SAFE SOFTWARE UPDATES VIA MULTI-VERSION EXECUTION

PETR HOSEK
CRISTIAN CADAR

Imperial College London

Petr Hosek is a recipient of the **Google European Fellowship in Software Engineering** and this research is supported in part by this Google Fellowship

"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.

—**F. Brooks,** 1975
The Mythical Man-Month

***More than 14.8~24.4% for major operating system patches**
Yin, Z., Yuan, D., Zhou, Y., Pasupathy, S., and Bairavasundaram, L. *How Do Fixes Become Bugs?* ESEC/FSE'11

Software updates often present a high risk

Many admins (70%) and users refuse to upgrade software

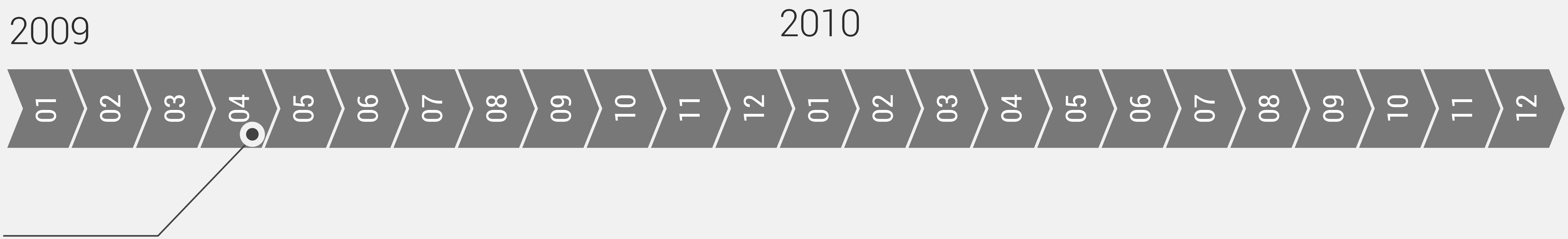
Reliance on outdated versions flawed with vulnerabilities

Crameri, O., Knezevic, N., Kostic, D., Bianchini, R., Zwaenepoel, W. *Staged deployment in Mirage, an integrated software upgrade testing and distribution system.* SOSP'07

Real-world **example**

LIGHTTPD
fly light.

Real-world **example**

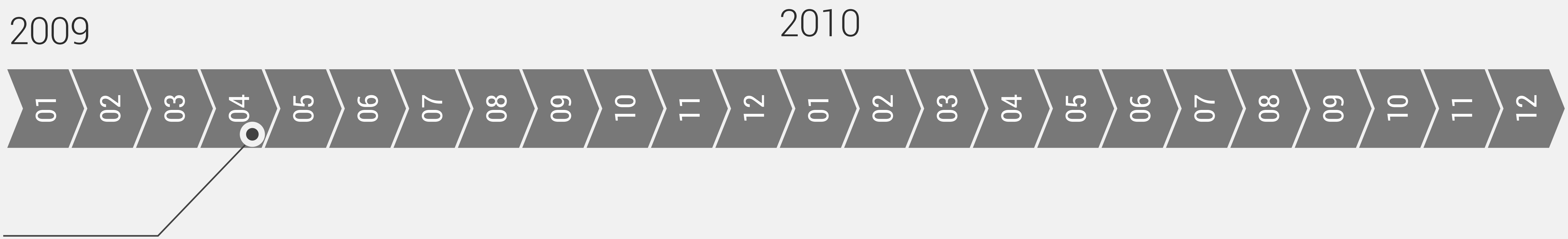



```
for (h = 0, i = 0; i < etag->used; ++i)
  h = (h << 5) ^ (h >> 27) ^ (etag->ptr[i]);
```

HTTP ETag hash value computation in `etag_mutate`

LIGHTTPD
fly light.

Real-world **example**

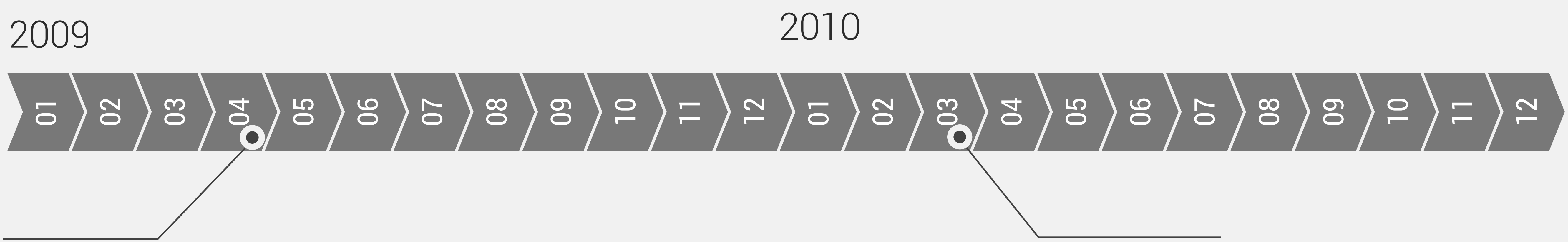



```
for (h = 0, i = 0; i < etag->used - 1; ++i)
  h = (h << 5) ^ (h >> 27) ^ (etag->ptr[i]);
```

HTTP ETag hash value computation in `etag_mutate`

LIGHTTPD
fly light.

Real-world **example**



Bug diagnosed in issue tracker

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

HTTP ETag hash value computation in `etag_mutate`

LIGHTTPD
fly light.

Real-world **example**

```
etag_mutate(con->physical.etag, srv->tmp_buf);
```

File (re)compression in `mod_compress_physical`

Bug diagnosed in issue tracker

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

HTTP ETag hash value computation in `etag_mutate`

LIGHTTPD fly light.

Real-world **example**

```
if (use_etag) {
    etag_mutate(con->physical.etag, srv->tmp_buf);
}
```

File (re)compression in `mod_compress_physical`


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

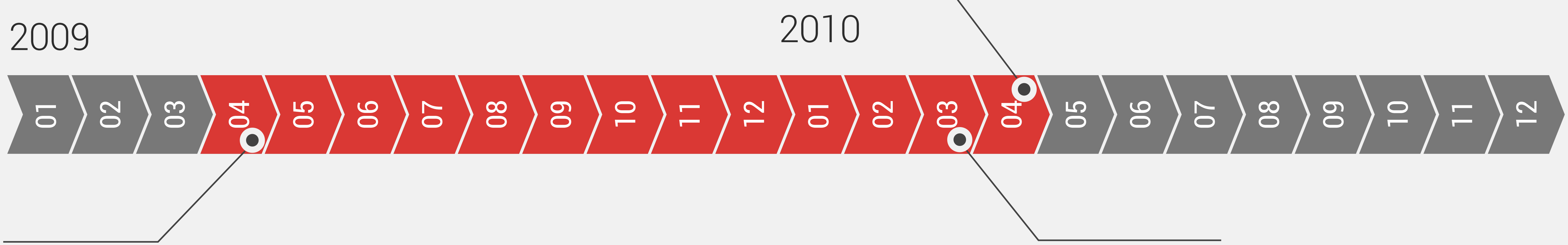
HTTP ETag hash value computation in `etag_mutate`

Bug diagnosed in issue tracker

Real-world **example**

```
if (use_etag) {
    etag_mutate(con->physical.etag, srv->tmp_buf);
}
```

File (re)compression in `mod_compress_physical`



Bug diagnosed in issue tracker

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

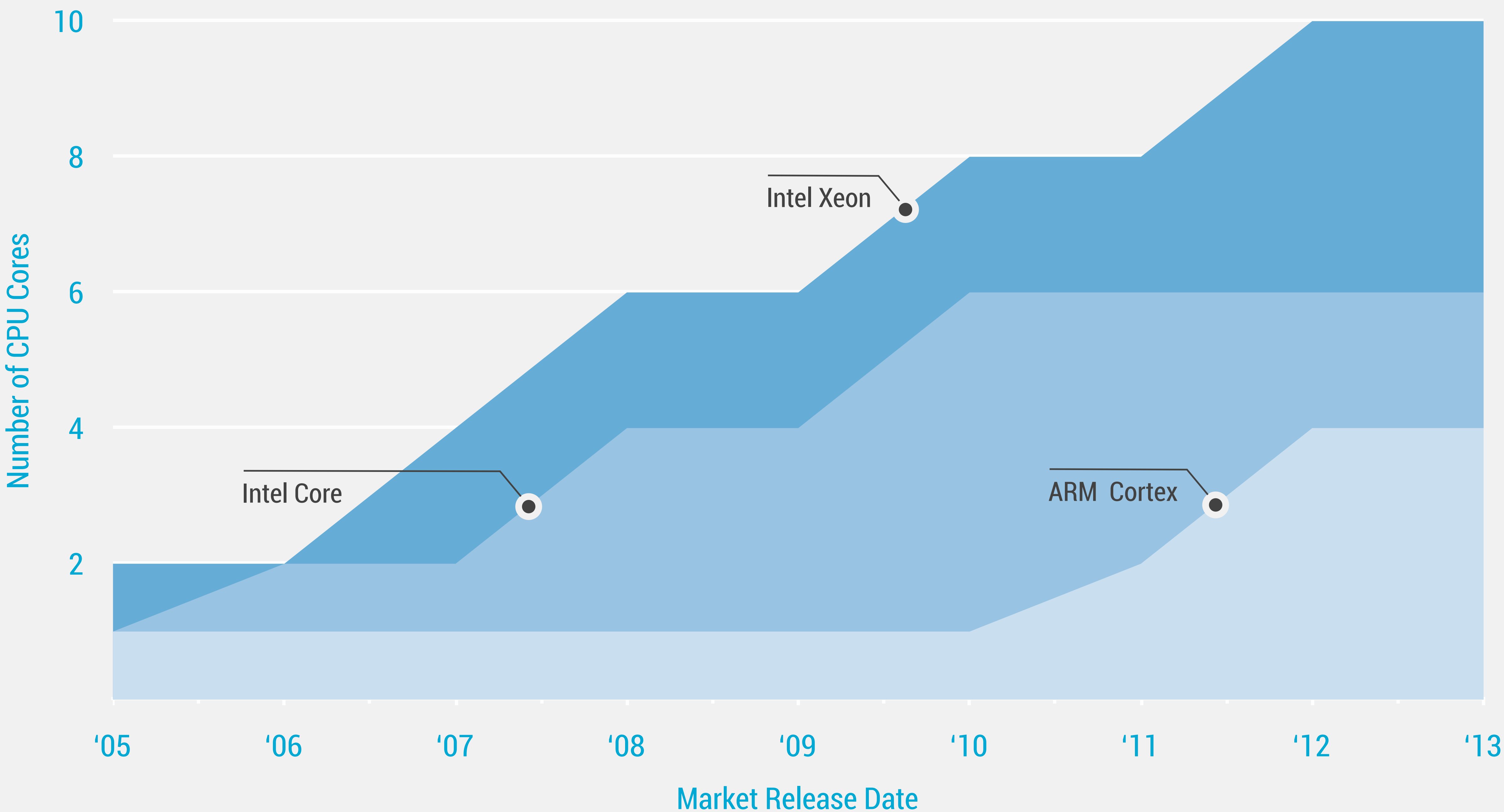
HTTP ETag hash value computation in `etag_mutate`

Goal

Improve the execution of upgraded software to provide:

- Benefits of the newer version
- Stability of the older version

Multi-core CPU becoming a standard



Abundance of resources and a high degree of parallelism with no benefit to inherently sequential applications

Cadar, C., Pietzuch P., Wolf, A. L. *Multiplicity computing: A vision of software engineering for next-generation computing platform applications*. FoSER'10

Multi-version execution based approach

Run the new version in parallel with the existing one

Synchronise the execution of the two versions

Use output of correctly executing version at any given time

Can be extended to work with multiple versions

Synchronisation possible at multiple levels of abstraction:

- Application inputs-outputs
- Function/library calls
- System calls

System calls define **external behaviour**

VERSION 1	VERSION 2

VERSION 1

```
void fib(int n)
{
    int f[n+1];
    f[1] = f[2] = 1;
    for (int i = 3; i <= n; ++i)
        f[i] = f[i-1] + f[i-2];

    printf("%d\n", f[n]);
}
```

VERSION 2

```
void fib(int n)
{
    int a = 1, b = 1;
    for (int i = 3; i <= n; ++i) {
        int c = a + b;
        a = b, b = c;
    }
    printf("%d\n", b);
}
```

```
int main(int argc, char **argv)
{
    fib(5);
    fib(6);
}
```

Example testing code
Tested with both implementations

System calls define **external behaviour**

VERSION 1

```
void fib(int n)
{
    int f[n+1];
    f[1] = f[2] = 1;
    for (int i = 3; i <= n; ++i)
        f[i] = f[i-1] + f[i-2];

    printf("%d\n", f[n]);
}
```

VERSION 2

```
void fib(int n)
{
    int a = 1, b = 1;
    for (int i = 3; i <= n; ++i) {
        int c = a + b;
        a = b, b = c;
    }
    printf("%d\n", b);
}
```

```
...
write(1, "5\n", 2) = 2
write(1, "8\n", 2) = 2
...
```

Snippet of system call trace
Obtained using *strace* tool

```
int main(int argc, char **argv)
{
    fib(5);
    fib(6);
}
```

Example testing code
Tested with both implementations

System calls define **external behaviour**

VERSION 1	VERSION 2

VERSION 1

```
void fib(int n)
{
    int f[n+1];
    f[1] = f[2] = 1;
    for (int i = 3; i <= n; ++i)
        f[i] = f[i-1] + f[i-2];

    printf("%d\n", f[n]);
}
```

```
...
write(1, "5\n", 2) = 2
write(1, "8\n", 2) = 2
...
```

Snippet of system call trace
Obtained using *strace* tool

VERSION 2

```
void fib(int n)
{
    int a = 1, b = 1;
    for (int i = 3; i <= n; ++i) {
        int c = a + b;
        a = b, b = c;
    }
    printf("%d\n", b);
}
```

```
...
write(1, "5\n", 2) = 2
write(1, "8\n", 2) = 2
...
```

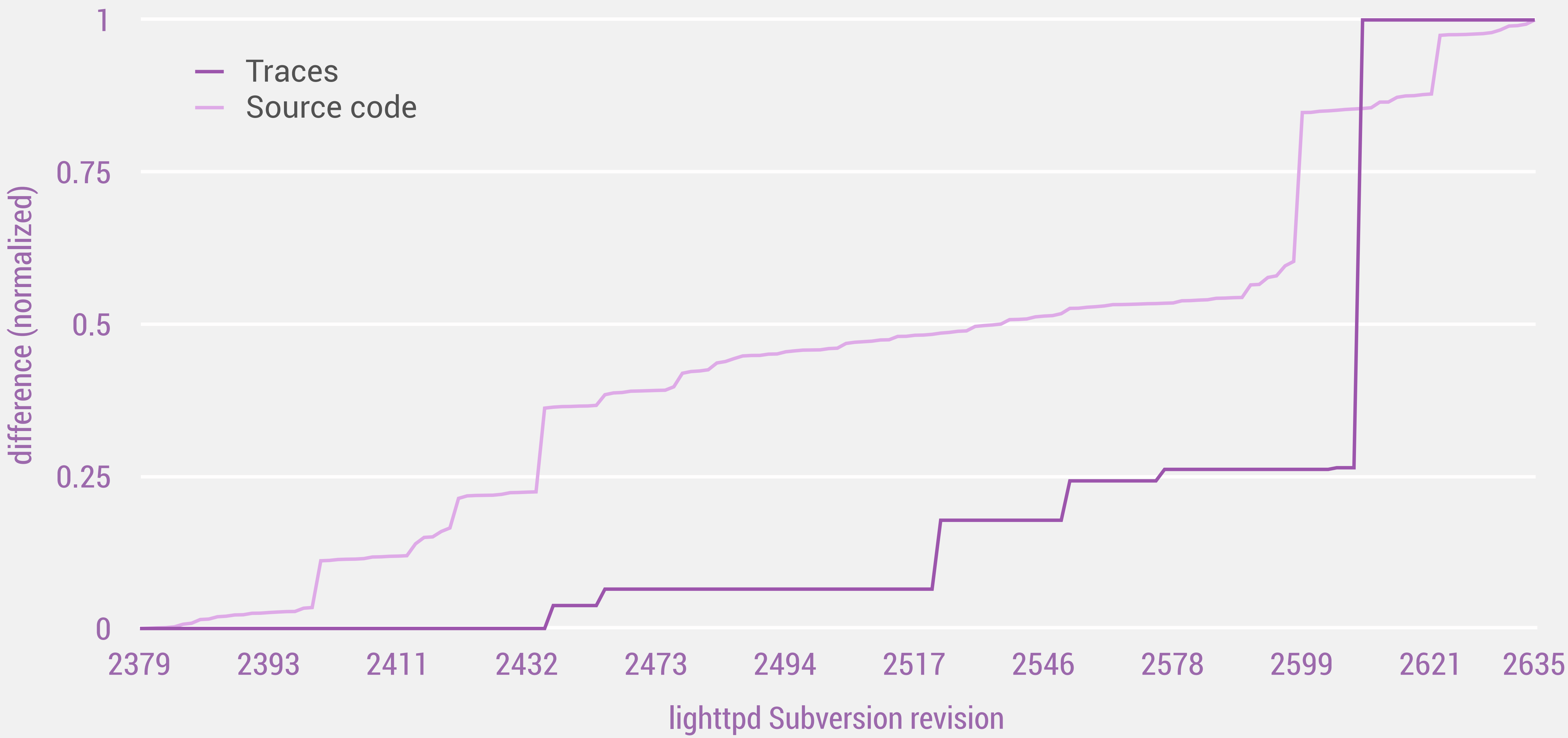
Snippet of system call trace
Obtained using *strace* tool

```
int main(int argc, char **argv)
{
    fib(5);
    fib(6);
}
```

Example testing code
Tested with both implementations

External behaviour **evolves sporadically**

95% of revisions introduce *no change*



Measured using *lighttpd* regression suite on 164 revisions

Taken on Linux kernel 2.6.40 and glibc 2.14 using *strace* tool and custom post-processing (details in the paper)

LIGHTTPD **1.4.22**

LIGHTTPD **1.4.23**

Synchronisation and fail-recovery strategy

LIGHTTPD **1.4.22**

LIGHTTPD **1.4.23**

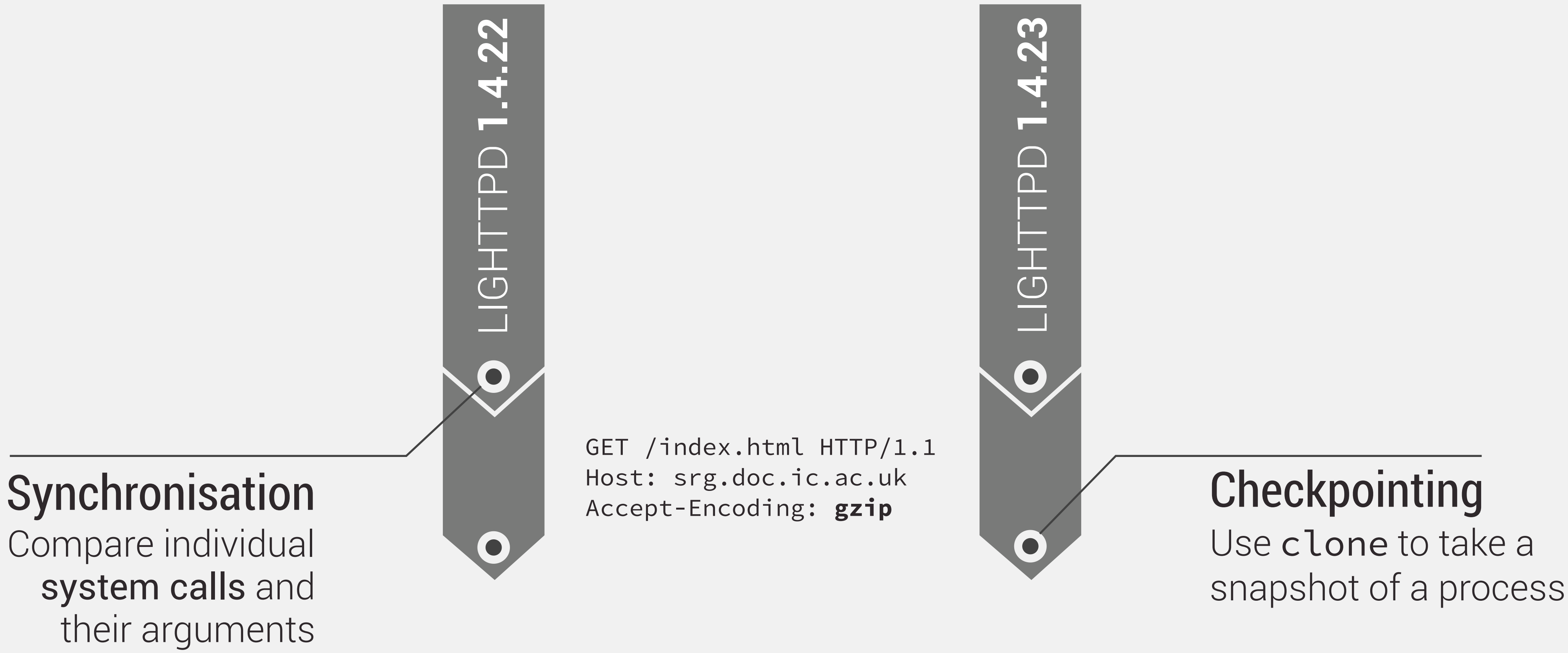
Synchronisation

Compare individual **system calls** and their arguments

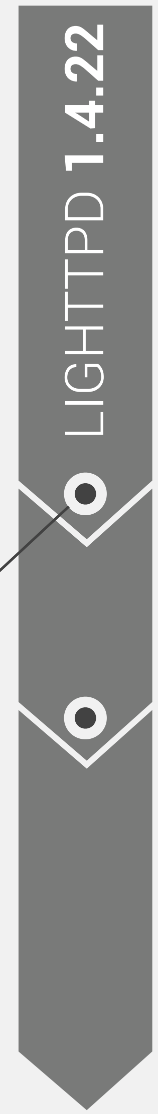
Synchronisation and fail-recovery strategy

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

Synchronisation and fail-recovery strategy



Synchronisation and fail-recovery strategy



Synchronisation

Compare individual **system calls** and their arguments

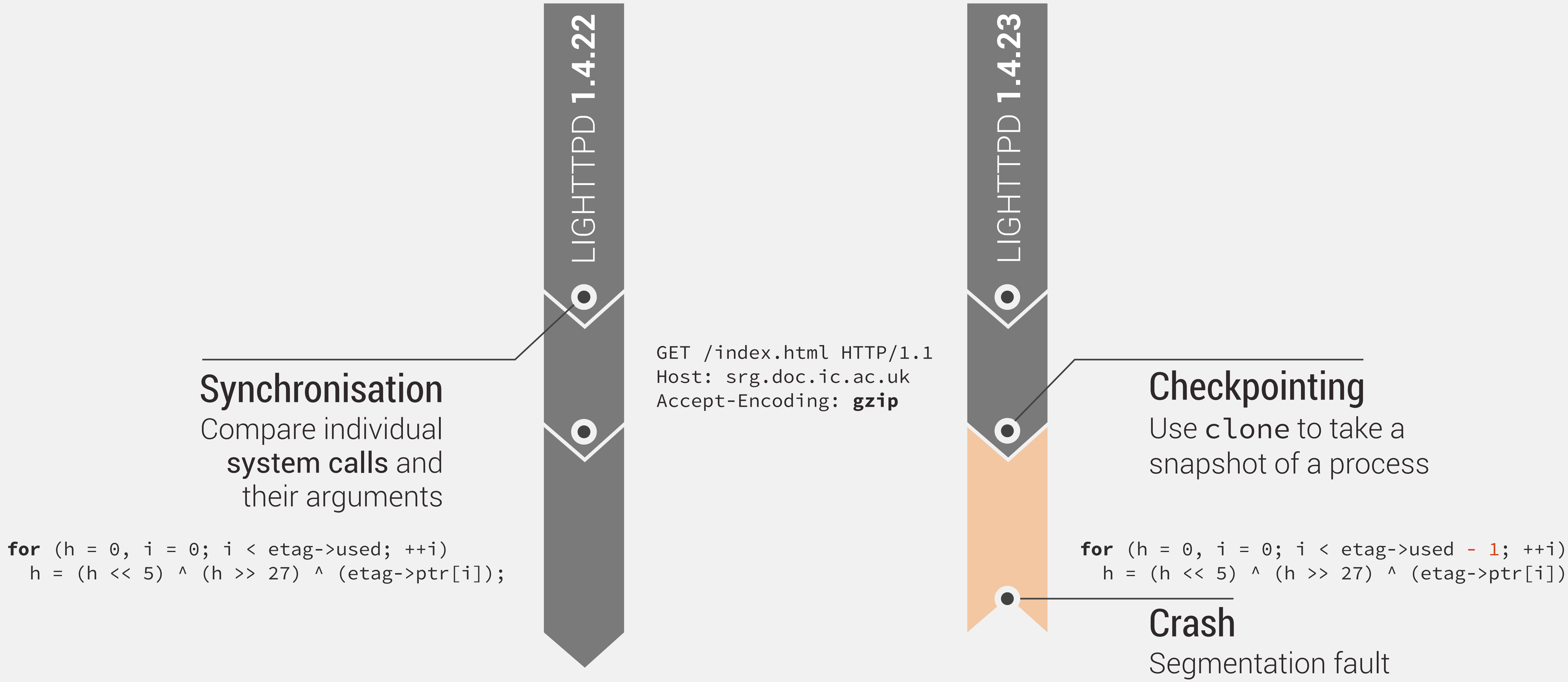
```
for (h = 0, i = 0; i < etag->used; ++i)
  h = (h << 5) ^ (h >> 27) ^ (etag->ptr[i]);
```

```
GET /index.html HTTP/1.1
Host: srg.doc.ic.ac.uk
Accept-Encoding: gzip
```

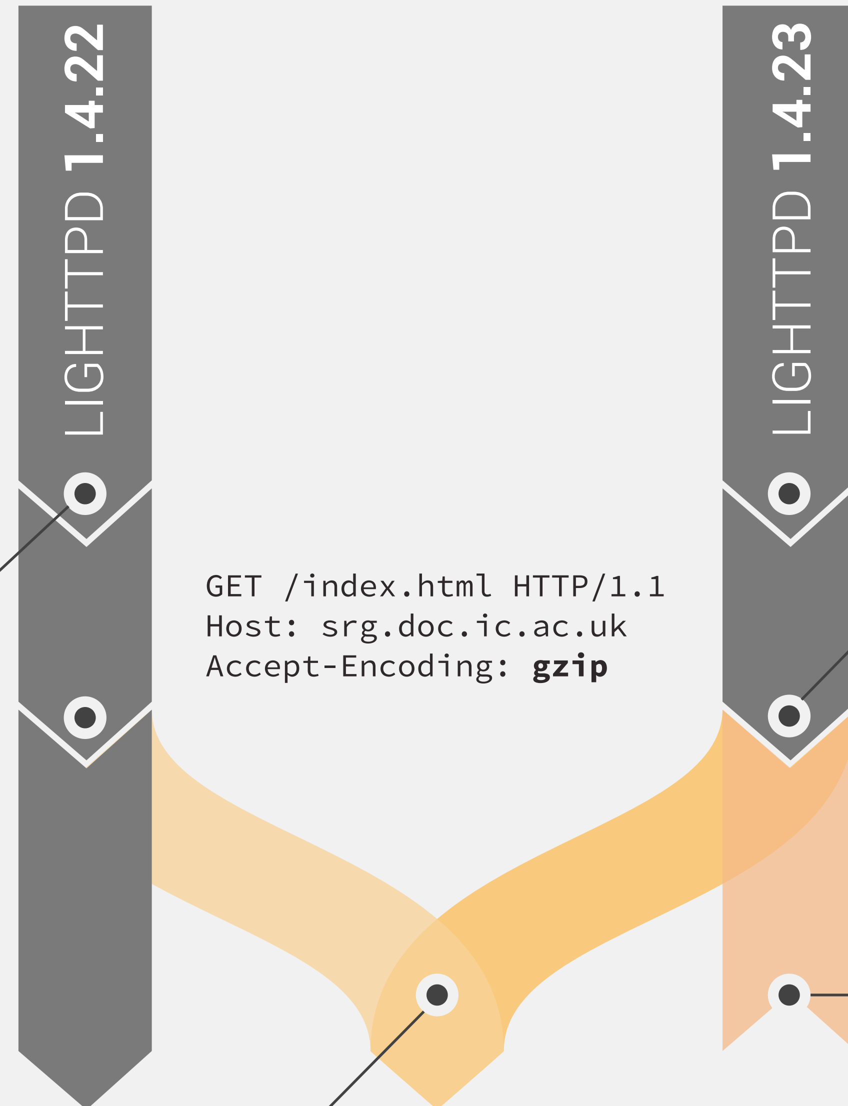
Checkpointing

Use `clone` to take a snapshot of a process

Synchronisation and fail-recovery strategy



Synchronisation and fail-recovery strategy



Synchronisation

Compare individual **system calls** and their arguments

```
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

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

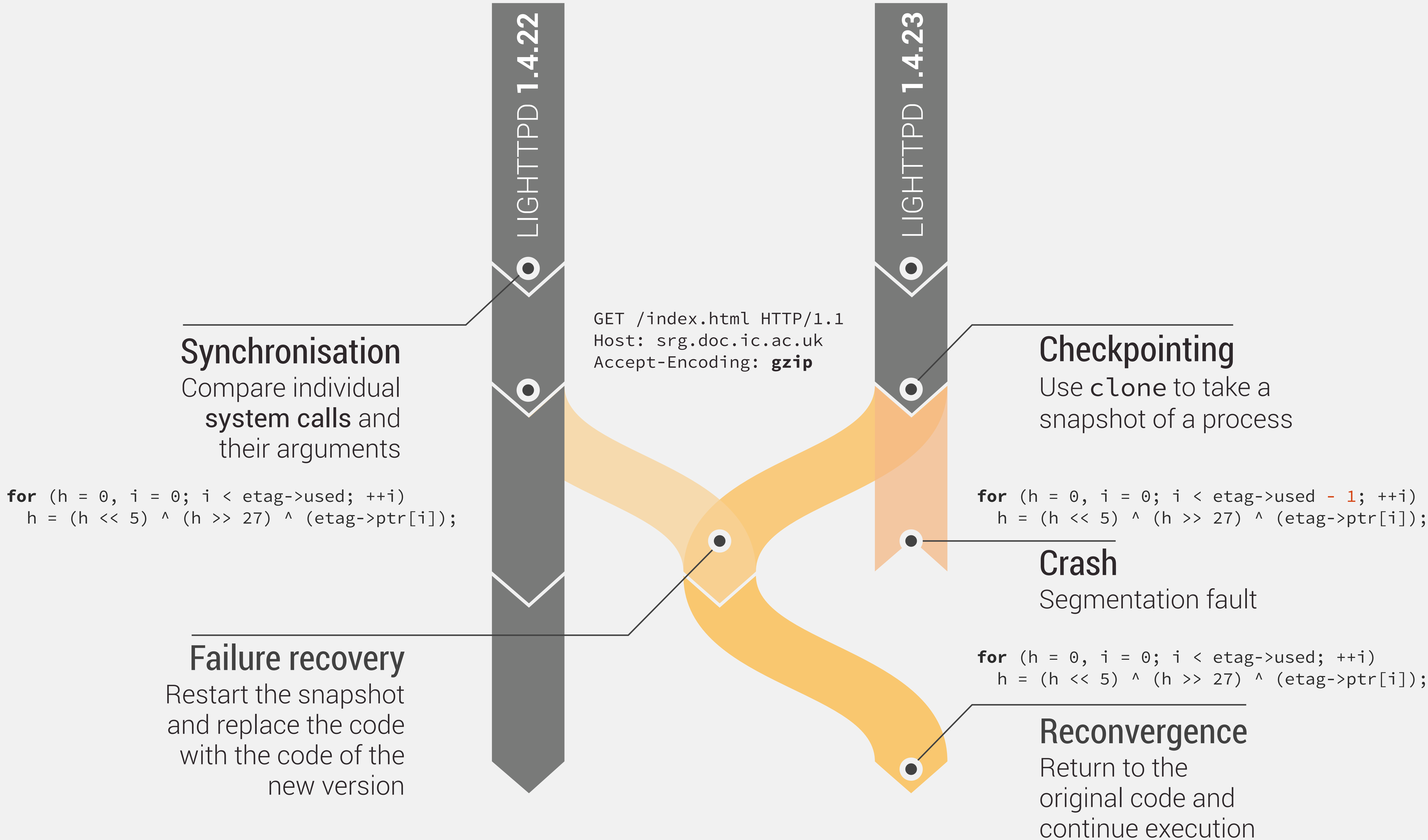
Crash

Segmentation fault

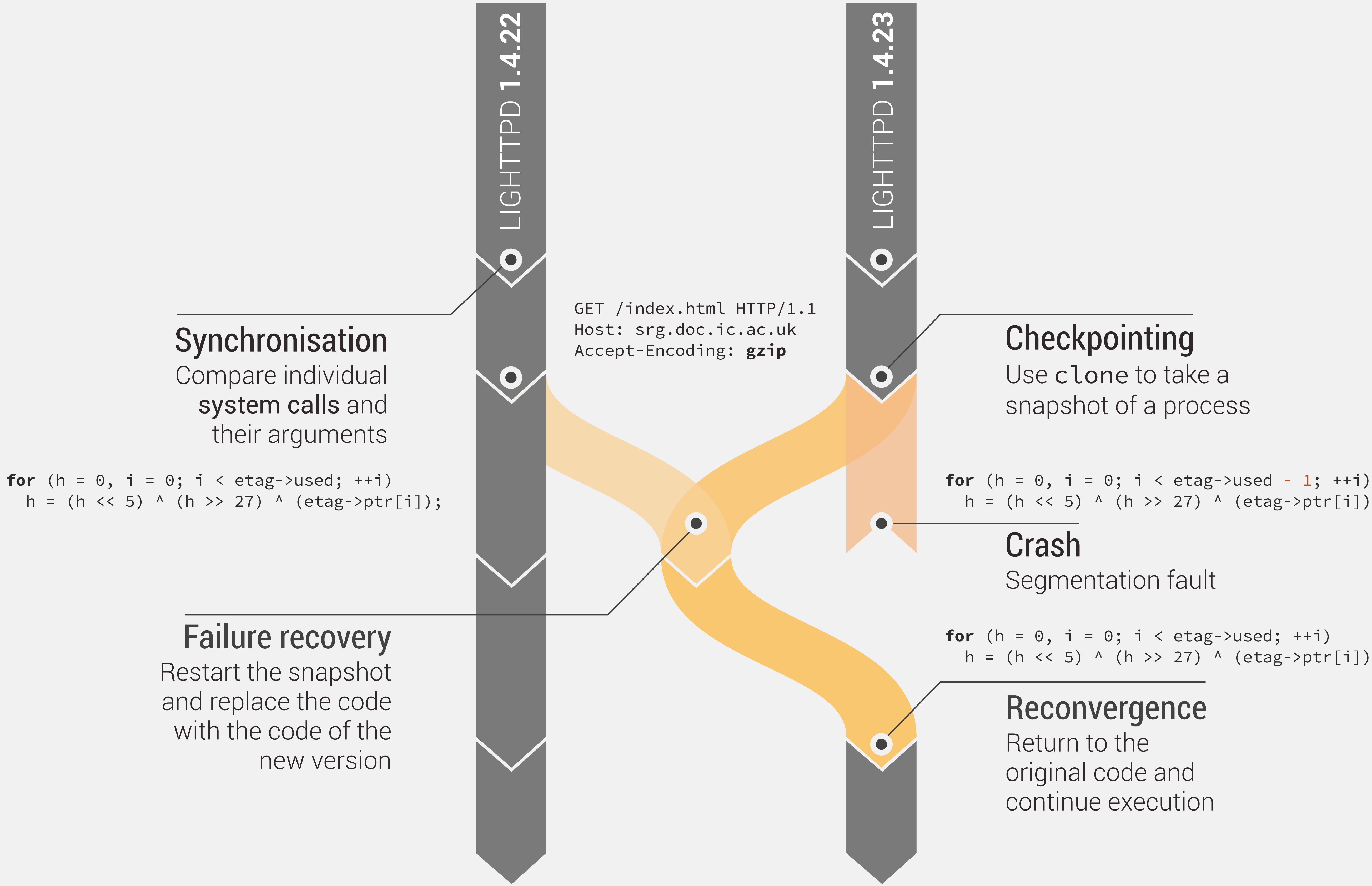
Failure recovery

Restart the snapshot and replace the code with the code of the new version

Synchronisation and fail-recovery strategy



Synchronisation and fail-recovery strategy



Synchronisation and fail-recovery strategy

Recover crashing version using the state of the other one:

Assumes small bug *propagation distance*

Crashes are the only type of observable divergences

The non-crashing version used as an *oracle*

Guarantees

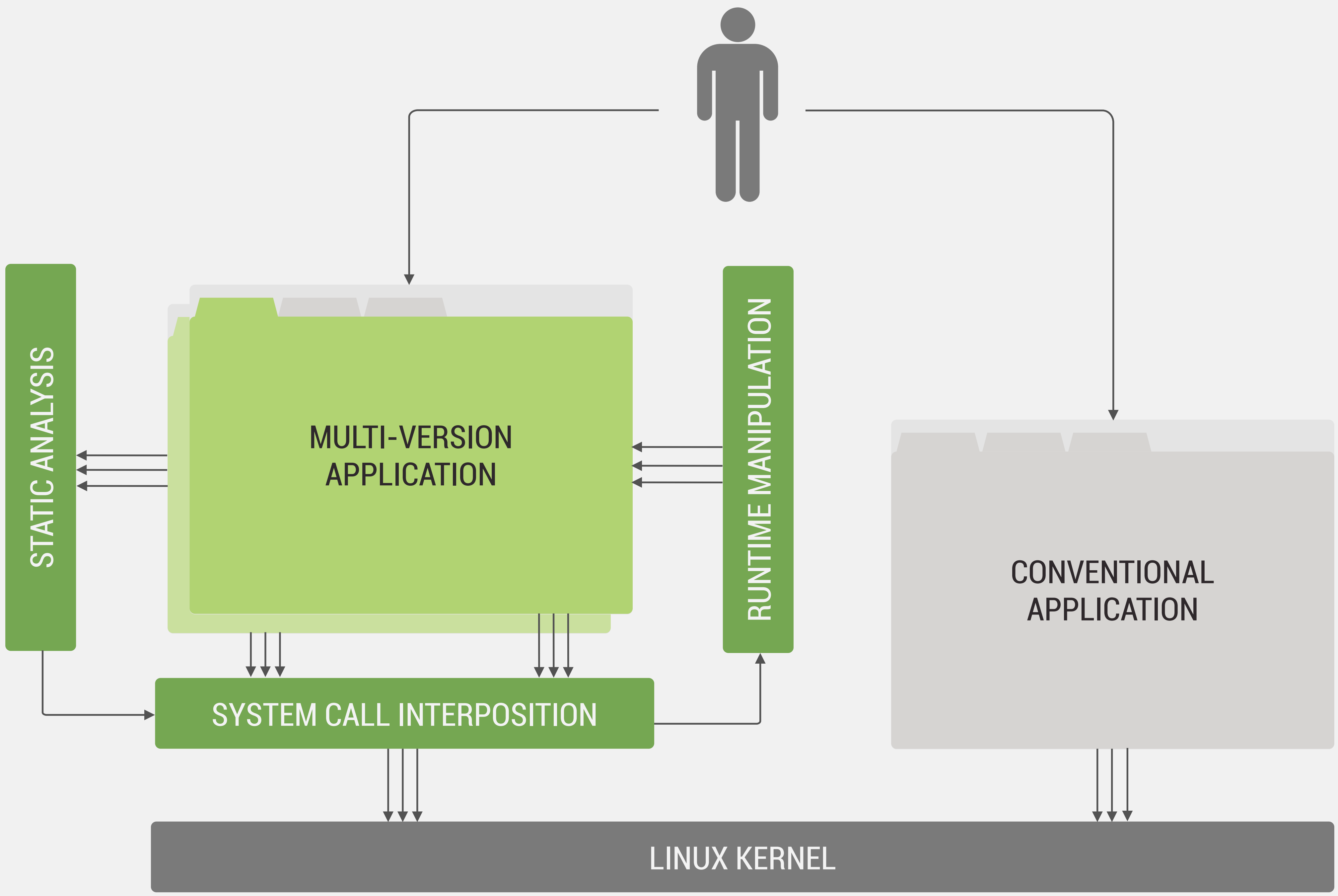
Recovery is successful if versions exhibit the same externally observable behaviour after recovery:

- If unrecoverable, continue with the non-crashing version
- Do not attempt to survive errors we cannot handle

Suitable for type of changes and applications:

- Changes which do not affect memory layout
  - *e.g., refactorings, security patches*
- Applications which provide synchronisation points
  - *e.g., servers structured around the main dispatch loop*
- Where reliability is more important than performance
  - *e.g., interactive apps, some server scenarios*

Mx architecture



Implementation for x86 and x86-64 Linux

Combines binary static analysis, lightweight checkpointing and runtime code patching

Completely transparent, runs on unmodified binaries

Runs two versions with small differences in behaviour

Focus on application crashes and recovery

Multi-eXecution Monitor

Execute and monitor multi-version applications:

- Intercepting system calls (via `ptrace` interface)
- Semantically comparing system calls arguments
- Environment virtualisation (e.g. files and sockets)

Runtime Execution Manipulator

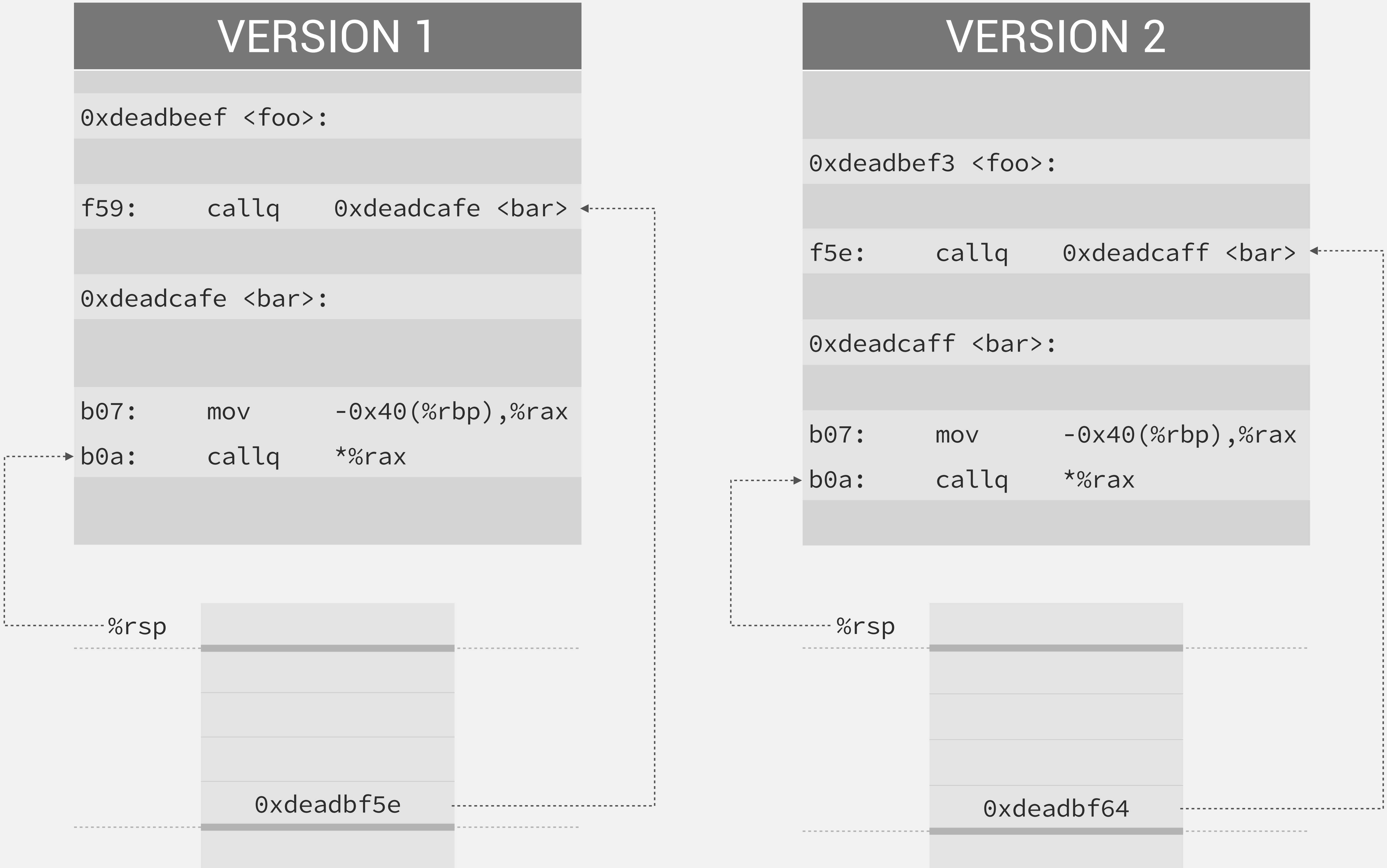
Runtime code patching and fault recovery:

- OS-level checkpointing (using `clone` syscall)
- Runtime stack rewriting (`libunwind`)
- Breakpoint insertion and handling

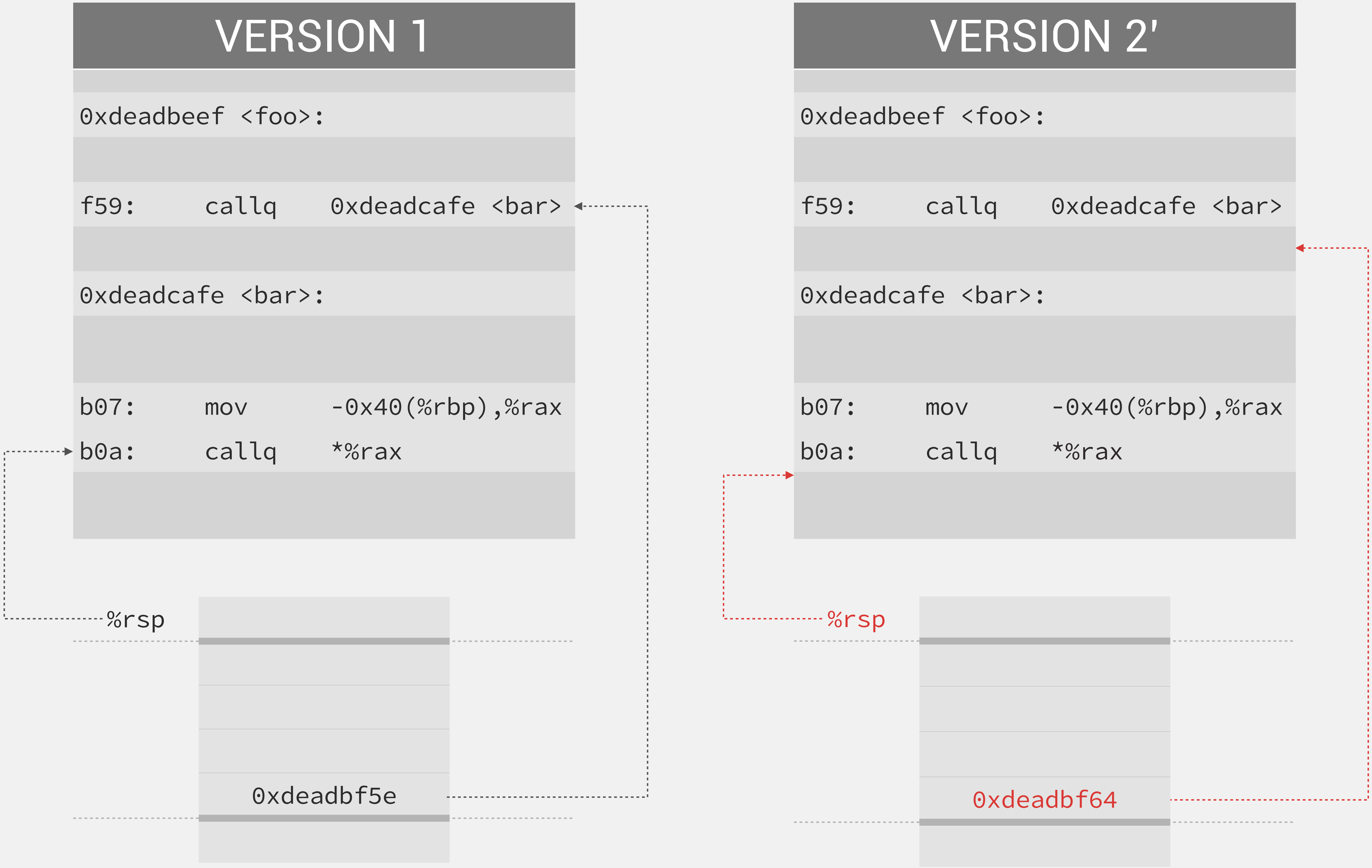
Static Executable Analyser

Create various mappings between the two version binaries:

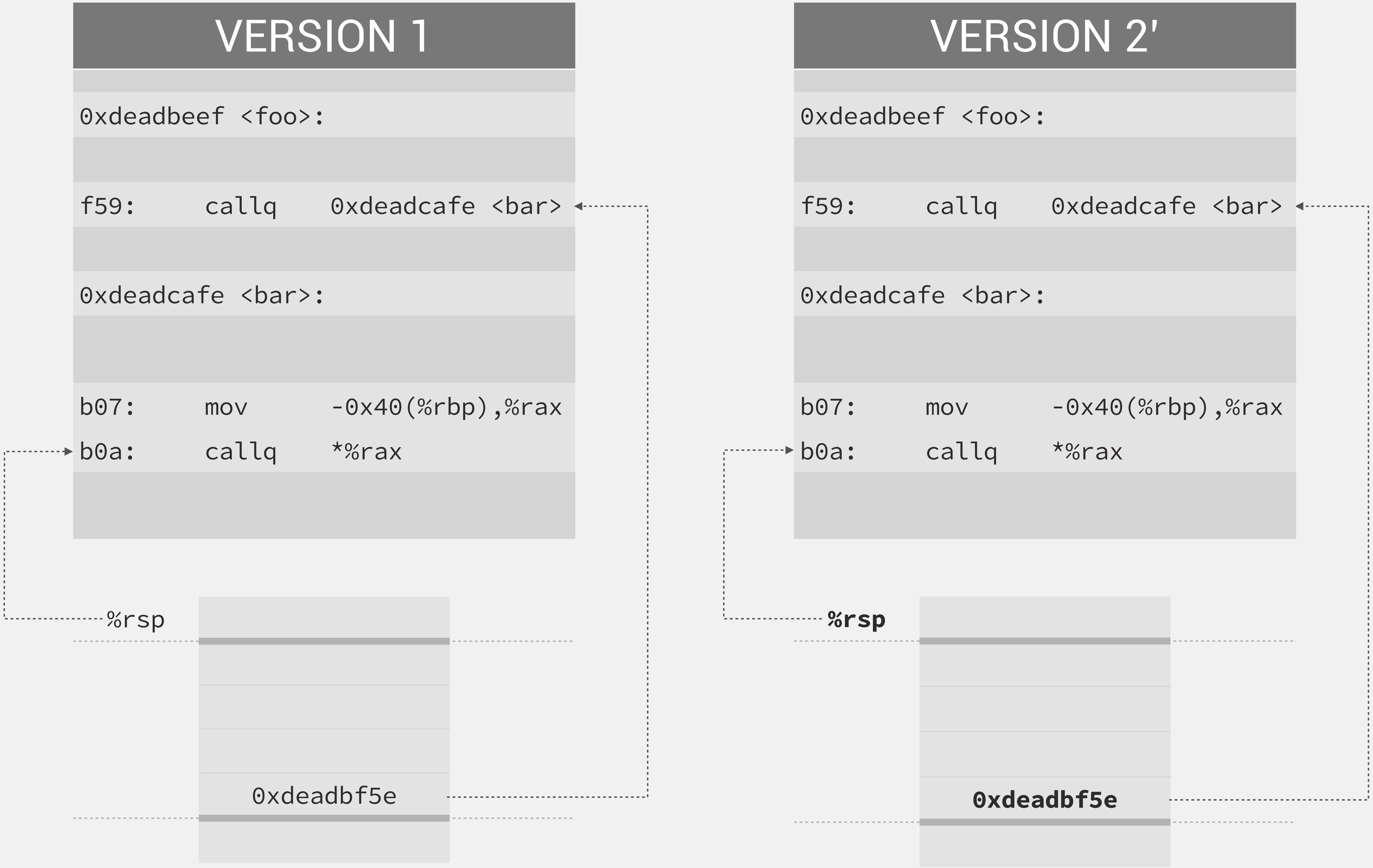
- Extracting function symbols from binaries (`libbfd`)
- Machine code disassembling and analysis (`libopcodes`)
- Binary call graph reconstruction and matching



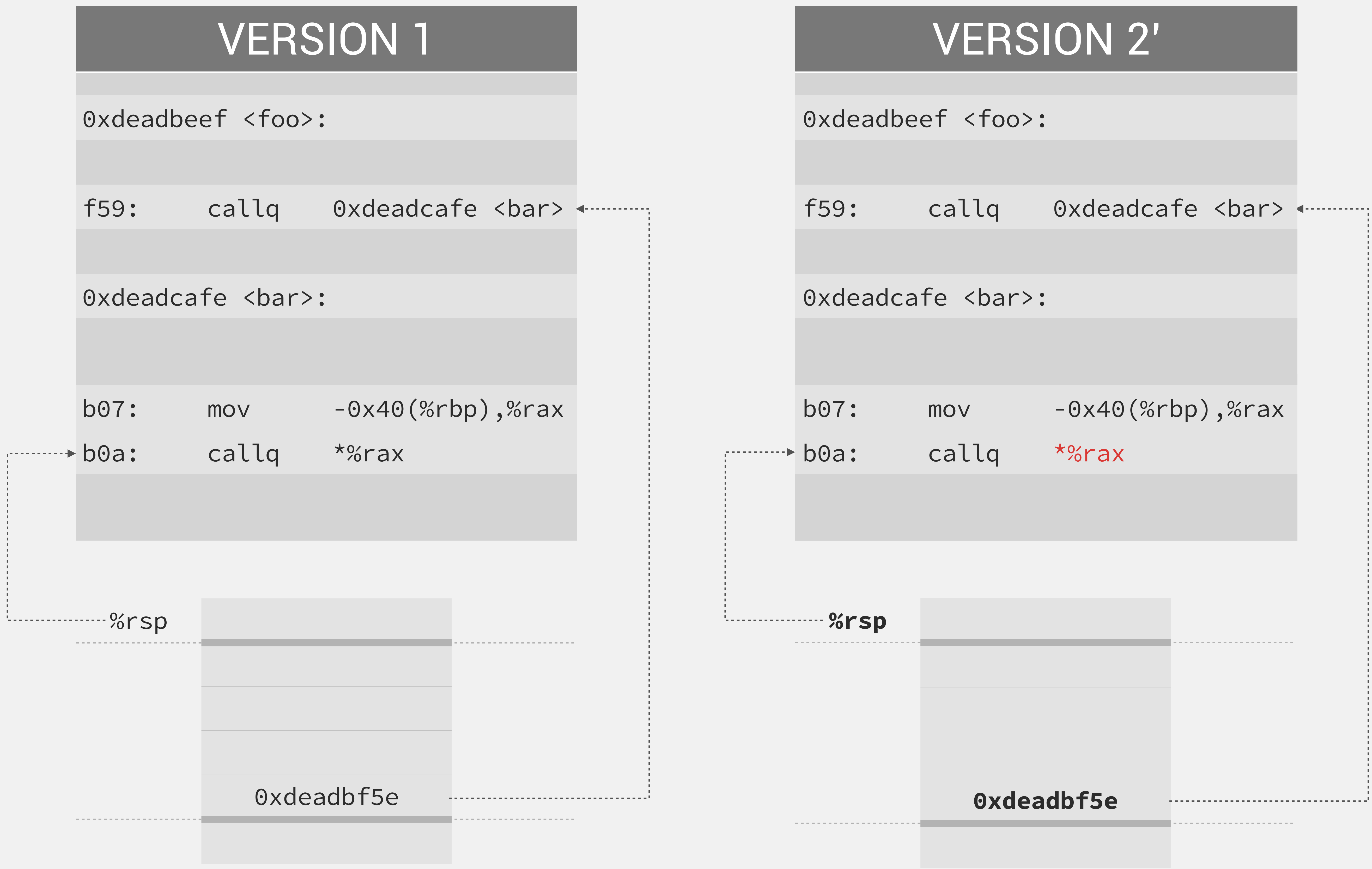
Execution stack rewriting



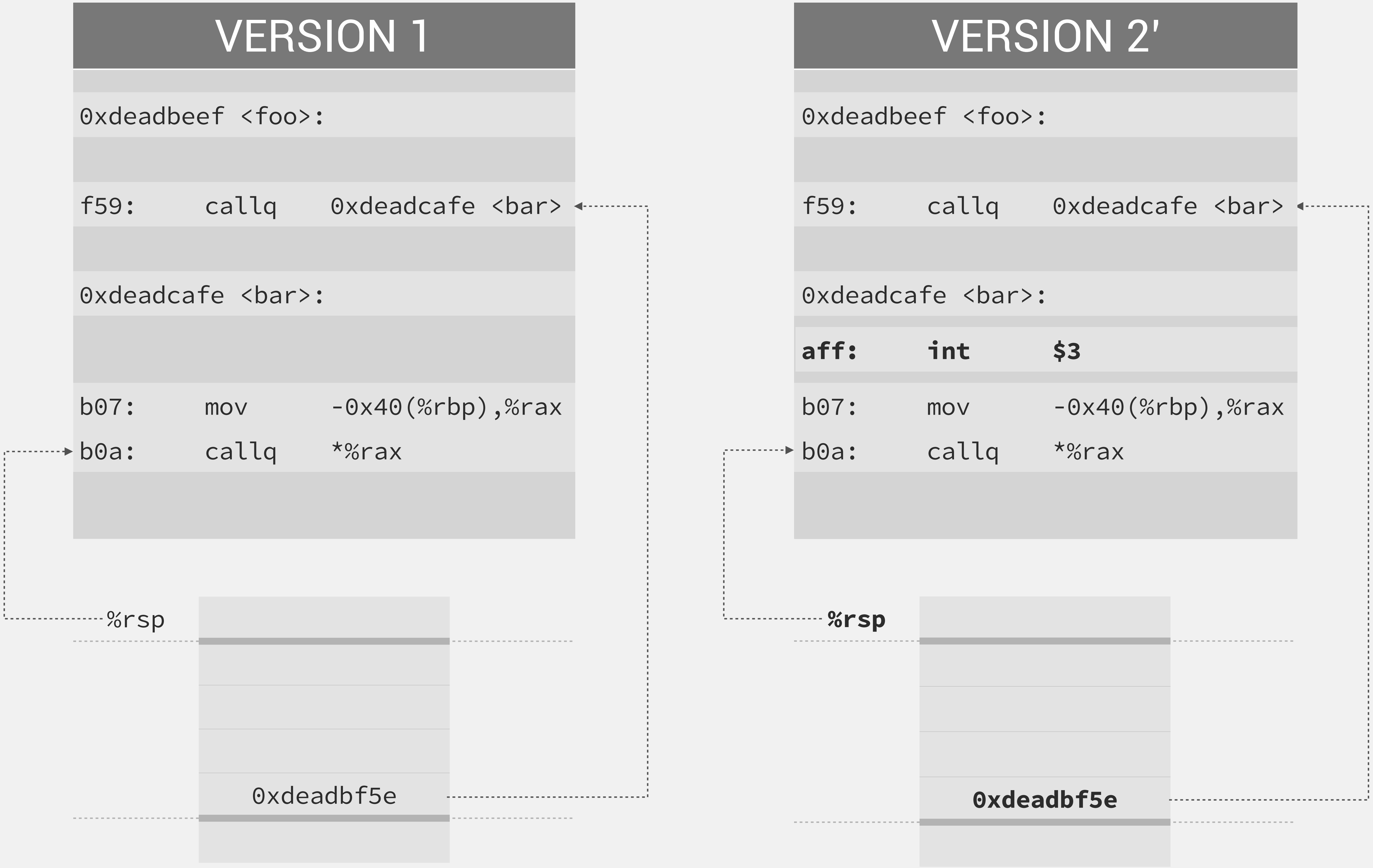
Execution stack rewriting



Execution stack rewriting



Execution stack rewriting



Execution stack rewriting

Survived a number of crash bugs
in several popular server applications

redis

In-memory NoSQL database

```
robj *o = lookupKeyRead(c->db, c->argv[1]);
if (o == NULL) {
    addReplySds(c,sdscatprintf(sdsempty(),
      "*%d\r\n",c->argc-2));
    for (i = 2; i < c->argc; i++) {
        addReply(c,shared.nullbulk);
    }
    return;
} else {
    if (o->type != REDIS_HASH) {
        addReply(c,shared.wrongtypeerr);
        return;
    }
}
addReplySds(c,sdscatprintf(sdsempty(),
  "*%d\r\n",c->argc-2));
```

Redis regression bug #344 introduced during refactoring
HMGET command implementation in `hmgetCommand` function

Survived a number of crash bugs
in several popular server applications

redis

In-memory NoSQL database

```
robj *o, *value;
o = lookupKeyRead(c->db,c->argv[1]);
if (o != NULL && o->type != REDIS_HASH) {
    addReply(c,shared.wrongtypeerr);
    return;
}
addReplySds(c,sdscatprintf(sdsempty(),
  "*%d\r\n",c->argc-2));
for (i = 2; i < c->argc; i++) {
    if (o != NULL && (value =
          hashGet(o,c->argv[i])) != NULL) {
        addReplyBulk(c,value);
        decrRefCount(value);
    } else {
        addReply(c,shared.nullbulk);
    }
}
```

~~return;~~ — Missing `return` statement

Redis regression bug #344 introduced during refactoring
HMGET command implementation in `hmgetCommand` function

Interactive applications:

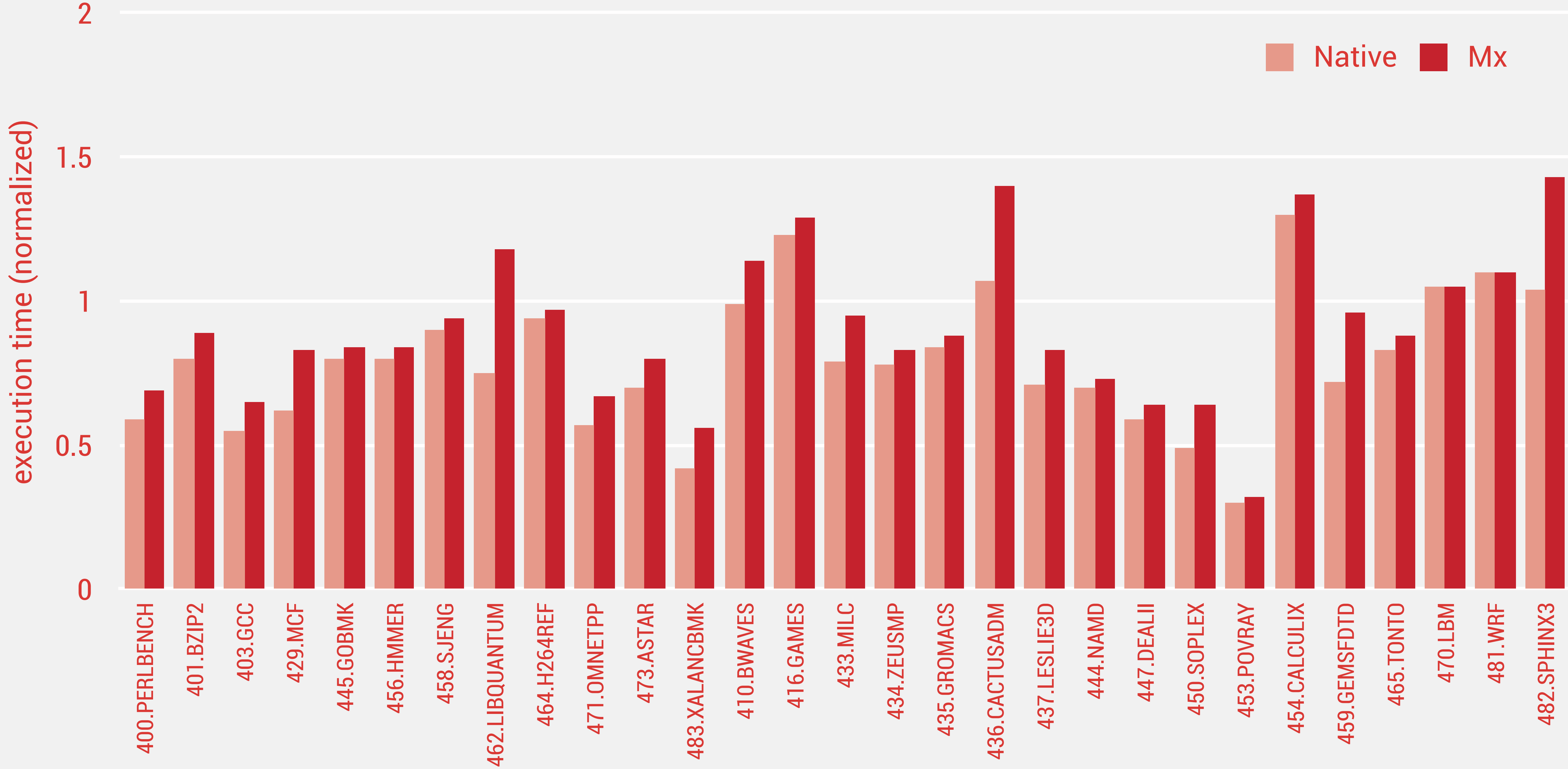
UTILITY	BUG	TIME SPAN
md5sum sha1sum	Buffer underflow	1,124 revs. (1 year 7 months)
mkdir mkfifo mknod	`NULL`-pointer dereference	2,937 revs. (over 4 years)
cut	Buffer overflow	1,201 revs. (2 years 3 months)

Server applications:

APPLICATION/ISSUE	BUG	TIME SPAN
lighttpd #2169	Loop index underflow	87 revs. (2 months 2 days)
lighttpd #2140	Off-by-one error	12 revs. (2 months 1 day)
redis #344	Missing `return` statement	27 revs. (6 days)

17.91% overhead on SPEC CPU2006

over single version despite **2x** utilisation cost



Measured using SPEC CPU2006 1.2

Taken on 3.50 GHz Intel Xeon E3 1280 with 16 GB of RAM, Linux kernel 3.1.9

Interactive applications:

UTILITY	INPUT SIZE	OVERHEAD
md5sum sha1sum	<1.25MB	<100ms (imperceptible)
mkdir mkfifo mknod	<115 nested directories	<100ms (imperceptible)
cut	<1.10MB	<100ms (imperceptible)

Measured using Coreutils 6.10
Taken on 3.50 GHz Intel Xeon E3 1280 with 16 GB of RAM, Linux kernel 3.1.9

Server applications:

APPLICATION	SCENARIO	OVERHEAD
lighttpd	localhost/network	2.60x – 3.49x
	distant networks	1.01x – 1.04x
redis	localhost/network	3.74x – 16.72x
	distant networks	1.00x – 1.05x

Measured using redis-benchmark and http_load
Taken on 3.50 GHz Intel Xeon E3 1280 with 16 GB of RAM, Linux kernel 3.1.9

Summary

Novel approach for improving software updates:

Based on multi-version execution

Mx can survive crash bugs in real apps

Many opportunities for future work:

Better performance overhead

Tolerance to system call divergencies

Support for more complex code changes

Support for non-crashing type of divergences