Varan and Mx: Safe Software Updates via Multi-version Execution

Petr Hosek  Cristian Cadar
Software Reliability Group
Department of Computing

Imperial College
London

Hong Kong University of Science and Technology 28th January 2015
Motivation

Software evolves, with new versions and patches being released frequently

Software updates often present a high risk

Many users refuse to upgrade their software…

…relying instead on outdated versions flawed with vulnerabilities or missing useful features and bug fixes

Many admins (70% of those interviewed) refuse to upgrade

Cameri, O., Knezevic, N., Kostic, D., Bianchini, R., Zwaenepoel, W. *Staged deployment in Mirage, an integrated software upgrade testing and distribution system.* SOSP’07
“The fundamental problem with program maintenance is that fixing a defect has a substantial (20-50%) chance of introducing another. So the whole process is two steps forward and one step back.”

— Fred Brooks, 1975

≥14.8~24.4% for major operating system fixes

Yin, Z., Yuan, D., Zhou, Y., Pasupathy, S., and Bairavasundaram, L.
How Do Fixes Become Bugs? In ESEC/FSE’ 11
One solution: Patch Testing
[joint work with Marinescu, ESEC/FSE'13]

**KATCH** automatically tests each submitted patch, looking for potential bugs it introduces.

Study on all patches in 19 applications over a combined period of 6 years:

- Significantly improved patch coverage
- Found previously unknown bugs

Of course, bugs inevitably make it into released code

Paul Marinescu, Cristian Cadar
*KATCH: High-Coverage Testing of Software Patches. In ESEC/FSE’13*
Single-threaded event-driven web server

Powers several popular sites such as YouTube, Wikipedia, Meebo

HTTP ETag hash value computation in `etag_mutate`

```c
for (h = 0, i = 0; i < etag->used; ++i)
    h = (h << 5) ^ (h >> 27) ^ (etag->ptr[i]);
```
April 2009

Old bug fixed,
New bug introduced

HTTP ETag hash value computation in etag_mutate

```c
for (h = 0, i = 0; i < etag->used - 1; ++i)
    h = (h << 5) ^ (h >> 27) ^ (etag->ptr[i]);
```

File (re)compression in mod_compress_physical

```c
if (use_etag)
    etag_mutate(con->physical.etag, srv->tmp_buf);
}
```
Safe Updates via Multi-Version Execution

When a new version becomes available
Run it in parallel with the old versions!

REDIS $v_1$
REDIS $v_2$
REDIS $v_3$

...  

REDIS $v_{N-1}$
REDIS $v_N$

N = available (idle) cores
Safe Updates via Multi-Version Execution

When a new version becomes available
Run it in parallel with the old versions!
Synchronise all versions to act as one to the outside world
Safe Updates via Multi-Version Execution

When a new version becomes available
Run it in parallel with the old versions!
Synchronise all versions to act as one to the outside world
Transparently survive crashes occurring in some versions
Safe Updates via Multi-Version Execution

Could do so until enough confidence is gained in the new version(s)
Or as long as enough idle cores are available
MultiCore CPUs becoming standard
...with no benefit to inherently sequential apps

Idle parallel resources, with no benefit to inherently sequential applications

Cristian Cadar, Peter Pietzuch, Alex Wolf  *Multiplicity computing: A vision of software engineering for next-generation computing platform applications.* FoSER’10
Similar Idea: Automatically Generated Variants

Run two variants with stacks growing in different directions [Orchestra]
   Any divergence is a possible attack: fail safe
Run multiple variants with different placement of objs in mem [DieHard]
   Survive some errors due to memory corruption
Challenges of Multi-Version and Multi-Variant Execution

Common challenges:

- Synchronise and virtualise the executions of multiple versions efficiently

Specific to multi-version execution

- Allow for (small) differences in behaviour

Our proposed solution addresses both of these
Synchronisation

Possible at different levels of abstraction/granularity

- Application inputs/outputs
- Library calls
- System calls
Synchronisation at System Call Level

Advantages
General
System calls the only way to interact with outside world
Small number of system call types

Monitor

Operating System

System calls

\[ V_1 \]
System calls

\[ V_2 \]
System calls

\[ \cdots \]
System calls

\[ V_N \]
### Version 1

```c
void even_odd(int *a, size_t len) {
    int i, even = 0;
    for (i=0; i<len; i++)
        if (a[i] % 2 == 0)
            even++;
    printf("%d\n", even);
    printf("%d\n", len - even);
}
```

```c
int arr[] = { 6, 3, 2, 4 };
even_odd(arr, 4);
... write(1, "3\n", 2) = 2
write(1, "1\n", 2) = 2
...```

### Version 2

```c
void even_odd(int *a, size_t len) {
    int i, odd = 0;
    for (i=len-1; i>=0; i--)
        if (a[i] % 2 != 0)
            odd++;
    printf("%d\n", len - odd);
    printf("%d\n", odd);
}
```

**System Calls Define External Behavior**
95% of lighttpd revisions introduce no change*

Measured using lighttpd regression suite on 164 revisions (~10 months)

*Taken on Linux kernel 2.6.40 and glibc 2.14 using strace tool and custom post-processing (details in [ICSE'13])
Wait for other version

Notify monitor
Notify monitor

Compare syscalls
Perform syscall
Propagate results
Wait for other version

Notify monitor

VERSION 1

VERSION 2

MONITOR
Perform syscalls
ptrace(PTRACE_GETREGS, 7, {...}, NULL)
ptrace(PTRACE_SETREGS, 7, {...}, {...})
ptrace(PTRACE_SYSCALL, 7, {...}, NULL)

read(8, “PING\n”, 1024)
--- SIGTRAP ---
read(6, “PING\n”, 1024)
--- SIGTRAP ---
getpid()
--- SIGTRAP ---
read(6, “PING\n”, 1024)

ptrace(PTRACE_GETREGS, 7, {...}, NULL)
ptrace(PTRACE_SETREGS, 7, {...}, {...})
ptrace(PTRACE_SYSCALL, 7, {...}, NULL)

read(8, “PING\n”, 1024)
ptrace(PTRACE_GETREGS, 7, {...}, NULL)
process_vm_writev(7, {...}, 1, {...}, 1, 0)
ptrace(PTRACE_SETREGS, 7, {...}, {...})
ptrace(PTRACE_SYSCALL, 7, {...}, NULL)

--- System call enter ---
APPLICATION
--- System call exit ---
--- System call nullification ---
--- Writeback results ---
--- Monitor ---
Disadvantages of ptrace

Slow
For each system call run by each version, the monitor runs several system calls (and traps)
Does not scale well to large number of versions
Multi-version execution runs no faster than the slowest version

Inflexible
Lockstep execution requires the same sequence of system calls
Varan
Distributed Highly-Concurrent Multi-Version Monitor

http://godzilla.wikia.com/wiki/Varan
Varan

**Performance**
- Low performance overhead
- Scales to large number of versions

**Flexibility**
- Does not require lockstep execution
  - Tolerance to minor differences
- Enables novel applications
REDIS
0x4050f0 <anetRead>:

405130: callq <read@plt>

GLIBC
0xdeadbeef <__libc_read>:

2a: jmpq $0x13cd0

MONITOR
0x13cd0 <syscall_enter>:

...
Shared memory ring buffer
LEADER

MONITOR

Shared memory ring buffer
LEADER

MONITOR

Shared memory ring buffer
Asynchronous execution
Asynchronous execution
Enforcing causal ordering
VARAN: Performance Evaluation
Varan

Performance

- Low performance overhead
- Does not require lockstep execution
- Scales to large number of versions
ptrace(PTRACE_GETREGS, 7, {...}, NULL)
ptrace(PTRACE_SETREGS, 7, {...}, {...})
ptrace(PTRACE_SYSCALL, 7, {...}, NULL)
read(8, "PING\r\n", 1024)
ptrace(PTRACE_GETREGS, 7, {...}, NULL)
process_vm_writev(7, {?}, 1, {?}, 1, 0)
ptrace(PTRACE_SETREGS, 7, {...}, {...})
ptrace(PTRACE_SYSCALL, 7, {...}, NULL)
--- SIGTRAP ---
read(6, "PING\r\n", 1024)
--- SIGTRAP ---
getpid()
--- SIGTRAP ---
read(6, "PING\r\n", 1024)
Taken on 3.40 GHz Intel Core i7-2600 with 8 GB of RAM, Linux kernel 3.11.0
Safe updates via multi-version execution
Handling crashes in some of the versions
April 2009

Old bug fixed,
New bug introduced

HTTP ETag hash value computation in etag_mutate

```c
for (h = 0, i = 0; i < etag->used - 1; ++i)
    h = (h << 5) ^ (h >> 27) ^ (etag->ptr[i]);
```

File (re)compression in mod_compress_physical

```c
if (use_etag)
    etag_mutate(con->physical.etag, srv->tmp_buf);
```
Case 1: Follower crashes

LIGHTTPD 2437
Monitor

Remove from followers list

Latency perceived by the client remains ~5ms
Monitor
Case 2: Leader crashes

Latency perceived by the client remains ~5ms

Leader re-election
Advanced key-value store server

Powers several popular services such as GitHub and Flickr
HMGET command hmgetCommand function

refactor

Bug introduced Apr 13, 2010

Bug may result in losing some or even all of the stored data
Latency perceived by the client remains ~42.36μs

Case 1: Follower crashes
Case 2: Leader crashes

Latency increases from 42.36μs to 122.62μs
Handling divergences between versions

Using rewrite rules
95% of lighttpd revisions introduce no change*

Measured using lighttpd regression suite on 164 revisions (~10 months)

*Taken on Linux kernel 2.6.40 and glibc 2.14 using strace tool and custom post-processing (details in [ICSE'13])
if (!i_am_root && (geteuid() == 0 || getegid() == 0)) {

    #ifdef HAVE_GETUID
    #ifndef HAVE_ISSETUGID
    static int l_issetugid() {
        return (geteuid() != getuid() ||
                getegid() != getgid());
    }
    #define issetugid l_issetugid
    #endif
    #endif

    if (!i_am_root && issetugid()) {

        # define issetugid l_issetugid
        # endif
        #endif

        if (!i_am_root && issetugid()) {

    }}
BPF filter

```c
ld  event[0]
jeq #108, getegid  /* __NR_getegid */
jeq #2, open       /* __NR_open */
jmp  bad
getegid:
  ld [0]            /* offsetof(struct event_data, nr) */
  jeq #102, good    /* __NR_getuid */
open:
  ld [0]            /* offsetof(struct event_data, nr) */
  jeq #104, good    /* __NR_getgid */
bad: ret #0         /* SECCOMP_RET_KILL */
good: ret #0x7fff0000 /* SECCOMP_RET_ALLOW */
```
Handling different crashes in multiple versions
Via failure recovery
Scope: Surviving crash errors occurring at different times

Current limitation: implemented in a ptrace-based system, with lockstep execution
Failure Recovery: Runtime Code Patching

System call X

System call Y

V1

V2

...
Synchronisation and failure recovery mechanism

**Synchronisation**
Compare individual system calls and their arguments

```c
for (h = 0, i = 0; i < etag->used; ++i)
    h = (h << 5) ^ (h >> 27) ^ (etag->ptr[i]);
```

**Failure recovery**
Restart from the checkpoint and replace the code with the code of the new version

```c
for (h = 0, i = 0; i < etag->used; ++i)
    h = (h << 5) ^ (h >> 27) ^ (etag->ptr[i]);
```

**Checkpointing**
Use `clone` to take a snapshot of a process

```c
for (h = 0, i = 0; i < etag->used - 1; ++i)
    h = (h << 5) ^ (h >> 27) ^ (etag->ptr[i]);
```

**Crash**
Segmentation fault

```c
for (h = 0, i = 0; i < etag->used; ++i)
    h = (h << 5) ^ (h >> 27) ^ (etag->ptr[i]);
```

**Reconvergence**
Return to the original code and continue execution

```c
for (h = 0, i = 0; i < etag->used; ++i)
    h = (h << 5) ^ (h >> 27) ^ (etag->ptr[i]);
```
Failure Recovery: Suitable Scenarios

Errors with a small propagation distance
  “Localized” around a small portion of code

Applications which provide “natural” synchronization points
  E.g., servers structured around a main dispatch loop

Changes which do not affect memory layout
  E.g., refactorings, security patches
Failure Recovery: Guarantees?

Assumes that recovery is successful if versions exhibit the same external behavior after recovery

If unrecoverable, drops the crashed version

(By design, Mx does not attempt to survive errors it cannot handle)
Runtime code patching and fault recovery

- OS-level checkpointing (using clone syscall)
- Code segment replacement*
- Runtime stack manipulation
- Breakpoint insertion and handling (for indirect fun calls)

*Currently with compiler support
Stack Patching

Version 1

```c
void foo() {
    ...
    read(1, buf, 3);
    ...
}
```

Version 2 (patched)

```c
void foo() {
    ...
    read(1, buf, 3);
    ...
}
```

Ret Addr: 0xDEADBEEF:

Ret Addr: 0xAAACCC

Ret Addr: 0xAAAACCC

Ret Addr: 0xAAAABBB

Ret Addr: 0xAAAACCC

Ret Addr: 0xBEEFDEAD

Ret Addr: 0xDEADBEEF

Ret Addr: 0xBEEFDEAD

Ret Addr: 0xAAAACCC

Ret Addr: 0xAAAABBB
Indirect Calls

### Version 1

```
fptr = bar;
...
void bar(int x) {
    ...
}
void foo() {
    ...
    fptr(1);
    ...
}
```

Memory

```
fptr: 0x12345678
```

### Version 2 (patched)

```
fptr = bar;
...
void bar(int x) {
    ...
    INT 3
    ...
}
void foo() {
    ...
    fptr(1);
    ...
}
```

Memory

```
fptr: 0x876543210
```
Static Binary Analyzer

Create various mappings between the two version binaries

Static analysis of binary executables
Extracting function symbols from binaries (libbfd)
Machine code disassembling and analysis (libopcodes)
Binary call graph reconstruction and matching
Evaluation: survived several crash bugs

<table>
<thead>
<tr>
<th>Utilities</th>
<th>Application</th>
<th>Bug</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>md5sum</td>
<td>Buffer overflow</td>
</tr>
<tr>
<td></td>
<td>sha1sum</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mkdir</td>
<td>NULL-ptr dereference</td>
</tr>
<tr>
<td></td>
<td>mkfifo</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mknod</td>
<td></td>
</tr>
<tr>
<td></td>
<td>cut</td>
<td>Buffer overflow</td>
</tr>
<tr>
<td>Servers</td>
<td>lighttpd #1</td>
<td>Loop index underflow</td>
</tr>
<tr>
<td></td>
<td>lighttpd #2</td>
<td>Off-by-one error</td>
</tr>
<tr>
<td></td>
<td>redis</td>
<td>Missing return</td>
</tr>
</tbody>
</table>
**Mx and Varan**

Promising new approach for improving software updates

Based on multi-version execution

Our prototypes can survive crash bugs in real software updates

Varan’s novel architecture incurs a low performance overhead and can handle system call divergences

**Many opportunities for future work**

Support for more complex code changes in Mx & more complex divergences in Varan

Improve memory consumption

Explore new other applications, e.g., live sanitization

Can multiple software versions be effectively combined to increase software reliability and security?
*Mx and Varan: Safe Software Updates via Multi-version Execution*

**[ASPLOS 2015]** Hosek and Cadar, VARAN the Unbelievable An Efficient N-version Execution Framework

**[ICSE 2013]** Hosek and Cadar, Safe Software Updates via Multi-version Execution

**[HotSwUp 2012]** Cadar and Hosek, Multi-version software updates