then xfgen will generate C code for this function. We describe this process next.

Migrating types. xfgen generates transformer code from $old_type \rightarrow new_type$ specifications in approximately the same manner as for variables. For specifications involving only a single **struct** field (as for the cid field in Example 3), xfgen will generate code for the rest of the fields in the manner described below, and then insert the hand-written code for the new or changed field.

A generated function for an unchanged type simply recursively invokes the migration functions for the immediate children of the type. For example, suppose we generated a migration for **struct** list in Example 3. Then, xfgen would produce code that retains the old key value by copying it (generated migrations for primitive types are simple assignments); recursively invokes the user-provided transformer for data for the val field; inserts the user-provided code to assign to the cid field; and recursively invokes itself on the target of the next field, assuming it is not NULL.

Pointers must be handled carefully by the generated code. For non-NULL pointers, the generated code checks a global *migration map* to see if the pointer has been mi-

E_PTRARRAY(S) – size of ptd-to array
E_ARRAY(S) – size of array
E_OPAQUE – non-traversed pointer
E_FORALL(@t) – polymorphism intro.
E_VAR(@t) – refer to type var
E_INST(typ) – instantiate poly. type

Fig. 4. xfgen type annotations

grated before; if so it returns the old target. Doing so maintains the shape of the heap and avoids infinite loops when traversing cyclic data structures. Otherwise it calls the appropriate migration for the pointer's target. If the pointer is to a global or local variable, then the address of this variable is different in the new version. As such, the code will look up the corresponding address and redirect to it. If the pointer target's type has truly changed (and so must have an explicit transformer), the generated code mallocs space to store the result and then frees the old-version pointer. This is what happens for next in Example 3—since the **struct** increased in size, new memory is allocated for it and all fields must be copied. As an optimization, since the generated code for an unchanged **struct** reuses the old memory, migrations ignore non-pointer fields, retaining the old values. Once the source and target addresses (which may be the same) have been determined, they are added to the migration map.

xfgen-generated code may uselessly traverse portions of the heap that do not contain changed data. If the programmer knows that a particular data structure contains only pointers into the heap (and not to global or local variables), and that no pointed-to objects require transformation, she can create transformers that truncate the traversal. For example, if field f of **struct** foo (transitively) points only to unchanged heap data, the developer could write

```
struct foo.f → struct foo.f: { $out = $in; }
```

to shallow-copy the field.

Recall from Section 3.3 that we assume there are no pointers into the middle of migratable objects. To help check this assumption, we provide an execution mode in which the xfgen-created migrations use an interval tree to record the start and end of each object they encounter. A migration 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. Supporting pointers to the interior of objects is future work.

Type annotations. xfgen sometimes needs type information beyond what is normally available in C. For example, without further guidance, xfgen would generate an incorrect migration for mapping in Example 1. It would assume that mapping points to a single data element, rather than an array of elements. In Kitsune, extra type information is provided by the programmer as annotations, shown in Figure 4. kitc recognizes these annotations and adds the information supplied by them to the .ts files.

The annotations, inspired by Deputy [Condit et al. 2007], are straightforward. E_PTRARRAY(S) provides a size S for a pointed-to array. For example, we would change line 3 of Figure 2 to

```
data * E_PTRARRAY(config_size) mapping;
```

By default, xfgen assumes that t∗ values for all types t are annotated with E_PTRARRAY(1); explicit annotations override this default. Annotation E_ARRAY(S) provides a size S for array fields at the end of a **struct** (such fields can be left unsized in C). For both of these annotations, S can be an integer constant, a global variable, or a co-located **struct** field. E_OPAQUE annotates pointers that should be copied as val-

Table II. Kitsune benchmark programs

Program	# Vers		LoC
vsftpd	14	(1.1.0–2.0.6)	12,202
redis	5	(2.0.0–2.0.4)	13,387
Tor	13	(0.2.1.18–0.2.1.30)	76,090
memcached*	3	(1.2.2–1.2.4)	4,181
icecast*	5	(2.2.0–2.3.1)	15,759
snort*	4	(2.9.2–2.9.2.3)	214,703

*Multi-threaded

Table III. Kitsune modifications to support updating

Program	Upd	Ctrl	Data	E_*	Oth	Σ	v→v	t→t	Σ	xf LoC
vsftpd	6	26	17+8	6+14	28+8	83+30	9	21	30	101
redis	1	2	3	43	50	99	0	5	5	37
Tor	1	39	37+6	19	57	153+6	16	15	31	189
memchd*	4	9	13	20	66	112	12	10	22	27
icecast*	11+1	22+3	14+9	32+3	41	120+16	25	50	75	200
snort*	2	136+16	118	183+2	54+18	493+36	111	64	175	297

*Multi-threaded

ues, rather than recursed inside during traversals (so we could use this annotation on the foo.f field to truncate traversal in the above example, rather than define a manual transformation).

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 E_VAR(@t) *val;
  struct list E_INST(@t) *next;
} E_FORALL(@t);
```

E_FORALL(@t) introduces polymorphism, E_VAR(@t) refers to type variable @t, and E_INST(@t) instantiates a polymorphic type with type @t. For comparison, this example is equivalent to the following Java generic linked list:

```
class List<T> { // like E_FORALL(@T)
  T val;         // like E_VAR(@T)
  List<T> next; // like E_INST(@T)
}
```

With generics, we can write **struct** list E_INST(**int**) *x to declare that x is a list of **int**s. Generated migration code will invoke the migration function that is appropriate for the list elements' instantiated type.

5. EXPERIENCE AND APPLICATIONS

To evaluate Kitsune, we used it to develop dynamic updates for six widely deployed open-source server programs.[6] This section quantifies the programming effort of applying Kitsune to these applications; Section 6 quantifies Kitsune's performance.

[6]The source code for these programs is freely available at http://kitsune-dsu.com/.

To measure programmer effort, we tallied the number and kinds of changes we made to the programs, and the number and variety of xfgen specifications we wrote for state transformation. Overall, we made few code changes—between 99 and 529 LoC—to support updating, with most changes only to the initial version. Likewise, xfgen specifications were generally small, averaging 3–4 lines per changed variable or type. These numbers are comparable to prior work.

5.1. Applications

We applied Kitsune to programs that maintain in-process state that would be beneficial to preserve during an update. *Vsftpd* is a popular open-source FTP server. *Redis* is a key-value database used by several high-traffic services, including guardian.co.uk and craigslist.org. *Tor* is a popular onion-router that provides anonymous Internet access. *Memcached* is a widely used, high-performance data caching system employed by sites such as Twitter and Facebook. *Icecast* is a popular music streaming server. *Snort* is an open-source network intrusion detection and prevention system with millions of downloads and nearly 400,000 registered users. 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 lost (memcached) or expensive to restore (redis) following an update. Vsftpd also serves as a useful benchmark because several other DSU systems have used it for evaluation [Neamtiu et al. 2006; Makris and Bazzi 2009; Chen et al. 2011; Hayden et al. 2011].

Table II 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. We tested that all programs behaved correctly before, during, and after updates were applied; we say more about the tests we used at the start of Section 6.2.

5.2. 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 actual releases.

Table III summarizes the Kitsune-related source code modifications we made, tabulating the number of update points added (Upd); the number of lines of code needed for control migration and data migration (Ctrl and Data, respectively);[7] the number of type annotations for xfgen (E_*); the number of lines changed for other reasons (Oth); and their sum. Changes from the Oth column are explained in the subsections below for the respective applications. 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 migration and update points were particularly stable, essentially because the top-level control structure of the programs rarely changed. 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. 529 LoC for 215 KLoC Snort. On reflection, this trend makes sense. Changes to support control migration depend on the number and location of update

[7]Specifically, control migration changes comprise calls to kitsune_is_updating and kitsune_is_updating_from. Data migration changes include calls to kitsune_do_automigrate, MIGRATE_GLOBAL, and MIGRATE_LOCAL, and uses of the kitsune_no_automigrate attribute.

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 can be viewed as a stable program feature that is added to the program and maintained (with minimal effort) as the program evolves.

The rightmost columns of Table III list the xfgen specifications we wrote for each program's updates. 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. The median lines of xfgen code was 1, with some larger transformations such as one that switches on a signal value to determine the type of certain data, which affects its transformation.

Overall, we found both the control and data migration code relatively easy to write. Following the structure of the example given in Section 2.2, we added conditionals to avoid re-initializing data we wished to preserve across the update, and to direct control back to the correct update points. State transformation code was also relatively easy to write. Most state required either no or very simple transformation along the lines of Example 1 in Section 4.1. Perhaps the trickiest part was adding type annotations to data structures. While in many cases these annotations were obvious (specifying generic types or bounded arrays) or prompted by xfgen, a missing annotation was sometimes hard to debug. For example, failing to annotate a pointer as an array would result in the generated code not migrating all of the state, ultimately leading to a later crash. Fortunately, since Kitsune uses normal compilation and linking, we could use gdb to debug these problems directly.

Now we consider the particulars of each program, and when possible, compare the magnitude of these changes against those required by prior systems. In general, using Kitsune seems to require more program changes than prior systems, but the total sum of changes is still small.

5.2.1. Vsftpd. Many of the changes we made to vsftpd were typical across our benchmarks: we added type annotations 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, which comprise the connection acceptance loop, two loops to implement privilege-isolated logins, the main FTP command processing loop, and a loop left running in a privileged parent process that implements commands such as chown.

The most interesting change we made to vsftpd was to handle I/O. Instead of calling the recv library routine directly, vsftpd calls a wrapper that calls recv, restarting it if it is interrupted, e.g., by the receipt of a signal. We inserted one update point in the wrapper, just after the recv, so that interruption can initiate an update. To simplify the control-flow changes needed, rather than give the update point its own name, we reused the name of the update point in the loop that initiated the wrapped call; this is safe is because this loop will reinitiate the call when the update completes.

Other DSU systems. Neamtiu et al. [2006] 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 113 LoC changed and 101 lines of state transformation for Kitsune.

Makris and Bazzi [2009] also updated vsftpd using UpStare for a shorter streak. UpStare version 0.12.9 (which we compare against experimentally in Section 6) includes 14 state-mapping files for vsftpd, containing 4,644 lines of C code between them.

While most of this code is automatically generated by UpStare's patch generator, 322 lines are manually added [Makris 2009]. Of these lines, 300 are bookkeeping code, and 22 have deeper semantic meaning. xfgen's design goal is to obviate the need for such bookkeeping code, and the remaining 22 lines would be expressed as xfgen transformation rules.

The presence of these user-defined state mappings effectively reduces the number of update points to two, from 613 in versions that do not require user-defined mappings. This reduction in flexibility is an obvious consequence of requiring manual state mappings, since a programmer could hardly be expected to instrument mapping between states at hundreds of update points, but it removes one of the key advantages of UpStare.

5.2.2. 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 reference global variables (whose addresses change with each updated version). For example, we wrote a 19-line specification to direct the traversal through a **void**$*$ field by consulting an integer key to determine the field's type. Finally, redis uses a custom allocator, which xfgen does not support, so we modified the redis header files with preprocessor directives to redirect custom allocator calls to malloc and free.

Initially, the heap traversal of redis visited every key-value pair in memory. 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 modified redis (42 LoC changed) to store integer indices into a table in place of those pointers. Doing so obviates the need for a full heap traversal for all the updates in our streak, making the update times constant for all tested heap sizes. Programs that use Kitsune may benefit from a similar transformation if they maintain a large amount of state containing static pointers.

We are unaware of prior work applying DSU to redis.

5.2.3. Tor. Tor is one of the largest benchmark programs, at $\sim$76 KLoC. Adding DSU support required one update point in Tor's main loop. The larger number of control flow changes in comparison to other programs is a result of Tor's modular design. Twelve of the thirty three modules' module_init functions required simple two-line changes to prevent re-initialization of updated state. The remaining changes were made along the path from main to Tor's main loop, similarly to our other benchmarks. Most "data" changes were kitsune_no_automigrate attributes, directing Kitsune to skip over various constant strings used in parsing Tor's configuration file. We automigrated most global variables at the start of update execution, but manually migrated Tor's *network consensus* data structure (via a call to MIGRATE_GLOBAL). This update added a field to the network consensus, and the easiest way to update the current state was to use an existing routine in Tor to reload the consensus from disk, rather than duplicating code to read the new field from the on-disk consensus format in a custom transformer. Kitsune's manual migration API allowed us to wait for the rest of Tor's state to be consistent before reloading the consensus using standard Tor APIs.

The rest of our changes (counted above as "other") were primarily to add support to initiate an update via Tor's control interface, rather than a command-line signal, for easier testing.

We wrote eight xfgen rules, corresponding to 16 variables and 15 types. All the transformers for Tor data structures were straightforward, since type representation changes were rare in the streak we considered. Investigating further, we found that Tor's data representations tended to remain stable because most Tor data represents protocol messages, and these rarely change so as to preserve backward compatibility. The bulk of the remaining xfgen rules update function pointers for event handling stored in libevent and/or OpenSSL data structures.

We are unaware of prior work applying DSU to Tor.

5.2.4. Memcached. Memcached is a multi-threaded server implemented using libevent. Unlike Tor, memcached's main loop is in libevent's event_base_loop function. To work with Kitsune, we had to change main to install a libevent callback on SIGUSR2 to handle update notifications. When the main thread is notified of an update, it uses pipes, via libevent itself, to notify each child thread of the update; each thread then terminates in response. The main thread then performs the update, and when the update is complete all threads are restarted and work back to calls to event_base_loop in the new code. Interestingly, we needed to "sandwich" such calls with like-named 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 a thread is signaled, the event_base_loop call returns and initiates the update. When the program restarts, the update will complete when it reaches the update point (of the same name) just prior to the event_base_loop call in the new code.

There are two additional challenges in updating due to libevent: First, as with Tor, we needed to reinstall new function pointers in libevent after an update. Second, memcached installs the data associated with active connections in libevent, but does not retain its own pointers to that data. To enable updates to transform that data, we added code to maintain, in memcached itself, a list of active connections, which we can then use for state transformation.

Other DSU systems. Neamtiu and Hicks [2009] 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. Nishtala et al. [2013] discuss Facebook's custom version of memcached that supports dynamic updates: it stores its state in a ramdisk to which a restarted version can reconnect. The paper does not describe the amount of code affected by this change, nor does it discuss the limitations of this approach (e.g., aside from the obvious limitation that internal data structures must not be altered between versions).

5.2.5. 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. To modify icecast to support DSU with Kitsune, we located each of the thread creation points and then inspected the thread entry function and related code to make two types of changes. First, we identified the thread's long-running, event-handling loop to which we added a call to kitsune_update. Second, we added the necessary annotations to migrate local variables or skip initialization during thread startup.

In its standard configuration, icecast runs with 6 threads. 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. We replaced the blocking sleep calls with sleep calls that are interrupted by the update request signal, causing the sleeping thread to hit the update point immediately upon update request. The most complex icecast patch added a new thread to handle authentication (which required us to add an update point to the new authentication thread's code) and reduced the number of connection threads. To implement this patch, we wrote transformation code that uses a C API provided by the Kitsune runtime that exposes the set of active threads at the time the update was taken (including the entry function, initialization argument, and update point taken) and allows the developer to add and remove threads or modify the properties of existing threads.

Other DSU systems. Neamtiu and Hicks [2009] also considered updates to the same streak of icecast versions. 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. A large proportion of these specifications were simple rules to initialize added variables or fields. However, they also included more complex rules for adding and removing a thread, as mentioned above. Ginseng's patches do not attempt to remove this extra connection thread.

5.2.6. Snort. Snort is a widely used, open-source network intrusion detection and prevention system which works by inspecting and possibly blocking packets as they enter a network. It is the largest program we updated, at 215K lines of C code (counting only the code in the main distribution). Snort identifies suspicious packets using *rulesets* that are available by subscription. New rules are published every few days, while new Snort software versions are released every 1-3 months. Snort provides rule-reloading to allow users to update some of their rules without shutting down Snort. However, there is no built-in mechanism to dynamically update Snort itself; thus, updating it requires halting its protective services for the duration of a restart and losing all information for current packet streams.

Snort was challenging to dynamically update for two reasons. First, it makes extensive use of dynamically loaded shared objects (a.k.a., plugins), which are used for extended protocol analysis (such as POP, SSH, and DNP3), output formatting, and more. Snort 2.9.2 ships with 14 dynamic preprocessors that compile to shared objects, and we configured the plugin engine to load all of them at startup. Plugins create two challenges. First, code that interfaces with plugins makes heavy use of **void** ∗'s to simplify the type structure. For example, output plugins are stored in a linked list of closures (function pointers with an additional environment) as follows:

```
1 typedef struct _OutputFuncNode {
2     void *arg;
3     union {
4         OutputFunc fptr;
5         void *vfptr;
6     } fptr;
7     struct _OutputFuncNode *next;
8 } OutputFuncNode;
```

Many of the void pointers in Snort vary based on the configuration file. For example, with **struct** _OutputFuncNode, what a **void** ∗ arg points to depends on the logging format specified in the configuration file, which consequently affects which transformation rule to write. Snort provides several options for logging formats, such as a custom format called *Unified2*. If the configuration file specifies

Unified2 format, then **void** $*$ arg points to a **struct** Unified2Config, and we must write XF_INVOKE(XF_PTR($xform(Unified2Config, Unified2Config)),....) in the transformer. However, if the configuration file specifies a different format, then the corresponding transformation for that specified format is necessary.

The second challenge with Snort was that it contains a large amount of global state. This is not a problem in and of itself, but it was a problem for us: since we are not the Snort authors, we needed to pore over the code to understand the role and function of all this state, which was challenging due to Snort's heavy use of void pointers. In more detail: Snort's global policy user context structure contains a **void** pointer to an array of structure pointers with information such as protocol-specific function pointers, state loaded from Snort's configuration file, and information about the current packets being processed. There are several dozen of these large configuration structures, making up a large portion of the global state in Snort. Additionally, the main global SnortConfig structure contains information about nearly everything that gets loaded from the configuration and all available preprocessors; this SnortConfig structure alone contains well over 100 fields in Snort 2.9.2. Snort also keeps a substantial amount of information in hash tables. Although it varies based on the configuration, there are around 20 hash tables in Snort 2.9.2 when all of the default plugins are enabled. These hash tables maintain state on flowbits (to track information on packets that are part of a continuous stream), protocol-specific information, rule information, and port information. Many of the entries in the hash tables contain structures with function pointers and must be transformed on update. In addition to determining the actual types of these **void** $*$ function pointers, a big challenge in migrating the hash table data was making sure that all of the arrays were properly annotated with the correct lengths and generic types for each hash table type. Another challenge in data migration was that structures are often nested many layers deep, and a mistake somewhere in transforming the nested data (such as writing the wrong transformation rule for a **void** $*$) made debugging cumbersome.

As reported in Table III, the largest single category of changes required for Snort was E_$*$ annotations. A large number (106) of these annotations include labeling each plugin file with the appropriate namespace, as described in Section 3.2. The Other column includes naming anonymous structures/unions so that the proper transformation rules can be applied, writing wrappers around system calls so that they are not transformed, modifying a function signature so that Kitsune would recognize it, and adding an additional **#define** to fix a 64-bit issue with the version of CIL we use with Kitsune.

Control migration in Snort was fairly straightforward, involving redirecting around some initialization code before jumping to the main packet-processing loop, and adding some additional code to make sure the new version of the plugins were loaded during update.

We are unaware of prior work applying DSU to Snort.

6. PERFORMANCE

To evaluate Kitsune's performance, we conducted several experiments. We measured the slowdown on normal execution for the Kitsune versions of the servers compared to the originals. We found that there is essentially no overhead on normal execution, a result uniformly better than prior work. We also measured the time taken to perform an update, from signaling to completion. We found that the time required to apply an update ranges from 2ms up to 390ms, depending on the program; in all cases, the times seem acceptable for typical use. We show that multi-threaded programs quiesce quickly with Kitsune, typically in less that 1ms. We also show that Kitsune scales well and requires almost no additional update time for large amounts ($>$14GB) of in-memory state for memcached and redis.

Table IV. Normal execution performance overhead

Program	Orig (SIQR)	Kitsune	Ginseng	UpStare
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	48.05s (0.81s)	-2.00%	–	–
icecast 2.3.1	10.11s (2.27s)	-2.18%	–	–
snort 2.9.2	23.09s (0.50s)	-1.74%	–	–
32-bit, 1×3.6Ghz Pentium D (2 core), 2GB mem, Ubuntu 10.10				
vsftpd 2.0.3*	5.96s (0.01s)	+2.35%	+11.3%	+41.6%
vsftpd 2.0.3†	14.03s (0.02s)	+0.29%	+1.47%	+6.64%
memcached 1.2.4	101.40s (0.35s)	-0.49%	+18.4%	–
redis 2.0.4	46.13s (0.22s)	-0.01%	–	–
icecast 2.3.1	35.71s (0.68s)	+1.18%	-0.28%	–
snort 2.9.2	10.42s (0.50s)	+1.73%	–	–

*CD+LS benchmark, †file download benchmark

6.1. Normal execution overhead

We measured the overhead Kitsune adds to normal execution (as determined by compiling with the standard Makefile and running the resulting executable) for all programs except Tor, discussed separately below. For comparison, we also measured the overhead of Ginseng and UpStare 0.12.9 on programs they had previously benchmarked: vsftpd, memcached, and icecast for Ginseng, and vsftpd for UpStare.

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. (Note that switching to standard malloc slightly *improves* redis' performance, since the custom allocator performs extra bookkeeping for tracking memory usage.) For vsftpd, we measured two separate tests. The first, which we call the *CD+LS benchmark*, performs the following interaction 2K times: connect to the server, change directories, and retrieve a directory listing. The second, performed only on the 32-bit platform, establishes a connection and downloads a 32-byte binary file 200 times. This test, which we call the *file download benchmark*, closely resembles tests reported in the UpStare and Ginseng papers [Neamtiu et al. 2006; Makris and Bazzi 2009]. For icecast, we used a benchmark originally developed for Ginseng [Neamtiu and Hicks 2009] 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, to factor out network latency. For Snort, to eliminate the variance in live traffic, we measured the time it took Snort to process a folder of captured packet files ranging from 343 to 304,298 packets per file. (We found that the file size of the captured packet file caused overhead to vary $\pm 11\%$, so we used a range of file sizes in our test.)

Table IV 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, Ginseng, and UpStare (the median time for each 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 and UpStare is only distributed in binary packages, which are only available for 32-bit systems.

Table V. Kitsune update times

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	40.27ms (0.97ms)	38.37	43.60
icecast →2.3.1	171.34ms (34.44ms)	33.81	255.44
tor →0.2.1.30	11.81ms (0.12ms)	11.65	13.83
snort →2.9.2.3	390.37ms (5.82ms)	376.73	412.63
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	30.62ms (0.23ms)	30.05	32.40
icecast →2.3.1	136.35ms (60.03ms)	34.82	267.38
tor →0.2.1.30	10.43ms (0.46ms)	10.08	12.98
snort →2.9.2.3	327.16ms (5.62ms)	312.61	337.30

From this data, we can see that Kitsune adds essentially no overhead to normal execution: the performance differences range from -2.18% to 2.35%, which is well within the noise on modern systems [Mytkowicz et al. 2009]. In contrast, the overhead for Ginseng and UpStare is more significant: for Ginseng it is 11.3% and 18.4% for memcached and the vsftpd CD+LS benchmark, respectively, and for UpStare it is 41.6% for the CD+LS benchmark.

Previously published papers on Ginseng and UpStare report lower overheads for vsftpd than we report here. In particular, Ginseng's overhead was reported at 3% and UpStare's at 16% [Neamtiu et al. 2006; Makris and Bazzi 2009]. We believe the difference is that our CD+LS benchmark spends a high proportion of its time executing application code (handling a series of FTP commands and generating the directory listing), while the one used in these previous papers is simply a file download, which is mostly implemented as a single operating system call. In the latter case, the overhead introduced by the Ginseng and UpStare compilers is reduced because little application code is executed during the benchmark. We ran our second vsftpd benchmark—just a file download—to confirm this conjecture and measured 1.47% overhead for Ginseng and 6.64% for UpStare. These overheads are closer to the previously reported numbers, where the remaining difference is likely due in part to the use of a difference architecture, operating system, and C compiler version. We decided to report both benchmarks to provide a fuller picture, but the CD+LS benchmark more directly measures the overhead of Ginseng and UpStare, and is perhaps more representative. We note, for example, that UpStare reports overheads of 40% for an updateable version of PostgreSQL, for which query processing is largely implemented in the application, and thus highly affected by the UpStare's special-purpose compiler.

While we did not measure the overhead of Kitsune on Tor directly, we did test it by running a Tor relay in the wild. We chose this experiment because evaluating the raw performance overhead of Kitsune on Tor would have involved creating a synthetic Tor network, which would not have realistically approximated a real-world use case. Moreover, update times would most likely be hidden by network latency—a Tor user's visible performance is limited by the slowest link in the Tor circuit their connections are multiplexed over, and these connections are over the (slow, geographically distributed) Internet. This is unlike our other benchmark programs, as even though the interfaces to redis and memcached are network-based, those programs are typically hosted on high-bandwidth, low-latency links to their client applications.

We dynamically updated this relay from version 0.2.1.18 to version 0.2.1.30 (13 versions) 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.

6.2. Time required for an update

We also measured the time it takes to deploy an update with Kitsune, i.e., the total elapsed time for an update's preparation and execution, starting from when the update is signaled to when it has completed. The total time consists of roughly two parts: the time to reach *full quiescence* while each thread in a multi-threaded program pauses at an update point until all threads have reached one, and the time to perform control and data migration. Here we present direct measurements of the time taken to perform the last update in each program's streak. In the next two subsections we investigate full quiescence in more depth, and also consider the impact on updating for programs with large amounts of state.

Table V gives the median+SIQR, minimum, and maximum update times for the last program version we considered. 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 with the server and ensured that none of the connections were interrupted during or after the update. For redis and memcached, we inserted 1,000 and 15,000 key-value pairs, respectively, prior to update, and verified that the same key-value pairs were present after the update. For icecast, we established one connection to a music source and 10 clients receiving that stream prior to updating and listened to the uninterrupted stream during and post-update. For Tor, we fully bootstrapped as a client, establishing multiple circuits through the network and communicating with directory servers, and then applied the update and ensured the circuits were maintained after updating. For Snort, we generated network traffic consisting of web browsing traffic, streaming music traffic, and file downloading traffic, and we saved the traffic to file. We then replayed the traffic as fast as possible as using Snort's built-in replay capability as we applied the update and verified that the packet count and attack detection logs were the same in the update and non-update case. We also ran Snort on live traffic and recorded very similar update times.

For most programs, the update times are quite small. The two programs with slightly longer update times are icecast and snort. For icecast, the delay occurs while the Kitsune runtime waits for each thread to reach an update point. This time is lengthened by one-second sleeps sprinkled throughout several of these threads; thus, Kitsune's ability to break out of blocking sleeps (Section 3.1) is essentially for keeping the update times low. For snort, the delay occurs because it loads fifteen shared objects (fourteen plugins and the main plugin engine) dynamically during startup, and these new versions must also be loaded in during update. This adds $\sim$100ms to the update time (Section 3.2).

6.3. Time required for full quiescence

The design of several prior DSU systems focused on making update times fast, and identified the need to reach full quiescence as potentially imposing an unacceptable delay [Neamtiu and Hicks 2009; Chen et al. 2011; Chen et al. 2006]. Therefore we decided to study several multi-threaded programs in addition to the three already considered (snort, memcached, and icecast) to see, if we were to update them with Kitsune, whether the time to reach full quiescence would be acceptable. Our results in

this subsection show that the changes required to support full quiescence were small, and quiescence could be achieved fairly quickly (in less than 1ms).

For our study, we considered Apache httpd, iperf, Space Tyrant, and suricata, covering the domains of web hosting, Internet profiling, gaming, and intrusion detection, respectively. We discuss these programs in more depth below. For each program we added a handful of calls to kitsune_update to identify legal update points, and we made some changes of the kind described in Section 3.1. Then we measured the *quiescence time*, which is the time elapsed between the first thread reaching an update point (following the receipt of a SIGUSR2 signaling an available update) and the last thread reaching one.

Next, we describe the programs in more detail, along with the changes we made to them, and then discuss the experimental results.

6.3.1. Changes to support full quiescence. The first three columns of Table VI describe the size and thread structure of our four subject programs. We consider each in turn.

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 2 and summarized in the last three columns of Table VI. 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 Kitsune.

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. We used a workload of downloading a large file from the httpd server.

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.

Space Tyrant. Space Tyrant is a server for a text-based, multiplayer space strategy game. At startup, Space Tyrant has 3 threads, and it 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 to jump back to the update point when needed. Space Tyrant's threads required no additional modification. We updated Space Tyrant while it had 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 3.1.2) to break out of the loop when an update signal is received.

Table VI. Multithreaded quiescence experiment: Program thread and modification information

Program	**LoC Total**	**# of Threads**	**Upd Pts**	**Changed LoC (†)**	**Required Manual Chgs**
httpd-2.2.22	232,651	$2 + c^*, \quad c = 3$	5	7 (5)	3 (Cond. Var. Loop)
iperf-2.0.5	3,996	$3 + n^\circ, \quad n = 1$	5	8 (3)	1 (Cond. Var. Loop)
space-tyrant-0.354	8,721	$3 + 2n^\circ, \quad n = 5$	6	8 (6)	1 (Thread Sleeps)
suricata-1.2.1	260,344	$8 + c^*, \quad c = 3$	7	11 (6)	1 (libpcap break)

*Configurable: c workers $^\circ$Varies by n connected clients
†Calls to Kitsune excluding update

Table VII. Multithreaded quiescence experiment: Program quiescence times

Program	**w/Load (ms)**		**w/o Load (ms)**	
	All Chgs	**Upd only**	**All Chgs**	**Upd only**
httpd-2.2.22	0.185	0.230	0.123	0.150
iperf-2.0.5	0.193	DNQ	0.169	DNQ
space-tyrant-0.354	0.426	20.583	0.078	20.304
suricata-1.2.1	0.503	68.098	0.378	DNQ

DNQ = Does not quiesce.

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.

6.3.2. Experimental results. Table VII reports 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.

The *Upd only* columns show quiescence times when programs are only modified to contain update points, but not the changes to handle blocking calls tabulated in the rightmost column of Table VI. The *All chgs* columns measure quiescence times when the full set of changes is made. We find that dealing with blocking calls is crucial: for several of the programs, without such support the programs will either not quiesce at all, or take a long time to do so. When blocking calls are dealt with, we reach full quiescence quickly for both workloads.

6.4. Time and memory required for data migration

As mentioned in Section 6.2, update times depend in part on the time to perform data migration. xfgen-generated migrations could traverse significant portions of the heap, and thus for some updates the update time may vary with the size of the program state. Large heaps could also be a limiting factor when performing an update due to the need to allocate new data structures during an update. We consider both issues in this subsection.

Among our benchmark programs, the ones most likely to have a large heap are memcached and redis, which implement in-memory databases/caches. Thus, we performed a series of experiments in which we update memcached and redis with varying heap sizes. The experiments were performed on an Intel Core i7-3770K CPU @ 3.50GHz (8 cores) with 16GB RAM running 64-bit Ubuntu 12.04 LTS. The hard drive of this machine is 250GB at 5400 RPM with 8MB Cache (2.5” SATA 3.0Gb/s).

Table VIII. Memcached - Median of 11 Trials

Num Entries*	RSS (GB)	+Kit	+Kit w/Upd	Total (ms)	SIQR (ms)
1,000,000	2.17	0.013%	0.032%	2.737	0.248
2,000,000	4.34	0.007%	0.018%	2.677	0.959
3,000,000	6.50	0.004%	0.010%	2.698	0.338
4,000,000	8.67	0.003%	0.008%	2.817	0.156
5,000,000	10.85	0.003%	0.007%	2.650	0.133
6,000,000	13.00	0.002%	0.006%	2.684	0.239

*Entries are (∼10B,∼2KB) text pairs

Table IX. Redis- Large Heap/Large Entries - Median of 11 Trials

Num Entries*	RSS (GB)	+Kit	+Kit w/Upd	Cold Start (s)	Time (ms)	SIQR (ms)
1,000,000	2.07	0.017%	0.313%	5	22.445	0.324
2,000,000	4.14	0.008%	0.156%	11	20.097	1.244
3,000,000	6.22	0.006%	0.104%	17	24.683	2.726
4,000,000	8.26	0.004%	0.078%	22	19.877	2.969
5,000,000	10.38	0.003%	0.062%	28	21.584	4.279
6,000,000	12.45	0.003%	0.052%	34	16.643	1.268

*Entries are (∼1KB,∼1KB) text pairs

Table X. Redis- Large Heap/Small Entries - Median of 11 Trials

Num Entries*	RSS (GB)	+Kit	+Kit w/Upd	Cold Start (s)	Time (ms)	SIQR (ms)
16,000,000	2.99	0.012%	0.216%	6	21.048	5.113
32,000,000	5.97	0.006%	0.108%	12	22.639	4.137
48,000,000	9.33	0.004%	0.069%	19	22.538	6.034
64,000,000	11.95	0.000%	0.051%	23	19.596	5.846

*Entries are (∼10B,∼10B) text pairs

Table VIII shows how memcached update times vary with heap size when updated from version 1.2.3 to 1.2.4; in these runs, we allowed up to 13GB to be cached using the command line options because this is essentially the maximum the application can consume on this machine configuration. Each key is approximately 10 bytes and each value is approximately 2 KB; the length is approximate because each consists of a fixed-length string concatenated with an increasing integer, and 10 bytes and 2KB are the average sizes respectively. The first two columns of the table show the number of entries and maximum resident set size (RSS) of (normally compiled) memcached during the benchmark run. The third column shows the maximum RSS of Kitsune-enabled memcached, including the Kitsune executable and the driver program, expressed as a percentage increase over the RSS of normal memcached. The fourth column shows the maximum RSS of Kitsune-enabled memcached when the program updates during the benchmark run, thus giving a sense of the additional memory requirements during a dynamic update. This measurement includes all allocated data structures used for bookkeeping during the update and the memory used by having the additional program version loaded. The last two columns list the median time and the SIQR for 11 runs. From this table, we can see from the table that update times are essentially invariant with heap size, taking just a few milliseconds, and the additional memory overhead is also quite small.

Table XI. Redis-traverse Comparisons - Median of 11 Trials

Num Entries*	RSS (GB)	Deep w/Upd	Shallow w/Upd	Cold (s)	Deep (s)	Shallow (s)	Mod (ms)
1,000,000	2.07	39.1%	38.7%	5	3.743	3.544	22.445
2,000,000	4.14	39.3%	38.7%	11	7.532	7.087	20.097
3,000,000	6.22	40.4%	40.2%	17	11.886	11.264	24.683
4,000,000	8.26	39.3%	39.2%	22	15.340	14.493	19.877
5,000,000	10.38	40.9%†	38.9%†	28	22.332†	21.235†	21.584

*Entries are (~1KB,~1KB) text pairs †Page Faults

Tables IX and X show how update times for redis vary with heap size when updating from version 2.0.2 to 2.0.3, with ~1KB and ~10B keys and values, respectively. The first two columns of these tables again show the number of entries and resident set size. The third and fouth columns show the additional memory overhead incurred by running with Kitsune and by updating. The third-to-last columns show the cold start times (loading the database from disk) for comparison. The last two columns give the median update time and SIQR of 11 trials. From these results, we can see that, overall, update times remain nearly constant across all heap sizes and, as with memcached, memory overhead is also quite small.

Recall from Section 5.2.2 that we modified redis to eliminate the need to traverse the heap for our update streak. Table XI reports on an experiment in which we measured the update times from version 2.0.2 to 2.0.3 with ~1KB keys and values, but with the key-value structure type changed, thus forcing the heap to always be traversed.

The second column shows the maximum resident set size of (normally compiled) redis from Table IX for comparison. The third column shows the additional overhead if we were to traverse the entire heap, including all of the database entries, mallocing (deep-copying) a new structure for each traversed entry and immediately freeing the old entry once the new one is allocated. The fourth column shows the additional overhead if we were to traverse the entire heap but modifying memory in-place, obviating the need to malloc a new structure and free the old one. Columns three and four have a very similar memory footprint because the old copy of the structure is immediately freed when it is allocated in the deep-copy case, and no additional memory is allocated; the main source of memory overhead in this case is the bookkeeping necessary to traverse millions of entries. The number of these bookkeeping structures scale along with the size of the heap, so the overall percentages in columns three and four remain roughly constant.

The fifth column of the table shows the cold start times (loading the database from disk) for comparison. The sixth column reports update times when the traversal requires a deep-copy of all entries, and the next column reports update times when the traversal requires only modifying entries in-place. The last row of the table, 5,000,000 entries, incurred some page faults causing the update times to be a bit slower. Finally, the last column shows update times when traversals are not required (for comparison from Table IX). These results show that even in the worst case, when the entire heap must be traversed, a dynamic update is still faster than the cold start time for ~1KB sized key-value pairs.

7. RELATED WORK

In this section we consider recent work that supports DSU for programs written in C and C++; these languages impose stringent constraints on a DSU system's design. Table XII characterizes the mechanisms used to implement Kitsune and other recent C/C++ DSU systems. Ekiden [Hayden et al. 2011], Ginseng [Neamtiu et al. 2006],

Table XII. Comparing DSU systems for C/C++

	Code upd			Data upd			Timing	
	tramp	ind	prog	repl	shdw	wrap	nonactv	upd pts
DynaMOS[4]	×				×			
Ekiden			×	×				×
Ginseng[12345]		×				×		×
K42[25]		×		×			×	
Ksplice	×				×		×	
OPUS[2]	×			–	–	–	×	
LUCOS/POLUS[245]	×			×				
UpStare[23]			×	×				(×)
DynSec[4]	×				×			
PROTEOS			×	×				(×)
Kitsune			×	×				×

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

OPUS [Altekar et al. 2005], POLUS [Chen et al. 2011], UpStare [Makris and Bazzi 2009], and DynSec [Payer et al. 2013; Payer and Gross 2013] target applications, while Ksplice [Arnold and Kaashoek 2009], K42 [Baumann et al. 2005], LUCOS [Chen et al. 2006], DynaMOS [Makris and Ryu 2007], and PROTEOS [Giuffrida et al. 2013] support (or are) OS kernels. (LUCOS is essentially a version of POLUS that uses VMMs to effect changes in operating systems; all comments we make about the latter apply to the former.) 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. (A direct comparison to two systems, Ginseng and UpStare, appears at the end of the introduction.)

Code updates. Most systems effect code updates at the granularity of individual functions (or objects). As noted in the first column, Ksplice, OPUS, DynaMOS, POLUS, and DynSec insert, at run-time, a trampoline in the old function to jump to the function's new version.[8] As noted in the second column, Ginseng 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 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 execution and also inhibits inlining. Most onerously, neither trampolines nor indirections support updating functions that never (or rarely) exit, such as main, which changes relatively frequently [Hayden et al. 2012d], or functions that contain event-handling loops, such as the scheduling loop in the OS. In the best case, programmers must refactor the program to place long-running loop bodies in separate functions (e.g., using "loop extraction" [Neamtiu et al. 2006]).

[8]The first instruction of the function is replaced by a jump to a small piece of code, the *trampoline*, that executes the replaced instruction and then jumps to the function's new version.

The remaining four systems, UpStare, Ekiden, PROTEOS, and Kitsune, support more general changes by updating at the granularity of the whole program or process rather than individual functions. PROTEOS performs updates at the process-level by placing virtual end-points in its IPC implementation, which allows the kernel to atomically rebind the virtual endpoints at update time, replacing the whole process at once. UpStare loads in code for the new program and then performs *stack reconstruction*: the running program automatically *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.

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 code adds substantial performance overhead to 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,[9] 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. Ekiden, the precursor of Kitsune, effects updates by transferring state to a new-version process; it employs roughly the same API as Kitsune and enjoys its benefits of high flexibility and low overhead, but updates take longer and require more memory.

Data updates. Returning to Table XII, we can see that most systems handle data structure representation changes by using *object replacement*, in which the programmer, or system, 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* wherein programs are compiled so that **struct**s have an added version field and extra "slop" space to allow for future growth. Calls to mediator functions are inserted to access updatable objects, and these calls initiate transformation of those 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) the added slop space 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**∗);

[9]Compiling the updatable program to use position-independent code (PIC) sacrifices a register. However, modern servers are often already compiled with PIC to enable address-space layout randomization.

and (4) the whole-program static analysis underlying Ginseng's type wrapping has trouble scaling.

Object replacement adds the least overhead to normal execution, 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 much higher. Note that DSU for type-safe languages can avoid xfgen's traversal generation: the garbage collector can automatically find and initiate transformation of changed objects [Subramanian et al. 2009; Gilmore et al. 1997] without need of further type annotations.

That said, object replacement in Kitsune is not a panacea. For one, xfgen does not currently support migrating a pointer to the interior of an object before the object itself has been migrated. Type wrapping is not limited in this way because the address of the interior object will not change between versions. However, type wrapping does suffer from a related problem where other data structures may hold onto an address inside a wrapped type (what Neamtiu and Hicks call an *abstraction-violating alias* [Neamtiu et al. 2006]), which will delay the update. Another problem with object replacement in Kitsune is that all migration takes place at update time, potentially producing a lengthy pause in execution. Type wrapping postpones transformation of data until it is accessed, which amortizes the transformation cost over the post-update execution. However, type wrapping's lazy approach could have the undesirable of effect of delaying the pause until the application is performing a time-critical request.

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 changed code is not *active*; that is, no thread is running that code, and no thread's stack refers to it. While this restriction reduces post-update errors, it does not eliminate them [Hayden et al. 2012d], 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 [Neamtiu and Hicks 2009]. This approach accelerates 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 chances of errors [Hayden et al. 2012d]. 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. Therefore, the programmer must understand the impact of multiple code versions accessing the same data, which we imagine could be tricky when a data structure change corresponds to a change in semantics. Our experience with the simple barrier approach suggests these more sophisticated approaches, with higher programmer demands, may be unnecessary (as per Section 6.3).

PROTEOS takes a hybrid approach of allowing updates at almost any time but requiring "state quiescence" as defined by programmer-provided assertions written in a

domain-specific language, or even in full C. If these assertions pass at a marked event loop, an update may proceed.

Checkpointing. Checkpoint-and-restart systems [Roman 2002] 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. Ekiden effects an update by serializing and transferring state between an old-version process and a fresh new-version process—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 has a similar programming API but allows in-place code and data changes, for better performance.

Multi-threaded quiescence. Several systems [Arnold and Kaashoek 2009; Krieger et al. 2006; Subramanian et al. 2009] 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 prior work [Hayden et al. 2012d], 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 [Neamtiu and Hicks 2009] 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 update took effect at legal update points. A static analysis determines which program points are equivalent to update points [Neamtiu et al. 2008] 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 than Kitsune—1,068ms, 6ms, and 1ms, respectively—though the experimental setup is different.

UpStare [Makris and Bazzi 2009] 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 Kitsune. The UpStare paper does not report update times for any multi-threaded programs.

POLUS [Chen et al. 2011] 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 multi-threaded 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.

8. 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 from applying Kitsune to both single- and multi-threaded benchmarks show that Kitsune has essentially no performance overhead, multi-threaded programs show no barrier to expedient update, and code changes required to use Kitsune are comparable to, or only slightly more than, prior systems. We show that Kitsune scales well in systems with large amounts of state. We believe that the ideas and insights behind Kitsune could also be applied to C++ programs, though extending to kitc and xfgen to C++ would require non-trivial effort. 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.

Kitsune is freely available from http://kitsune-dsu.com/.

ACKNOWLEDGMENTS

We would like to thank Michail Denchev and Jonathan Turpie for help in the development and testing of Kitsune. Emery Berger, Miguel Castro, JP Martin, Cristi Zamfir, and the anonymous referees provided helpful comments on drafts of this paper. Kristis Makris helped us with the UpStare benchmarks. This work was supported by NSF grants CCF-0910530 and the partnership between UMIACS and the Laboratory for Telecommunication Sciences.

REFERENCES

Gautam Altekar, Ilya Bagrak, Paul Burstein, and Andrew Schultz. 2005. OPUS: Online Patches and Updates for Security. In *Proc. USENIX Security*. USENIX Association, Berkeley, CA, USA, 287–302.

Jeff Arnold and M. Frans Kaashoek. 2009. Ksplice: automatic rebootless kernel updates. In *Proc. EuroSys*. ACM, New York, NY, USA, 187–198.

A. Baumann, J. Appavoo, D. Da Silva, J. Kerr, O. Krieger, and R. W. Wisniewski. 2005. Providing Dynamic Update in an Operating System. In *Proc. USENIX ATC*. USENIX Association, 279–291.

Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. 2002. Reconsidering custom memory allocation. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '02)*. ACM, New York, NY, USA, 1–12. DOI:http://dx.doi.org/10.1145/582419.582421

Haibo Chen, Rong Chen, Fengzhe Zhang, Binyu Zang, and Pen-Chung Yew. 2006. Live updating operating systems using virtualization. In *Proc. VEE*. ACM, New York, NY, USA, 35–44.

Haibo Chen, Jie Yu, Chengqun Hang, Binyu Zang, and Pen-Chung Yew. 2011. Dynamic Software Updating Using a Relaxed Consistency Model. *IEEE Transactions on Software Engineering* 37, 5 (2011), 679–694.

Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. 2007. Dependent types for low-level programming. In *Proc. ESOP*. Springer-Verlag, Berlin, Heidelberg, 520–535. http://dl.acm.org/citation.cfm?id=1762174.1762221

S. Gilmore, D. Kirli, and C. Walton. 1997. *Dynamic ML without Dynamic Types*. Technical Report ECS-LFCS-97-378. LFCS, University of Edinburgh. http://www.dcs.ed.ac.uk/home/stg/DynamicML/dynamic.ps.gz

Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. 2013. Safe and automatic live update for operating systems. In *Proceedings of the 18th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*. 279–292.

Christopher M. Hayden, Stephen Magill, Michael Hicks, Nate Foster, and Jeffrey S. Foster. 2012a. Specifying and Verifying the Correctness of Dynamic Software Updates. In *Proc. International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*. Springer-Verlag, Berlin, Heidelberg, 278–293.

Christopher M. Hayden, Karla Saur, Michael Hicks, and Jeffrey S. Foster. 2012b. A Study of Dynamic Software Update Quiescence for Multithreaded Programs. In *Proc. HotSWUp*. IEEE, 6–10.

Christopher M. Hayden, Edward K. Smith, Michail Denchev, Michael Hicks, and Jeffrey S. Foster. 2012c. Kitsune: Efficient, General-purpose Dynamic Software Updating for C. In *Proceedings of the ACM Conference on Object-Oriented Programming Languages, Systems, and Appilcations (OOPSLA)*. ACM, New York, NY, USA, 249–264.

Christopher M. Hayden, Edward K. Smith, Eric A. Hardisty, Michael Hicks, and Jeffrey S. Foster. 2012d. Evaluating Dynamic Software Update Safety Using Efficient Systematic Testing. *IEEE Transactions on Software Engineering* 38, 6 (Dec. 2012), 1340–1354. DOI:http://dx.doi.org/10.1109/TSE.2011.101 Accepted September 2011.

Christopher M. Hayden, Edward K. Smith, Michael Hicks, and Jeffrey S. Foster. 2011. State Transfer for Clear and Efficient Runtime Upgrades. In *Proc. HotSWUp*. IEEE, 179–184.

Michael Hicks and Scott Nettles. 2005. Dynamic software updating. *ACM TOPLAS* 27, 6 (2005), 1049–1096.

Orran Krieger, Marc A. Auslander, Bryan S. Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria A. Butrico, Mark F. Mergen, Amos Waterland, and Volkmar Uhlig. 2006. K42: building a complete operating system. In *EuroSys*, Yolande Berbers and Willy Zwaenepoel (Eds.). ACM, 133–145.

LiveRebel 2013. LiveRebel. http://zeroturnaround.com/software/liverebel/. (2013).

Kristis Makris. 2009. *Whole-Program Dynamic Software Updating*. Ph.D. Dissertation. Arizona State University.

Kristis Makris and Rida Bazzi. 2009. Immediate Multi-Threaded Dynamic Software Updates Using Stack Reconstruction. In *USENIX ATC*. USENIX Association, Berkeley, CA, USA, 31–31.

Kristis Makris and Kyung Dong Ryu. 2007. Dynamic and Adaptive Updates of Non-Quiescent Subsystems in Commodity Operating System Kernels. In *Proc. EuroSys*. ACM, New York, NY, USA, 327–340.

Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2009. Producing wrong data without doing anything obviously wrong!. In *Proc. ASPLOS*. ACM, New York, NY, USA, 265–276.

Iulian Neamtiu and Michael Hicks. 2009. Safe and Timely Dynamic Updates for Multi-threaded Programs. In *Proc. PLDI*. ACM, New York, NY, USA, 13–24.

Iulian Neamtiu, Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. 2008. Contextual Effects for Version-Consistent Dynamic Software Updating and Safe Concurrent Programming. In *Proc. POPL*. ACM, New York, NY, USA, 37–50.

Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. 2006. Practical Dynamic Software Updating for C. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*. ACM, New York, NY, USA, 72–83.

George C. Necula, Scott Mcpeak, Shree P. Rahul, and Westley Weimer. 2002. CIL: Intermediate language and tools for analysis and transformation of C programs. In *In International Conference on Compiler Construction*. 213–228.

Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation (nsdi'13)*. USENIX Association, Berkeley, CA, USA, 385–398. http://dl.acm.org/citation.cfm?id=2482626.2482663

Mathias Payer, Boris Bluntschli, and Thomas R. Gross. 2013. DynSec: On-the-fly Code Rewriting and Repair. In *Proceedings of the Workshop on Hot Topics in Software Upgrades (HotSWUp)*.

Mathias Payer and Thomas R. Gross. 2013. Hot-patching a web server: A case study of ASAP code repair. In *PST*. 143–150.

Luís Pina and Michael Hicks. 2013. Rubah: Efficient, General-purpose Dynamic Software Updating for Java. In *Proceedings of the Workshop on Hot Topics in Software Upgrades (HotSWUp)*.

Eric Roman. 2002. *A Survey of Checkpoint/Restart Implementations*. Technical Report. Lawrence Berkeley National Laboratory, Tech.

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

Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. 2009. Dynamic Software Updates: A VM-centric Approach. In *Proc. PLDI*. ACM, New York, NY, USA, 1–12.

Received June XXXX; revised June XXXX; accepted June XXXX