A DSL Approach to Reconcile Equivalent Divergent Program Executions

Luís Pina
Daniel Grumberg
Anastasios Andronidis
Cristian Cadar

{l.pina / daniel.grumberg14 / a.andronidis15 / c.cadar}@imperial.ac.uk
Imperial College London
London, UK

July 13th, 2017

2017 USENIX Annual Technical Conference (ATC)
What are “Equivalent Divergent Program Executions”? And why should I care about reconciling them?
Equivalent Divergent Program Executions

>./hello1
Hello world

>./hello2
Hello world
Equivalent Divergent Program Executions

>./hello1
Hello world
>ldd hello1
libc.so.6

>./hello2
Hello world
>ldd hello2
libc.so.6
jemalloc.so.6
Equivalent Divergent Program Executions

>./hello1
Hello world
>ldd hello1
libc.so.6

>strace hello1
write(1, "Hello world", 11)

>./hello2
Hello world
>ldd hello2
libc.so.6
jemalloc.so.6

>strace hello2
write(1, "Hello ", 6)
write(1, "world", 5)
Multi-Version Execution (MVE)

Run multiple versions as one
Multi-Version Execution (MVE)

Run multiple versions as one

- Improves reliability
  
<table>
<thead>
<tr>
<th>malloc(WEIRD_NUMBER)</th>
<th>jemalloc(WEIRD_NUMBER)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEGFAULT</td>
<td>OK</td>
</tr>
</tbody>
</table>
Multi-Version Execution (MVE)

Run multiple versions as one

- Improves reliability
  - `malloc(WEIRD_NUMBER)` vs. `jemalloc(WEIRD_NUMBER)`
  - SEGFAULT vs. OK

- Improves security
  - `>strace hello1`
    - write(1,"Hello world")
  - `>strace hello2`
    - write(1,"Hello, ")
    - fork()
    - execve("/bin/sh")
    - write(1,"world!")
Multi-Version Execution (MVE)

Run multiple versions as one

- Improves reliability
  
  malloc(WEIRD_NUMBER)  jemalloc(WEIRD_NUMBER)
  SEGFAULT  OK

- Improves security
  
  >strace hello1
  write(1,"Hello world")

  >strace hello2
  write(1,"Hello, ")
  fork()
  execve("/bin/sh")
  write(1,"world!")

Versions should be diverse but equivalent
Versions should be **diverse** but **equivalent**

What about **equivalent** executions that issue **divergent** sequences of system calls?

```
> strace hello1
write(1,"Hello world", 11)
```

```
> strace hello2
write(1,"Hello ", 6)
write(1,"world" , 5)
```
Versions should be **dive**rse but **equi**valent

What about **equivalent** executions that issue **dive**rgent sequences of system calls?

```
>strace hello1
write(1,"Hello world", 11)

>strace hello2
write(1,"Hello ", 6)
write(1,"world" , 5)
```

Describe the divergences with a **Domain Specific Language (DSL)**
Versions should be diverse but equivalent. What about equivalent executions that issue divergent sequences of system calls?

```
> strace hello1
write(1,"Hello world", 11)
=> write(1,"Hello ", 6)
write(1,"world", 5)
```
Versions should be diverse but equivalent.

What about equivalent executions that issue divergent sequences of system calls?

```
> strace hello1
> strace hello2
write(1,"Hello world", 11)  =>  write(1,"Hello ", 6)
write(1,"world" , 5)
```
MVE Architecture

Varan

- User
- Leader
- Shared Memory
- Follower
- OS Kernel

Recorded

Replayed

DSL

read(0,_,128) = 6 // "foobar"

read(0,_,128)

read(0,_,128)
MVE Architecture

Varan

Varan

Shared Memory

Leader

User

read(0, _, 128)

OS Kernel

Follower

Recorded

Replayed

DSL
MVE Architecture

Varan

Varan

Shared Memory

read(0, _, 128)

Leader

Follower

User

read(0, _, 128)

OS Kernel

Recorded

Replayed

DSL
MVE Architecture

Varan

User \arrow{ Leader } \arrow{ Varan } \arrow{ Shared Memory }

read(0, _, 128) = 6 // "foobaz"

OS Kernel
MVE Architecture

Varan

User

Leader

Shared Memory

read(0, _, 128)

6, "foobar"

Follower

OS Kernel
MVE Architecture

Varan

Varan

Shared Memory

read(0,_,128)

6, "foobar"

Recorded

Replayed

DSL

19
MVE Architecture

Varan

**Varan**

- **Leader**
  - `read(0, _, 128)`
  - 6, "foobar"

- **Shared Memory**
  - `read(0, _, 128) = 6 // "foobar"`

- **Follower**

**User**

**OS Kernel**
MVE Architecture

Varan

Varan

Shared Memory

Leader

read(0, _, 128)

6, "foobar"

Follower

OS Kernel
MVE Architecture

Varan

```
read(0, _, 128)
6, "foobar"
...  
```

Recorded
Replayed

DSL
MVE Architecture

Varan

Varan

Shared Memory

Recorded

... 

read(0, _, 128)

... 6, "foobar"

... 

Replayed

OS Kernel

Leader

Follower

User

read(0, _, 128) = 6 // "foobar"

...
DSL Architecture

Recorded

... read "foobar"

Replayed

... read "foobar"

DSL

write(1, "Hello world", 11) => write(1, "Hello ", 6), write(1, "world", 5)
DSL Architecture

Recorded

... read "foobart"

Replayed

... read "foobart"

match

match

DSL

write(1, "Hello world", 11) => write(1, "Hello ", 6), write(1, "world", 5)

1. DSL

store

nop

exec

sched_yield

2. DSL

store

nop

exec

sigaction

sig1

sigaction

sig2

? sigaction

sig2

sigaction

sig1

match 1

match

25
DSL Architecture

Recorded

...  
read  
"foobar"

Replayed

...  
read  
"foobar"

write(1, "Hello world", 11) => write(1, "Hello ", 6), write(1, "world", 5)
DSL Rules

- **Default rule**: `read(_,_,_) as r => r`

- **Actions**
  - MATCH
  - NOP
  - SKIP
  - EXECUTE
  - STORE

- **Further examples**
  - Hello world
  - `nothing` keyword
  - C predicates
write(1, "Hello world", 11) => write(1, "Hello ", 6),
    write(1, "world" , 5)
Hello World Rule

write(1, "Hello world", 11) => write(1, "Hello ", 6),
write(1, "world" , 5)
Hello World Rule

write(1, "Hello world", 11) => write(1, "Hello ", 6), write(1, "world" , 5)
Hello World Rule

write(1, "Hello world", 11) => write(1, "Hello ", 6),
write(1, "world", 5)
Hello World Rule

write(1, "Hello world", 11) => write(1, "Hello ", 6),
write(1, "world", 5)
Hello World Rule

write(1, "Hello world", 11) => write(1, "Hello ", 6), write(1, "world" , 5)
Hello World Rule

Recorded

... 
read "foobar"
write "Hello world"

Replayed

... 
read "foobar"
write "Hello"
write "world"

write(1, "Hello world", 11) => write(1, "Hello ", 6), write(1, "world", 5)
Hello World Rule

Recorded

<table>
<thead>
<tr>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
</tr>
<tr>
<td>write</td>
</tr>
<tr>
<td>?</td>
</tr>
</tbody>
</table>

Replayed

<table>
<thead>
<tr>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
</tr>
<tr>
<td>write</td>
</tr>
<tr>
<td>write</td>
</tr>
<tr>
<td>sched_yield</td>
</tr>
</tbody>
</table>
nothing => sched_yield()
nothing Keyword

```
nothing => sched_yield()
```
nothing => sched_yield()
nothing => sched_yield()
<table>
<thead>
<tr>
<th>Recorded</th>
<th>Replayed</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>read &quot;foobar&quot;</td>
<td>read &quot;foobar&quot;</td>
</tr>
<tr>
<td>write &quot;Hello world&quot;</td>
<td>write &quot;Hello &quot;</td>
</tr>
<tr>
<td>sigaction sig1</td>
<td>write &quot;world&quot;</td>
</tr>
<tr>
<td>sigaction sig2</td>
<td>sched_yield</td>
</tr>
<tr>
<td>?</td>
<td>sigaction sig2</td>
</tr>
<tr>
<td></td>
<td>sigaction sig1</td>
</tr>
<tr>
<td></td>
<td>sigaction sig1</td>
</tr>
</tbody>
</table>
C Predicates
and multiple left-hand side

// extern int sig1, sig2;
sigact(sig,_,_) { $(sig) == sig1; } as s1,
sigact(sig,_,_) { $(sig) == sig2; } as s2  =>  s2, s1
sigact(sig,_,_) { $(sig) == sig1; } as s1,
sigact(sig,_,_) { $(sig) == sig2; } as s2 => s2, s1
sigact(sig,_,_) { $(sig) == sig1; } as s1,
sigact(sig,_,_) { $(sig) == sig2; } as s2 => s2, s1
C Predicates

```
sigact(sig,_,_) { $(sig) == sig1; } as s1,
sigact(sig,_,_) { $(sig) == sig2; } as s2  =>  s2, s1
```
C Predicates

```c
sigact(sig,_,_) { $(sig) == sig1; } as s1,
sigact(sig,_,_) { $(sig) == sig2; } as s2 => s2, s1
```
C Predicates

Recorded

1. read "foobar"
2. write "Hello world"

sigaction sig1

sigaction sig2

? 

Replayed

read "foobar"
write "Hello world"
sched_yield

sigaction sig2

sigaction sig1

DSL

match 1

? 

match

sigact(sig,_,_) { $(sig) == sig1; } as s1,
sigact(sig,_,_) { $(sig) == sig2; } as s2 => s2, s1
### C Predicates

<table>
<thead>
<tr>
<th>Recorded</th>
<th>Replayed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>read &quot;fooboo&quot;</td>
<td>read &quot;fooboo&quot;</td>
</tr>
<tr>
<td>write &quot;Hello world&quot;</td>
<td>write &quot;Hello &quot;</td>
</tr>
<tr>
<td>sigaction sig1</td>
<td>write &quot;world&quot;</td>
</tr>
<tr>
<td>sigaction sig2</td>
<td>sched_yield</td>
</tr>
<tr>
<td>?</td>
<td>sigaction sig2</td>
</tr>
<tr>
<td></td>
<td>sigaction sig1</td>
</tr>
</tbody>
</table>

```plaintext
dsигact(sig,_,_) { $(sig) == sig1; } as s1, 
                 sigact(sig,_,_) { $(sig) == sig2; } as s2  =>  s2, s1
```
Deployment scenarios

- Different configurations
- Different releases
- Different dynamic analyses
Deployment scenarios
Different configurations

- Recorded  Redis minimal config
- Replayed 1  Redis with persistency (3 rules)
- Replayed 2  Redis with verbose logs (4 rules)
- Replayed 3  Redis with persistency and verbose logs (7 rules)
## Deployment scenarios

### Different releases

<table>
<thead>
<tr>
<th>ID</th>
<th>Redis Versions</th>
<th>Commits</th>
<th>Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.3.8 – 1.3.10</td>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1.3.10 – 1.3.12</td>
<td>105</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1.3.12 – 2.0.0</td>
<td>92</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2.0.0 – 2.0.5</td>
<td>34</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>2.0.5 – 2.2.0</td>
<td>730</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>2.2.0 – 2.2.15</td>
<td>110</td>
<td>2</td>
</tr>
</tbody>
</table>
Deployment scenarios

Analyses

Recorded  Native
Replayed 1  Asan (3 rules)
Replayed 2  Msan (1 rule)
Replayed 3  Tsan (5 rules)
Replayed 4  Valgrind (14 rules)¹

¹Expands to 31 rules through group syntactic-sugar
Deployment scenarios
Analyses

Recorded  Native
Replayed 1  Asan (3 rules)
Replayed 2  Msan (1 rule)
Replayed 3  Tsan (5 rules)
Replayed 4  Valgrind (14 rules)\(^1\)

\(^1\)Expands to 31 rules through \textit{group} syntactic-sugar

- \texttt{git} (log, blame, diff, tag)
- \texttt{openssh} (ssh, ssh-keygen)
- \texttt{htop}
- \texttt{vim}
Deployment scenarios

Analyses

Recorded  Native
Replayed 1  Asan (3 rules)
Replayed 2  Msan (1 rule)
Replayed 3  Tsan (5 rules)
Replayed 4  Valgrind (14 rules)\(^1\)

\(^1\)Expands to 31 rules through group syntactic-sugar

\(\times\)

\(\text{git (log, blame, diff, tag)}\)
\(\text{openssh (ssh, ssh-keygen)}\)
\(\text{htop}\)
\(\text{vim}\)
Finding these rules must be hard...
Finding these rules must be hard...

It isn’t
Finding Rules

1. strace -o native.log native
2. strace -o valgrind.log valgrind
3. vimdiff native.log valgrind.log
Finding Rules

vimdiff

<table>
<thead>
<tr>
<th>native.log</th>
<th>valgrind.log</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(3, ..., 4096)</td>
<td>gettid()</td>
</tr>
<tr>
<td>lseek(3, -2347, SEEK_CUR)</td>
<td>write(1029, &quot;D&quot;, 1)</td>
</tr>
<tr>
<td>read(3, ..., 4096)</td>
<td>sigprocmask([], ~[...])</td>
</tr>
<tr>
<td>lseek(3, -2347, SEEK_CUR)</td>
<td>read(3, ..., 4096)</td>
</tr>
<tr>
<td>gettid()</td>
<td>sigprocmask(~[...], NULL)</td>
</tr>
<tr>
<td>read(1028, &quot;D&quot;, 1)</td>
<td>gettid()</td>
</tr>
<tr>
<td>lseek(3, -2347, SEEK_CUR)</td>
<td>sigprocmask([], ~[...])</td>
</tr>
<tr>
<td>read(3, ..., 4096)</td>
<td>read(3, ..., 4096)</td>
</tr>
<tr>
<td>sigprocmask([...], NULL)</td>
<td>sigprocmask( [...], NULL)</td>
</tr>
<tr>
<td>gettid()</td>
<td>gettid()</td>
</tr>
<tr>
<td>read(1028, &quot;E&quot;, 1)</td>
<td>read(1028, &quot;E&quot;, 1)</td>
</tr>
<tr>
<td>close(3)</td>
<td>close(3)</td>
</tr>
</tbody>
</table>
Finding Rules

vimdiff

native.log

read(3, ..., 4096)

valgrind.log

gettid()
write(1029, "D", 1)
sigprocmask([], ~[[...]])
read(3, ..., 4096)
sigprocmask(~[[...]], NULL)
gettid()
read(1028, "D", 1)

gettid()
write(1029, "E", 1)
sigprocmask([], ~[[...]])
read(3, ..., 4096)
sigprocmask([ [...]], NULL)
gettid()
read(1028, "E", 1)
Finding Rules

vimdiff

native.log

read(_, _, _) as r =>

read(3, ..., 4096)

valgrind.log

gettid(),
write(1029, _, 1),
sigprocmask(_, _),
r,
sigprocmask(_, _),
gettid(),
read(1028, _, 1)

gettid()
write(1029, "E", 1)
sigprocmask([], ~[...])
read(3, ..., 4096)
sigprocmask([...], NULL)
gettid()
read(1028, "E", 1)
Rule synthesis algorithm

- Rules with the shape: `syscall as s => ..., s, ...`
- Input: recorded and replayed traces
- Output: set of candidate rules
- Was able to find **16 out of 19** applicable rules
  - Non-determinism and infrequent syscalls impact quality of rules
- Details in the paper
Conclusion

- Increases the applicability of multi-version execution
  - For reliability and security
  - State-of-the-art MVE struggles with divergences
- Simple expressive language for reconciling system call sequences
  - Recorded and replayed
  - DSL provides the required action to tolerate divergences
- Necessary rules are easy to identify
  - `vimdiff` of strace logs
- Automatic algorithm to synthetize rules
  - From equivalent strace logs
We’re hiring!

▶ Post-doc position in Software Systems and Program Analysis
▶ Starting in November 2017, apply until August 2017
▶ Up to 17 months, possibly extendable to 24
▶ Details: https://srg.ic.ac.uk/vacancies
A DSL Approach to Reconcile Equivalent Divergent Program Executions

- Increases the applicability of multi-version execution
  - For reliability and security
  - State-of-the-art MVE struggles with divergences
- Simple expressive language for reconciling system call sequences
  - Recorded and replayed
  - DSL provides the required action to tolerate divergences
- Necessary rules are easy to identify
  - vimdiff of strace logs
- Automatic algorithm to synthesize rules
  - From equivalent strace logs

Luís Pina, Daniel Grumberg, Anastasios Andronidis, Cristian Cadar
Imperial College London