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

`malloc(WEIRD_NUMBER)`	`jemalloc(WEIRD_NUMBER)`
`SEGFAULT`	`OK`

Multi-Version Execution (MVE)

Run multiple versions as one

- Improves reliability

```
malloc(WEIRD_NUMBER)        jemalloc(WEIRD_NUMBER)
SEGFAULT                    OK
```

- Improves security

```
>strace hello1              >strace hello2
write(1,"Hello world")      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              >strace hello2
write(1,"Hello world")      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 **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)
```

Describe the divergences with a **Domain Specific Language (DSL)**

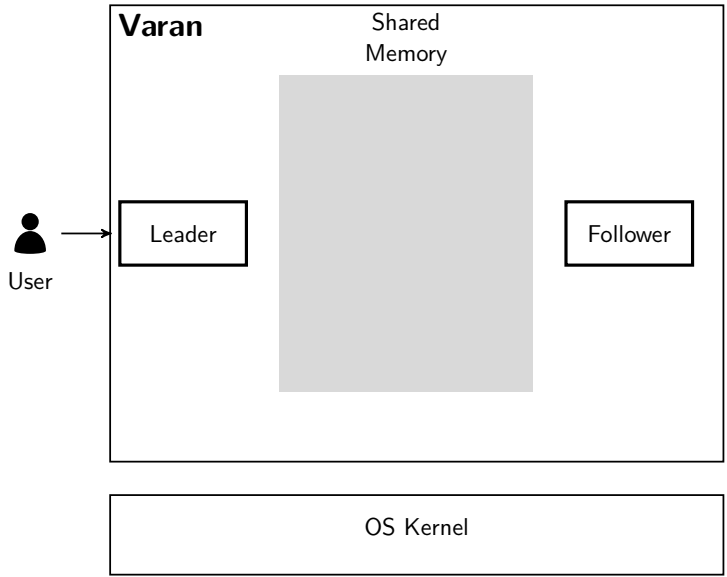
```
write(1,"Hello world", 11)
```

```
write(1,"Hello ", 6)
write(1,"world" , 5)
```

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

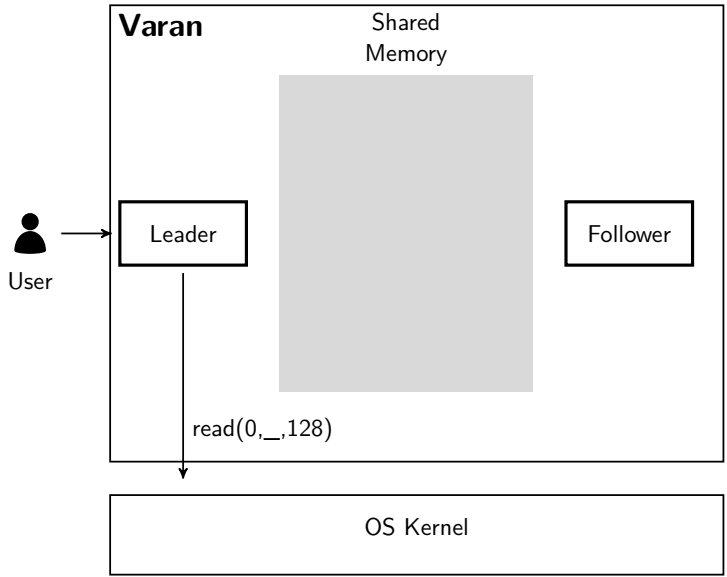
MVE Architecture

Varan



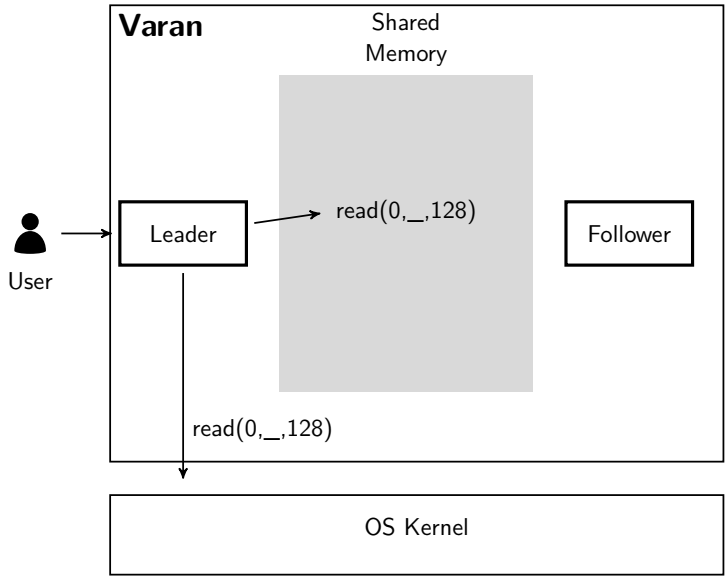
MVE Architecture

Varan



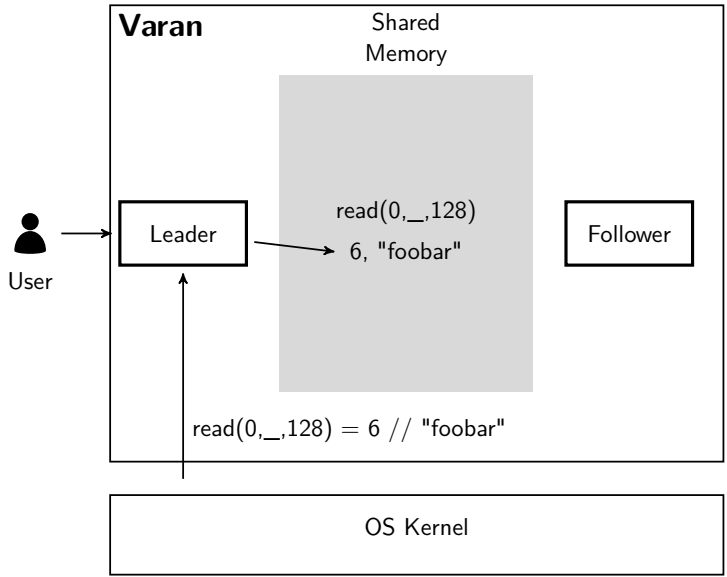
MVE Architecture

Varan



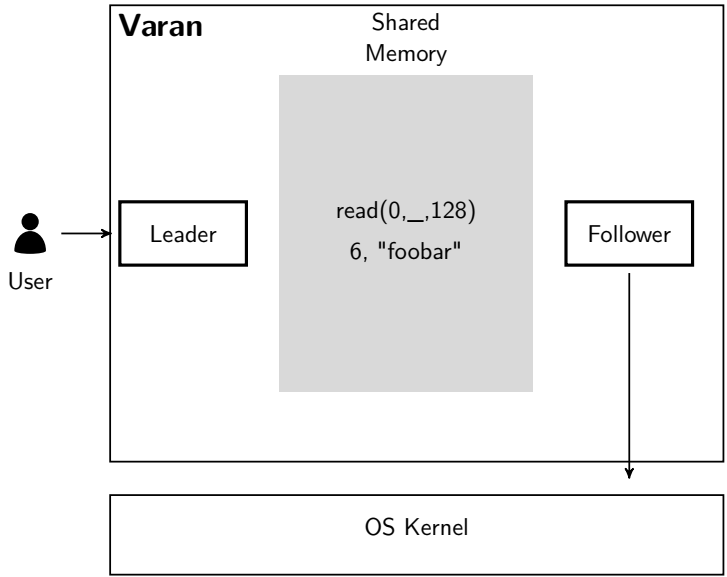
MVE Architecture

Varan



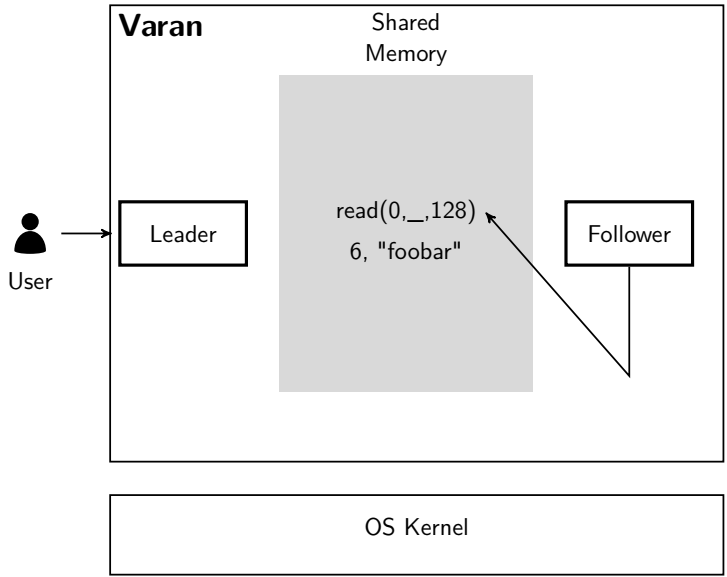
MVE Architecture

Varan



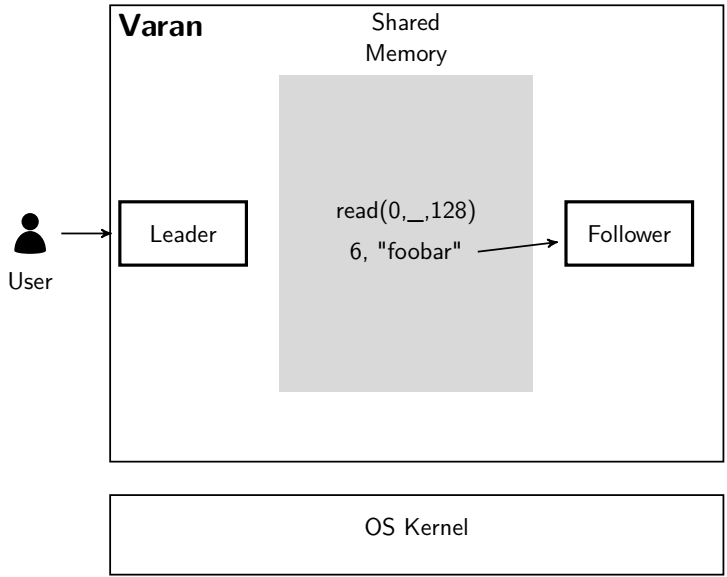
MVE Architecture

Varan



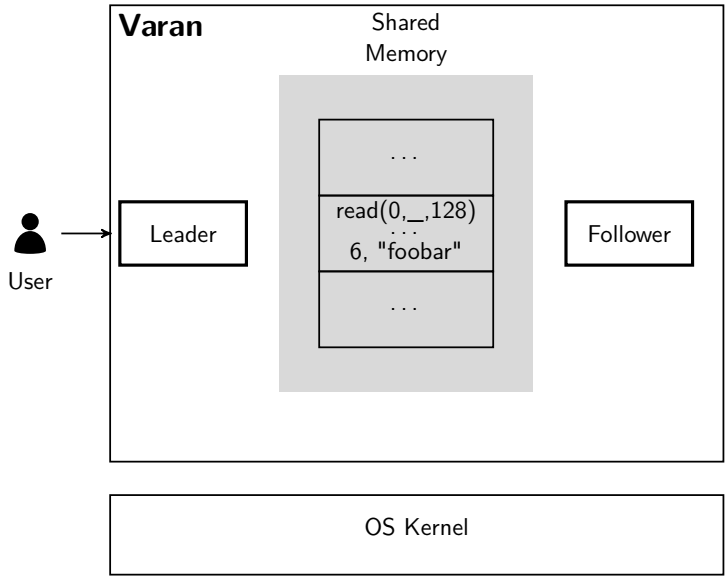
MVE Architecture

Varan



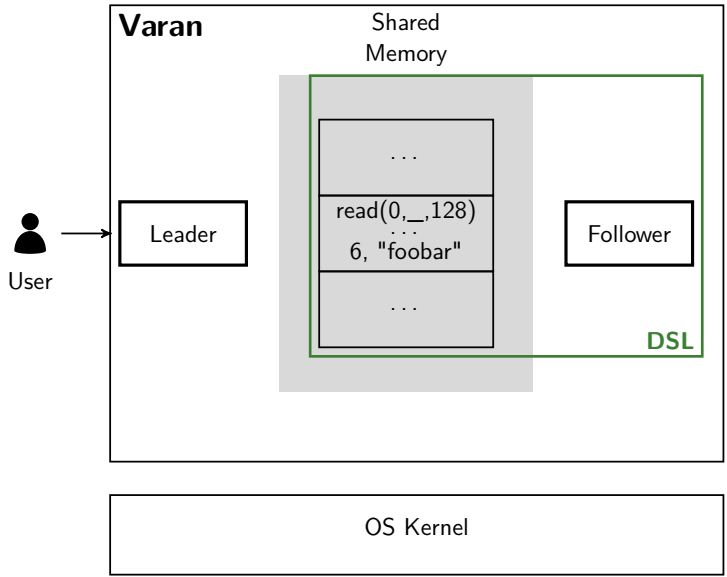
MVE Architecture

Varan



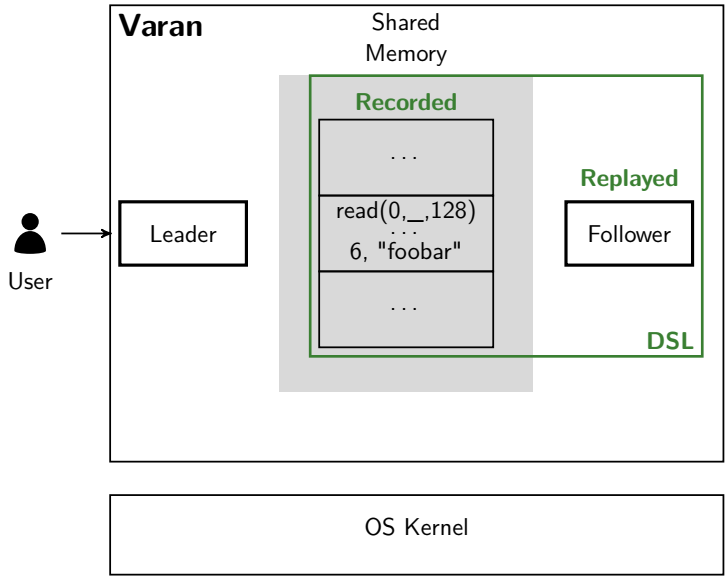
MVE Architecture

Varan

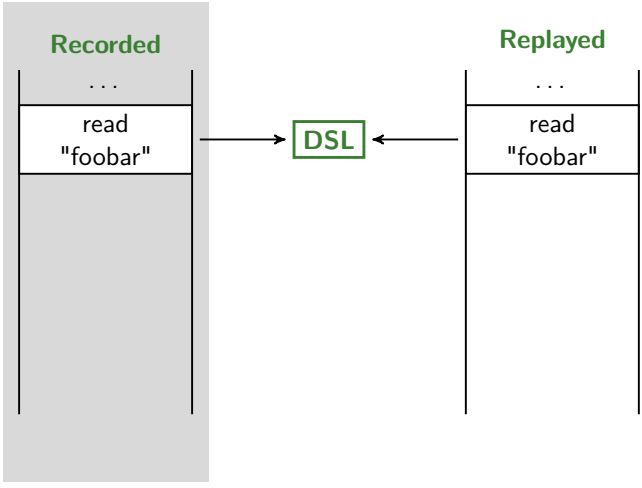


MVE Architecture

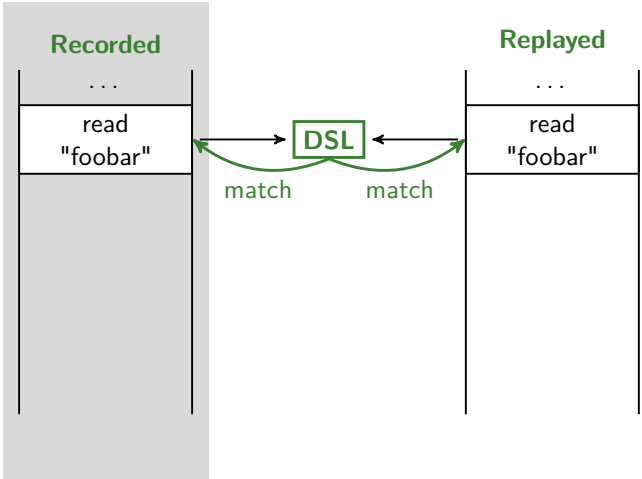
Varan



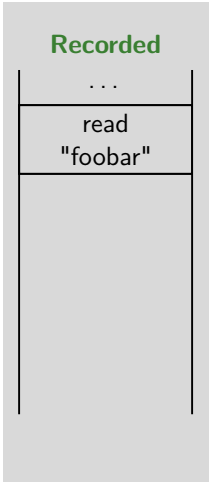
DSL Architecture

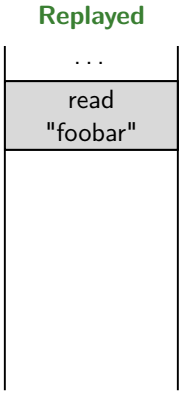


DSL Architecture



DSL Architecture





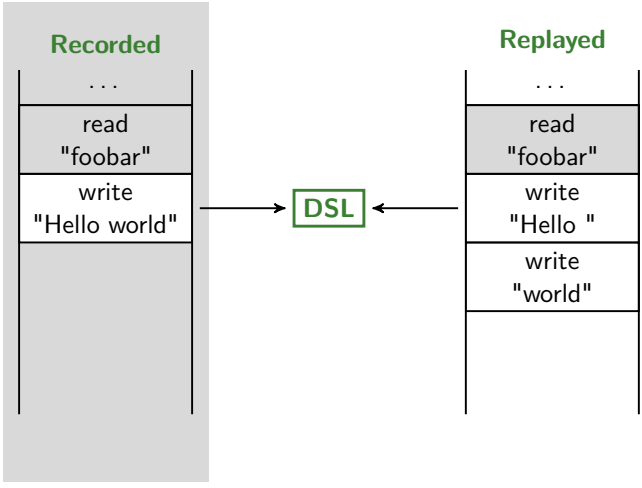
DSL Rules

- **Default rule:** `read(_,_,_) as r => r`
- **Actions**
  - `MATCH`
  - `NOP`
  - `SKIP`
  - `EXECUTE`
  - `STORE`
- **Further examples**
  - Hello world
  - **nothing** keyword
  - C predicates

Hello World Rule

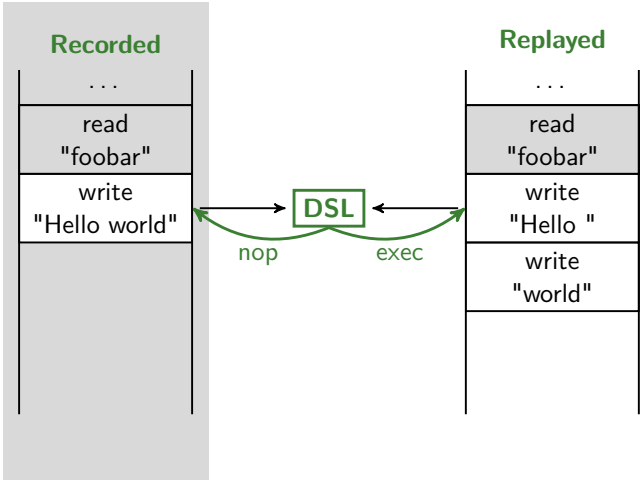
```
write(1, "Hello world", 11)  =>  write(1, "Hello ", 6),
                                  write(1, "world" , 5)
```

Hello World Rule



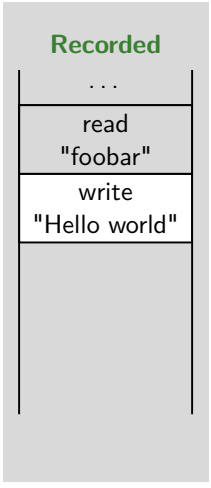

```
write(1, "Hello world", 11)  =>  write(1, "Hello ", 6),
                                  write(1, "world" , 5)
```

Hello World Rule



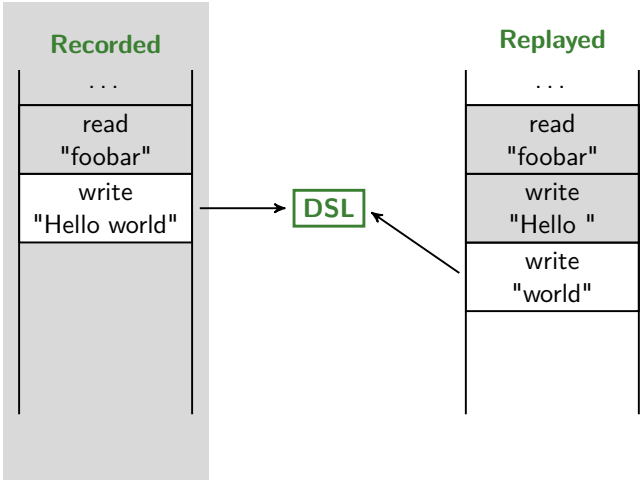

```
write(1, "Hello world", 11)  =>  write(1, "Hello ", 6),
                                  write(1, "world" , 5)
```

Hello World Rule



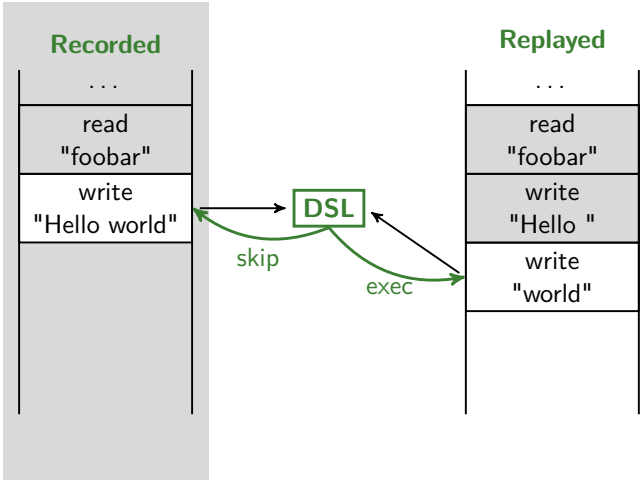

```
write(1, "Hello world", 11)  =>  write(1, "Hello ", 6),
                                  write(1, "world" , 5)
```

Hello World Rule



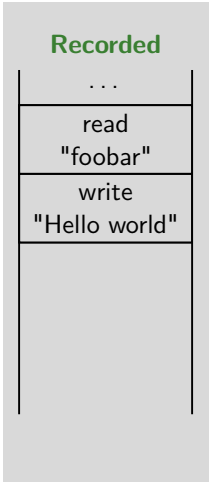

```
write(1, "Hello world", 11)  =>  write(1, "Hello ", 6),
                                  write(1, "world" , 5)
```

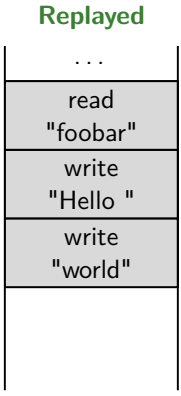
Hello World Rule




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

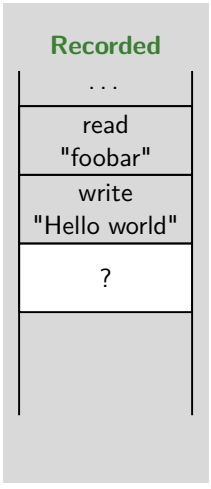
Hello World Rule

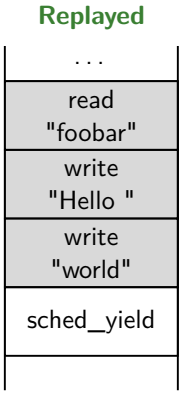





```
write(1, "Hello world", 11)  =>  write(1, "Hello ", 6),
                                  write(1, "world" , 5)
```

Hello World Rule

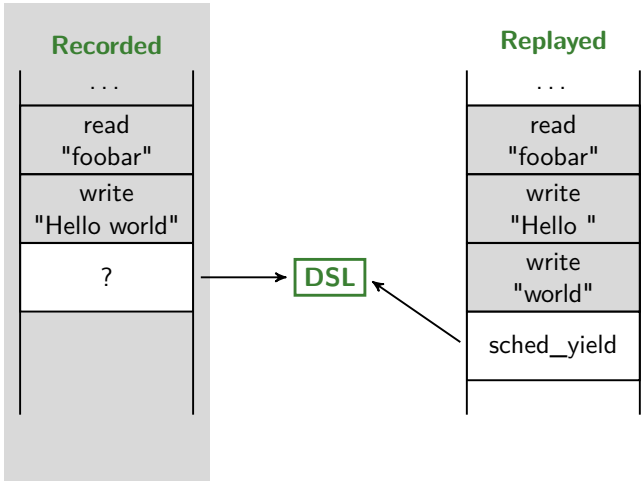




nothing Keyword

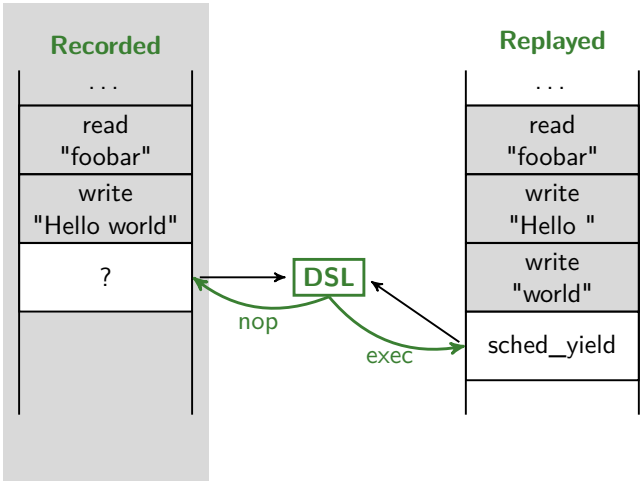
```
nothing  =>  sched_yield()
```

nothing Keyword



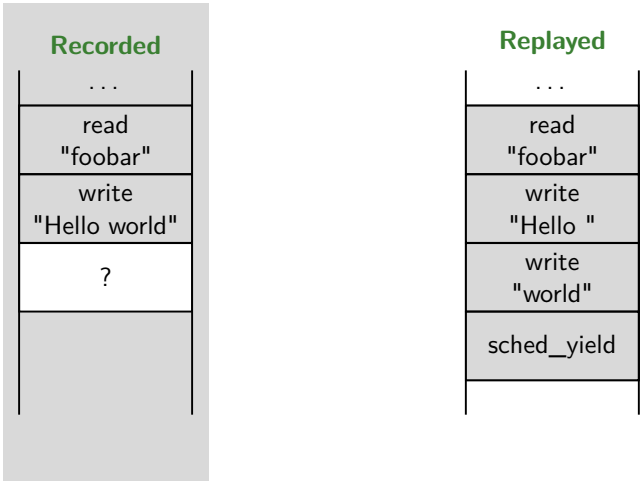

```
nothing  =>  sched_yield()
```

nothing Keyword



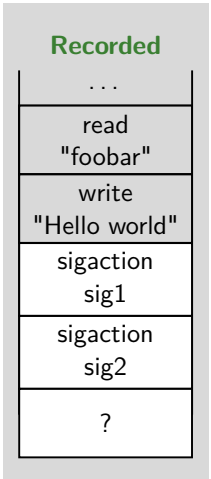

```
nothing  =>  sched_yield()
```

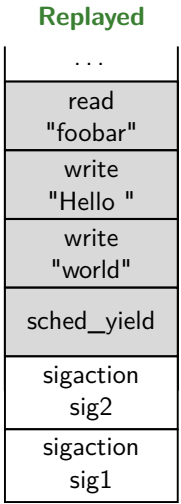
nothing Keyword




```
nothing  =>  sched_yield()
```

nothing Keyword



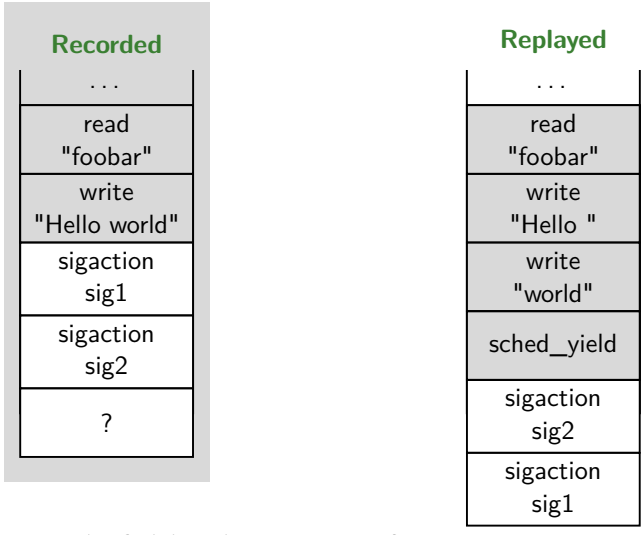


C Predicates

and multiple left-hand side

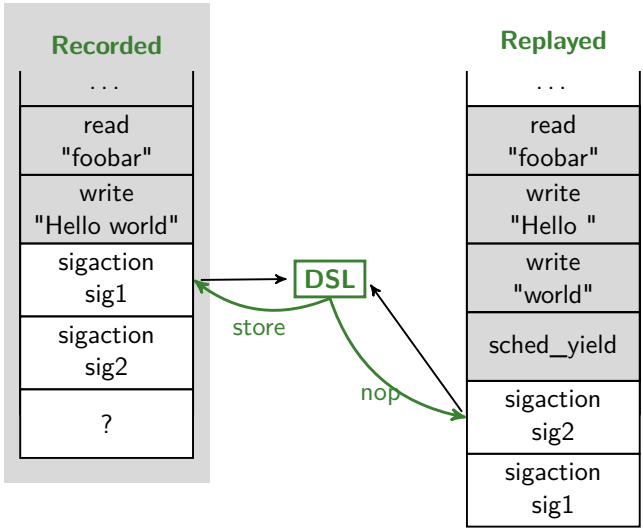
```
// extern int sig1, sig2;
sigact(sig,_,_) { $(sig) == sig1; } as s1,
sigact(sig,_,_) { $(sig) == sig2; } as s2   =>  s2, s1
```

C Predicates



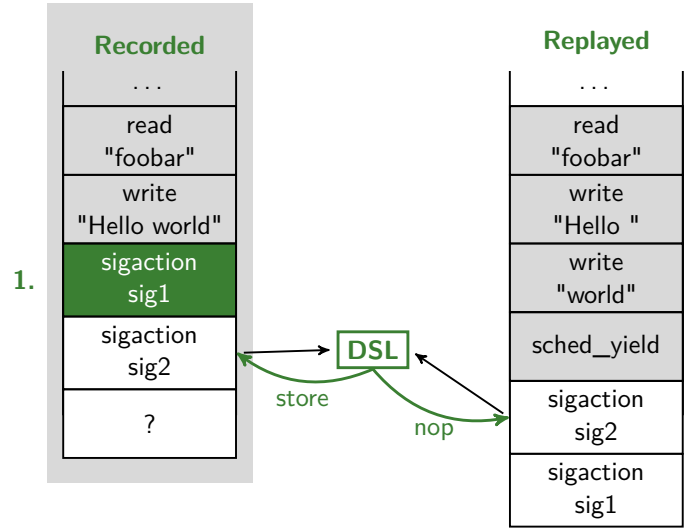

```
sigact(sig,_,_) { $(sig) == sig1; } as s1,
sigact(sig,_,_) { $(sig) == sig2; } as s2   =>  s2, s1
```

C Predicates



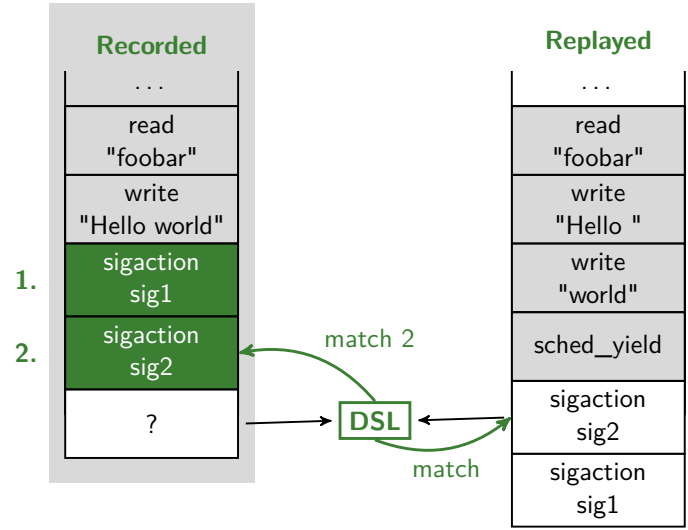

```
sigact(sig,_,_) { $(sig) == sig1; } as s1,
sigact(sig,_,_) { $(sig) == sig2; } as s2    =>   s2, s1
```

C Predicates



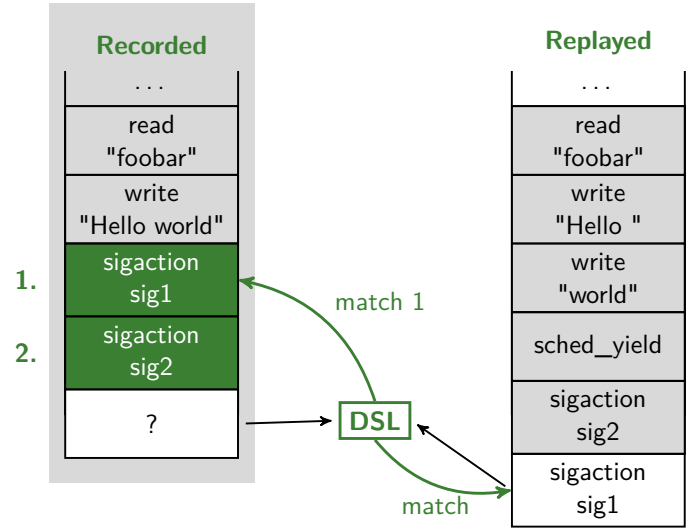

```
sigact(sig,_,_) { $(sig) == sig1; } as s1,
sigact(sig,_,_) { $(sig) == sig2; } as s2    =>   s2, s1
```

C Predicates



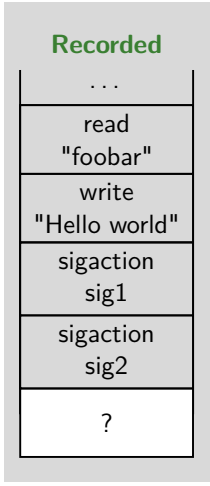

```
sigact(sig,_,_) { $(sig) == sig1; } as s1,
sigact(sig,_,_) { $(sig) == sig2; } as s2    =>   s2, s1
```

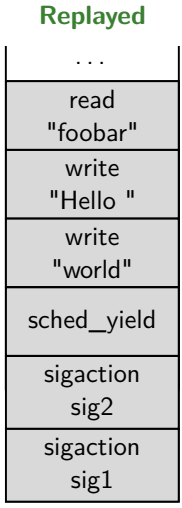
C Predicates




```
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
```

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

ID	Redis Versions *Recorded – Replayed*	Commits	Rules	
1	1.3.8 – 1.3.10	40	0	6
2	1.3.10 – 1.3.12	105	0	
3	1.3.12 – 2.0.0	92	1	
4	2.0.0 – 2.0.5	34	1	
5	2.0.5 – 2.2.0	730	3	
6	2.2.0 – 2.2.15	110	2	

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 syntatic-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]

- **git** (log, blame, diff, tag)
- **openssh** (ssh, ssh-keygen)
- **htop**
- **vim**

[1] Expands to 31 rules through group syntatic-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]

- **git** (log, blame, diff, tag)
- **openssh** (ssh, ssh-keygen)
- **htop**
- **vim**

[1] Expands to 31 rules through group syntatic-sugar

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

native.log

```
read(3, ..., 4096)

lseek(3, -2347, SEEK_CUR)

read(3, ..., 4096)

close(3)
```

valgrind.log

```
gettid()
write(1029, "D", 1)
sigprocmask([], ~[...])
read(3, ..., 4096)
sigprocmask(~[...], NULL)
gettid()
read(1028, "D", 1)
lseek(3, -2347, SEEK_CUR)
gettid()
write(1029, "E", 1)
sigprocmask([], ~[...])
read(3, ..., 4096)
sigprocmask( [...], NULL)
gettid()
read(1028, "E", 1)
close(3)
```

Finding Rules

`vimdiff`

native.log

```
read(3, ..., 4096)

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 synthethize 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 synthethize rules
  - From equivalent strace logs

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