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

Michael Hicks
work with Christopher M. Hayden, Ted K. Smith, Michail Denchev, Karla Saur, and Jeffrey Foster
Software updates are crucial

- 2010 NASDAQ hacking incident
  - investigators blame vulnerability on out-of-date software
- Vulnerabilities are most exploited after being disclosed [Bilge & Dumitras, CCS ’12]

Oracle Critical Patch Update Advisory - April 2009

Description
A Critical Patch Update is a collection of patches for multiple security vulnerabilities. ...

Due to the threat posed by a successful attack, Oracle strongly recommends that customers apply fixes as soon as possible. This Critical Patch Update contains 43 new security fixes across all products.
Software updates are disruptive

- Typically require **restarting** the program
- interrupts active users / processing
- makes services unavailable
Dynamic Software Updating (DSU)

• Update program **while it runs**
  - Avoid interruptions
  - Preserve critical program state

• Useful for:
  - **Non-stop services**
    - E.g., Financial processing, air traffic control, network infrastructure
  - Programs with **long-lived connections**
    - E.g., VPNs and media streaming
  - Long-running programs with **large in-memory state**
    - E.g., operating systems, caching servers, in-memory databases
DSU State of the Art

• Current DSU systems for C try to be transparent
  ▪ Aim to work on any input program, with no changes

• We think this is a bad idea!
  ▪ At odds with the reasons people use C
  ▪ Empirical study shows existing transparent update approaches allow incorrect updates [Hayden et al, TSE 2011]
    ▪ Not as transparent as they seem
Kitsune

- Favors explicitness over transparency
  - Kitsune treats DSU is a program feature
- Having the developer orchestrate DSU allows:
  - simpler DSU mechanisms
  - easier reasoning for developers
  - full flexibility
  - low-level control
  - better performance

Kitsune (fox) - a shapeshifter according to Japanese folklore
Key idea #1: Explicitness

• Implementation of DSU is **visible** in the program source code
  - **When** the program updates
  - **Which state** is necessary to update
  - **How** that state is **transformed**

• Benefits
  - **Simplifies reasoning** for programmers
  - **Simplifies mechanism** for updating
Key idea #2: Whole program updates

• Update by restarting program from main, after replacing entire program at once
  ▪ Status quo: program keeps running the current code, and subsequent function calls to new versions

• Benefits
  ▪ Can update active code (e.g., long-running loops) easily
  ▪ Can easily understand the updating process as it is explicit in the program text
    - Not hard to write
Kitsune Implementation

- Employs entirely standard compilation and tools
  - Only requires simple source-to-source translator
- Runtime library to support the updating process
- New tool called **xfgen** generates code to transform the existing state
  - based on simple developer-written specifications
- In essence: **automates what is easy** and tedious, and focuses developer’s effort on the hard parts
Results

• Applied Kitsune to **six** open-source **programs**
  - **memcached, redis, icecast, snort**: 3-6 mos. of releases
  - **Tor, vsftpd**: 2, and 4, years of releases, respectively

• Essentially **no performance overhead**

• **Fast update times** (< 40ms typically)

• **Manageable effort** to retrofit
  - 50-160 LOC per program (largely one-time effort)
    - Program sizes from 5KLOC up to 220KLOC
  - 27-200 LOC of xfgen specs across **all** releases
Programmer obligations

• To implement DSU as a program feature, Kitsune requires the programmer to modify the program:
  ▪ Choose update points: where updates may take place
  ▪ Code for data migration: identify the state to be transformed, and where it should be received in the new code
  ▪ Code for control migration: ensure execution reaches the right event loop when the new version restarts

• Also, have to specify state transformation rules for our tool xfgen
Example single-threaded server

typedef int data;
data *mapping;

void client_loop(int fd) {
    while (1) {
        // ... process client requests
    }
}

int main() {
    int l_fd, cl_fd;
    mapping = malloc(...);
    l_fd = setup_conn();
    while (1) {
        cl_fd = get_conn(l_fd);
        client_loop(cl_fd);
    }
}
Example single-threaded server

typedef int data;
data *mapping; // automigrated
int l_fd; // automigrated

void client_loop() {
    int cl_fd = get_conn(l_fd);
    while (1) {
        kitsune_update("client");
        // ... process client requests
    }
}

int main() {
    kitsune_do_automigrate();
    if (!kitsune_is_updating()) {
        mapping = malloc(...);
        l_fd = setup_conn();
    }
}

We also support migration of locals
Generalizes to multi-threaded programs

Redirect control to update point

if (kitsune_is_updating_from ("client")) {
    client_loop();
}
while (1) {
    kitsune_update("main");
    client_loop();
}

1. Choose update points
   One per long running loop

2. Add data migration code
   Globals migrated by default
   Initiate at start of main()

3. Add control migration code
   Avoid reinitialization
   Redirect control to update point

We also support migration of locals
Generalizes to multi-threaded programs
Example single-threaded server

typedef
int
data;
data*mapping;

int
l_fd;

void
client_loop()
{
int
cl_fd
=
get_conn(l_fd);
while
(1)
{
//...
process
client
requests
}
}

int
main()
{
mapping
=
malloc(...);
l_fd
=
setup_conn();
while
(1)
{
client_loop();
}
}
kitsune_update("main");
kitsune_update("client");
if (!kitsune_is_updaFng())
{
}
kitsune_do_automigrate();
if (kitsune_is_updaFng_from
("client"))
{
client_loop();
}
//automigrated
//automigrated


Kitsune runtime model

1. Load first version
2. Run it
3. Call back to driver when update ready
4. Load second version
5. Migrate and transform state
6. Free up old resources
7. Continue with new version
Transforming State with xfgen

- State may need to be transformed to work with the new program

```
Xform
typedef int data;
data *mapping;
typedef char *data;
data *mapping;

Xfgen tool
• Require programmer to write relevant xform
typedef data → typedef data: {
  $out = malloc(N);
  snprintf($out, N, "%d", $in);
}
```
Example

typedef data -> typedef data: { $out = (long)$in; }
INIT struct list.cid { $out = -1; }
struct list.next -> struct list.pnext

xfgen spec
Benchmark programs

- Very secure FTP daemon - file transfers, securely
- **Redis** - key/value server
- **Tor** - anonymous routing daemon
- **Memcached** - caching daemon
- **Icecast** - streaming music server
- **Snort** - intrusion detection system

- **memcached, redis, icecast, snort**: 3-6 mos. of releases
- **vsftpd, Tor**: 2-4 years of releases
  - 39 updates total, for all programs
Kitsune benchmarks: changes required

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

*Multi-threaded

<table>
<thead>
<tr>
<th>Program</th>
<th>Upd</th>
<th>Ctrl</th>
<th>Data</th>
<th>E.*</th>
<th>Oth</th>
<th>∑</th>
<th>v→v</th>
<th>t→t</th>
<th>∑</th>
<th>xf</th>
<th>LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>vsftpd</td>
<td>6</td>
<td>26</td>
<td>17+8</td>
<td>6+14</td>
<td>28+8</td>
<td>83+30</td>
<td>9</td>
<td>21</td>
<td>30</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td>redis</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>43</td>
<td>8</td>
<td>57</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>Tor</td>
<td>1</td>
<td>39</td>
<td>37+6</td>
<td>19</td>
<td>57</td>
<td>153+6</td>
<td>16</td>
<td>15</td>
<td>31</td>
<td>189</td>
<td></td>
</tr>
<tr>
<td>memcached*</td>
<td>4</td>
<td>9</td>
<td>13</td>
<td>20</td>
<td>66</td>
<td>112</td>
<td>12</td>
<td>10</td>
<td>22</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>icecast*</td>
<td>11+1</td>
<td>22+3</td>
<td>14+9</td>
<td>32+3</td>
<td>39</td>
<td>118+16</td>
<td>25</td>
<td>50</td>
<td>75</td>
<td>200</td>
<td></td>
</tr>
<tr>
<td>snort*</td>
<td>2</td>
<td>90+18</td>
<td>110+2</td>
<td>158</td>
<td>66</td>
<td>426+20</td>
<td>111</td>
<td>64</td>
<td>175</td>
<td>197</td>
<td></td>
</tr>
</tbody>
</table>
### Steady-state performance overhead

<table>
<thead>
<tr>
<th>Program</th>
<th>Orig (s)</th>
<th>Kitsune</th>
<th>Ginseng</th>
<th>UpStare</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>64-bit, 4×2.4Ghz E7450 (6 core), 24GB mem, RHEL 5.7</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vsftpd 2.0.6*</td>
<td>6.55s (0.04s)</td>
<td>+0.75%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>memchd 1.2.4</td>
<td>59.30s (3.25s)</td>
<td>+0.51%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>redis 2.0.4</td>
<td>46.83s (0.40s)</td>
<td>-0.31%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>icecast 2.3.1</td>
<td>10.11s (2.27s)</td>
<td>-2.18%</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program</th>
<th>Orig (s)</th>
<th>Kitsune</th>
<th>Ginseng</th>
<th>UpStare</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>32-bit, 1×3.6Ghz Pentium D (2 core), 2GB mem, Ubuntu 10.10</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vsftpd 2.0.3*</td>
<td>5.96s (0.01s)</td>
<td>+2.35%</td>
<td>+11.3%</td>
<td>+41.6%</td>
</tr>
<tr>
<td>vsftpd 2.0.3†</td>
<td>14.03s (0.02s)</td>
<td>+0.29%</td>
<td>+1.47%</td>
<td>+6.64%</td>
</tr>
<tr>
<td>memchd 1.2.4</td>
<td>101.40s (0.35s)</td>
<td>-0.49%</td>
<td>+18.4%</td>
<td>-</td>
</tr>
<tr>
<td>redis 2.0.4</td>
<td>43.88s (0.16s)</td>
<td>-1.21%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>icecast 2.3.1</td>
<td>35.71s (0.68s)</td>
<td>+1.18%</td>
<td>-0.28%</td>
<td>-</td>
</tr>
</tbody>
</table>

*CD+LS benchmark, †file download benchmark

- Overhead when **not** updating
- Overall: -2.18% to 2.35% overhead (in the noise)
  - (No performance measurements for snort yet)
Update times

- <40ms in all cases but icecast
  - Icecast includes 1s sleeps; icecast-nsp removes these

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

![Graph showing update times for various state sizes](image)

- Key difference is data representation: arrays vs. nested objects with pointers to static memory
Conclusions

• Kitsune treats DSU as a *program feature*
  - Deliberately makes update semantics apparent to programmer
  - Effort for changing program roughly corresponds to effort in thinking about DSU

• Results promising
  - Applied to 39 updates of 6 real applications
  - Most flexible, performant C DSU system to date

• Now exploring commercialization potential