Finding Race Conditions in Kernels

from fuzzing to symbolic execution

Meng Xu

July 16, 2020

The game of attack and defense

Bug finding

Exploitation

Profit

The game of attack and defense

Bug finding → **Exploitation** → **Profit**

Privacy violations in browsers **[CCS'15]**

Kernel double-fetch bugs **[SP'18]**

Concolic execution **[Security'18]**

Kernel file system bugs **[SOSP'19]**

File system data races **[SP'20]**

C to SMT Transpilation **[WIP]**

The game of attack and defense

Bug finding

Protection through diversity

Profit

Privacy violations in browsers **[CCS'15]**

Kernel double-fetch bugs **[SP'18]**

Concolic execution **[Security'18]**

Kernel file system bugs **[SOSP'19]**

File system data races **[SP'20]**

C to SMT Transpilation **[WIP]**

Comprehensive memory prot. **[ATC'17]**

Malicious document prot. **[Security'17]**

Information leak prot. **[TDSC'18]**

The game of attack and defense

Bug finding

Privacy violations in browsers **[CCS'15]**

Kernel double-fetch bugs **[SP'18]**

Concolic execution **[Security'18]**

Kernel file system bugs **[SOSP'19]**

File system data races **[SP'20]**

C to SMT Transpilation **[WIP]**

Protection through diversity

Comprehensive memory prot. **[ATC'17]**

Malicious document prot. **[Security'17]**

Information leak prot. **[TDSC'18]**

Recovery

Android security survey **[CSUR'16]**

1-day vuln. in OSS **[CCS'17]**

Android update attack **[ComSIS'18]**

IoT device resiliency **[SP'19]**

Secure router for smart homes **[in sub.]**

In this dissertation

Bug finding

Protection through diversity

Recovery

Privacy violations in browsers **[CCS'15]**

Kernel double-fetch bugs **[SP'18]**

Concolic execution **[Security'18]**

Kernel file system bugs **[SOSP'19]**

File system data races **[SP'20]**

C to SMT Transpilation **[WIP]**

Comprehensive memory prot. **[ATC'17]**

Malicious document prot. **[Security'17]**

Information leak prot. **[TDSC'18]**

Android security survey **[CSUR'16]**

1-day vuln. in OSS **[CCS'17]**

Android update attack **[ComSIS'18]**

IoT device resiliency **[SP'19]**

Secure router for smart homes **[in sub.]**

In this dissertation

Bug finding

Protection through diversity

Recovery

Privacy violations in browsers **[CCS'15]**

Kernel double-fetch bugs **[SP'18]**

Concolic execution **[Security'18]**

Kernel file system bugs **[SOSP'19]**

File system data races **[SP'20]**

C to SMT Transpilation **[WIP]**

Race conditions

Comprehensive memory prot. **[ATC'17]**

Malicious document prot. **[Security'17]**

DSC'18]

Android security survey **[CSUR'16]**

1-day vuln. in OSS **[CCS'17]**

Android update attack **[ComSIS'18]**

IoT device resiliency **[SP'19]**

Secure router for smart homes **[in sub.]**

Race conditions

Definition: Two memory accesses from different threads such that

1. They access the same memory location
2. At least one of them is a write operation
3. They may interleave without restrictions (i.e., locks, orderings, etc)

The classic race condition example

`count = 0`

Thread 1	Thread 2
`for(i=0; i<5; i++) {`	`for(i=0; i<5; i++) {`
`  count++;`	`  count++;`
`}`	`}`

What is the value of `count` when both threads terminate?

*Assume sequential consistency.

The classic race condition example

`count = 0`

Thread 1	Thread 2
`for(i=0; i<5; i++) {`	`for(i=0; i<5; i++) {`
`  count++;`	`  count++;`
`}`	`}`

What is the value of `count` when both threads terminate?

Any value between 5 to 10

*Assume sequential consistency.

The classic race condition example

`count = 0`

Thread 1	Thread 2

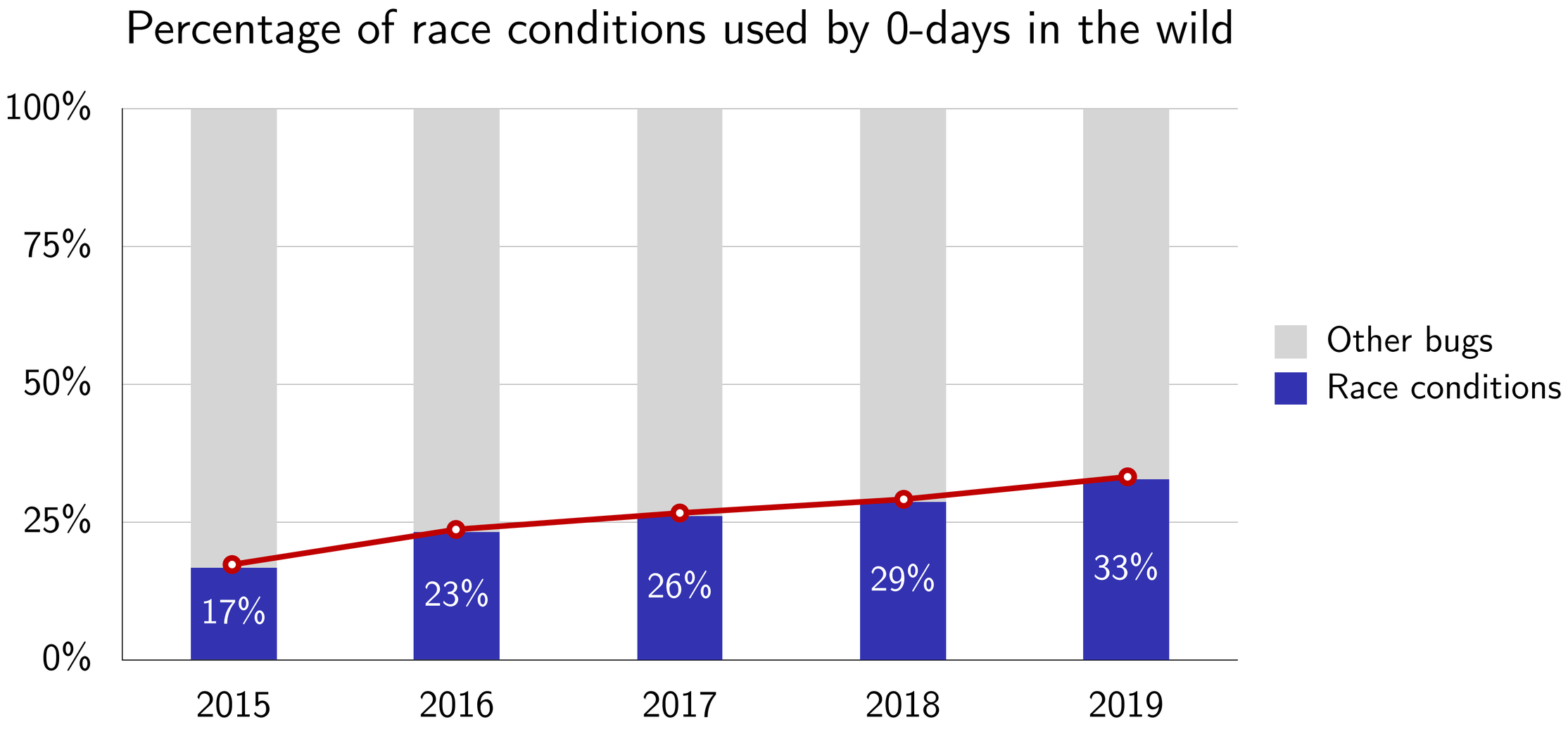
`for(i=0; i<5; i++) {`
`  lock(mutex);`
`  count++;`
`  unlock(mutex);`
`}`
	`for(i=0; i<5; i++) {`
	`  lock(mutex);`
	`  count++;`
	`  unlock(mutex);`
	`}`

What is the value of `count` when both threads terminate?

10

*Assume sequential consistency.

The rise of race conditions



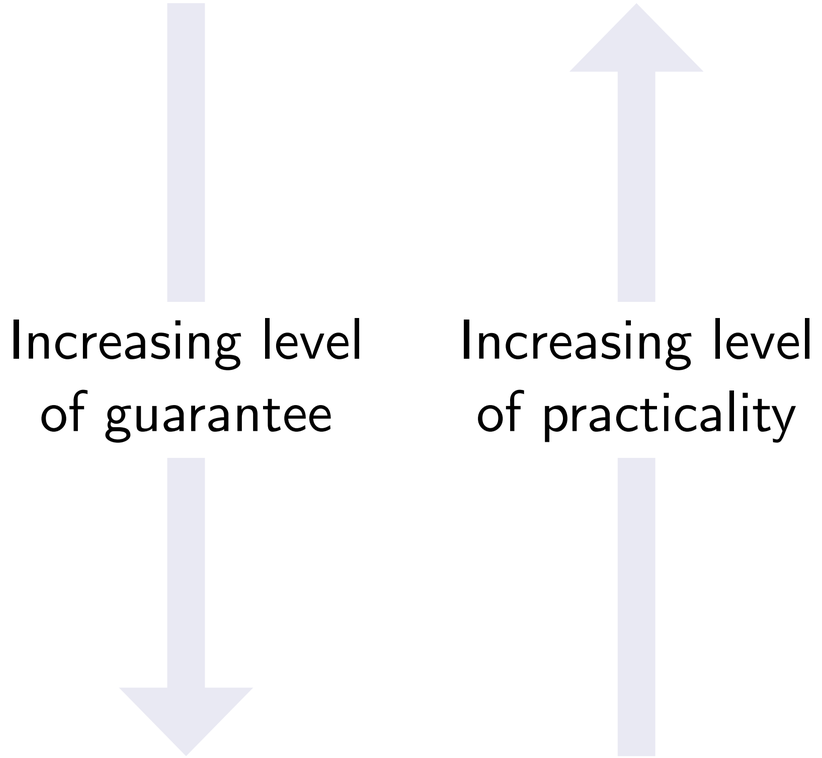
Agenda

1. What are race conditions?

2. Finding their presence with **fuzzing**?
   - [SP'20] Data races in file systems

3. Towards a more **systematic** methodology?
   - [SP'18] Symbolic race checking

4. Up to the extreme of **completeness and soundness**?
   - [WIP (CAV'21)] C to SMT transpilation



Agenda

1. What are race conditions?

2. Finding their presence with **fuzzing**?
   - [SP'20] Data races in file systems

3. Towards a more **systematic** methodology?
   - [SP'18] Symbolic race checking

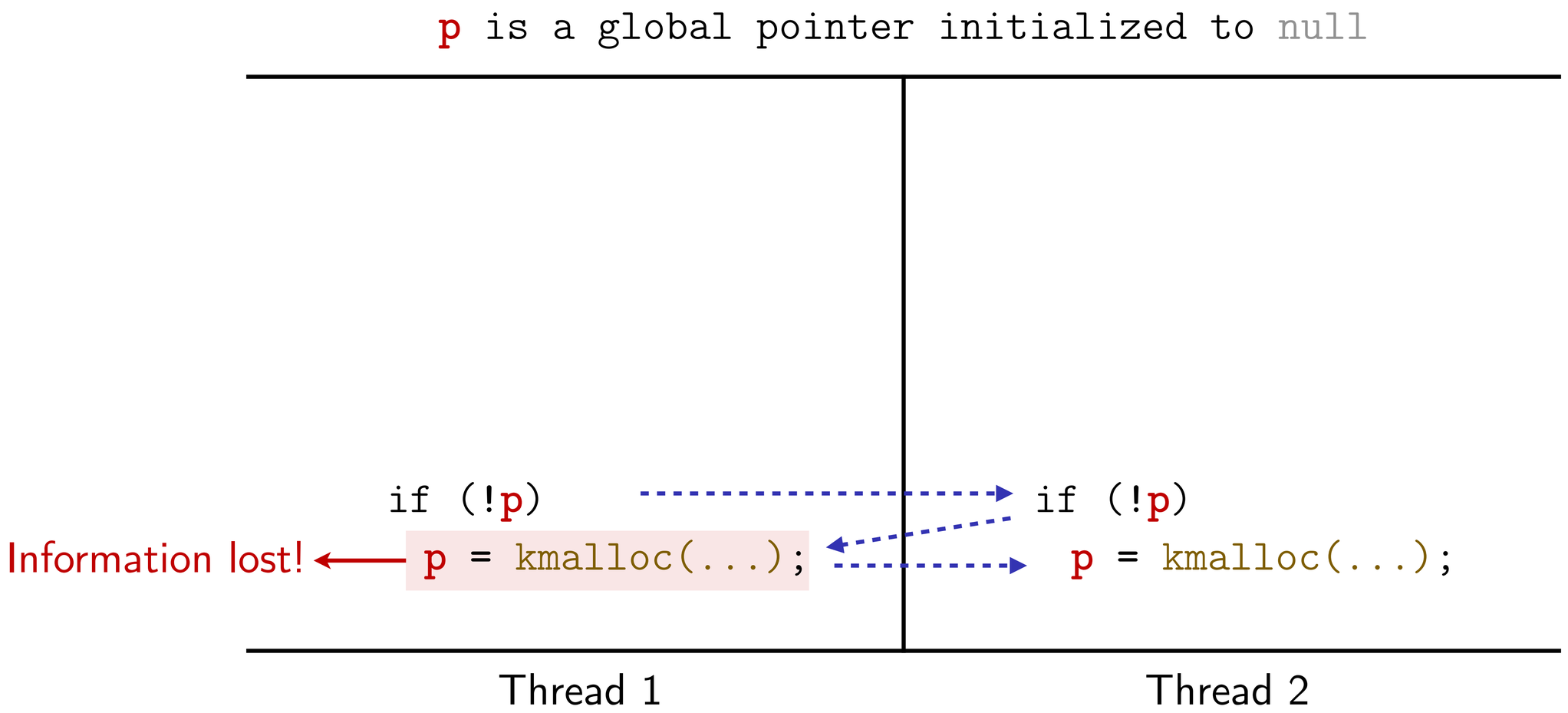
4. Up to the extreme of **completeness and soundness**?
   - [WIP (CAV'21)] C to SMT transpilation

High level of concurrency in the Linux kernel

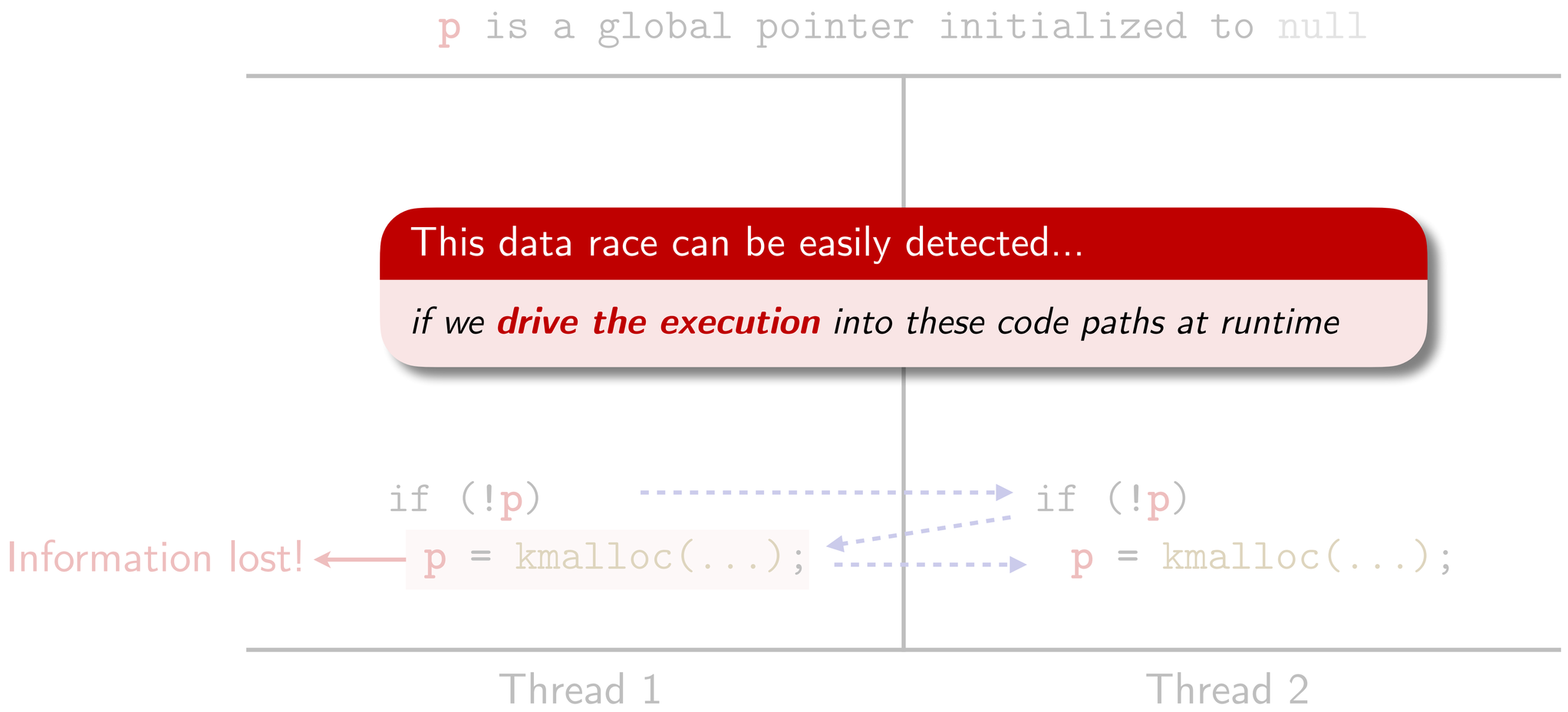
```
1  struct btrfs_fs_info {
2      /* work queues */
3      struct btrfs_workqueue *workers;
4      struct btrfs_workqueue *delalloc_workers;
5      struct btrfs_workqueue *flush_workers;
6      struct btrfs_workqueue *endio_workers;
7      struct btrfs_workqueue *endio_meta_workers;
8      struct btrfs_workqueue *endio_raid56_workers;
9      struct btrfs_workqueue *endio_repair_workers;
10     struct btrfs_workqueue *rmw_workers;
11     struct btrfs_workqueue *endio_meta_write_workers;
12     struct btrfs_workqueue *endio_write_workers;
13     struct btrfs_workqueue *endio_freespace_worker;
14     struct btrfs_workqueue *submit_workers;
15     struct btrfs_workqueue *caching_workers;
16     struct btrfs_workqueue *readahead_workers;
17     struct btrfs_workqueue *fixup_workers;
18     struct btrfs_workqueue *delayed_workers;
19     struct btrfs_workqueue *scrub_workers;
20     struct btrfs_workqueue *scrub_wr_completion_workers;
21     struct btrfs_workqueue *scrub_parity_workers;
22     struct btrfs_workqueue *qgroup_rescan_workers;
23     /* background threads */
24     struct task_struct *transaction_kthread;
25     struct task_struct *cleaner_kthread;
26 };
```

22 threads run
in the background!

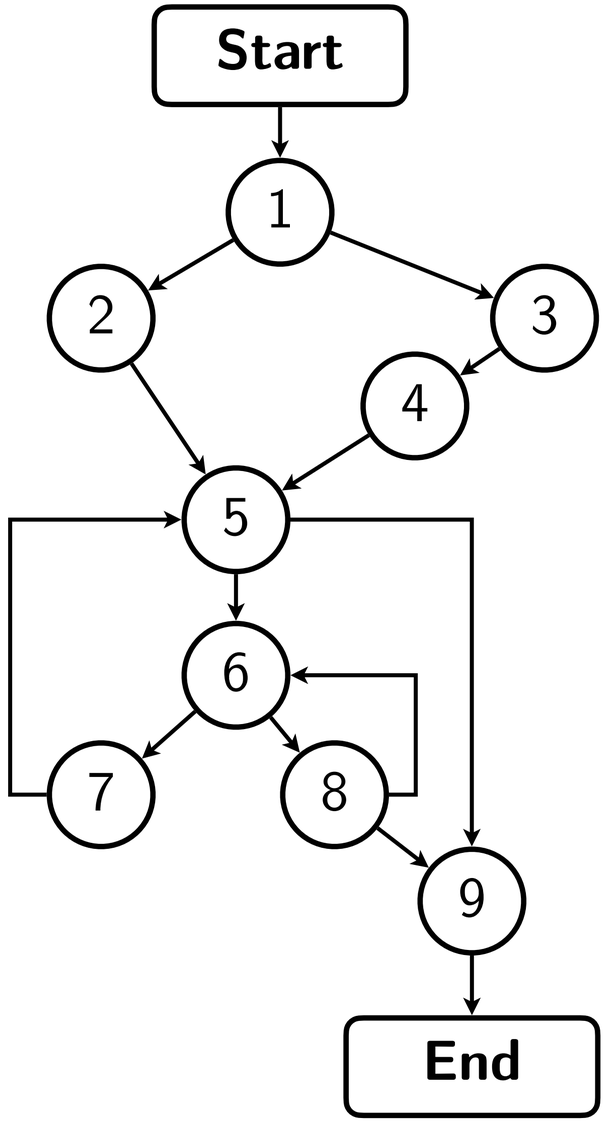
A data race in the kernel



A data race in the kernel

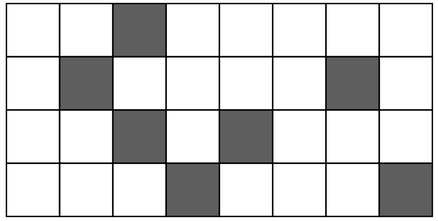


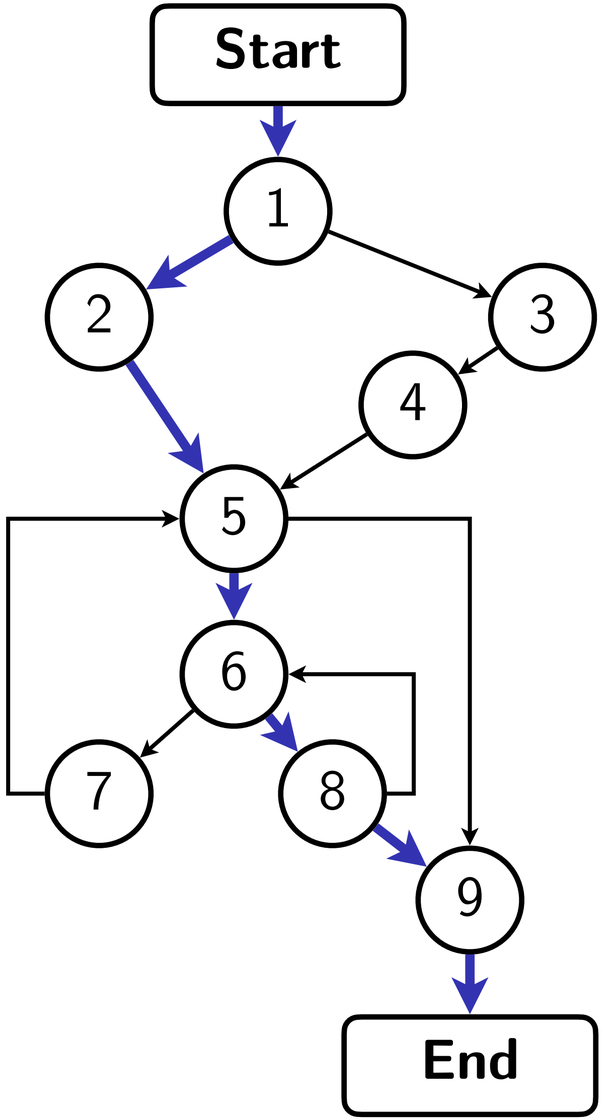
Fuzzing as a way to explore the program



Code coverage as an approximation

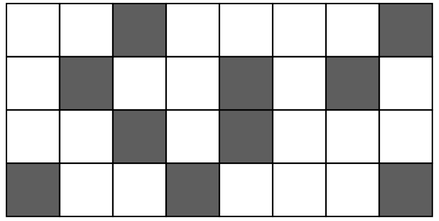
```
open(“some-file”, O_READ, ...)
```

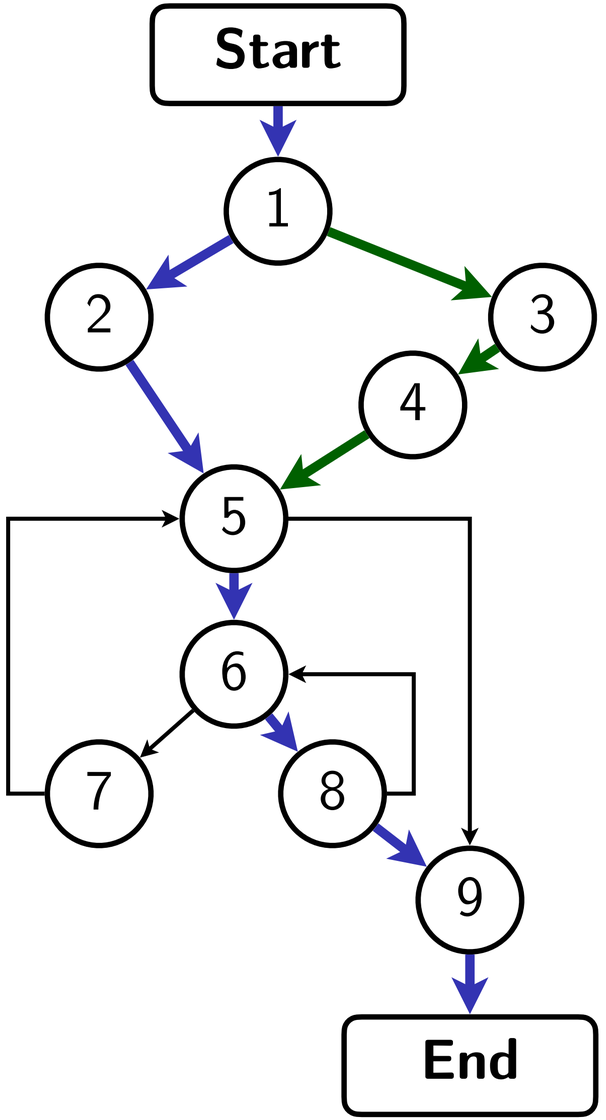




Code coverage as an approximation

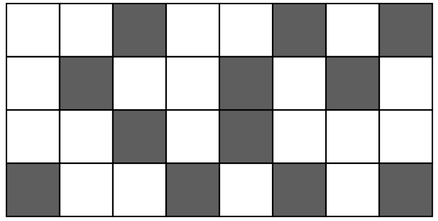
```
open(“some-file”, O_READ, ...)
open(“some-file”, O_WRITE, ...)
```

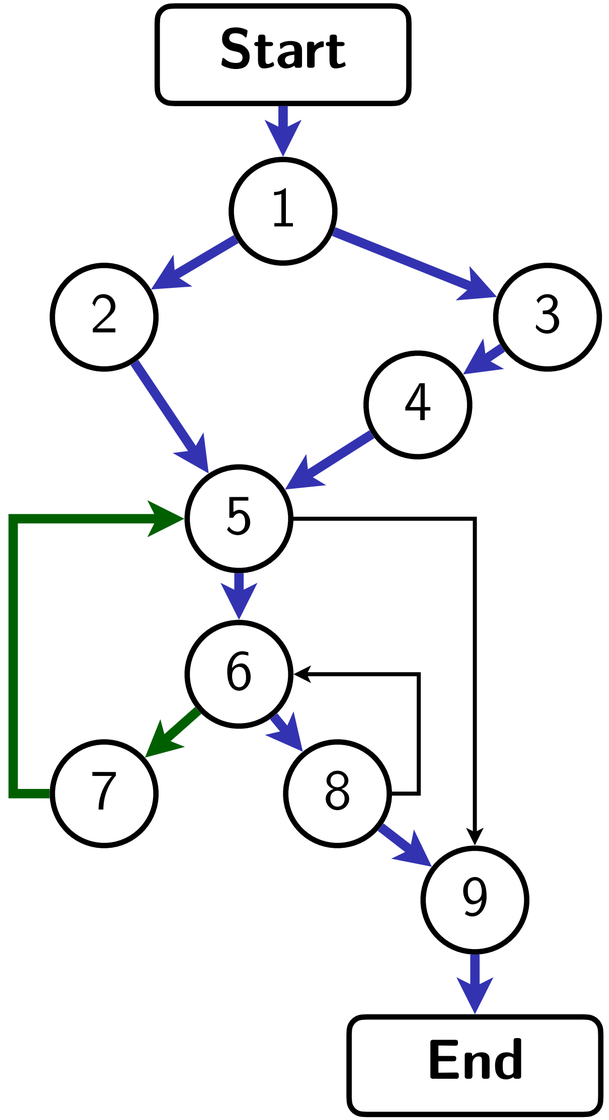




Code coverage as an approximation

```
open(“some-file”, O_READ, ...)
open(“some-file”, O_WRITE, ...)
open(“new-file”, O_READ, ...)
```

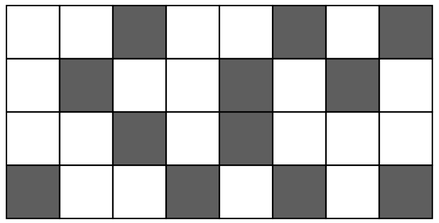


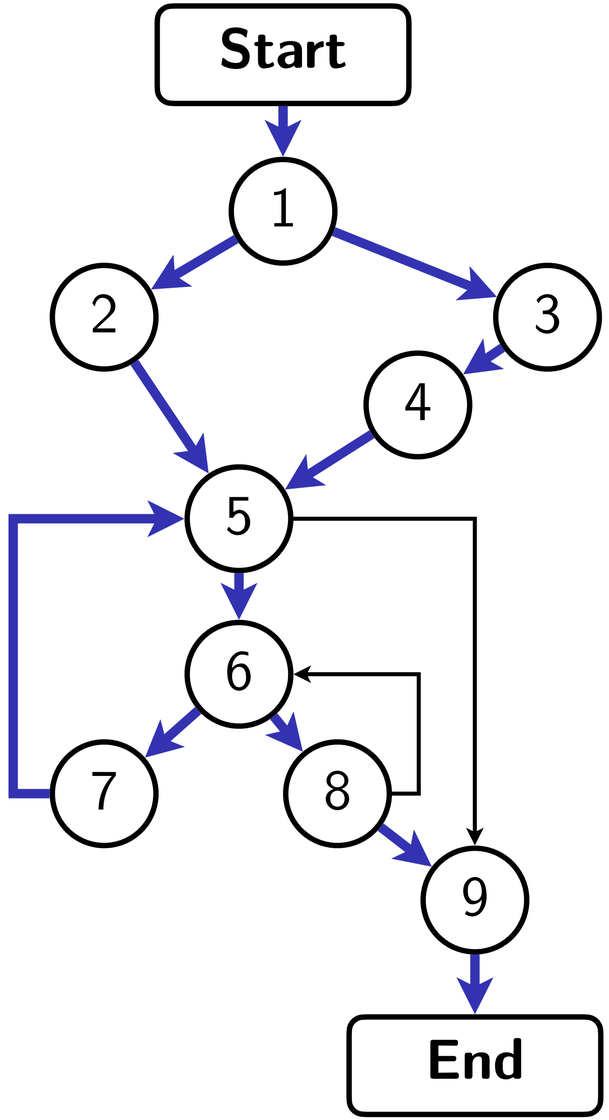


Code coverage as an approximation

```
open(“some-file”, O_READ, ...)
open(“some-file”, O_WRITE, ...)
open(“new-file”, O_READ, ...)
                    ⋮
                  20 trials
                    ⋮
open(“some-file”, O_RDWR, ...)
```

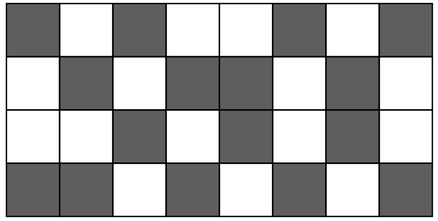
Coverage growth stalled!



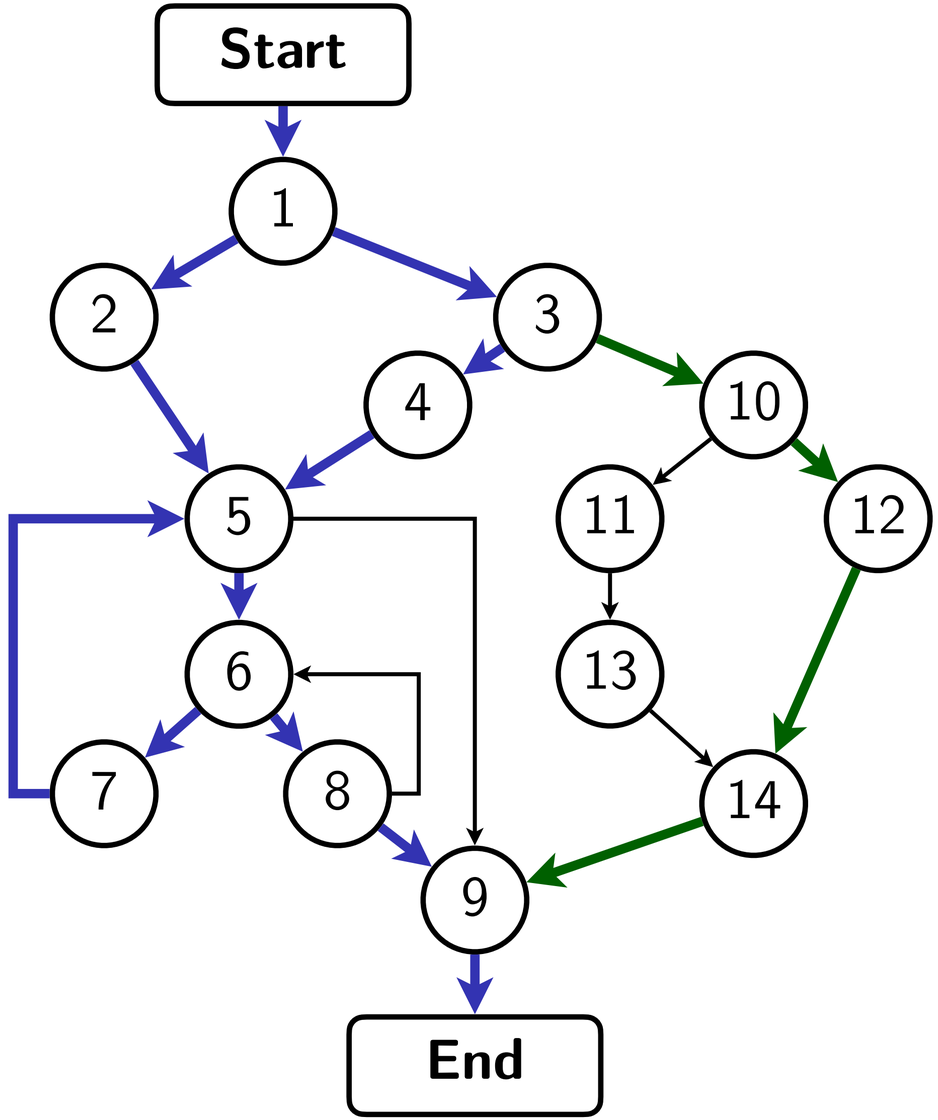


Code coverage as an approximation

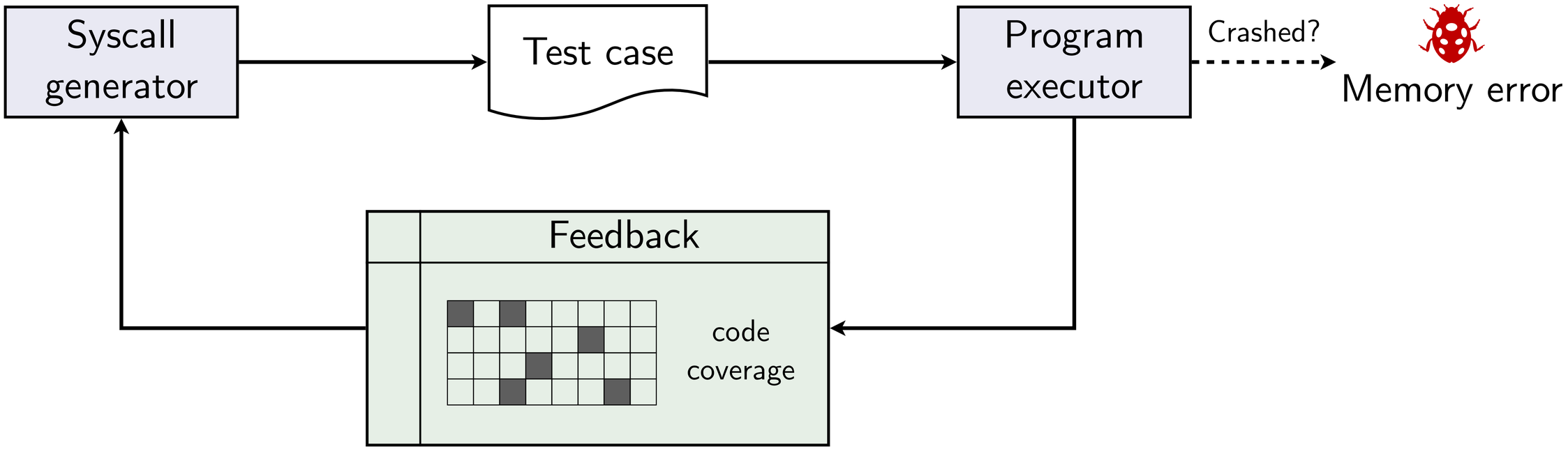
```
open(“some-file”, O_READ, ...)
open(“some-file”, O_WRITE, ...)
open(“new-file”, O_READ, ...)
                  ⋮
                  ⋮  20 trials
                  ⋮
open(“some-file”, O_RDWR, ...)
```



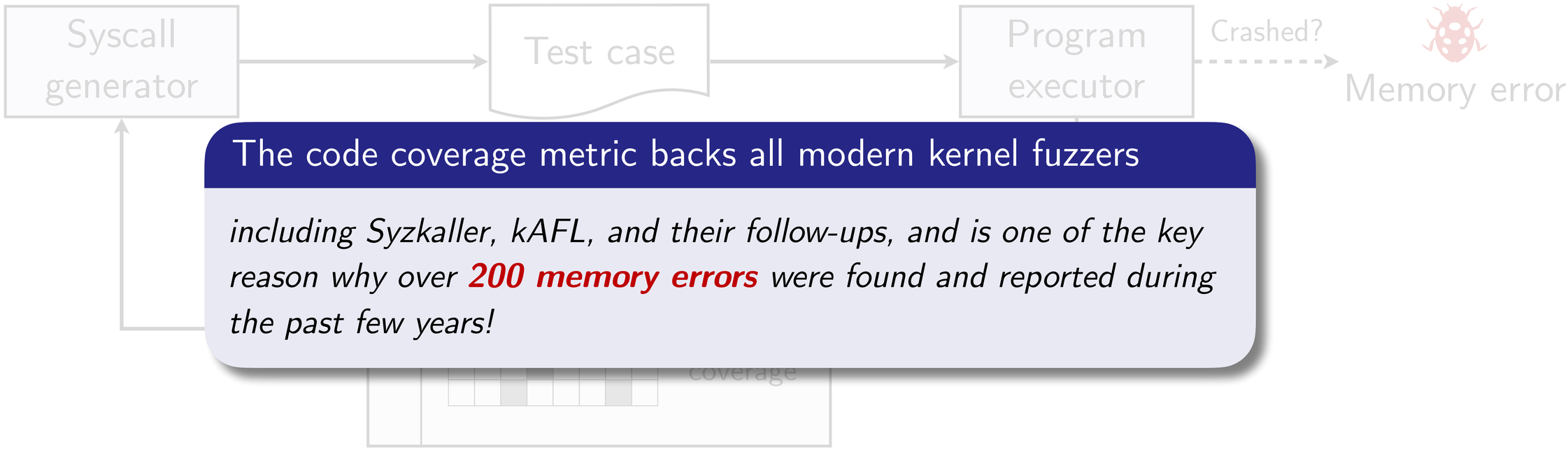
```
rename(“new-file”, “old-file”)
```



The conventional fuzzing process



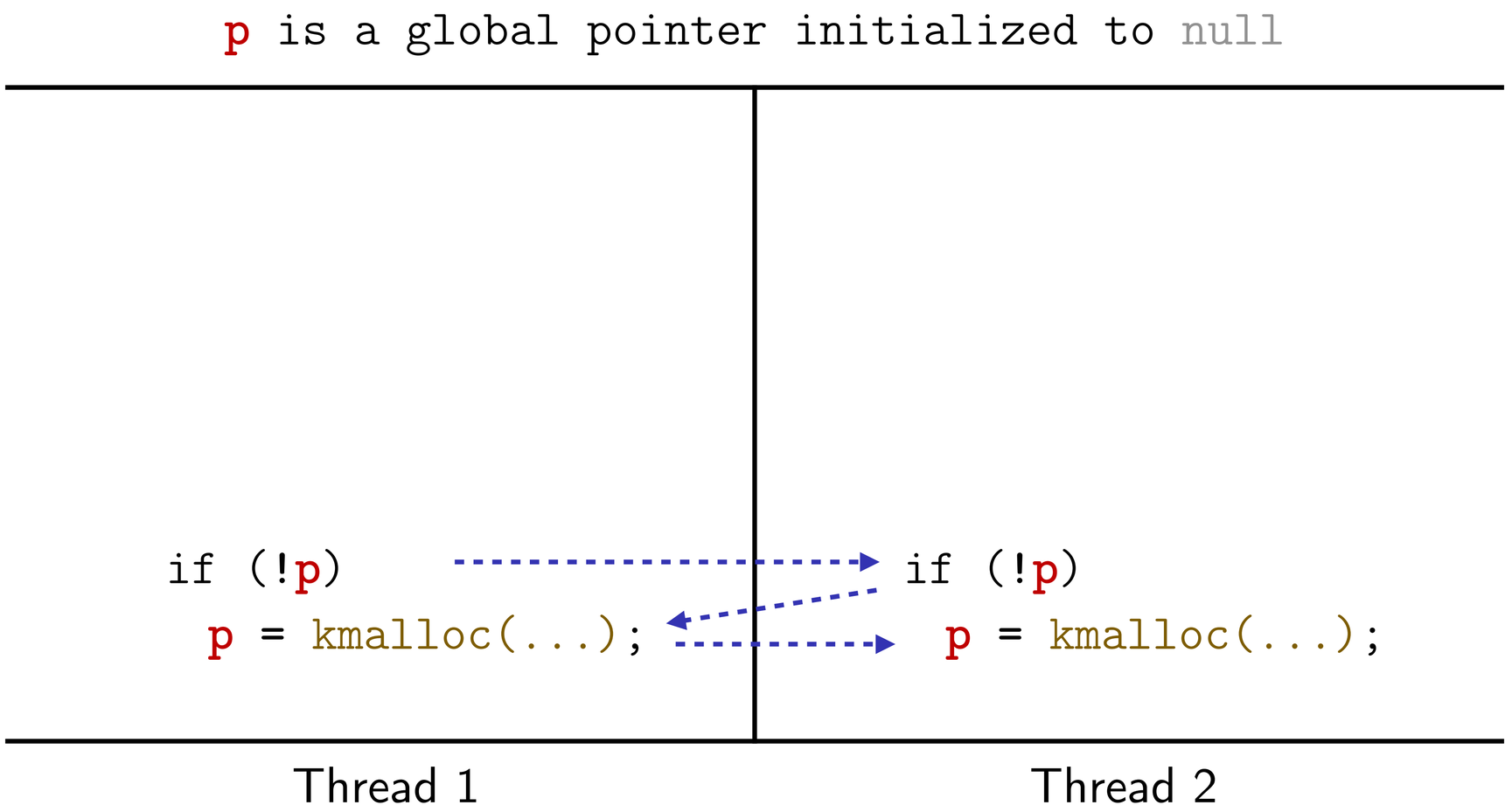
The conventional fuzzing process



The code coverage metric backs all modern kernel fuzzers

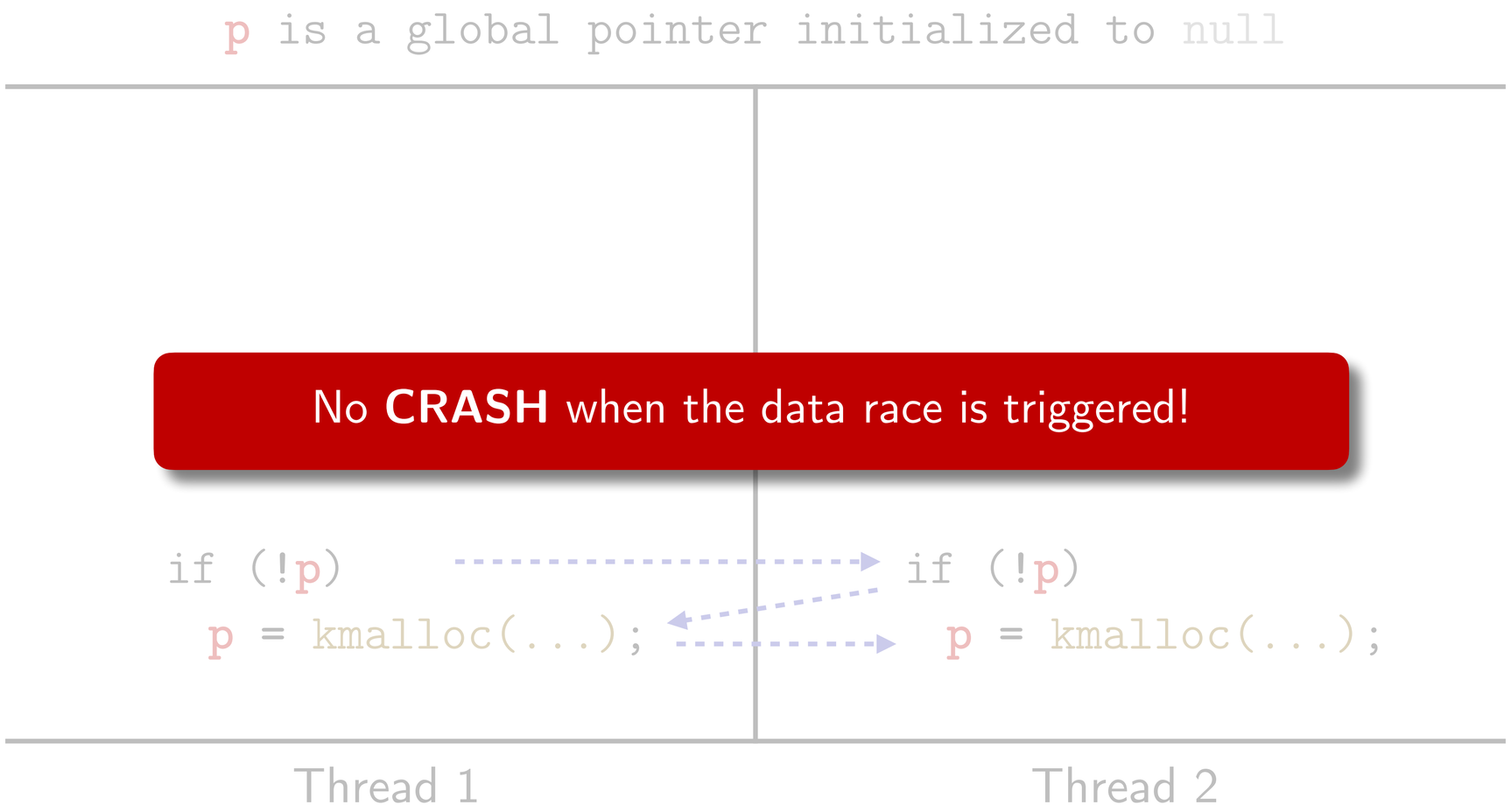
including Syzkaller, kAFL, and their follow-ups, and is one of the key reason why over ***200 memory errors*** *were found and reported during the past few years!*

Back to our data race example



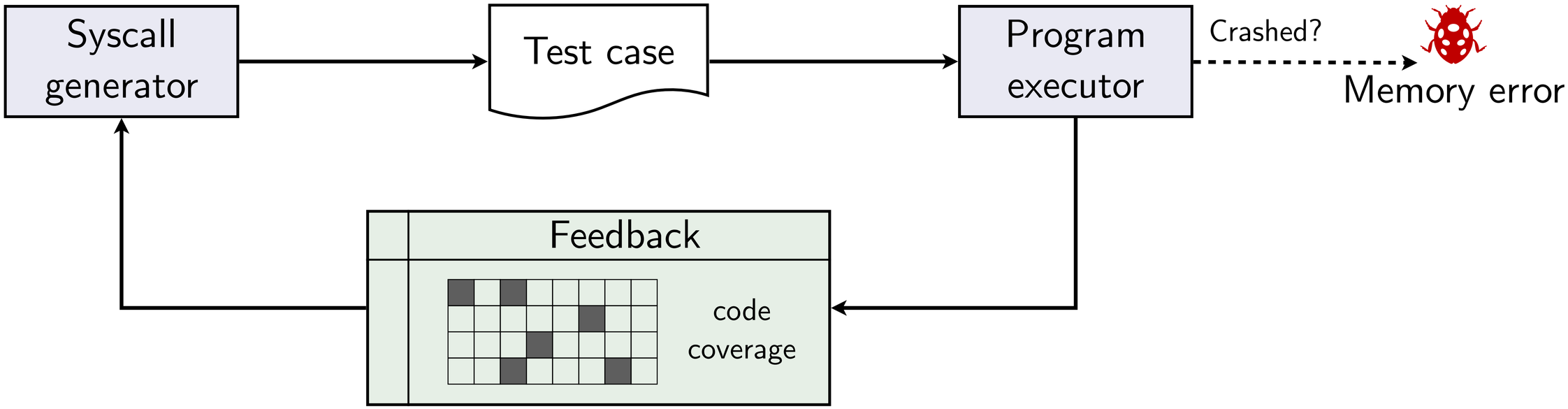
*Assume sequential consistency.

Back to our data race example

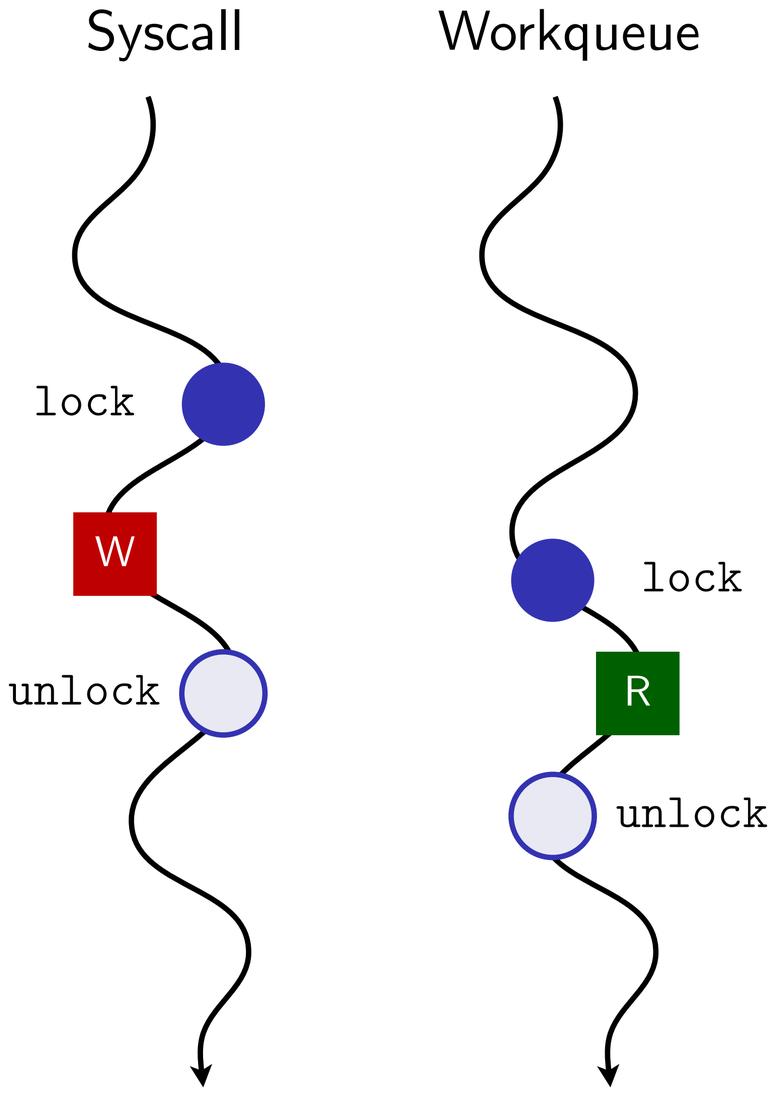


*Assume sequential consistency.

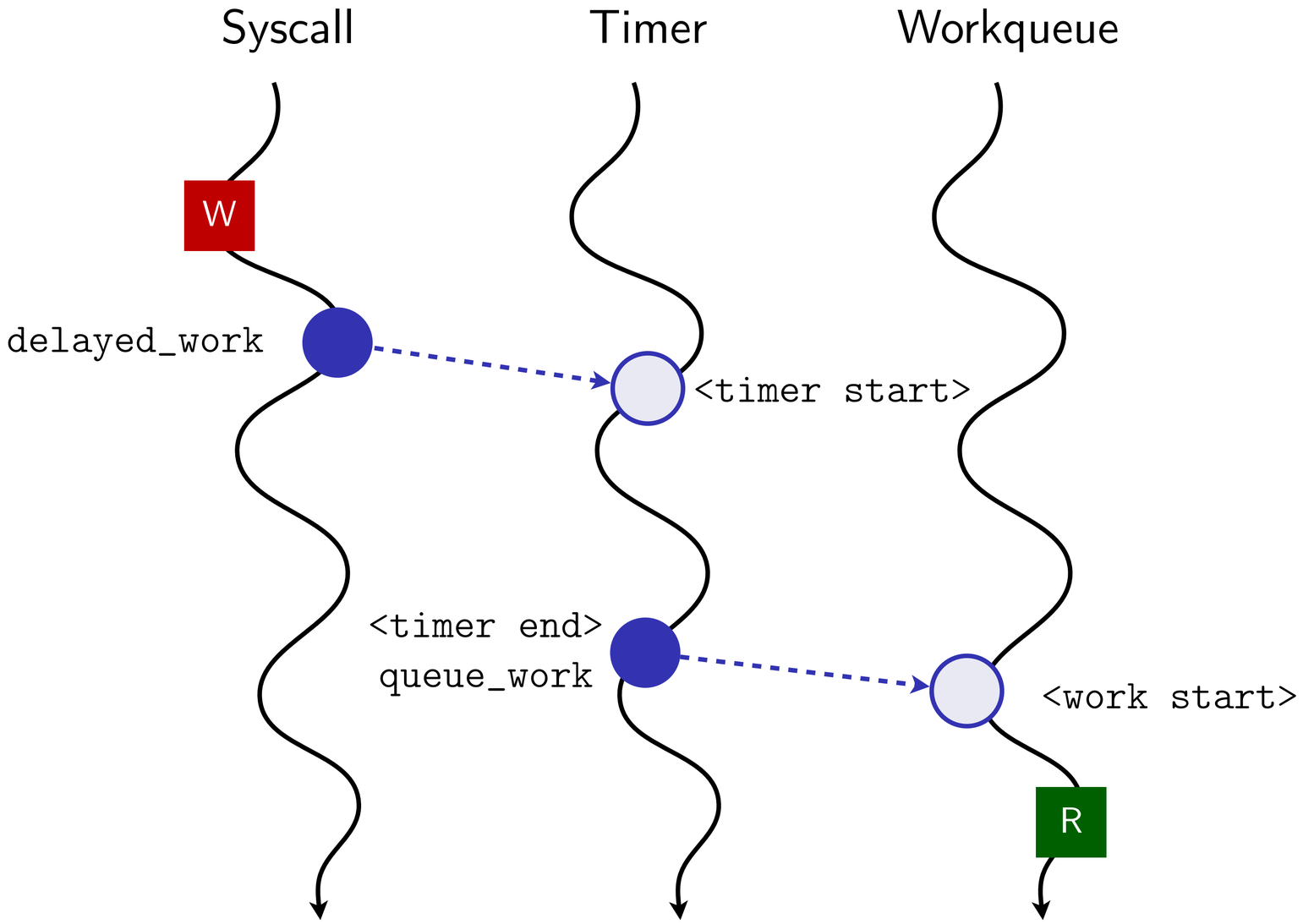
Traditional fuzzers rely on crash as a bug signal



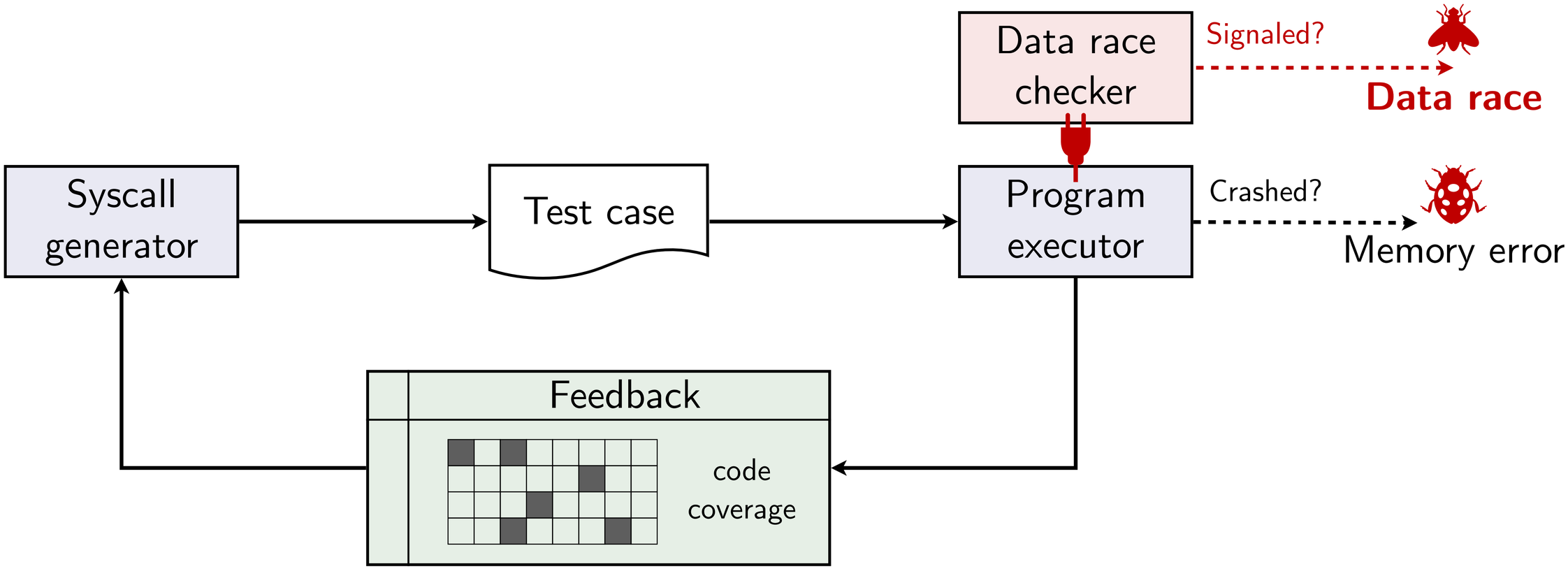
Checking data races - locking



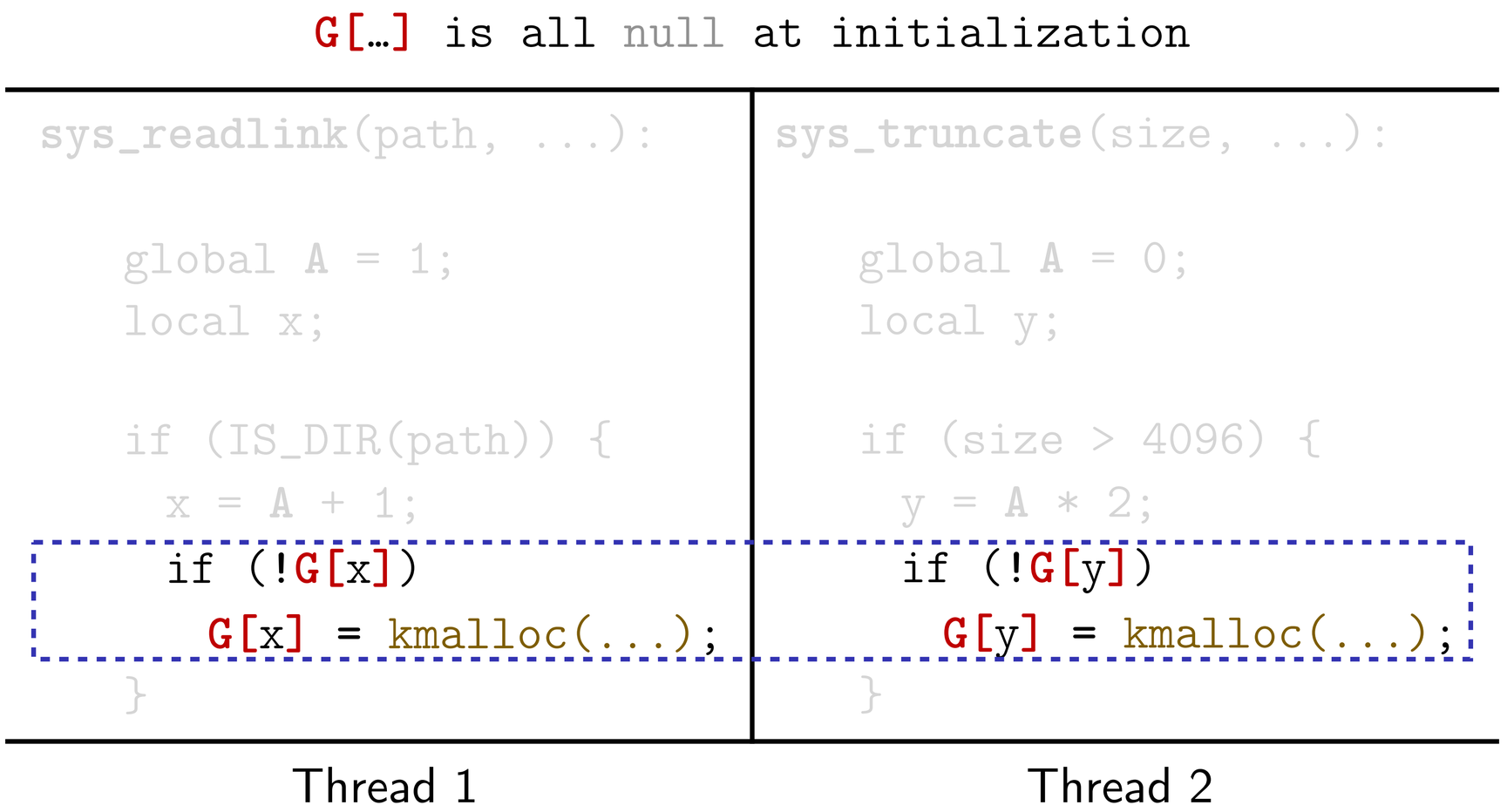
Checking data races - ordering (causality)



Bring out data races explicitly with a checker



A slightly complicated data race



*Assume sequential consistency.

A slightly complicated data race

`G[…]` is all `null` at initialization

Thread 1	Thread 2
`sys_readlink(path, ...):`	`sys_truncate(size, ...):`
`global A = 1;`	`global A = 0;`
`local x;`	`local y;`
`if (IS_DIR(path)) {`	`if (size > 4096) {`
`x = A + 1;`	`y = A * 2;`
`if (!G[x])`	`if (!G[y])`
`G[x] = kmalloc(...);`	`G[y] = kmalloc(...);`
`}`	`}`

*Assume sequential consistency.

Case simplified

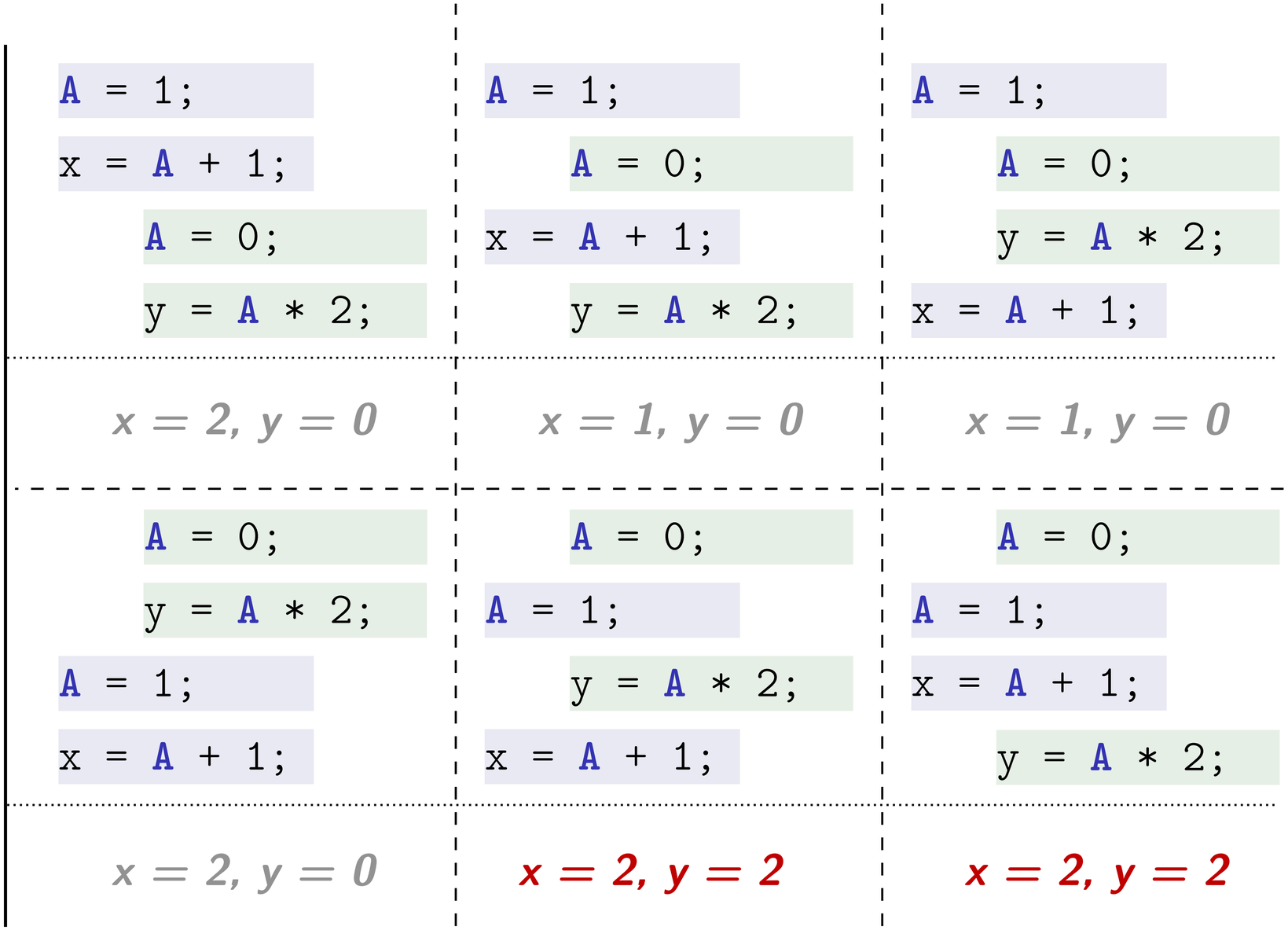
Thread 1	Thread 2
`A = 1;`	`A = 0;`
`x = A + 1;`	`y = A * 2;`

Can we reach x == y?

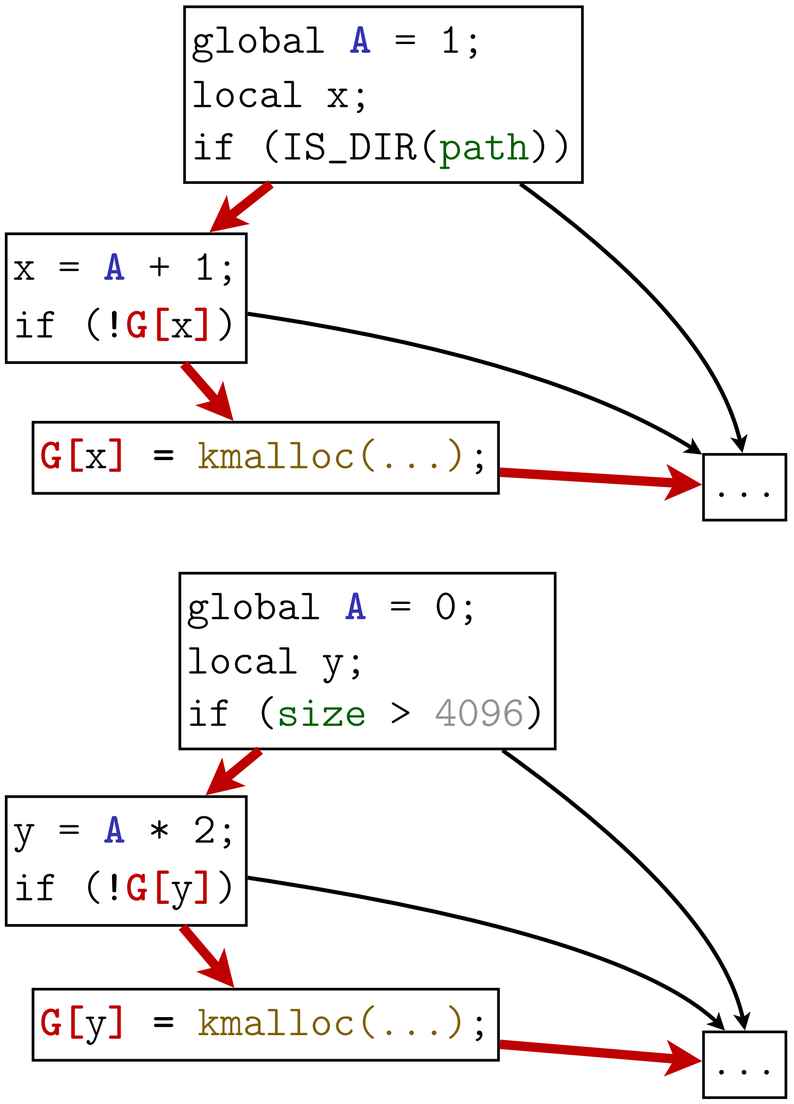
Case simplified

Thread 1	Thread 2
`A = 1;`	`A = 0;`
`x = A + 1;`	`y = A * 2;`

Can we reach x == y?



All interleavings yield to the same code coverage!




```
A = 1;
x = A + 1;
    A = 0;
    y = A * 2;
```

x = 2, y = 0

```
A = 1;
    A = 0;
x = A + 1;
    y = A * 2;
```

x = 1, y = 0

```
A = 1;
    A = 0;
    y = A * 2;
x = A + 1;
```

x = 1, y = 0

```
    A = 0;
    y = A * 2;
A = 1;
x = A + 1;
```

x = 2, y = 0

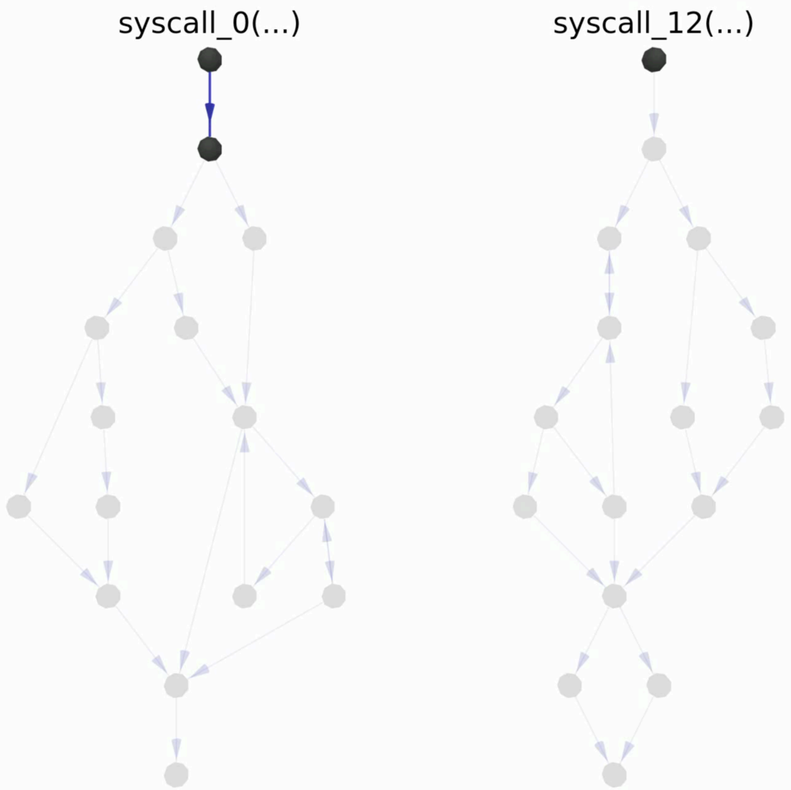
```
    A = 0;
A = 1;
    y = A * 2;
x = A + 1;
```

x = 2, y = 2

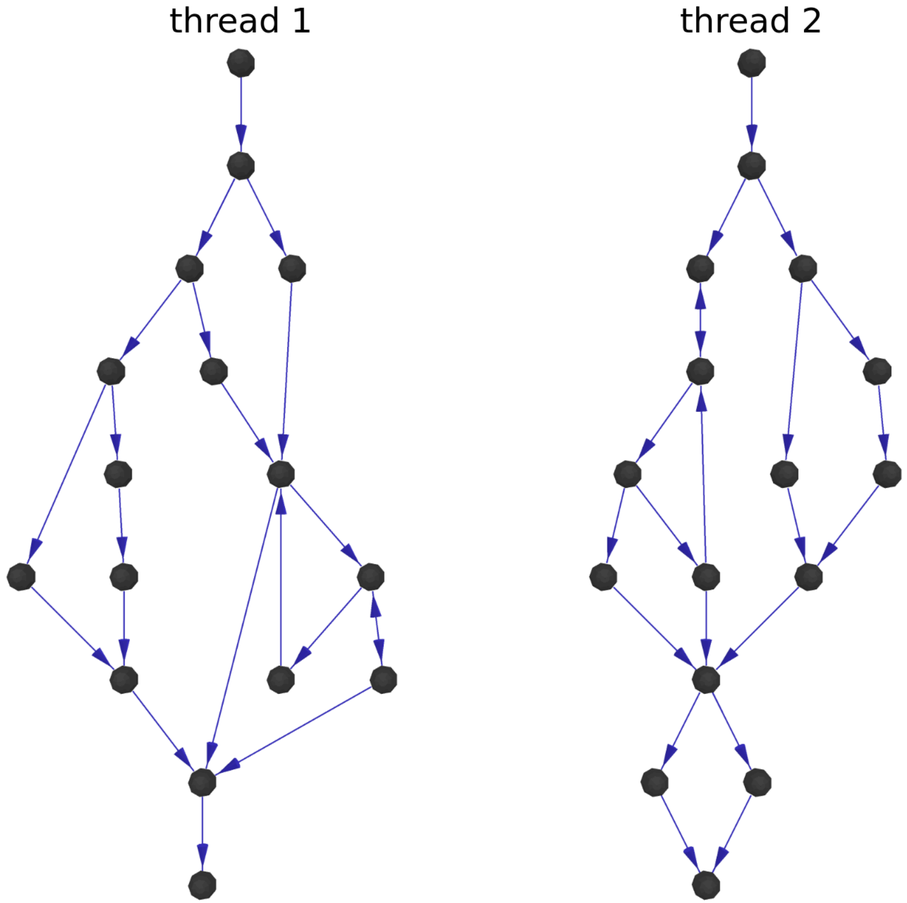
```
    A = 0;
A = 1;
x = A + 1;
    y = A * 2;
```

x = 2, y = 2

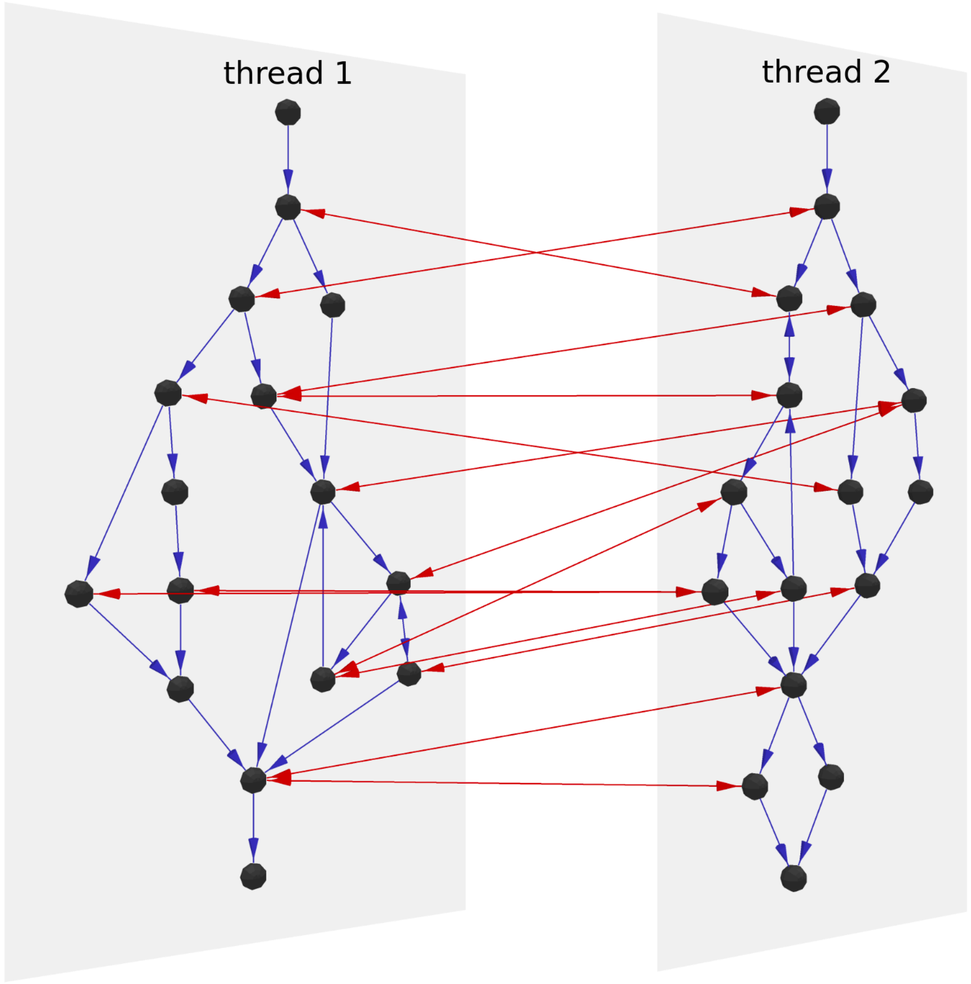
Incompleteness of CFG edge coverage



A multi-dimensional view of coverage in fuzzing

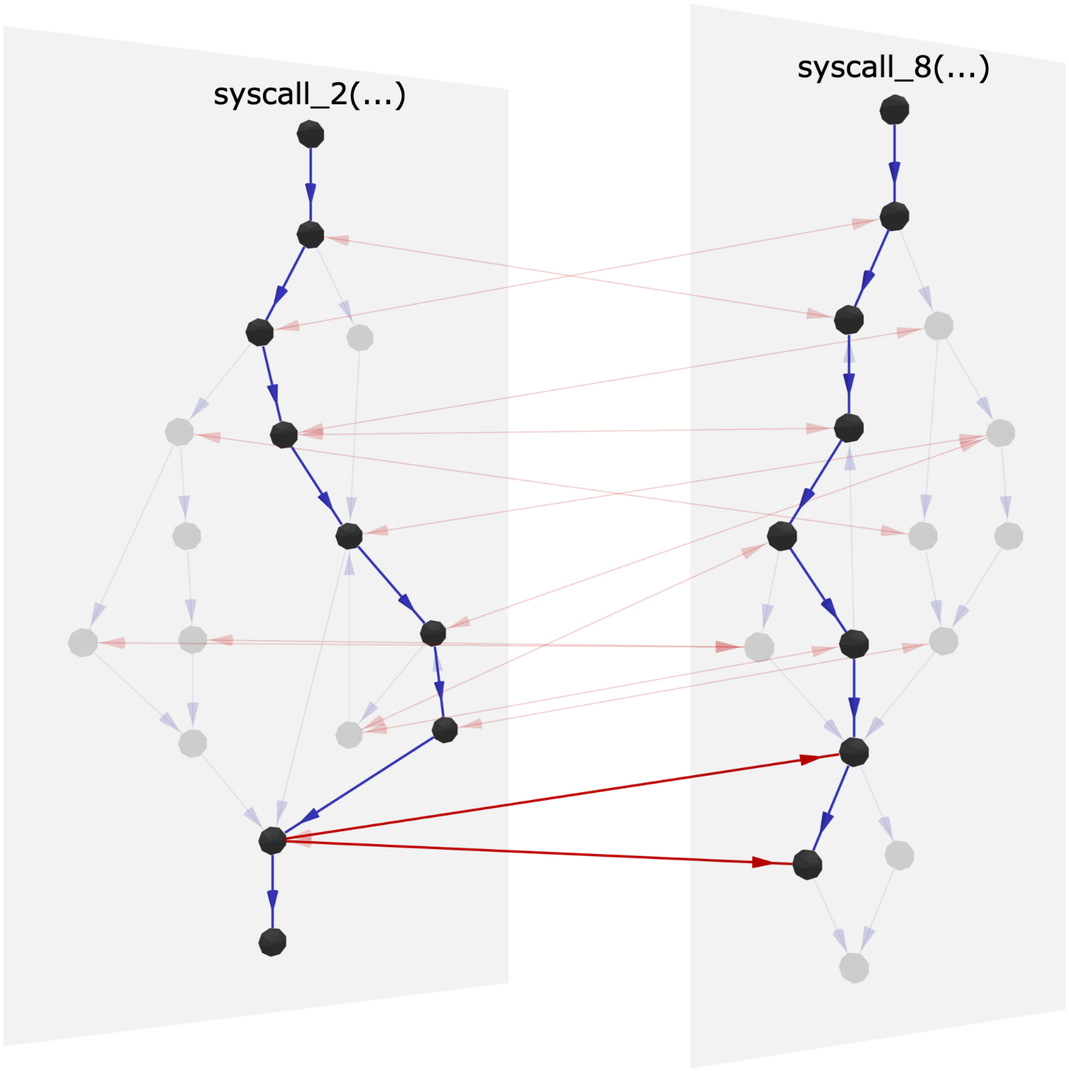


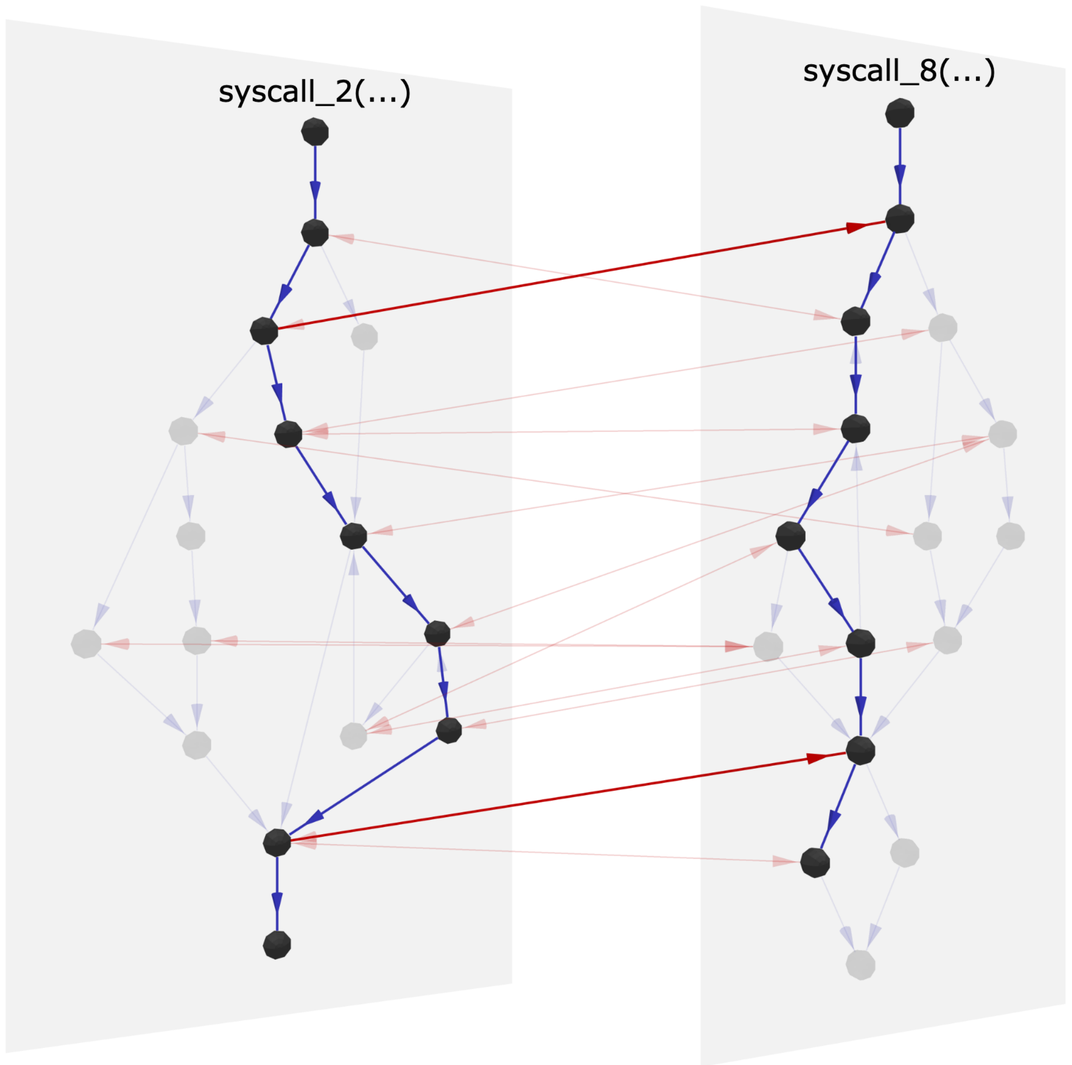
Edge-coverage only



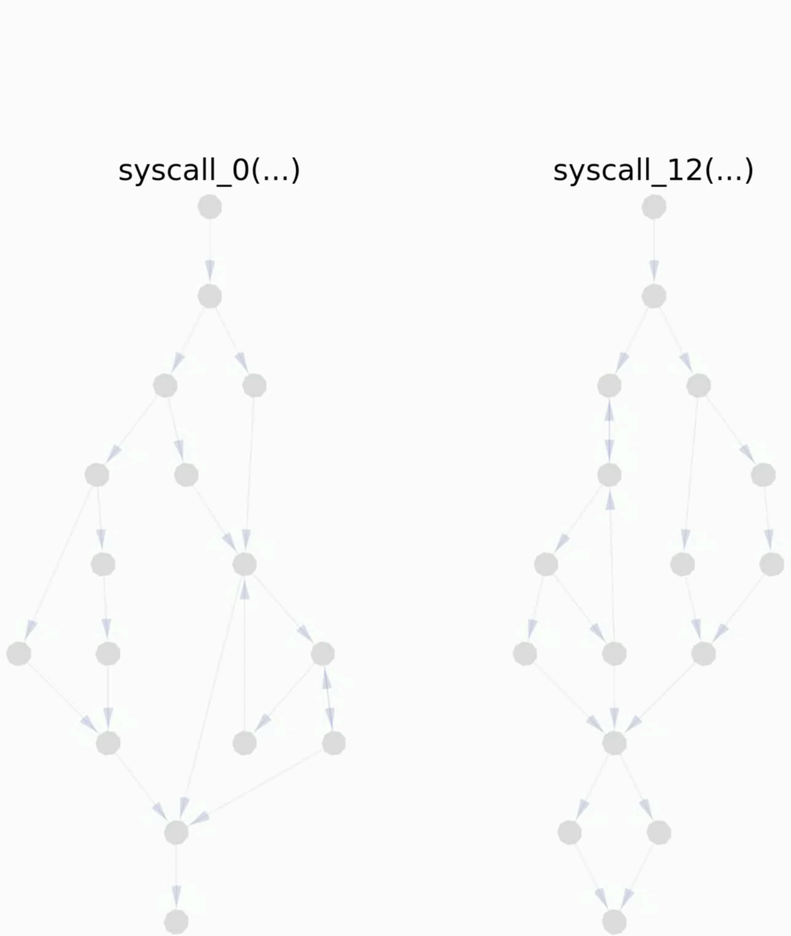
Krace

Visualizing the concurrency dimension

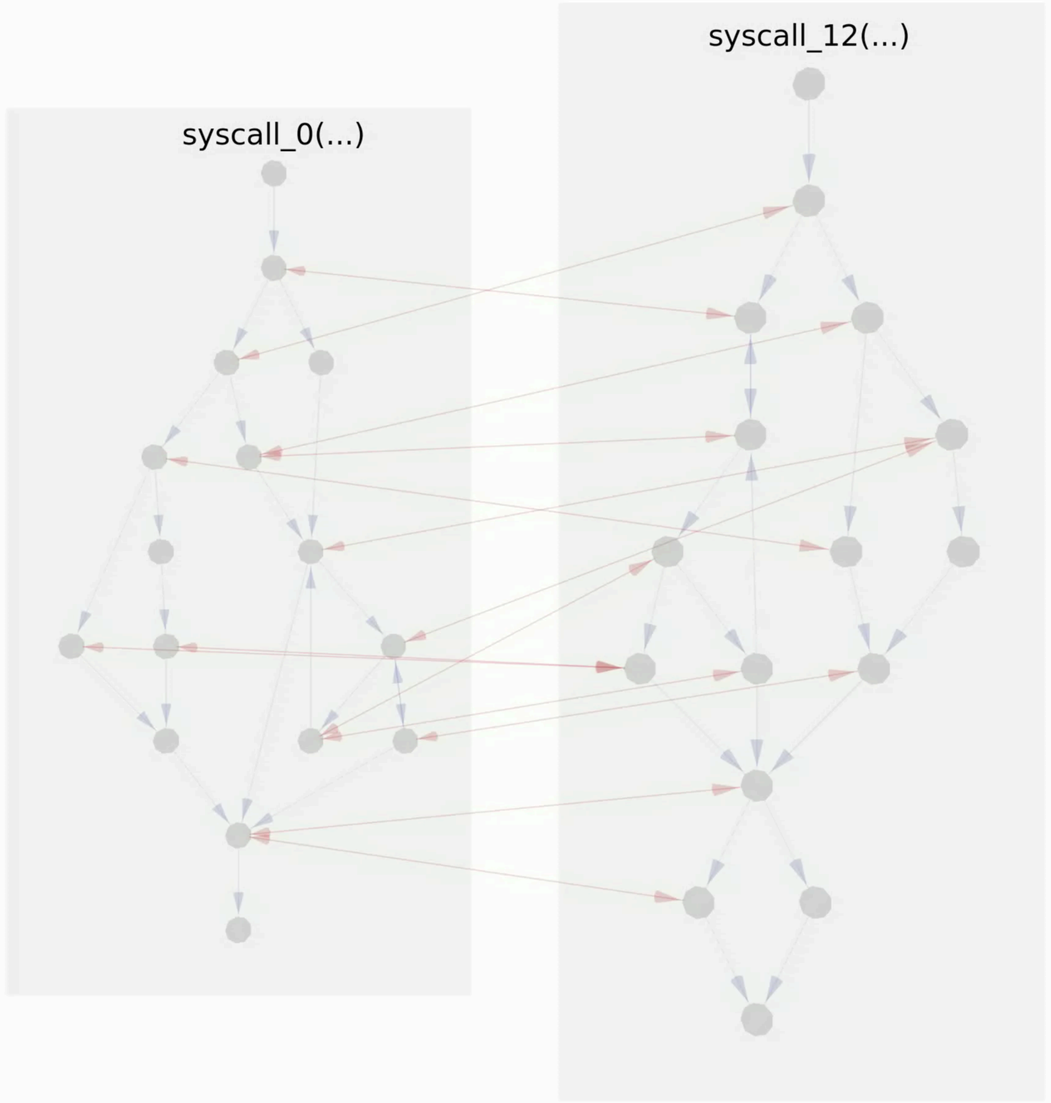




Visualizing the concurrency dimension

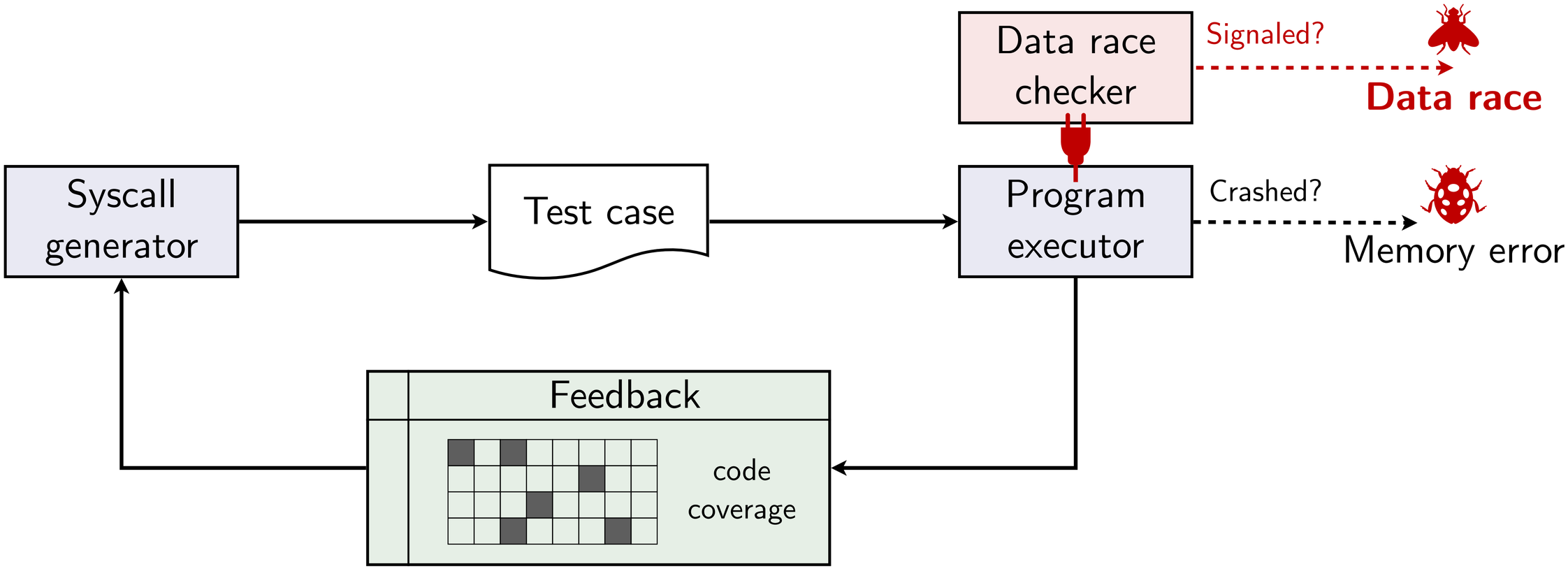


Edge-coverage only

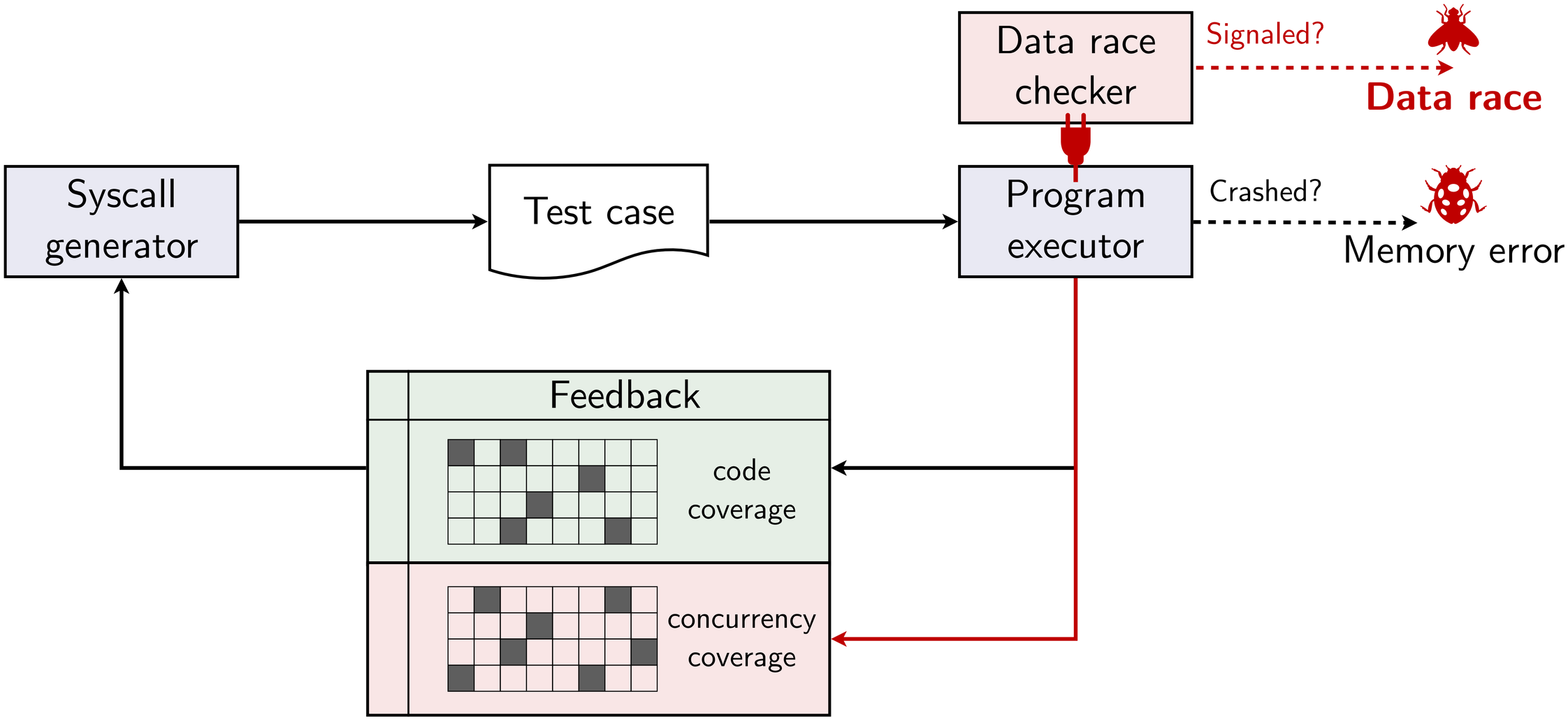


Krace

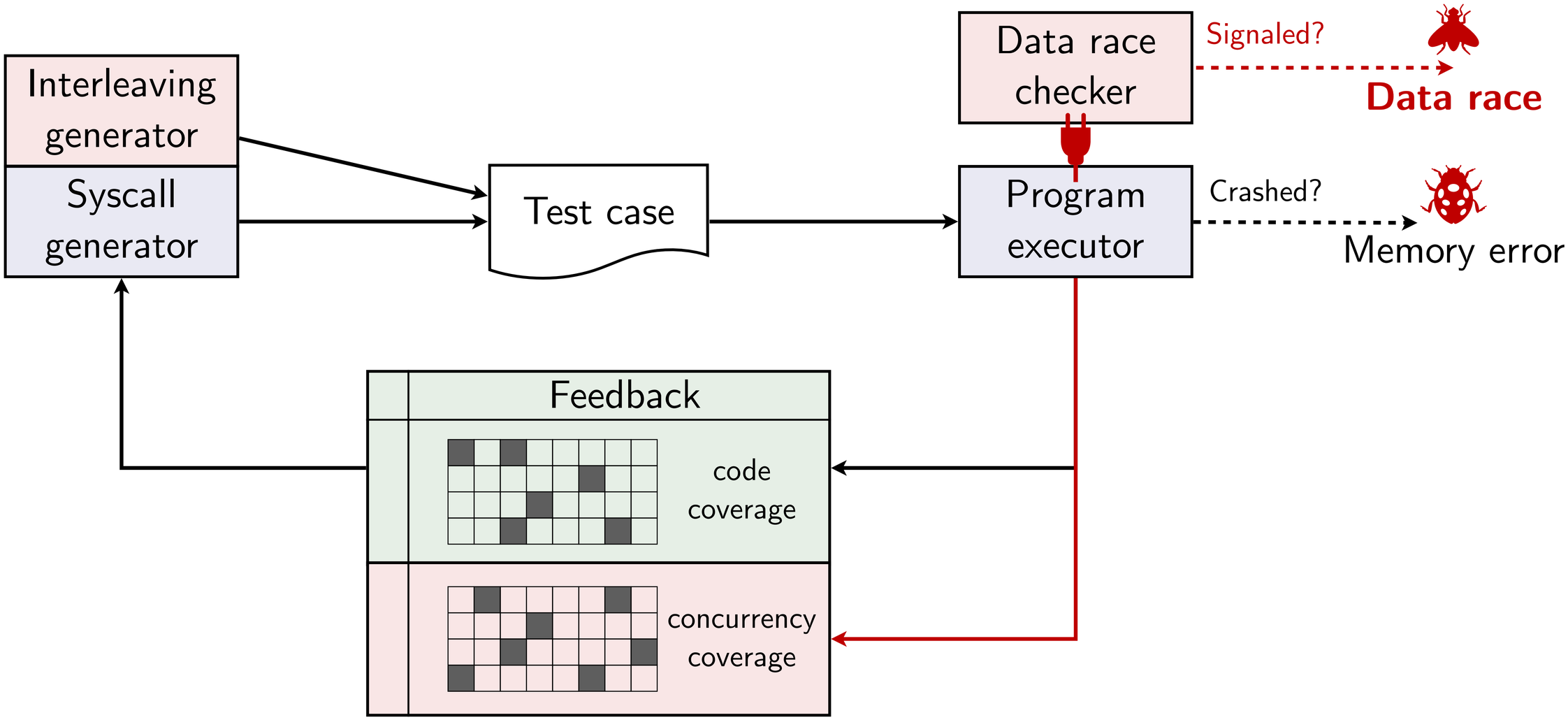
Bring fuzzing to the concurrency dimension



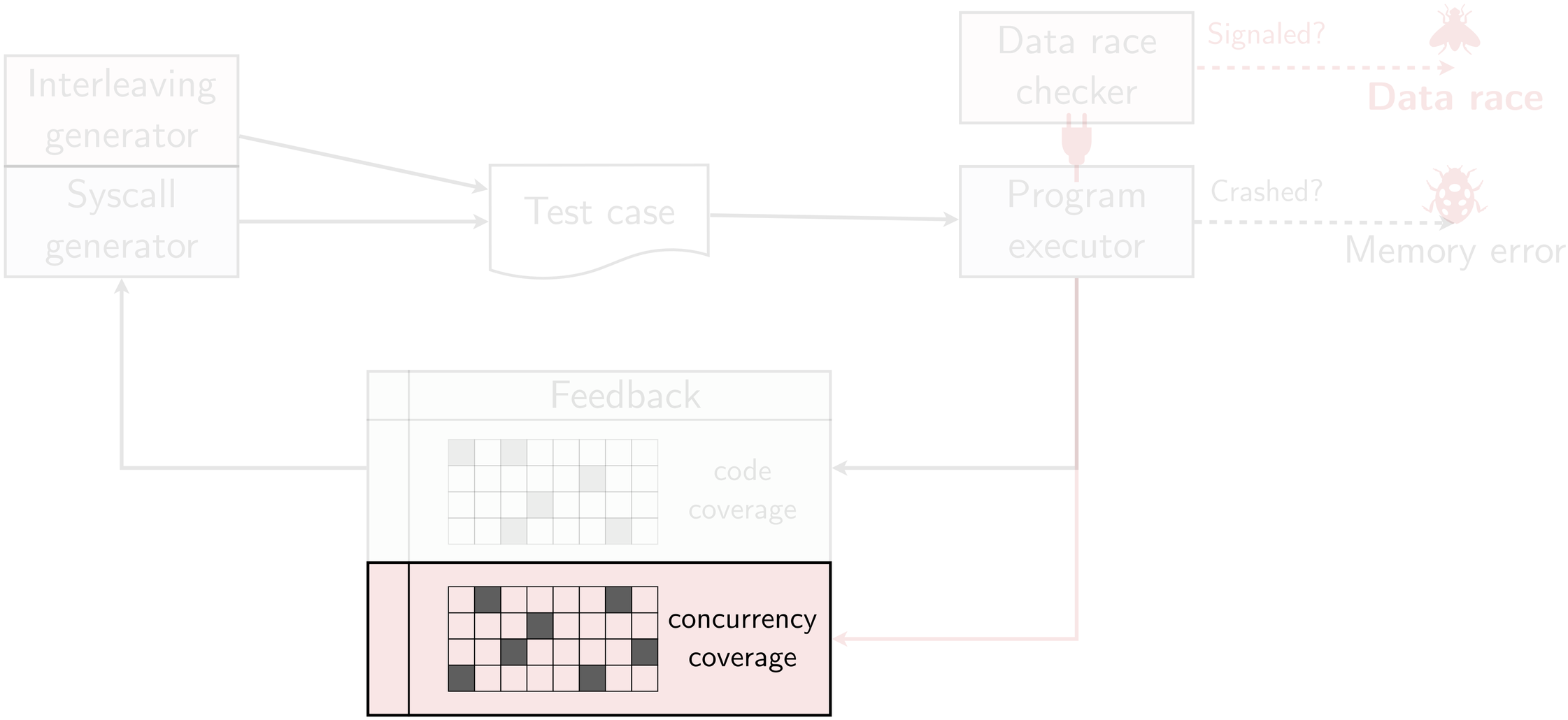
Bring fuzzing to the concurrency dimension



Bring fuzzing to the concurrency dimension



Concurrency coverage tracking



A straw-man solution

```
sys_readlink(path, ...):

i1  global A = 1;
i2  local x;

i3  if (IS_DIR(path)) {
i4    x = A + 1;
i5    if (G[x])
i6      kmalloc(...);
    }
```

Thread 1

```
sys_truncate(size, ...):

i7  global A = 0;
i8  local y;

i9  if (size > 4096) {
i10   y = A * 2;
i11   if (G[y])
i12     kmalloc(...);
    }
```

Thread 2

A straw-man solution

```
sys_readlink(path, ...):

i1  global A = 1;
i2  local x;

i3  if (IS_DIR(path)) {
i4    x = A + 1;
i5    if (G[x])
i6      kmalloc(...);
    }
```

Thread 1

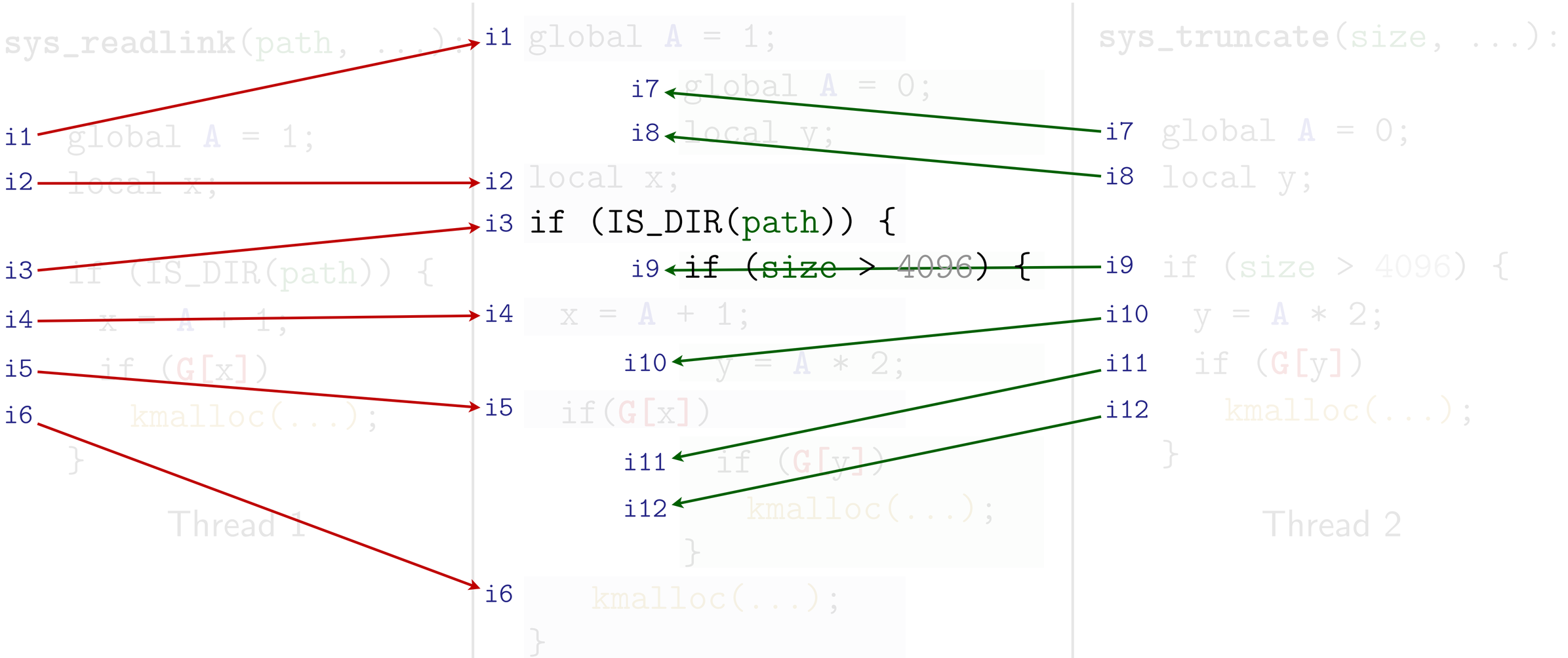
```
i1 global A = 1;
        i7 global A = 0;
        i8 local y;
i2 local x;
i3 if (IS_DIR(path)) {
        i9 if (size > 4096) {
i4   x = A + 1;
        i10   y = A * 2;
i5   if(G[x])
        i11   if (G[y])
        i12     kmalloc(...);
              }
i6     kmalloc(...);
   }
```

A possible interleaving

```
sys_truncate(size, ...):

i7  global A = 0;
i8  local y;

i9  if (size > 4096) {
i10   y = A * 2;
i11   if (G[y])
i12     kmalloc(...);
    }
```

Thread 2

A straw-man solution



Hash(i1, i7, i8, i2, ***i3***, ***i9***, i4, i10, i5, i11, i12, i6) = 7825

Hash(i1, i7, i8, i2, ***i9***, ***i3***, i4, i10, i5, i11, i12, i6) = 1356

A straw-man solution

```
sys_readlink(path, ...):
i1  global A = 1;
i2  local x;
i3  if (IS_DIR(path)
i4    x = A + 1;
i5    if (G[x])
i6      kmalloc(...);
    }
```

Thread 1

```
i1 global A = 1;
i6     kmalloc(...);
   }
```

A possible interleaving

```
sys_truncate(size, ...):
lobal A = 0;
ocal y;
f (size > 4096) {
y = A * 2;
if (G[y])
  kmalloc(...);
```

Thread 2

Number of possible interleavings of two threads

If two threads have m and n instructions respectively, then the number interleavings between them is given by:

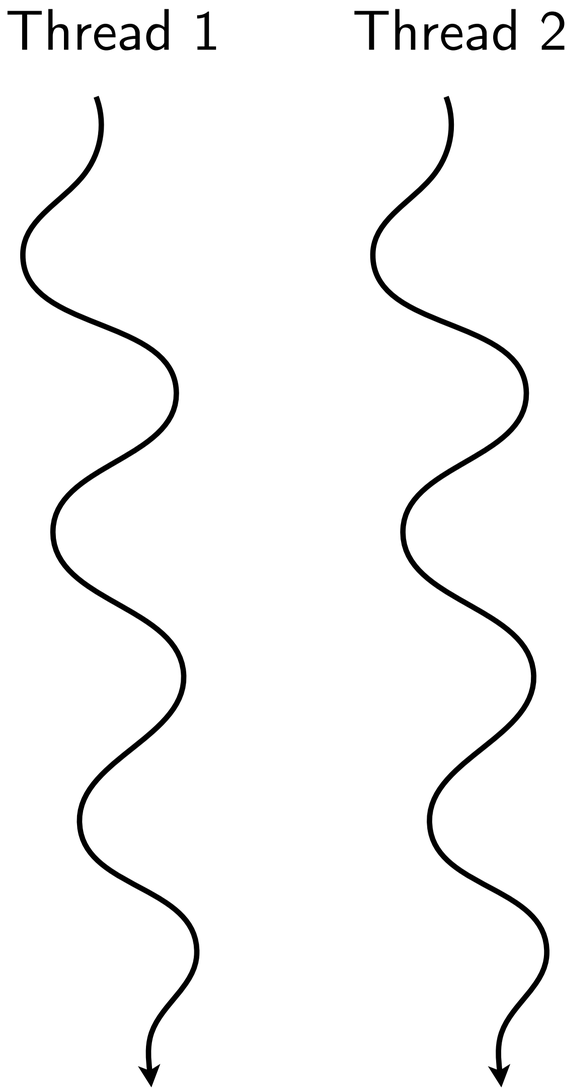
$$\frac{(m+n)!}{m! \times n!}$$

$m = n = 2$	$m = n = 4$	$m = n = 8$	$m = n = 16$
6	70	$13K$	$601M$

Observations on practical interleaving tracking

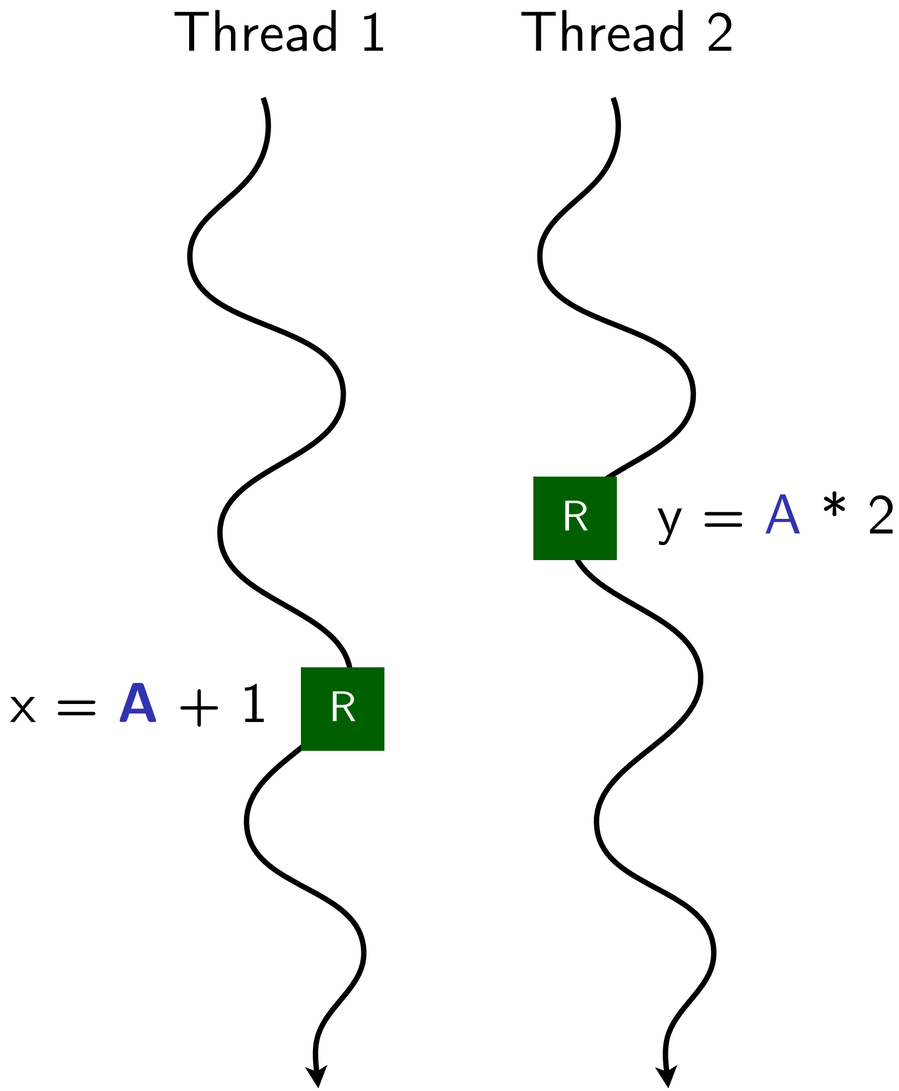
Observations on practical interleaving tracking

- Only interleaved accesses to shared memory matters
  - In an extreme case where two threads do not shared memory, they interleaving does not matter at all.



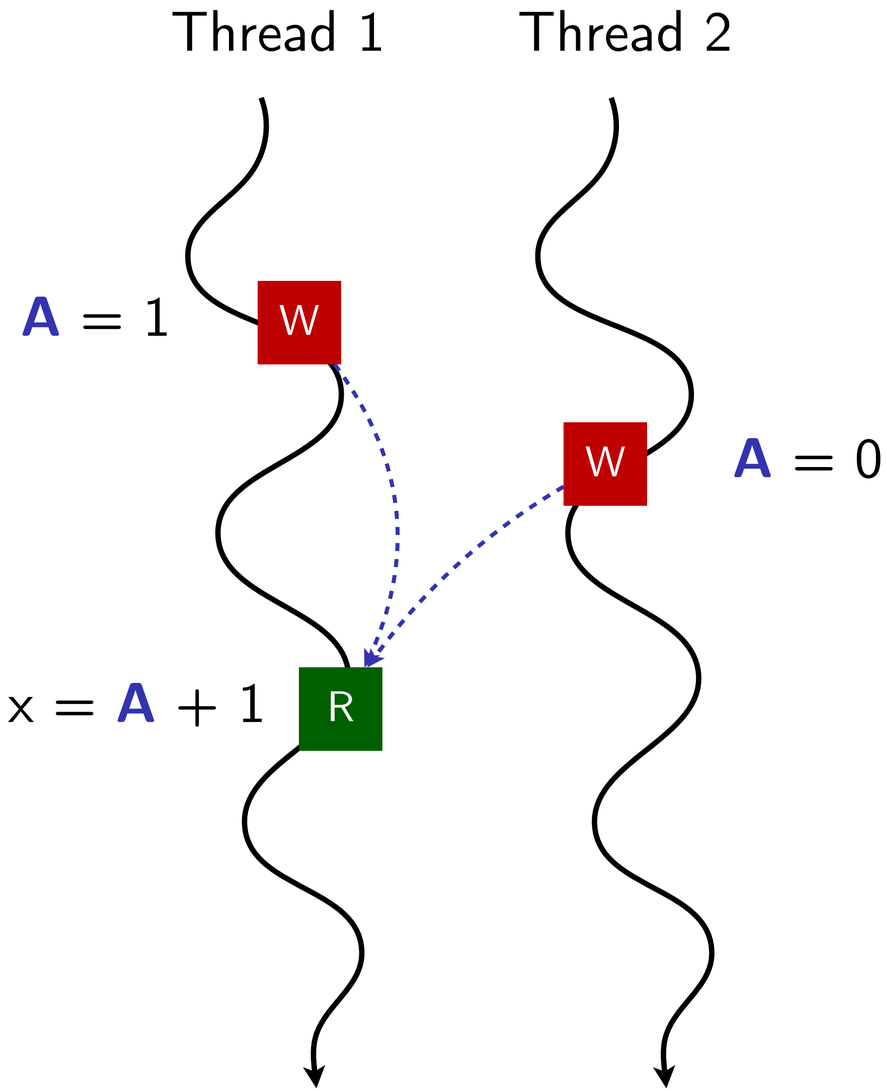
Observations on practical interleaving tracking

- Only interleaved accesses to shared memory matters
  - In an extreme case where two threads do not shared memory, they interleaving does not matter at all.
- Only interleaved **read-write** accesses to shared memory locations matters
  - In an extreme case where two threads only read from shared memory, they interleaving does not matter at all.



Observations on practical interleaving tracking

- Only interleaved accesses to shared memory matters
  - In an extreme case where two threads do not shared memory, they interleaving does not matter at all.
- Only interleaved **read-write** accesses to shared memory locations matters
  - In an extreme case where two threads only read from shared memory, they interleaving does not matter at all.
- Thread interleaving alters the **def-use relation** of memory locations!



Observations on practical interleaving tracking

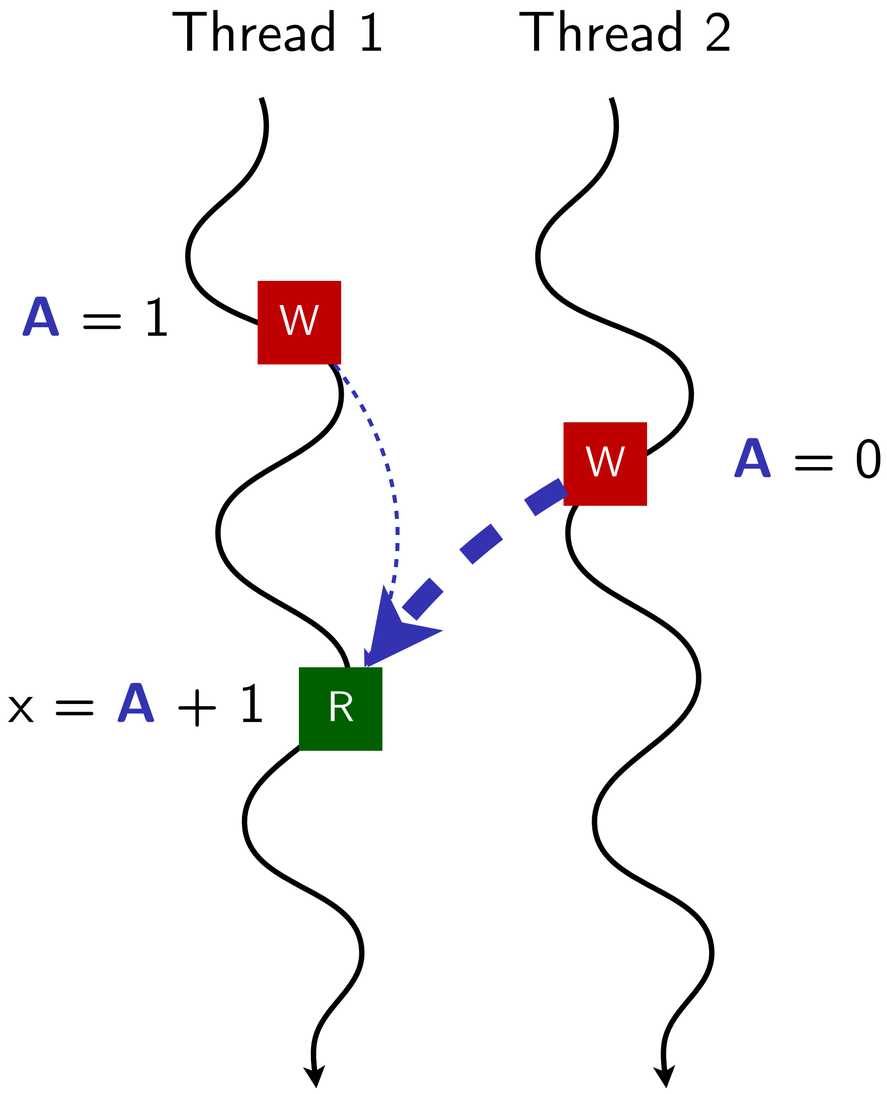
- Only interleaved accesses to shared memory matters
  - In an extreme case where two threads do not shared memory, they interleaving does not matter at all.

Interleaving approximation

Track cross-thread write-to-read (def-to-use) edges!

- Only interleaved accesses to shared memory matters
  - In an extreme case where two threads only read from shared memory, they interleaving does not matter at all.

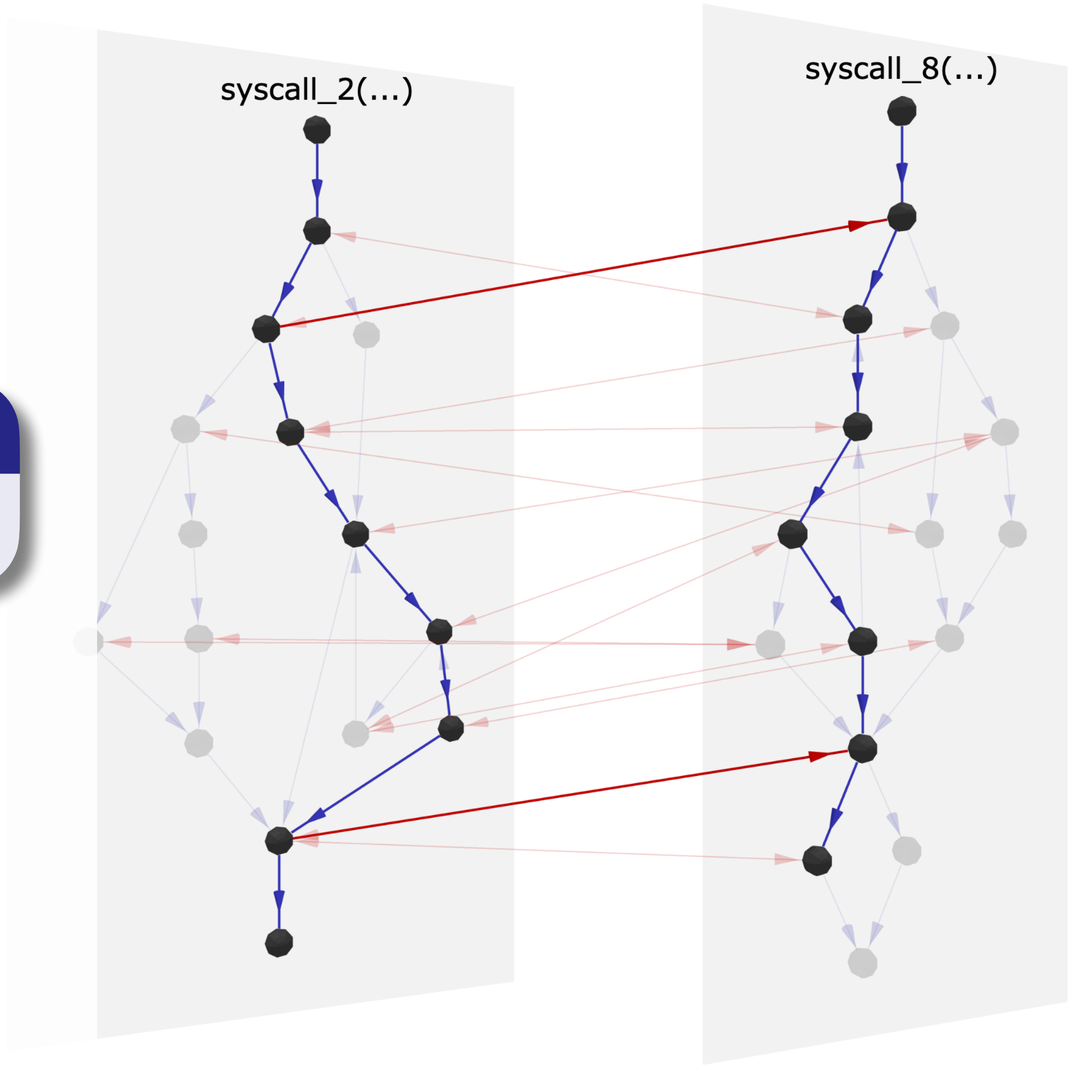
- Thread interleaving alters the **def-use relation** of memory locations!



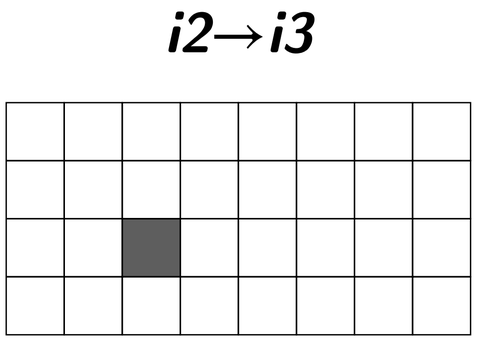
Observations on practical interleaving tracking

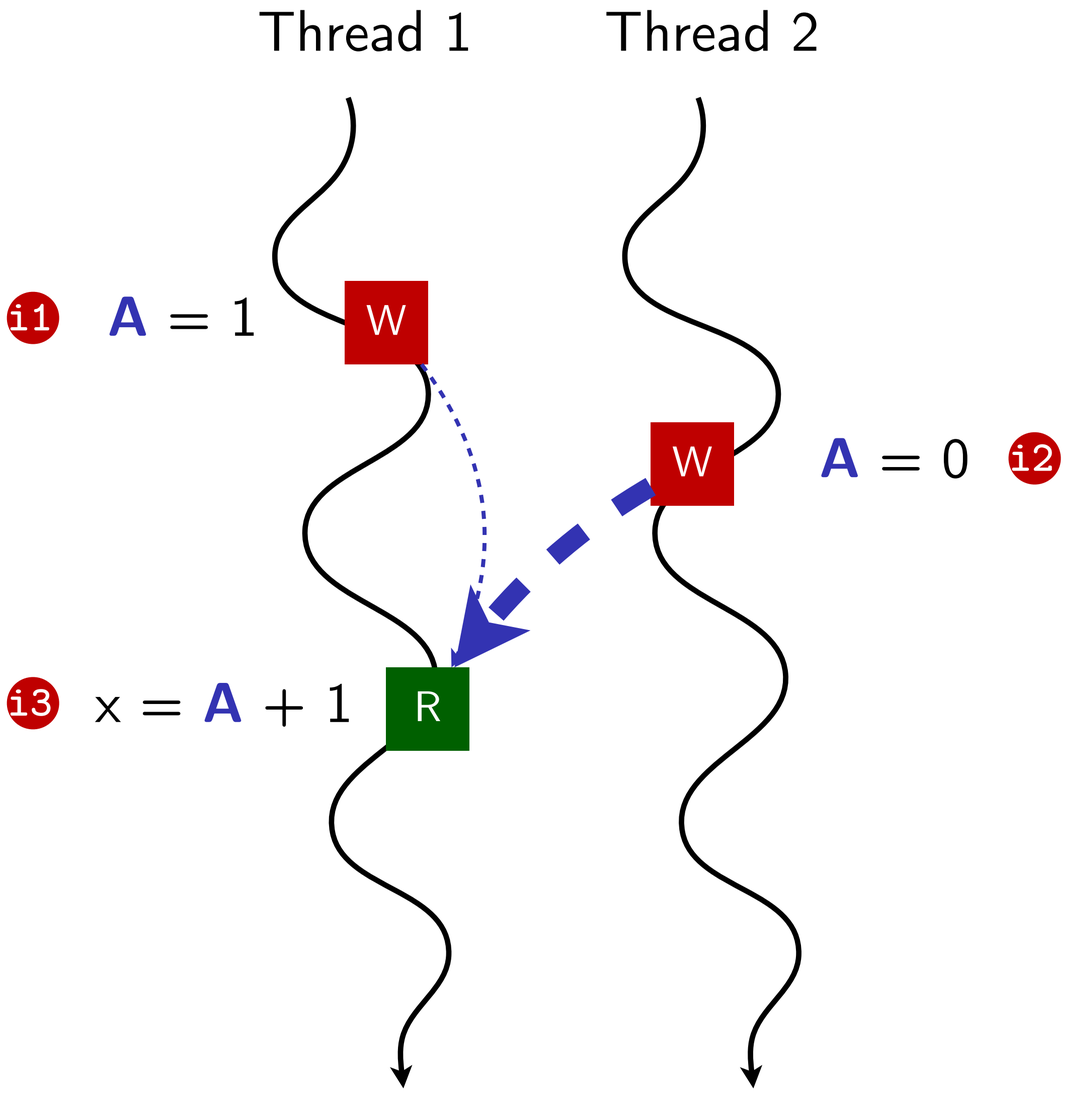
Interleaving approximation

Track cross-thread write-to-read (def-to-use) edges!



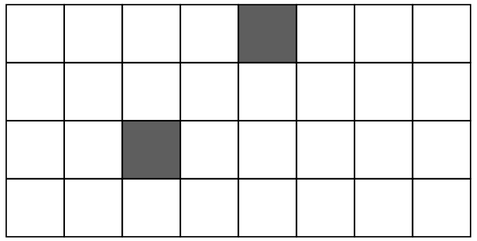
Aliased-instruction coverage

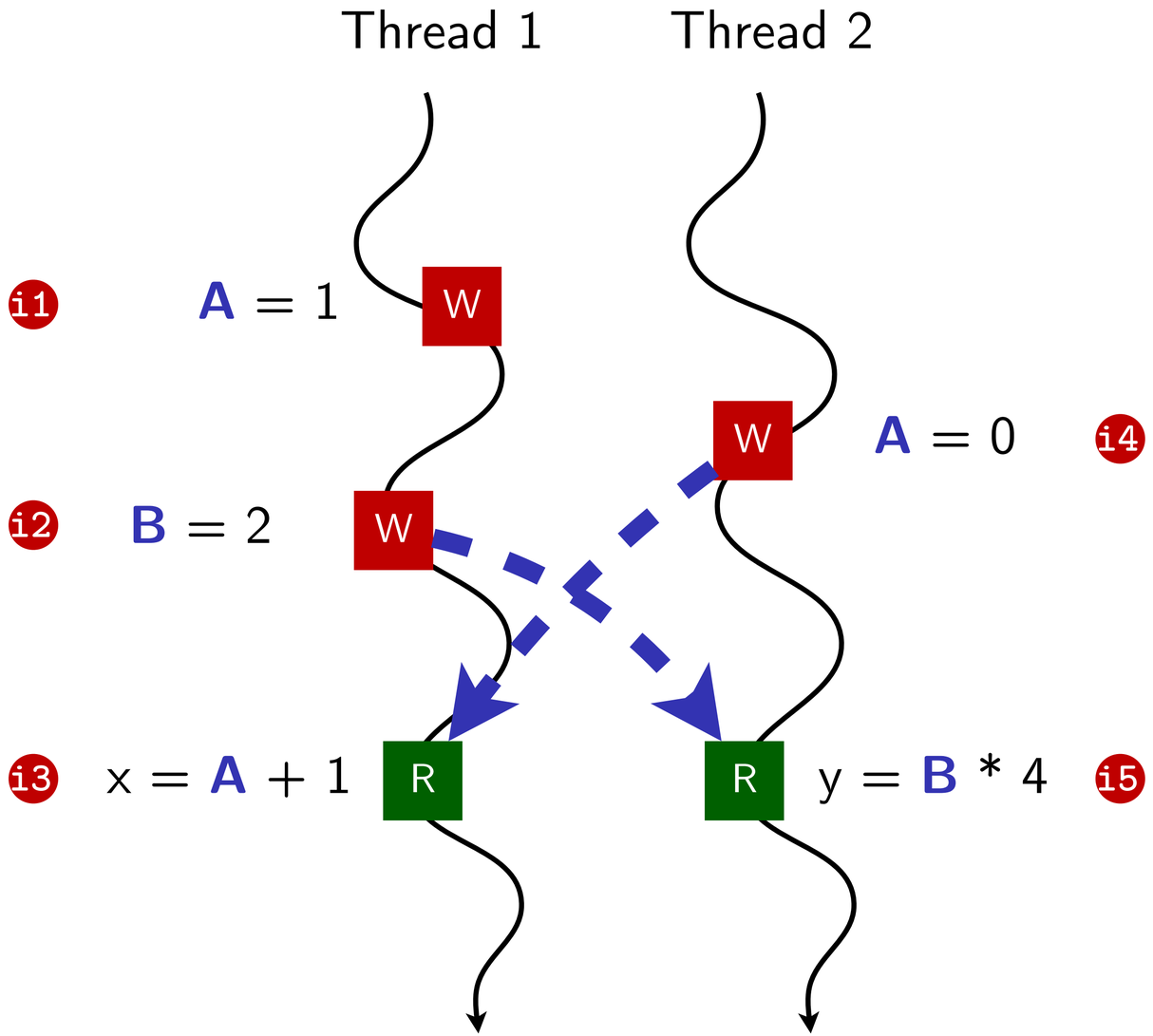




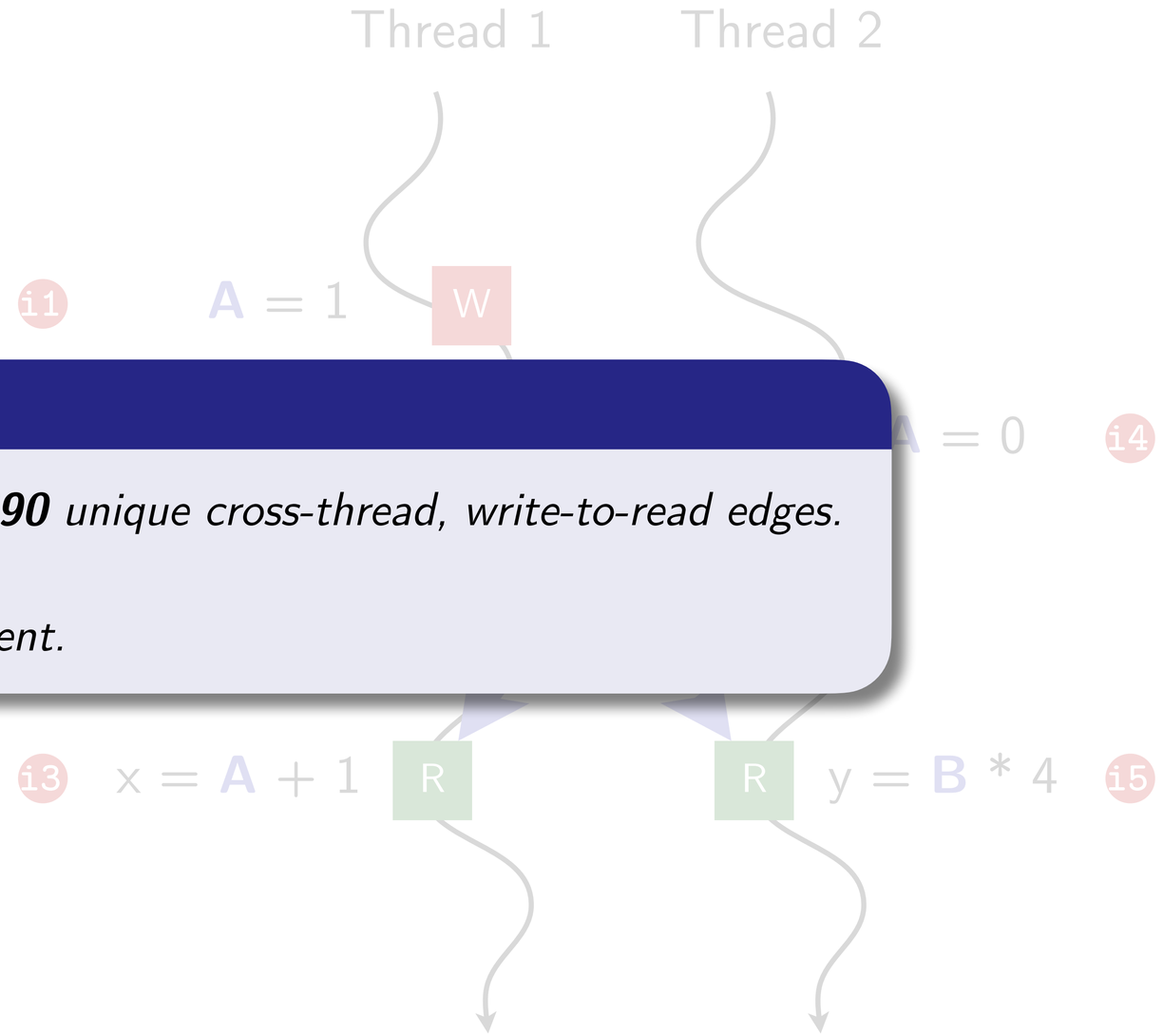
Aliased-instruction coverage

i2→i5, i4→i3





Aliased-instruction coverage

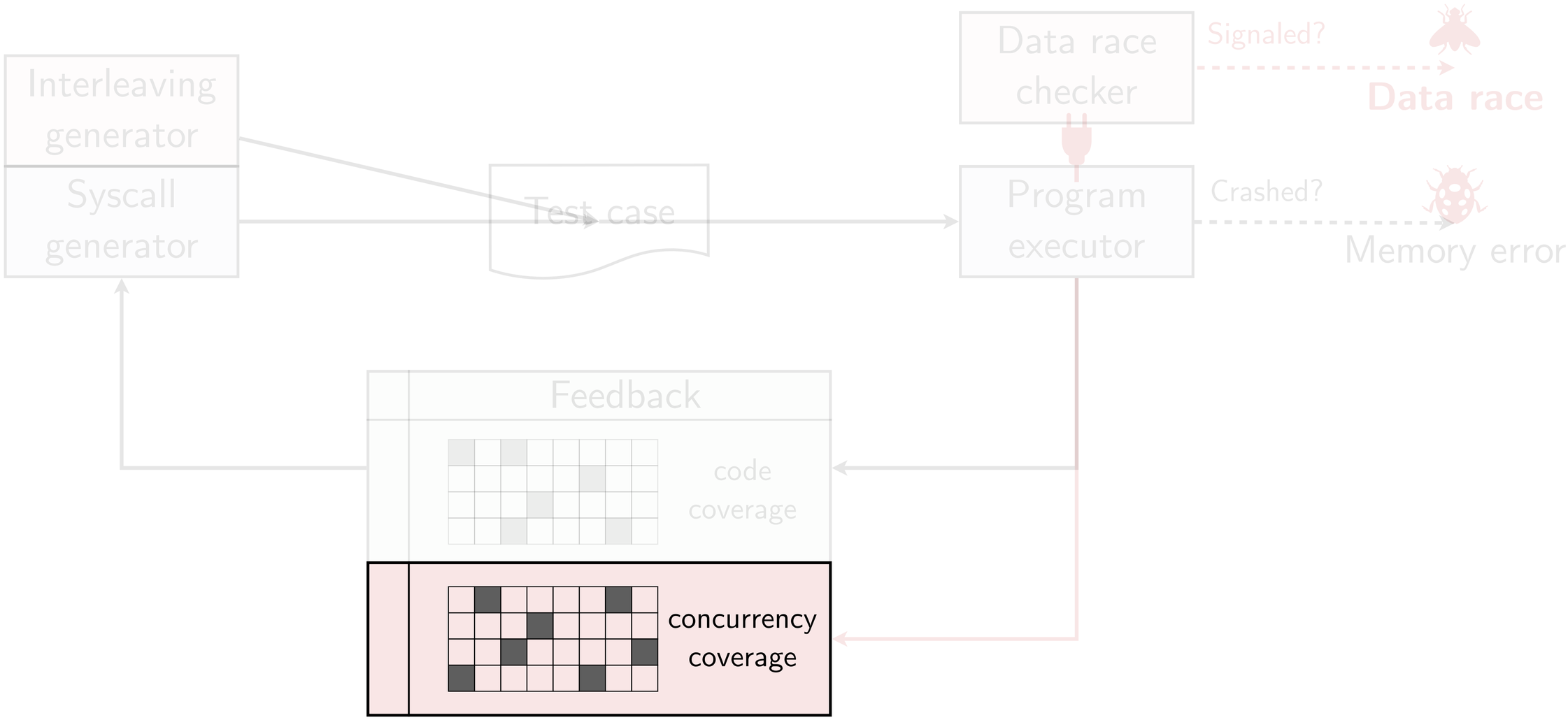


Concurrency coverage bitmap size

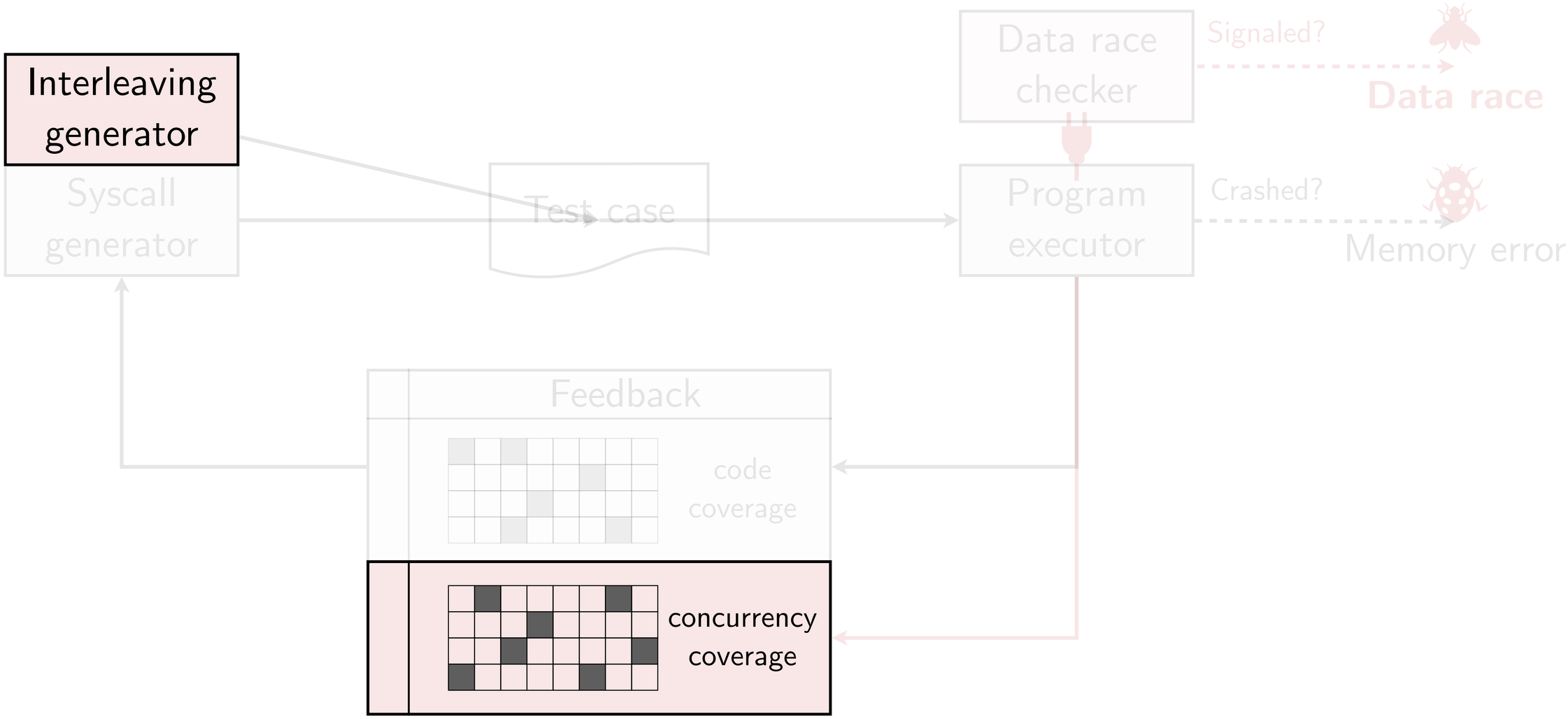
*During our experiment, we observed **63,590** unique cross-thread, write-to-read edges.*

$\rightarrow$ *a bitmap size of **128KB** will be sufficient.*

Concurrency coverage tracking

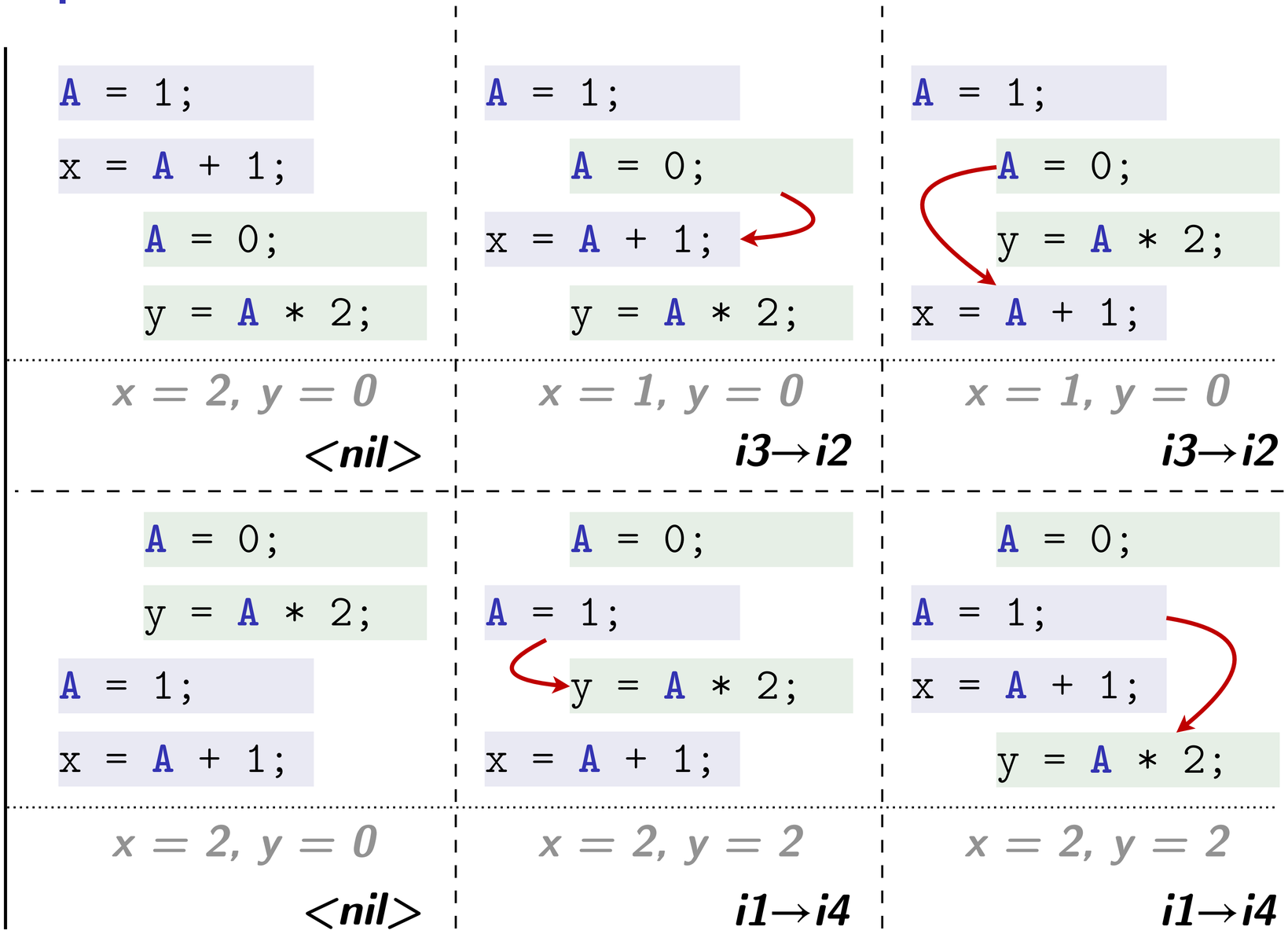


Interleaving exploration

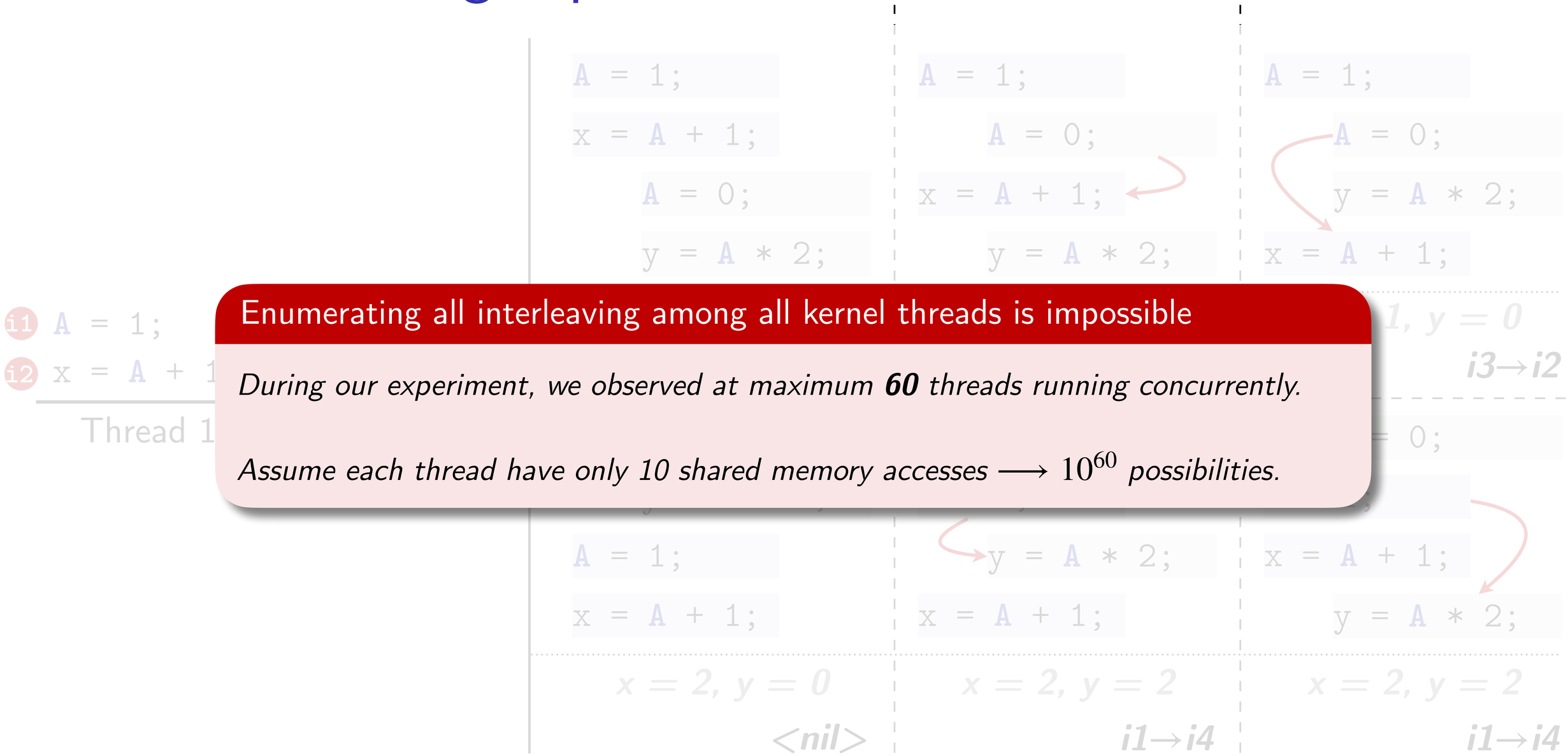


Active interleaving exploration - ideal case

Thread 1	Thread 2
i1 `A = 1;`	`A = 0;` i3
i2 `x = A + 1;`	`y = A * 2;` i4



Active interleaving exploration - ideal case



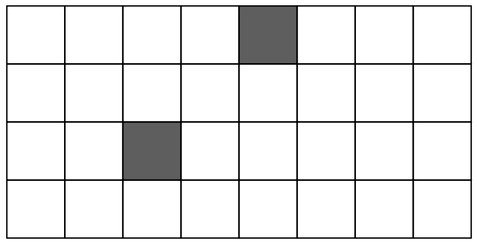
Enumerating all interleaving among all kernel threads is impossible

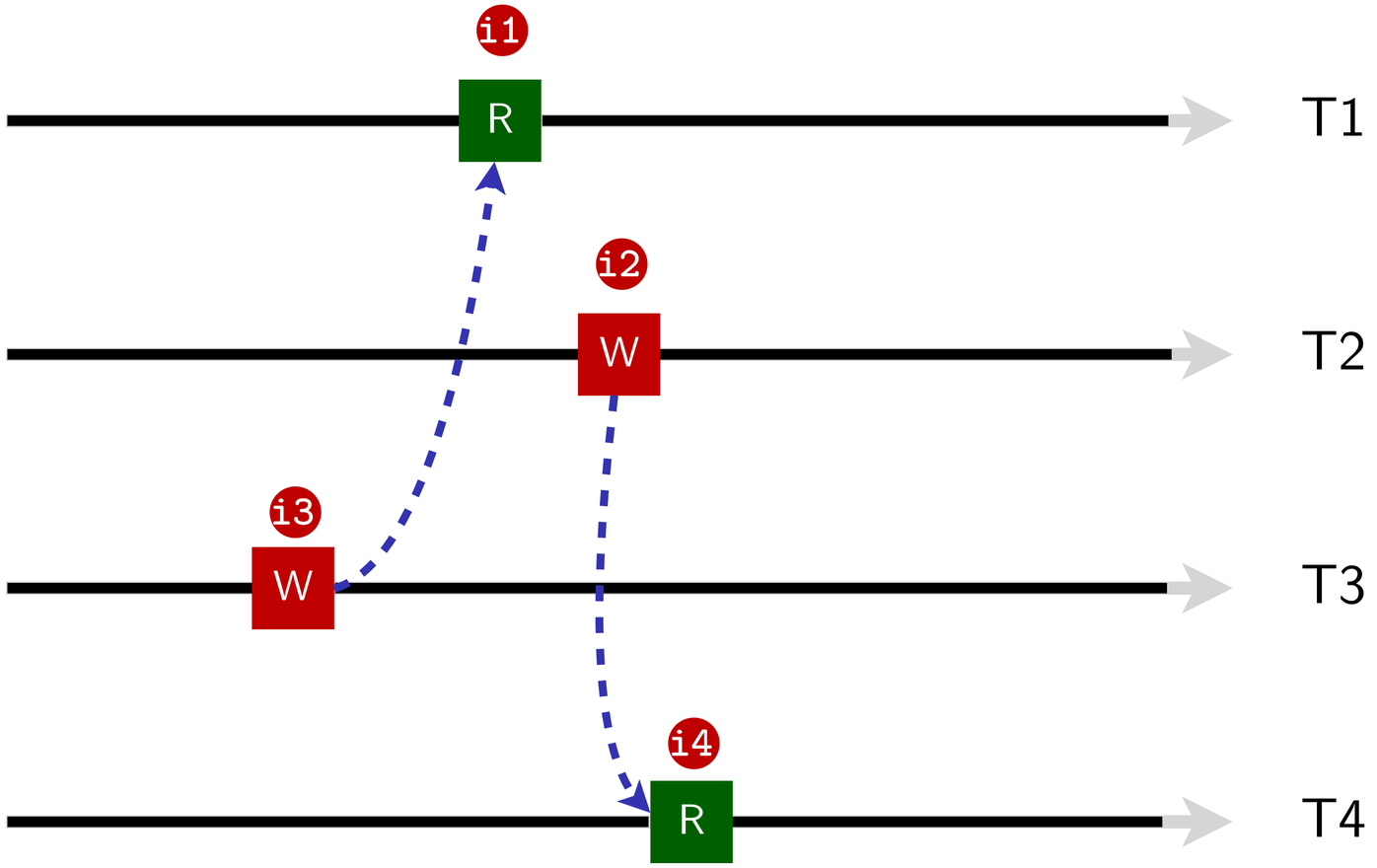
*During our experiment, we observed at maximum **60** threads running concurrently.*

Assume each thread have only 10 shared memory accesses $\longrightarrow 10^{60}$ *possibilities.*

Active interleaving exploration through delay injection

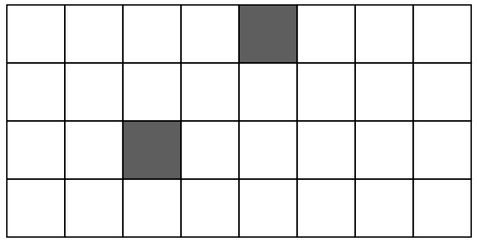
Concurrency coverage

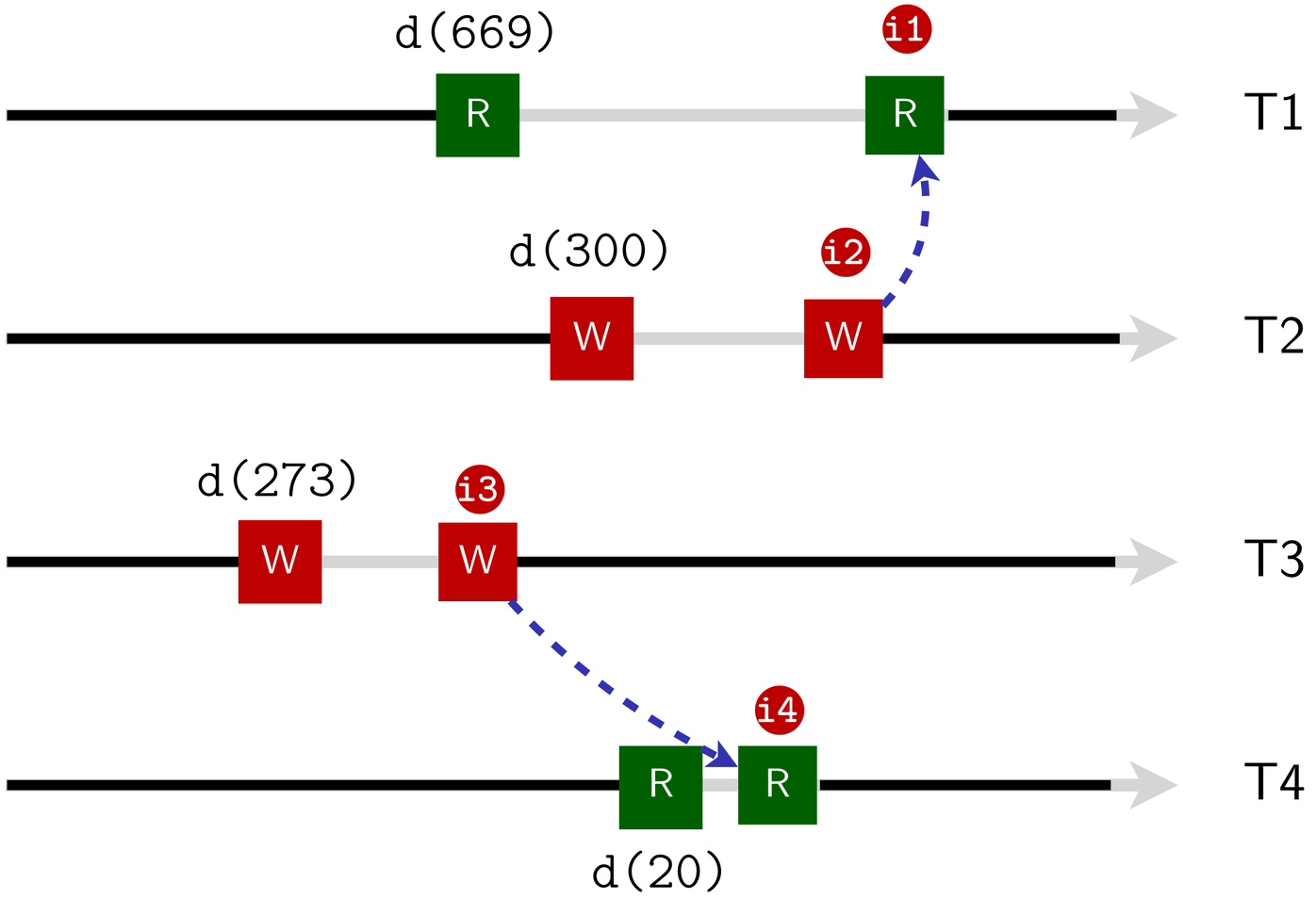




Active interleaving exploration through delay injection

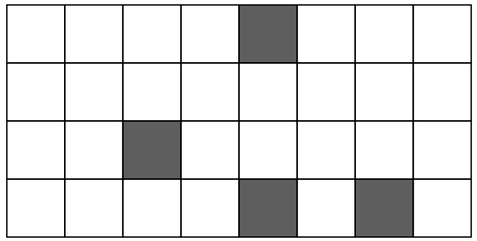
Concurrency coverage

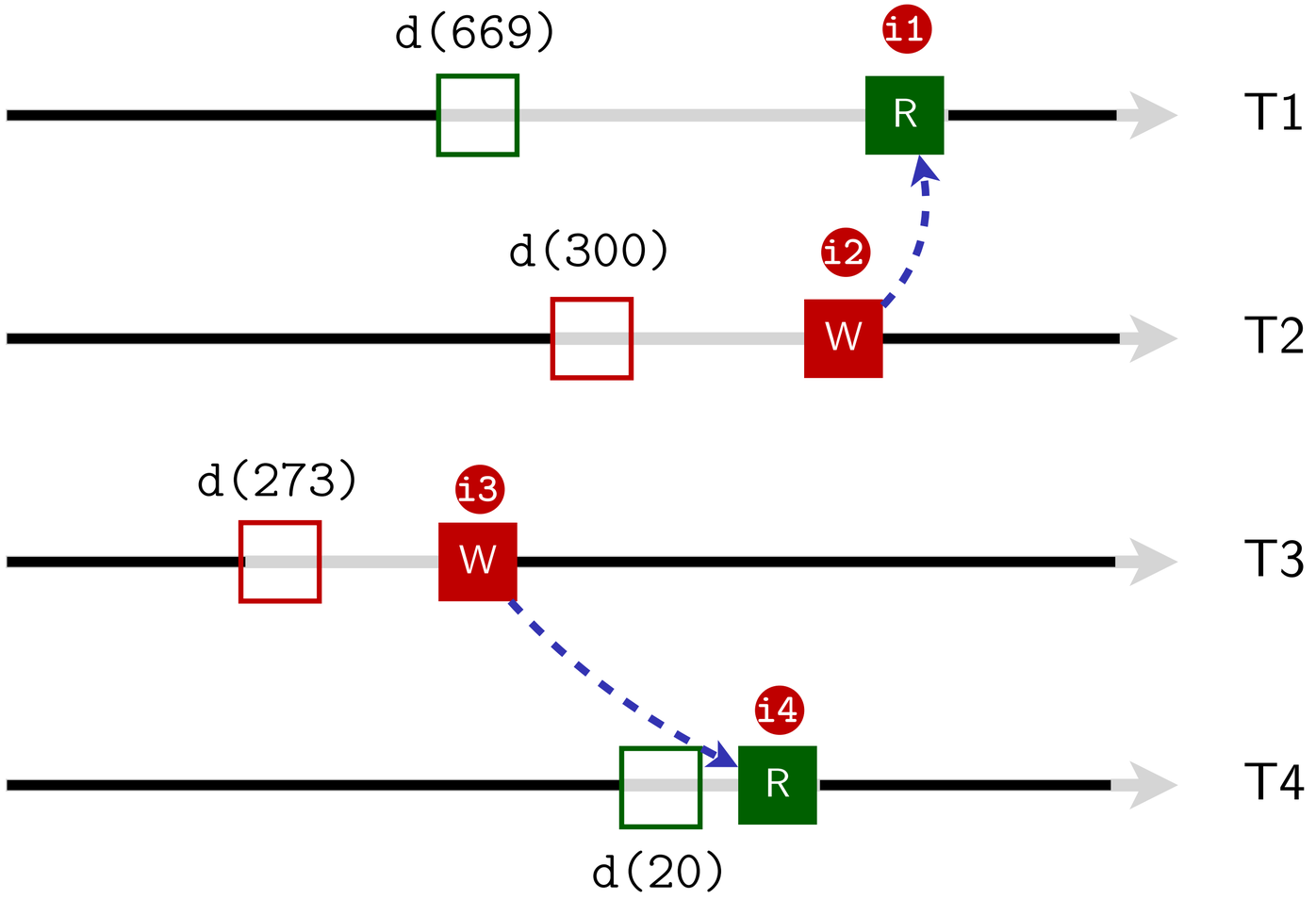




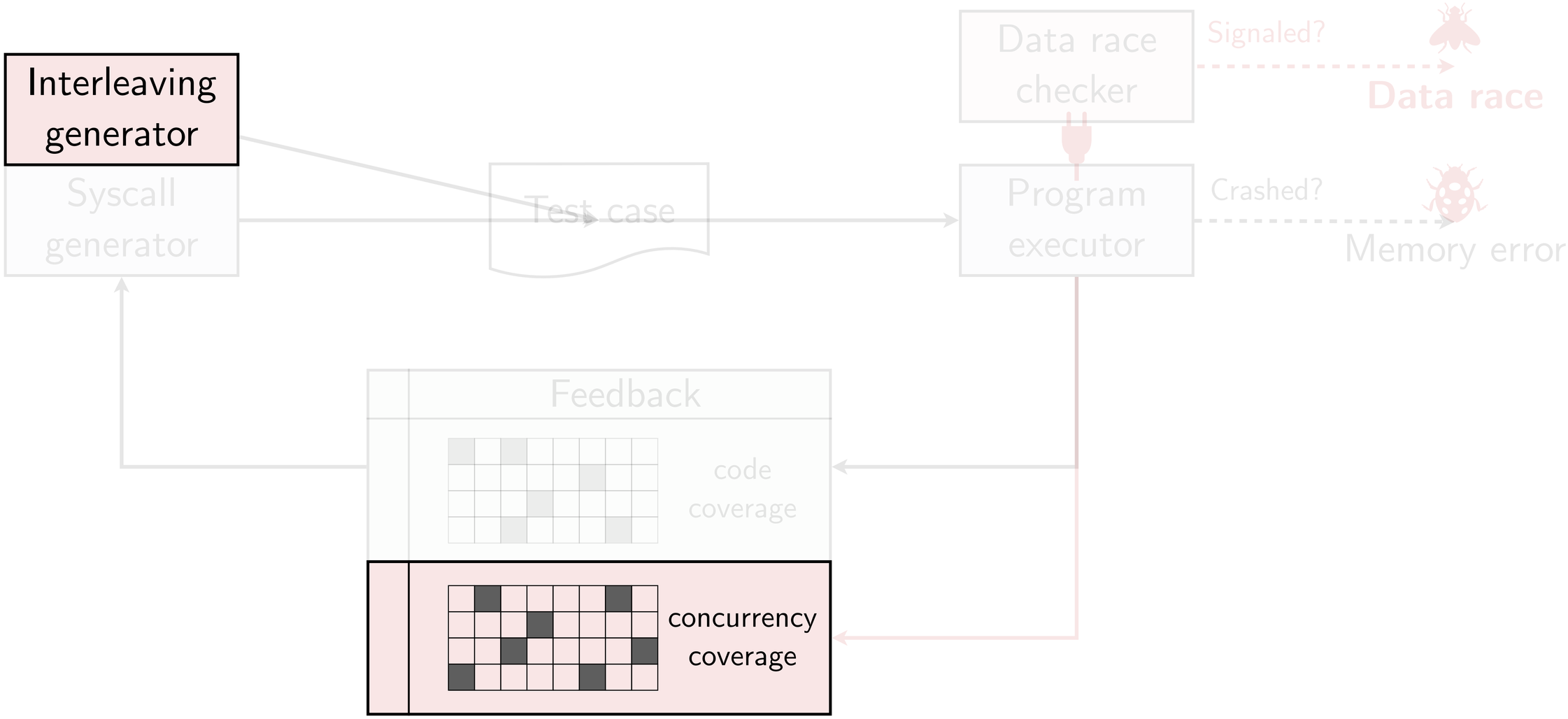
Active interleaving exploration through delay injection

Concurrency coverage

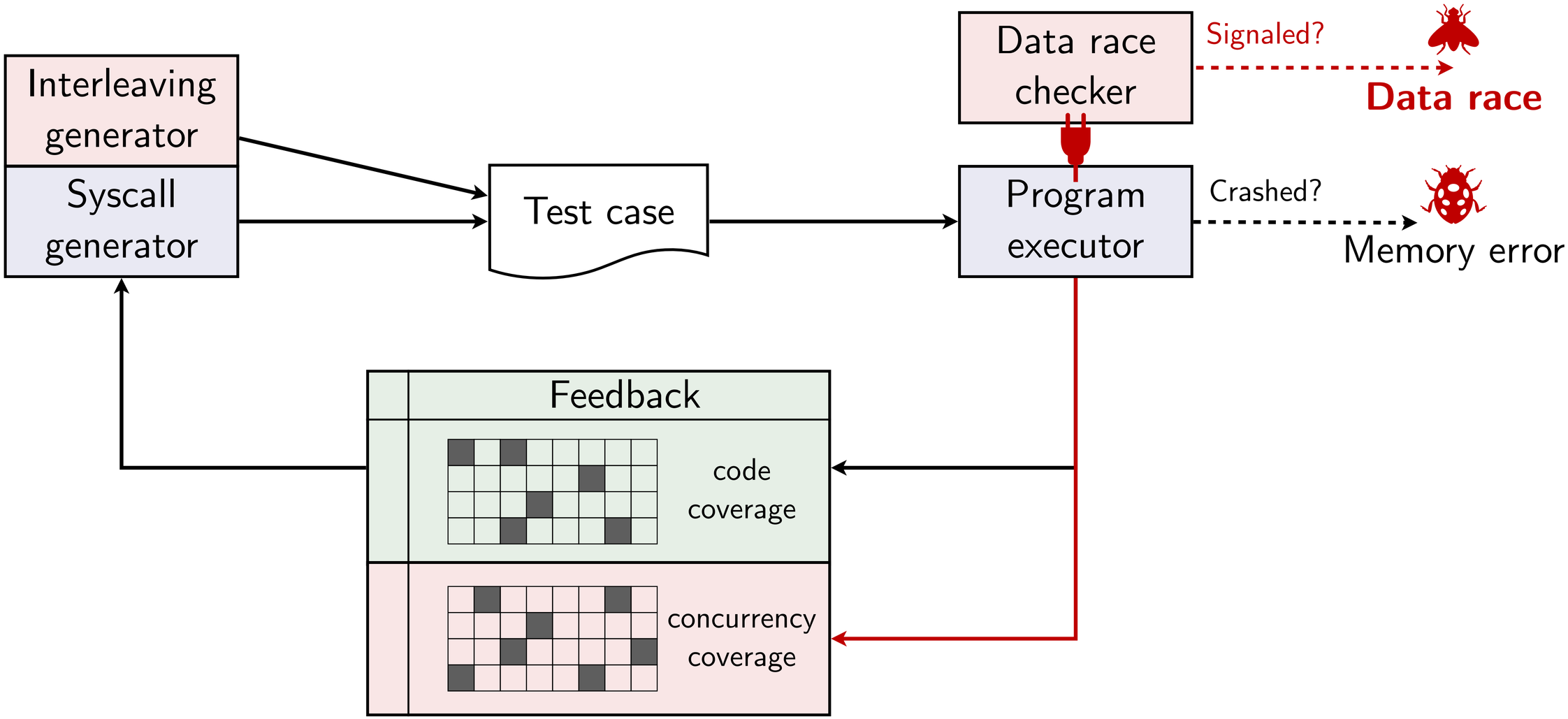




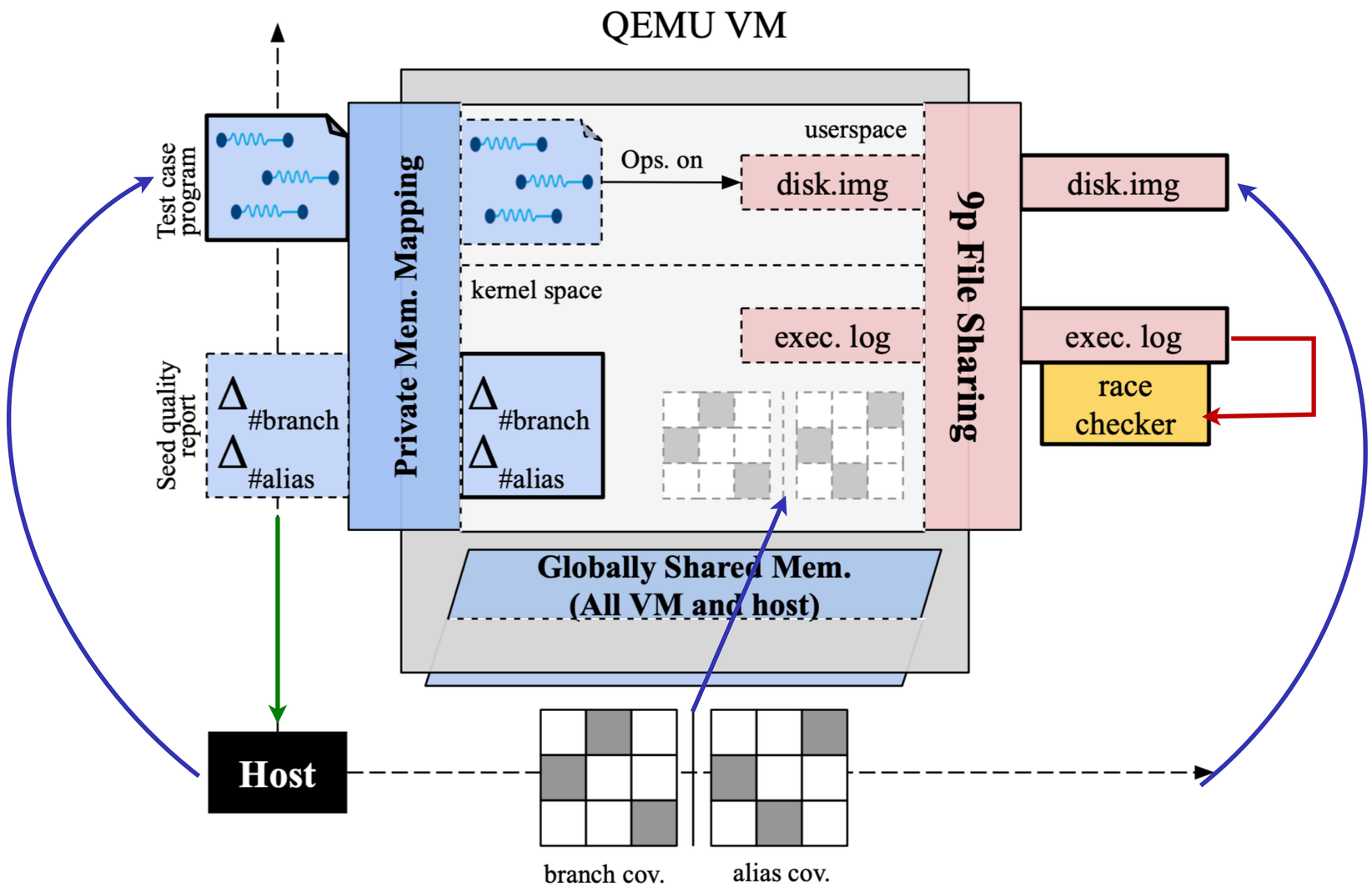
Interleaving exploration



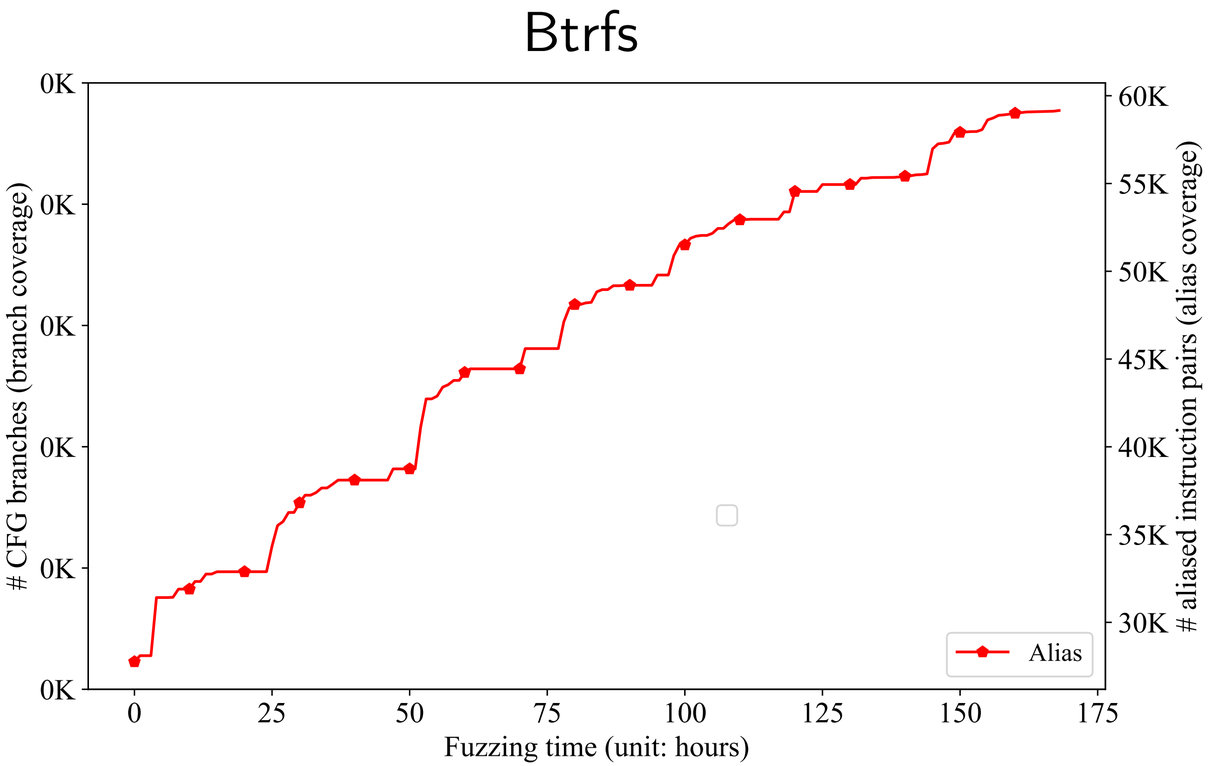
Bring them all together

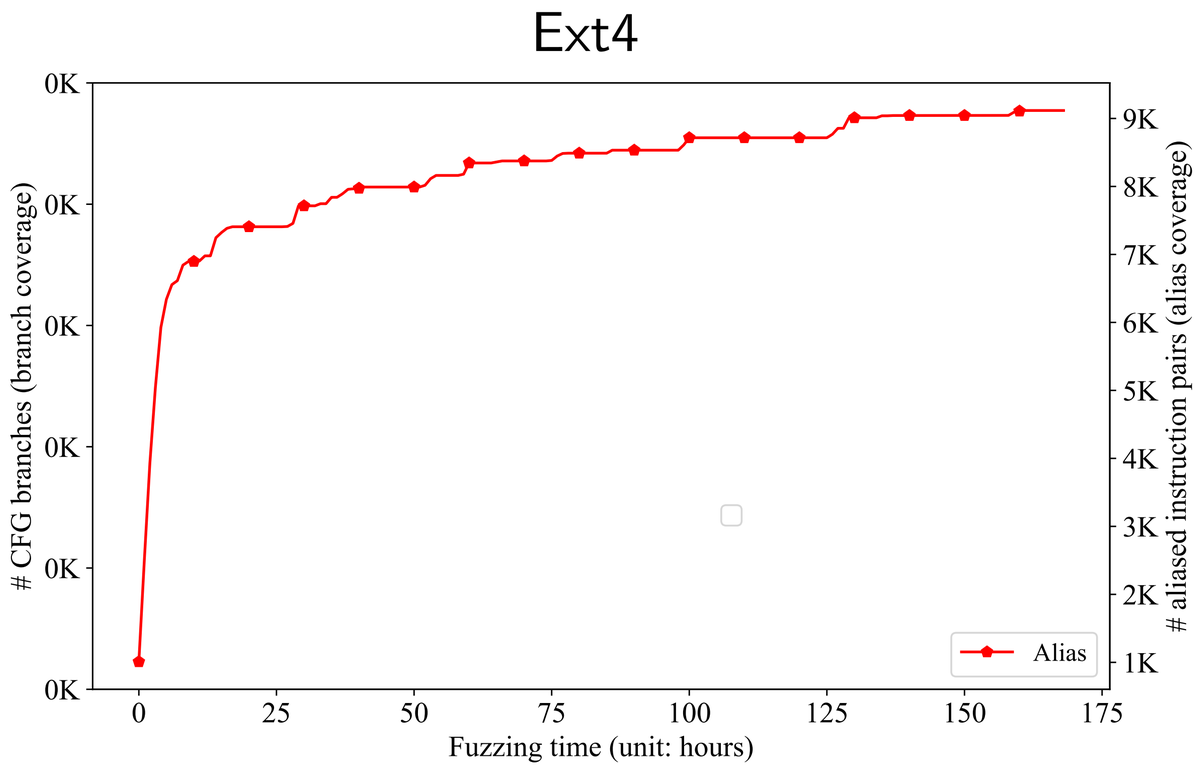


QEMU-based implementation



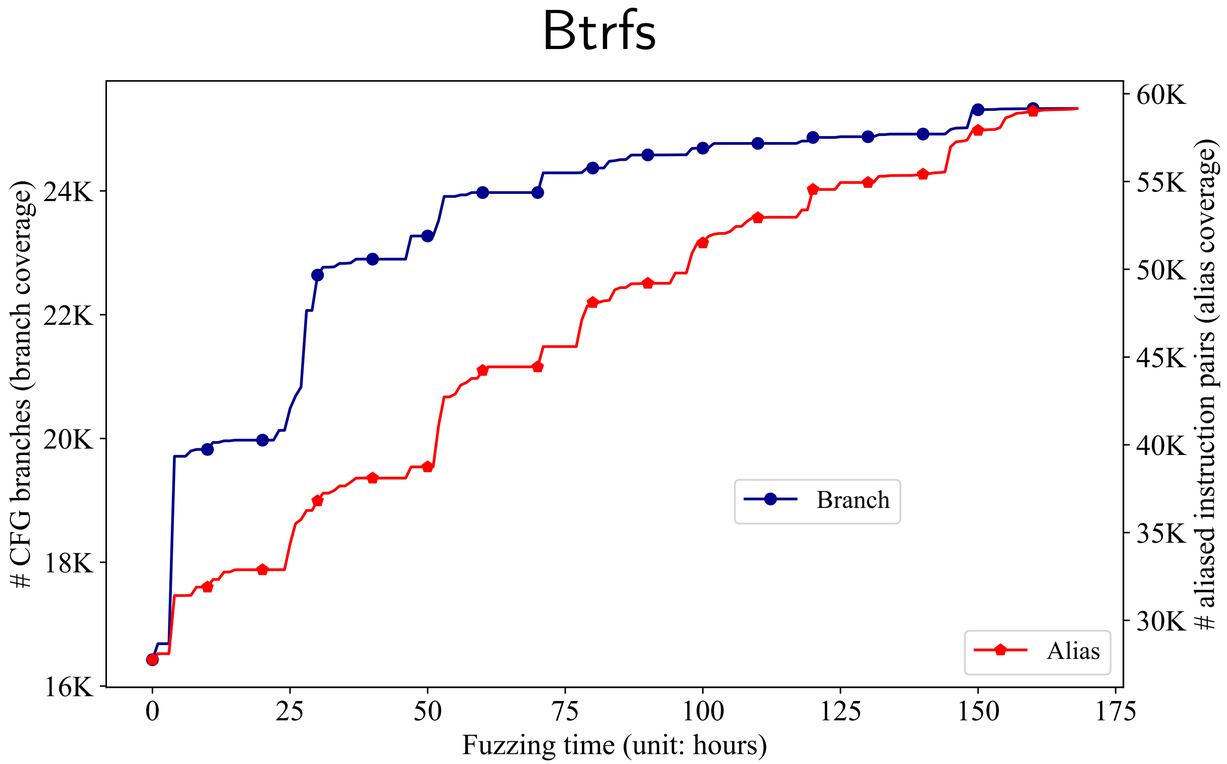
Alias coverage growth will be saturating

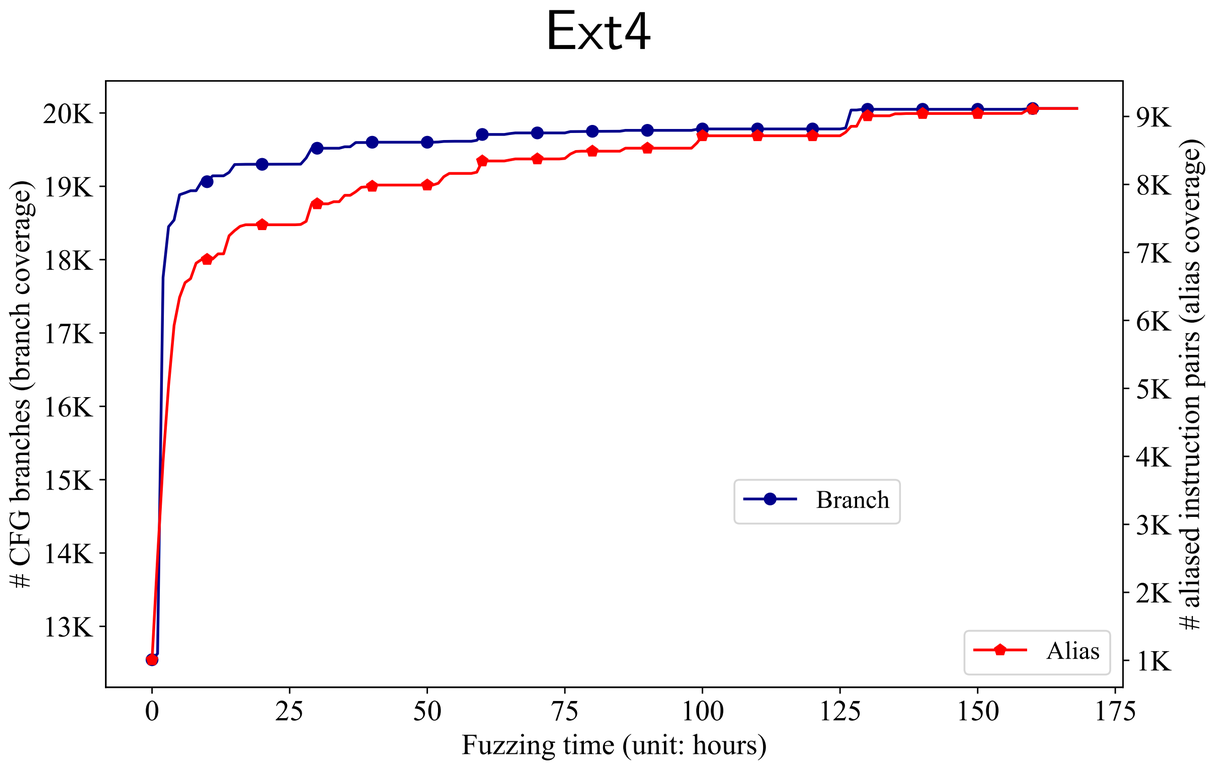




But file systems that are higher in concurrency level saturates much slower!

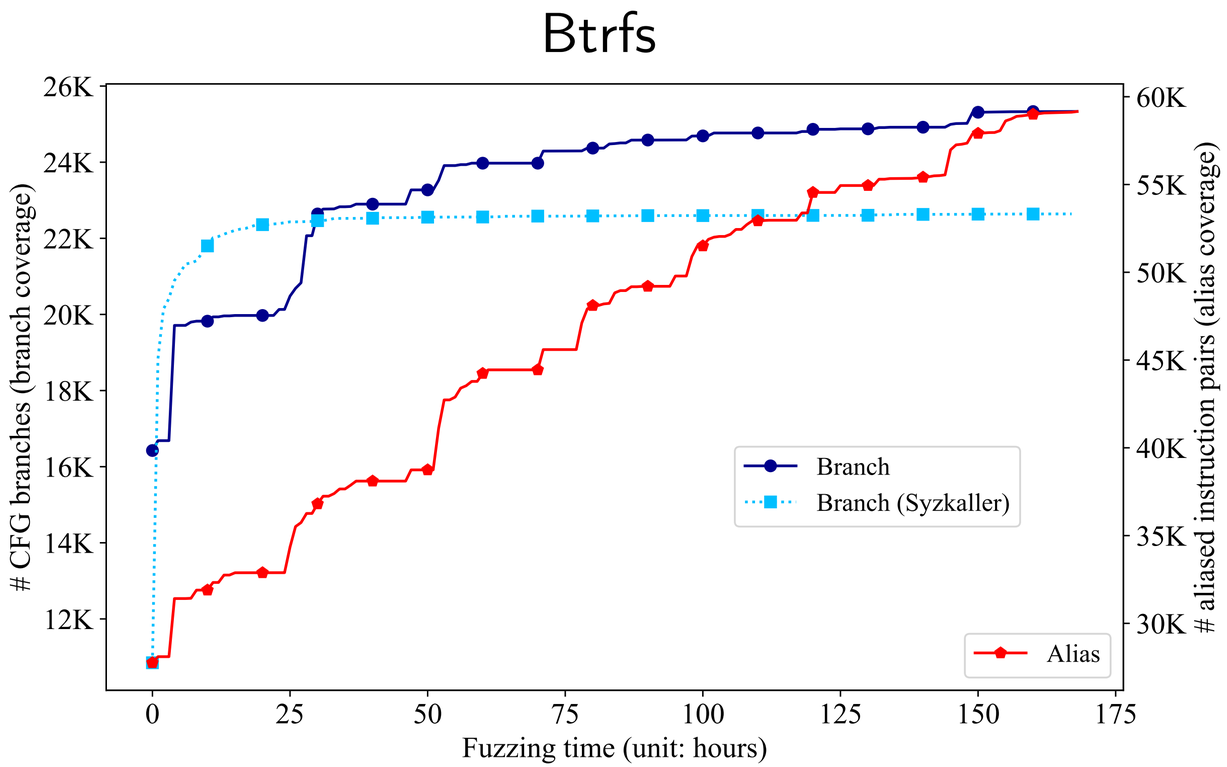
Edge and alias coverage goes generally in synchronization

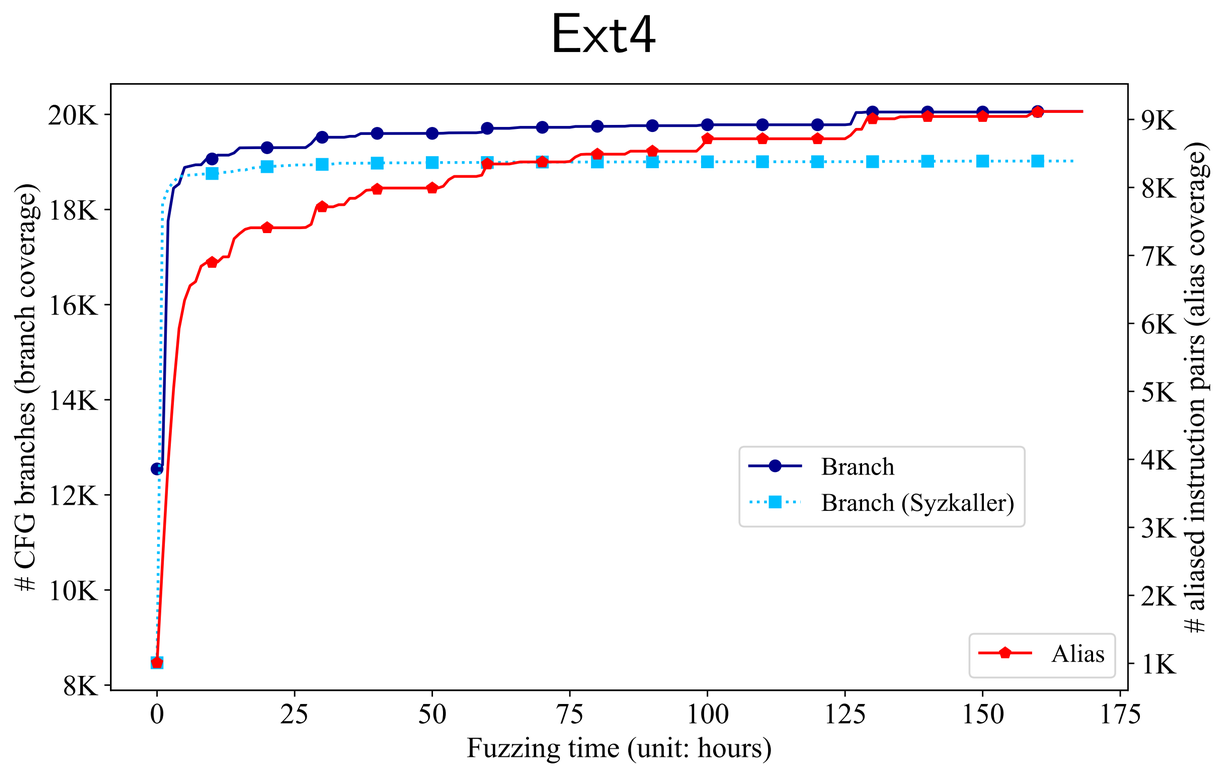




But there will be time when the edge coverage saturates
but alias coverage keeps finding new thread interleaving

Slightly more branch coverage than Syzkaller





This maybe due to the fact that we give each seed more chances (if they make progresses in alias coverage)

Bugs found by Krace

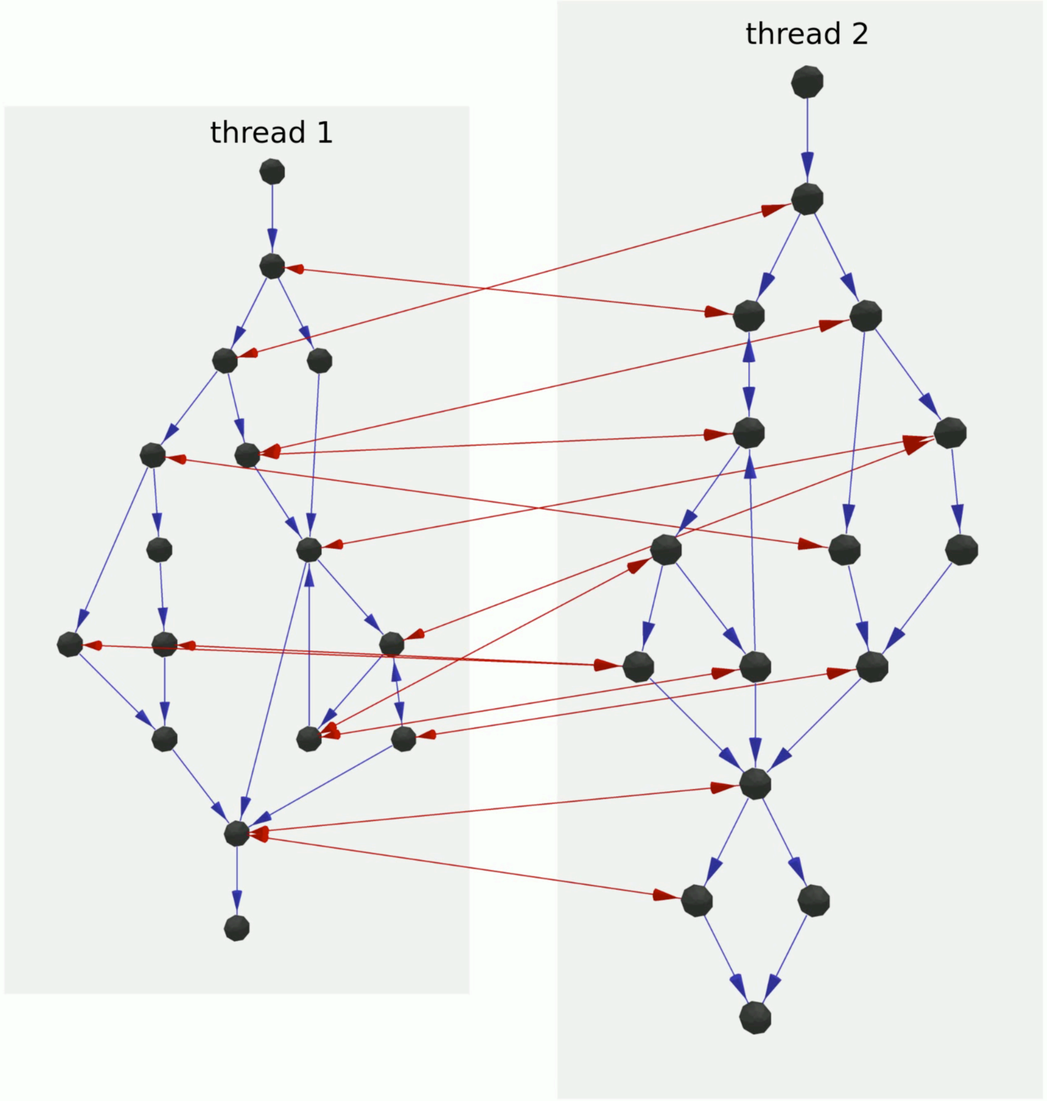
File system	# data races	# harmful confirmed
Btrfs	11	8
Ext4	4	1
VFS	8	2
Total	23	11

Comparison with related works

Structured input	**Application**
[Google] Syzkaller [SP'19] Janus [ICSE'19] SLF	[CCS'17] SlowFuzz [ICSE'19] DifFuzz [VLDB'20] Apollo .....
Seed selection	**Coverage metric**
[CCS'16] AFLFast [ASE'18] FairFuzz [FSE'19] Fudge	[SP'18] Angora [RAID'19] Benchmark **[SP'20] Krace**

Conclusion and contribution

Structured input	**Application**
[Google] Syzkaller [SP'19] Janus [ICSE'19] SLF	[CCS'17] SlowFuzz [ICSE'19] DifFuzz [VLDB'20] Apollo .....
Seed selection	**Coverage metric**
[CCS'16] AFLFast [ASE'18] FairFuzz [FSE'19] Fudge	[SP'18] Angora [RAID'19] Benchmark **[SP'20] Krace**



Common criticism about the fuzzing approach

1. How strong is the security guarantee?
   - e.g., 1 hour of fuzzing without any bugs, what does that mean?
2. Can you deterministically replay the bugs found?
   - Most likely not, because we do not directly control the scheduler.
3. How easy can these data races be triggered in reality?
   - Hard to argue, because fuzzing is essentially a probabilistic search.
4. What are the consequences out of these data races?
   - e.g., can you really alter the control flow or change some sensitive data?

Agenda

1. What are race conditions?

2. Finding their presence with **fuzzing**?
   - [SP'20] Data races in file systems

3. Towards a more **systematic** methodology?
   - [SP'18] Symbolic race checking

4. Up to the extreme of **completeness and soundness**?
   - [WIP (CAV'21)] C to SMT transpilation

Agenda

1. What are race conditions?

2. Finding their presence with **fuzzing**?
   - [SP'20] Data races in file systems

3. Towards a more **systematic** methodology?
   - [SP'18] Symbolic race checking

4. Up to the extreme of **completeness and soundness**?
   - [WIP (CAV'21)] C to SMT transpilation

SMT and symbolic execution

- SMT stands for **satisfiability modulo theories**
  - Given some basic math theories about Booleans, Integers, etc, decide whether some constraints are *satisfiable* or not ==> constraint solvers.
  - Example: what are the values of x and y that satisfy both constraints:
    - $x + y = 10$
    - $x + 2y = 20$
  - Manual solving: $x = 0, y = 10$

An SMT script is a problem description

- To describe the problem to an automated solver,
  - we need some standardized format ==> an SMT script
  - Continued from the example:

- $x + y = 10$
- $x + 2y = 20$

The problem we have in mind

```
(declare-const x Int)
(declare-const y Int)

(assert (= (+ x y) 10))
(assert (= (+ x (* 2 y)) 20))

(check-sat)
(get-model)
```

The SMT script we formulated

```
sat
(model
  (define-fun y () Int 10)
  (define-fun x () Int 0)
)
```

The answer given by Z3 SMT solver

Analogy with bug finding

Will variable "*s*" overflow in the program?

```
int loop(int x) {
  int s = 1;
  for (int i=1; i<=x; i++) {
    s *= i;
  }
  return s;
}
```

- $x + y = 10$
- $x + 2y = 20$
- ???

The problem we have in mind

```
(declare-const x Int)
(declare-const y Int)

(assert (= (??? x y) 10))
(assert (= (??? x (* 2 y)) 20))

(check-sat)
(get-model)
```

The SMT script we formulated

```
sat
(model
  (define-fun y () Int 10)
  (define-fun x () Int 0)
)
```

The answer given by Z3 SMT solver

Analogy with bug finding

Translating bug description — focus of the SP'18 paper

Symbolizing program — focus of the ongoing CAV'21 paper

Will variable "*s*" overflow in the program?

```
int loop(int x) {
  int s = 1;
  for (int i=1; i<=x; i++) {
    s *= i;
  }
  return s;
}
```

???

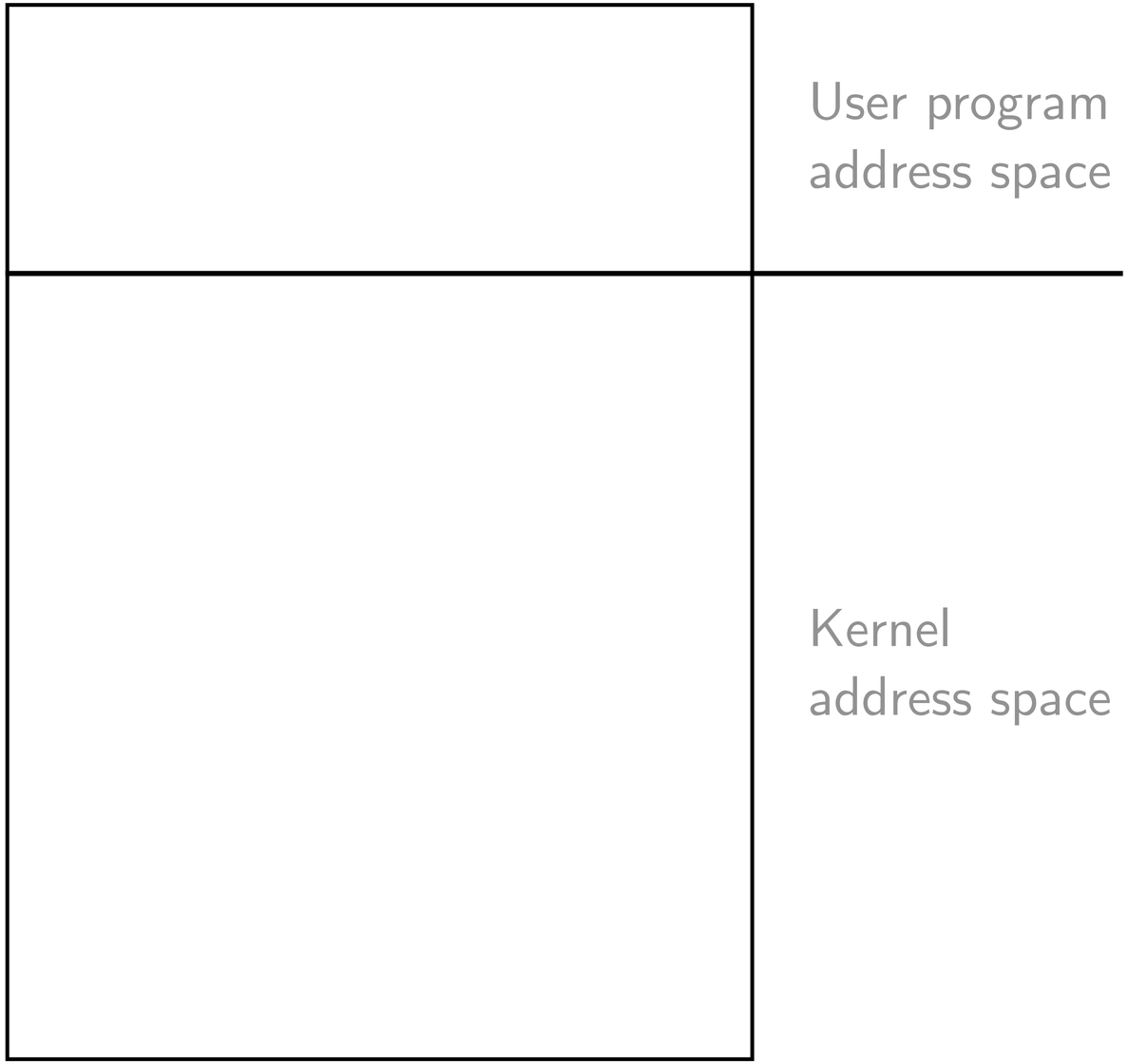
???

The problem we have in mind

The SMT script we formulated

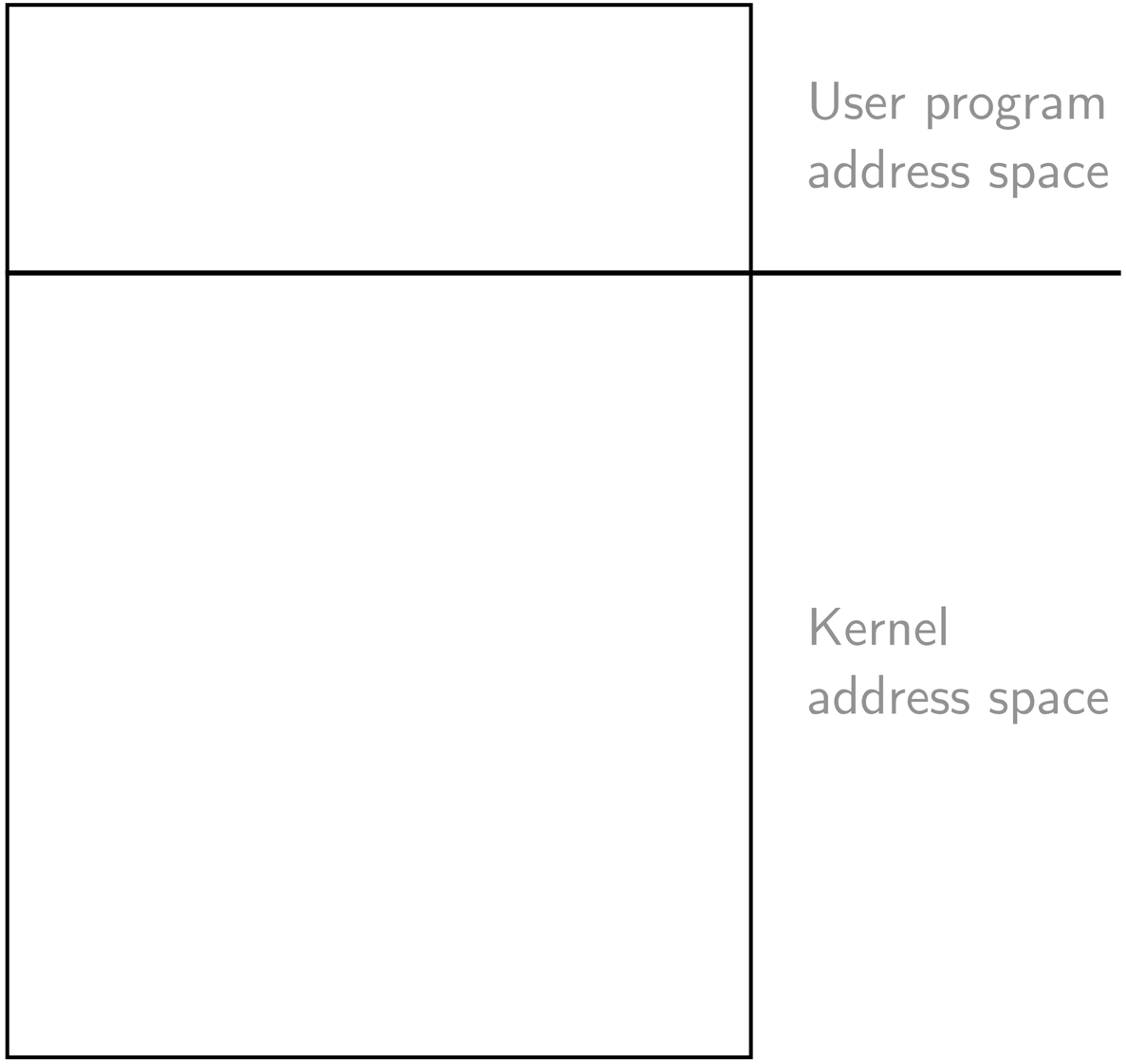
The answer given by Z3 SMT solver

What is a double-fetch bug?



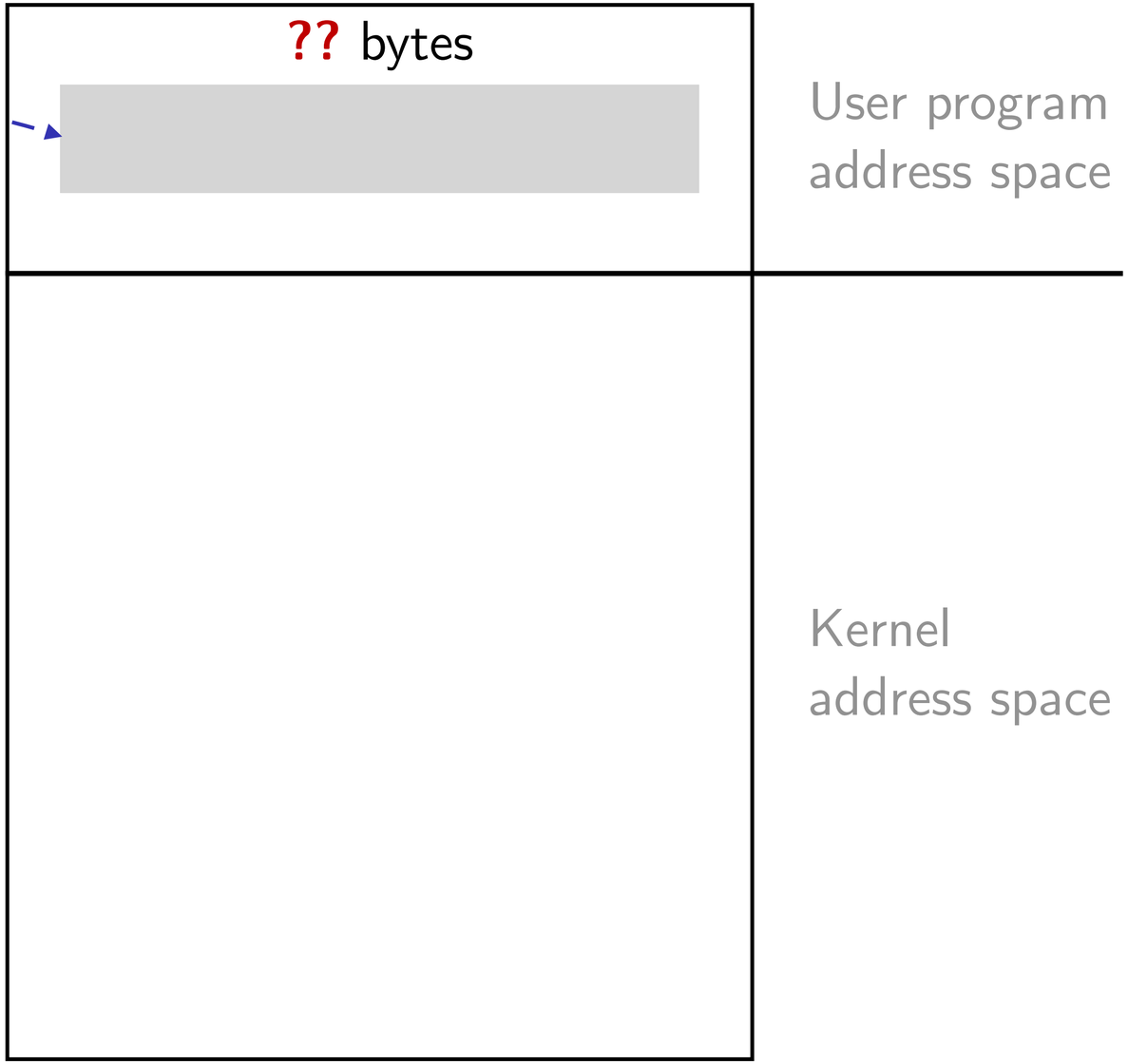
What is a double-fetch bug?

```
static int perf_copy_attr(
    struct perf_event_attr __user *uattr,
    struct perf_event_attr *kattr) {
```



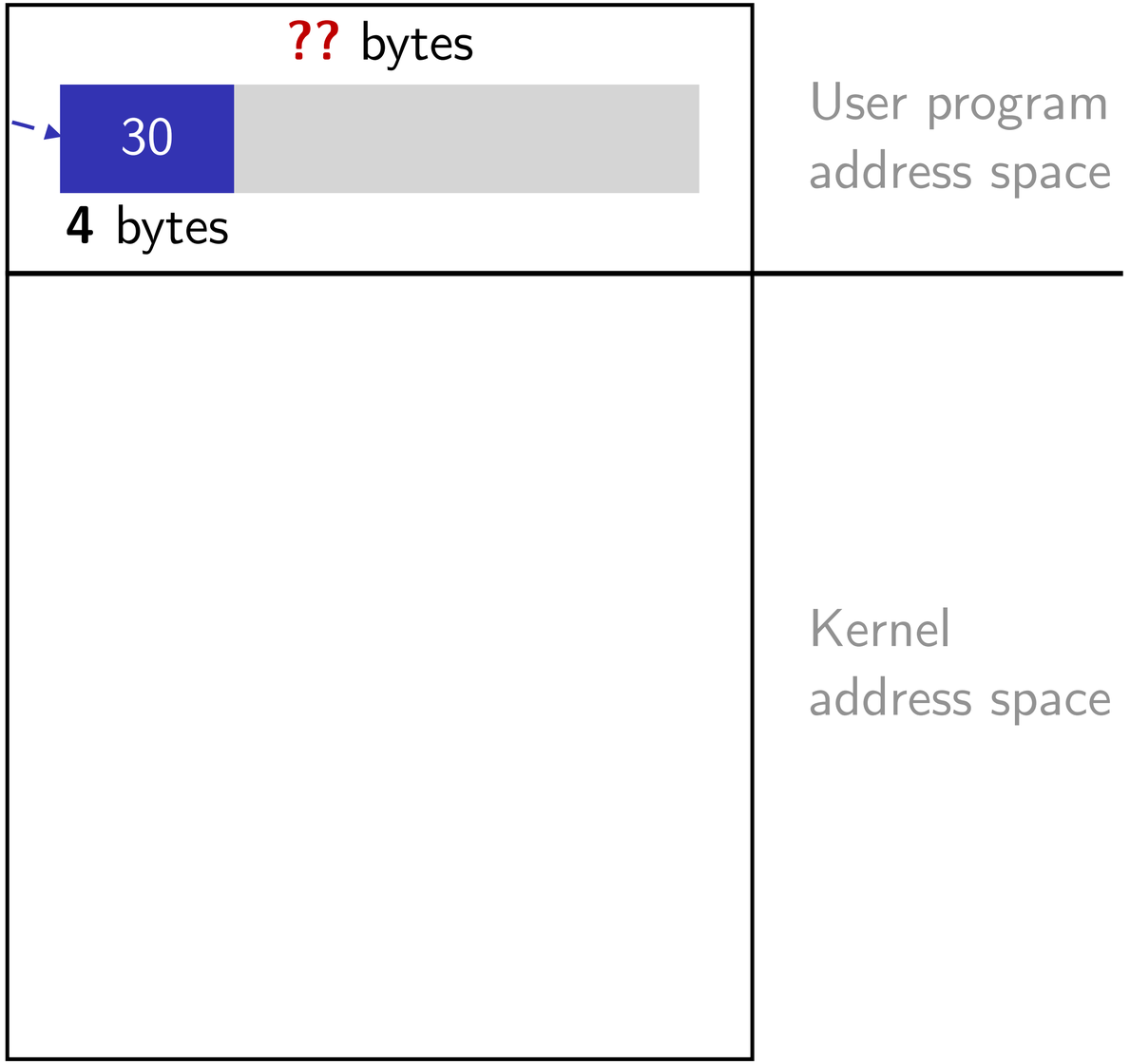
What is a double-fetch bug?

```
static int perf_copy_attr(
    struct perf_event_attr __user *uattr,
    struct perf_event_attr *kattr) {
```



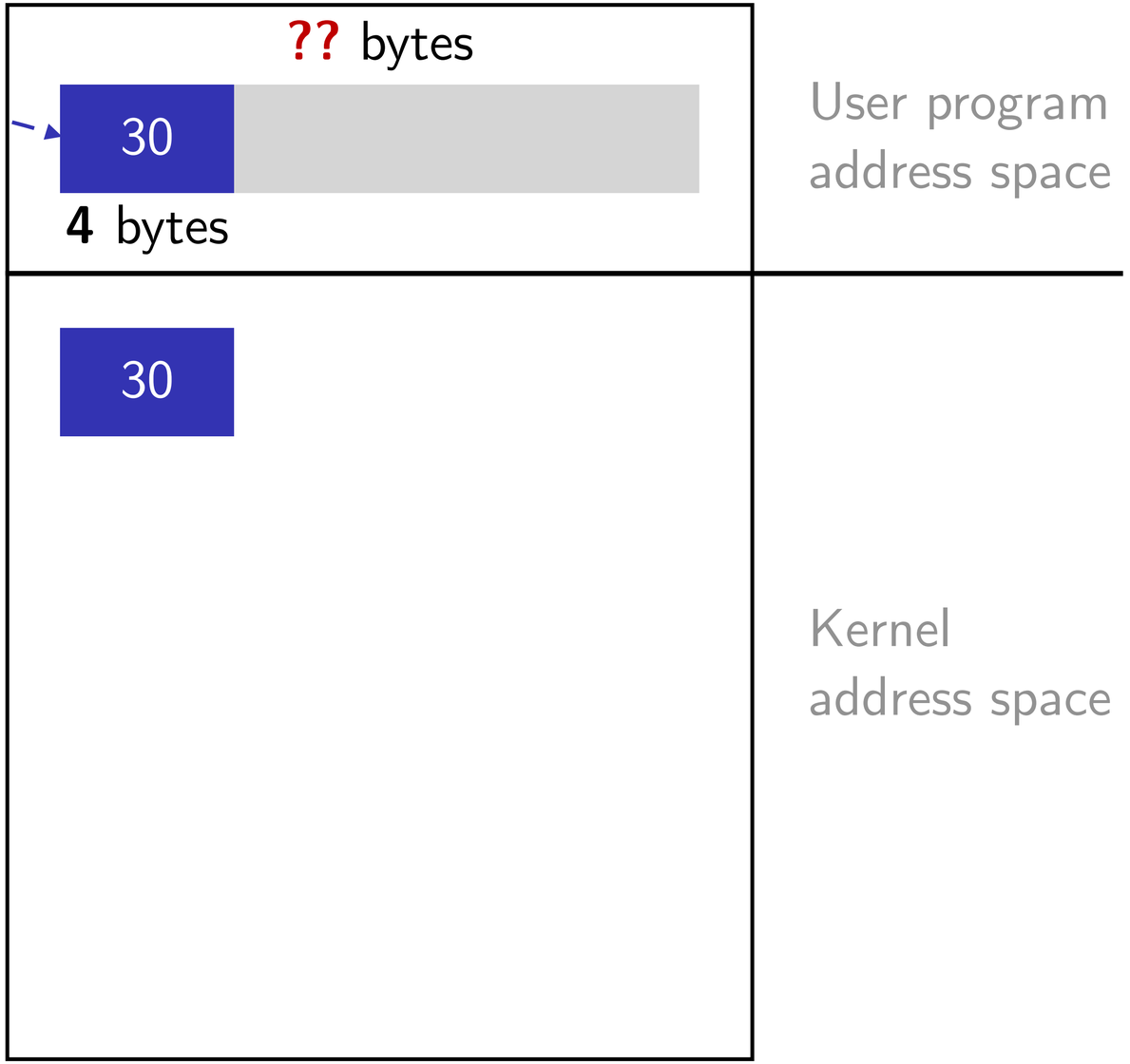
What is a double-fetch bug?

```
static int perf_copy_attr(
    struct perf_event_attr __user *uattr,
    struct perf_event_attr *kattr) {

  u32 size;
```



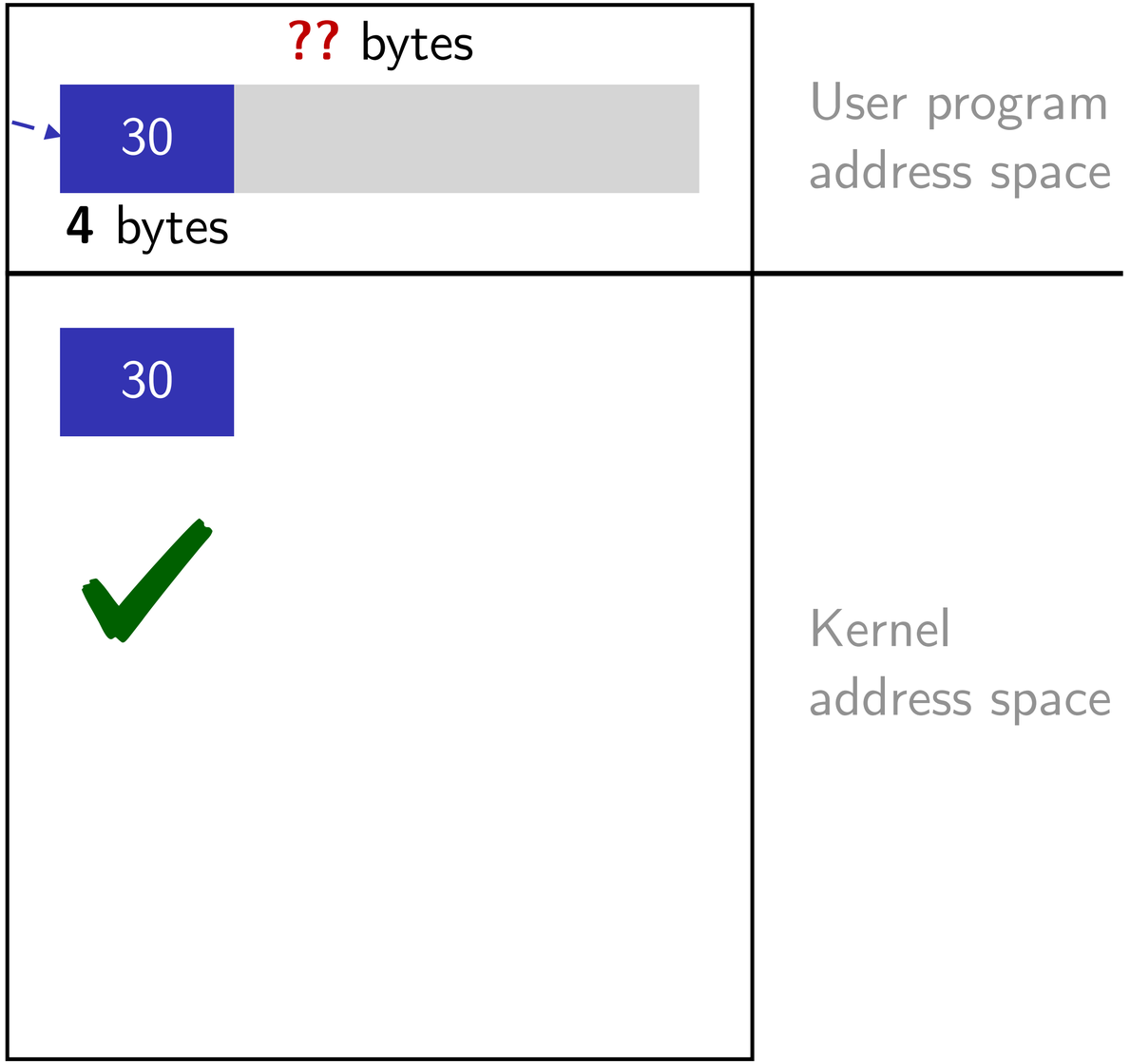
What is a double-fetch bug?

```
static int perf_copy_attr(
    struct perf_event_attr __user *uattr,
    struct perf_event_attr *kattr) {

  u32 size;

  /* first fetch */
  if (copy_from_user(&size, &uattr->size, 4))
    return -EFAULT;
```



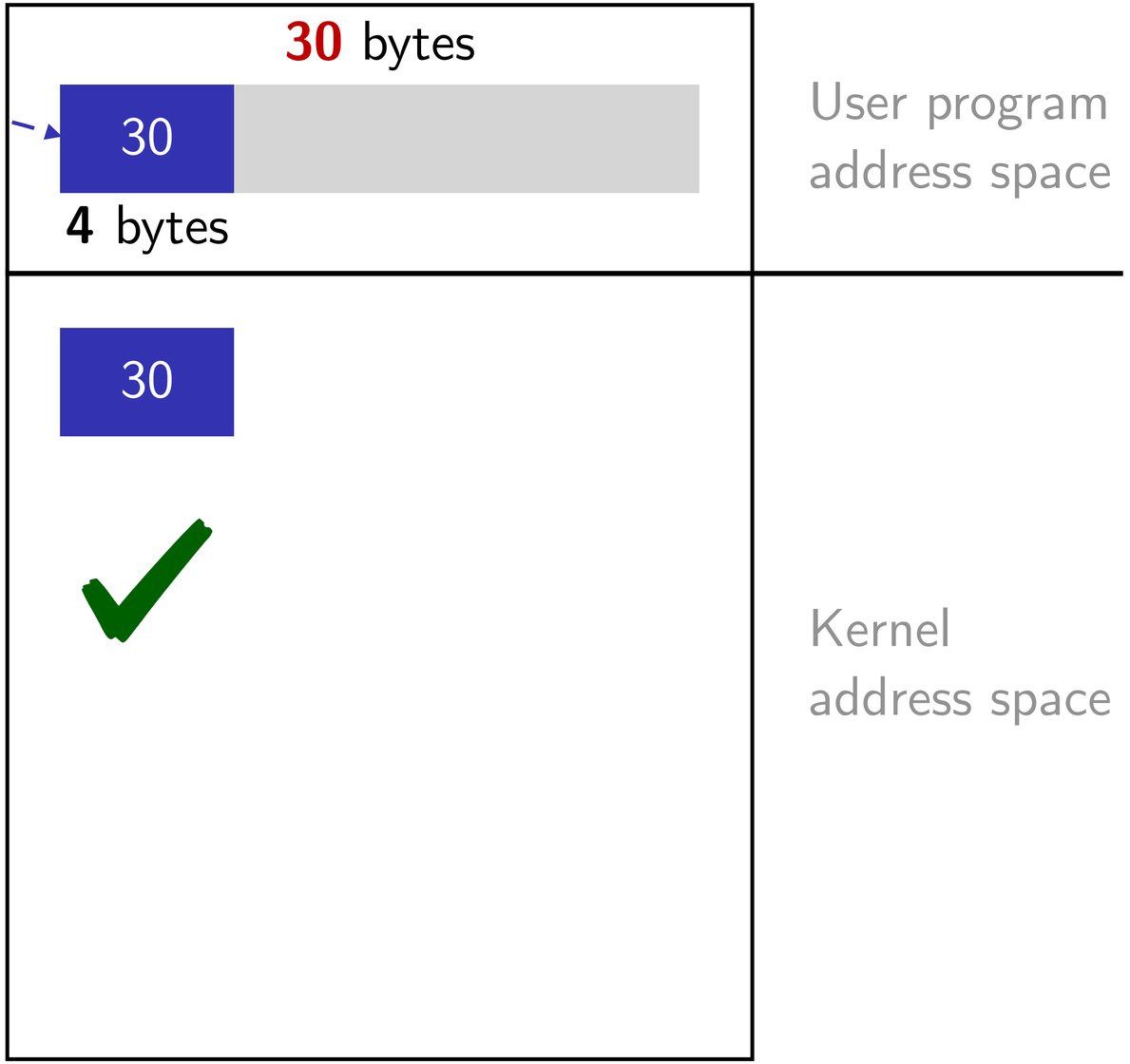
What is a double-fetch bug?

```
static int perf_copy_attr(
    struct perf_event_attr __user *uattr,
    struct perf_event_attr *kattr) {

  u32 size;

  /* first fetch */
  if (copy_from_user(&size, &uattr->size, 4))
    return -EFAULT;

  /* sanity checks */
  if (size > PAGE_SIZE || size < PERF_ATTR_SIZE_VER0)
    return -EFAULT;
```



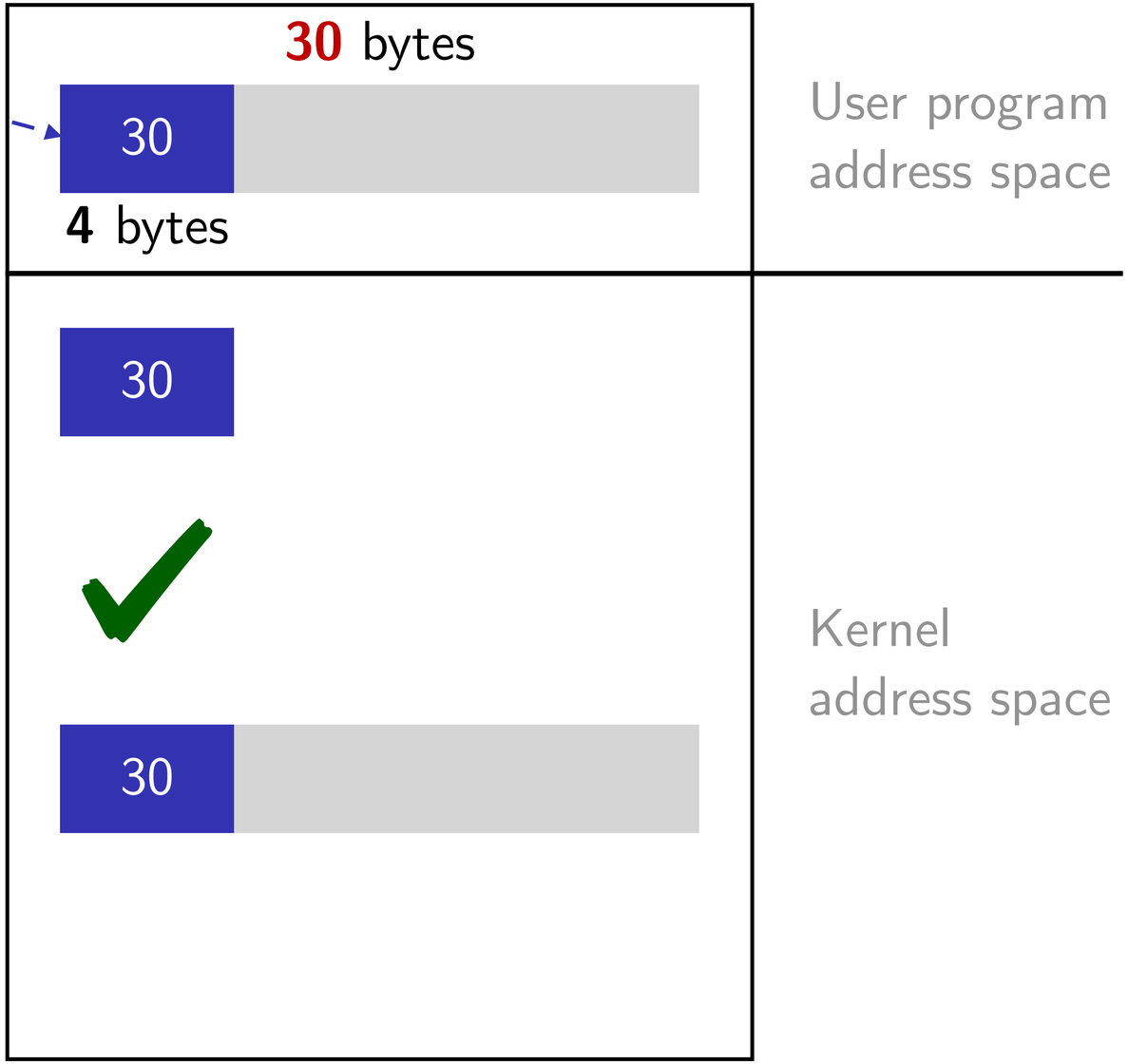
What is a double-fetch bug?

```
static int perf_copy_attr(
    struct perf_event_attr __user *uattr,
    struct perf_event_attr *kattr) {

  u32 size;

  /* first fetch */
  if (copy_from_user(&size, &uattr->size, 4))
    return -EFAULT;

  /* sanity checks */
  if (size > PAGE_SIZE || size < PERF_ATTR_SIZE_VER0)
    return -EFAULT;
```



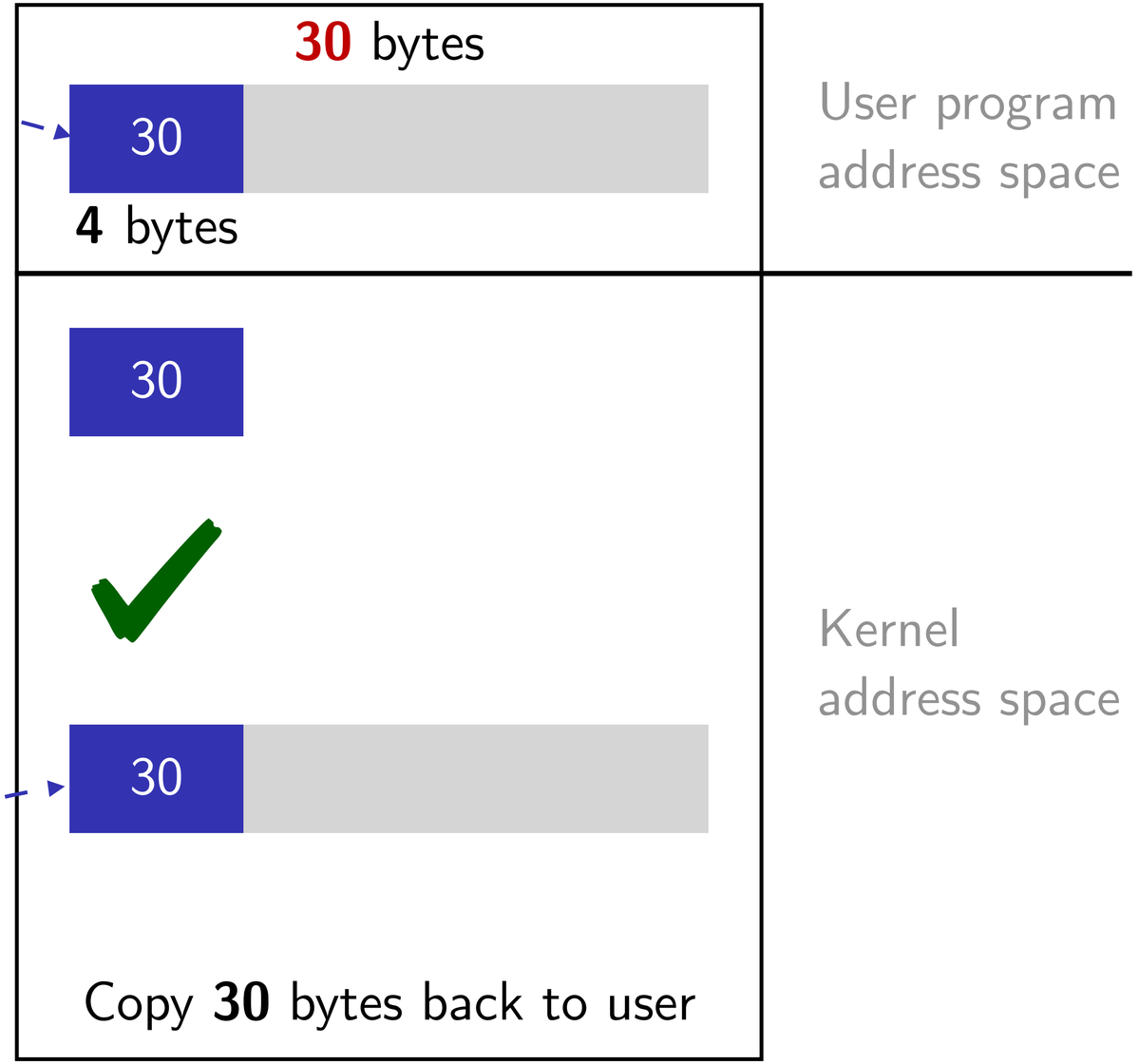
What is a double-fetch bug?

```
static int perf_copy_attr(
    struct perf_event_attr __user *uattr,
    struct perf_event_attr *kattr) {

  u32 size;

  /* first fetch */
  if (copy_from_user(&size, &uattr->size, 4))
    return -EFAULT;

  /* sanity checks */
  if (size > PAGE_SIZE || size < PERF_ATTR_SIZE_VER0)
    return -EFAULT;

  /* second fetch */
  if (copy_from_user(kattr, uattr, size))
    return -EFAULT;
  ......
}
```



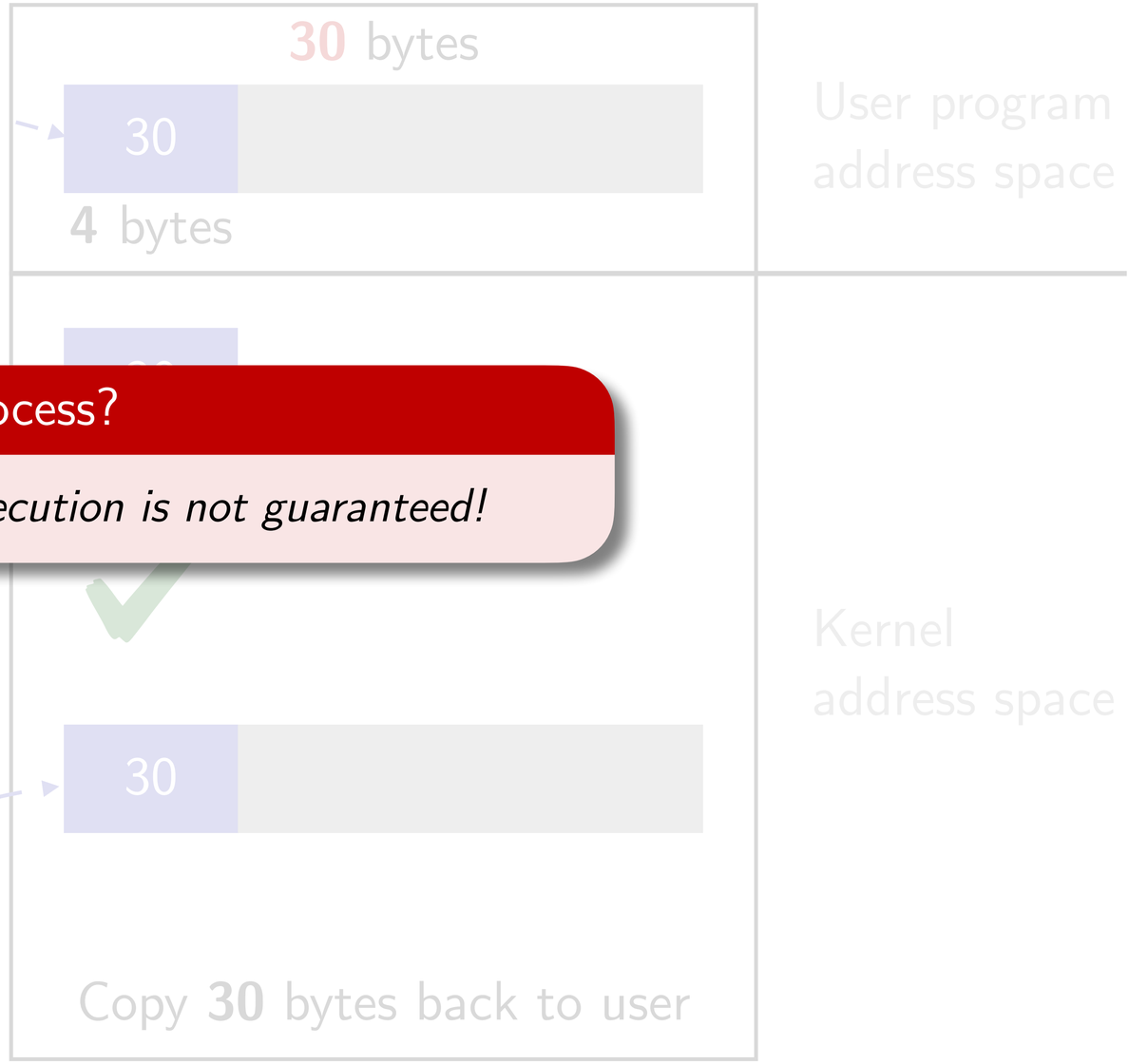
What is a double-fetch bug?

```
static int perf_copy_attr(
    struct perf_event_attr __user *uattr,
    struct perf_event_attr *kattr) {

  u32 size;

  /* first fetch */
  if (copy_from_user(&size, &uattr->size, 4))
    return -EFAULT;

  /* sanity checks */
  if (size > PAGE_SIZE || size < PERF_ATTR_SIZE_VER0)
    return -EFAULT;

  /* second fetch */
  if (copy_from_user(kattr, uattr, size))
    return -EFAULT;
  ......
}

copy_to_user(uattr, attr, attr->size))
```



The assumption failure

```
static int perf_copy_attr(
    struct perf_event_attr __user *uattr,
    struct perf_event_attr *kattr) {

  u32 size;

  /* first fetch */
  if (copy_from_user(&si
    return -EFAULT;

  /* sanity checks */
  if (size > PAGE_SIZE || size < PERF_ATTR_SIZE_VER0)
    return -EFAULT;

  /* second fetch */
  if (copy_from_user(kattr, attr, size))
    return -EFAULT;
  ......
}

copy_to_user(uattr, attr, attr->size))
```

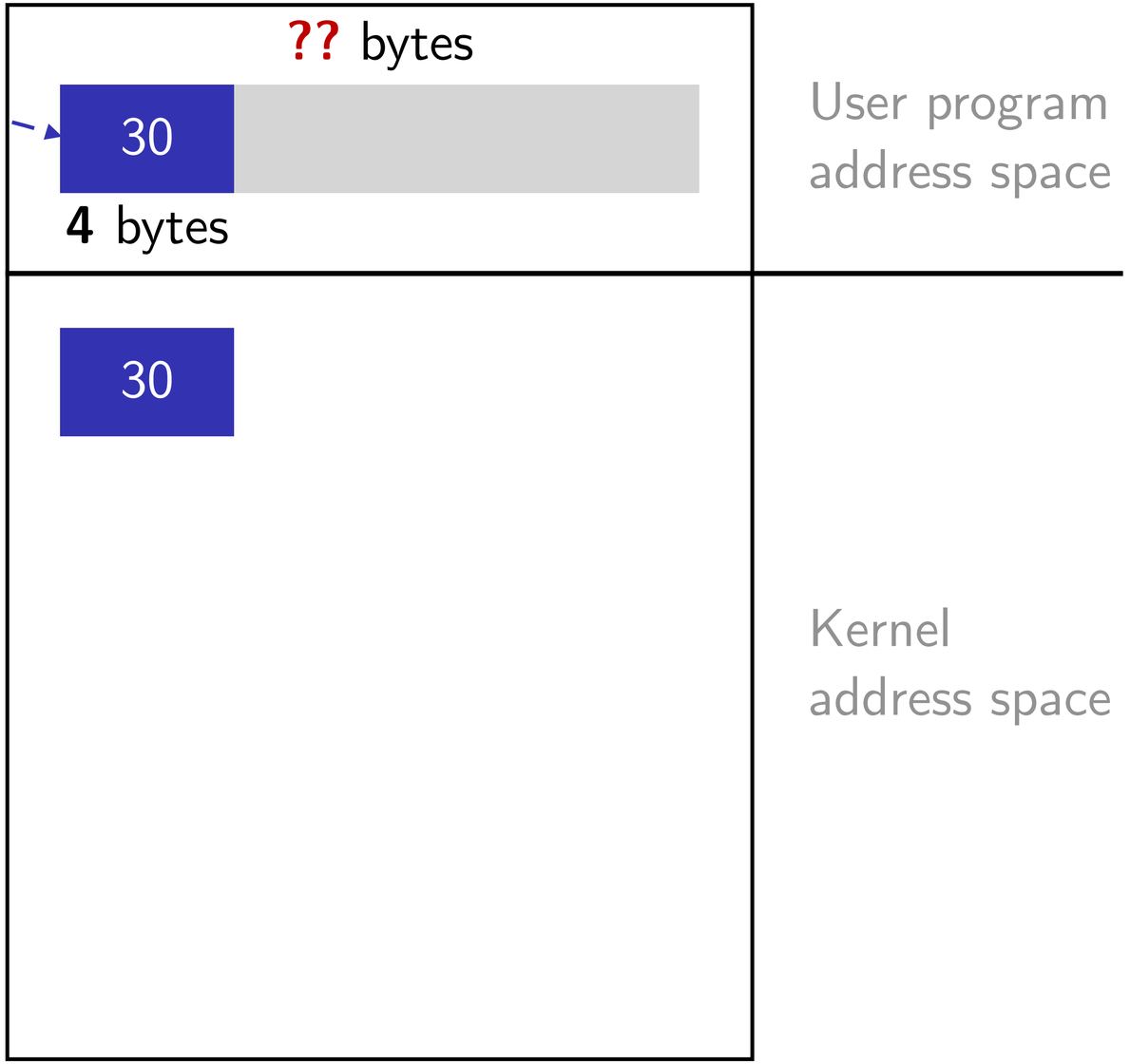


What can go wrong in this process?

*Data **atomicity** during syscall execution is not guaranteed!*

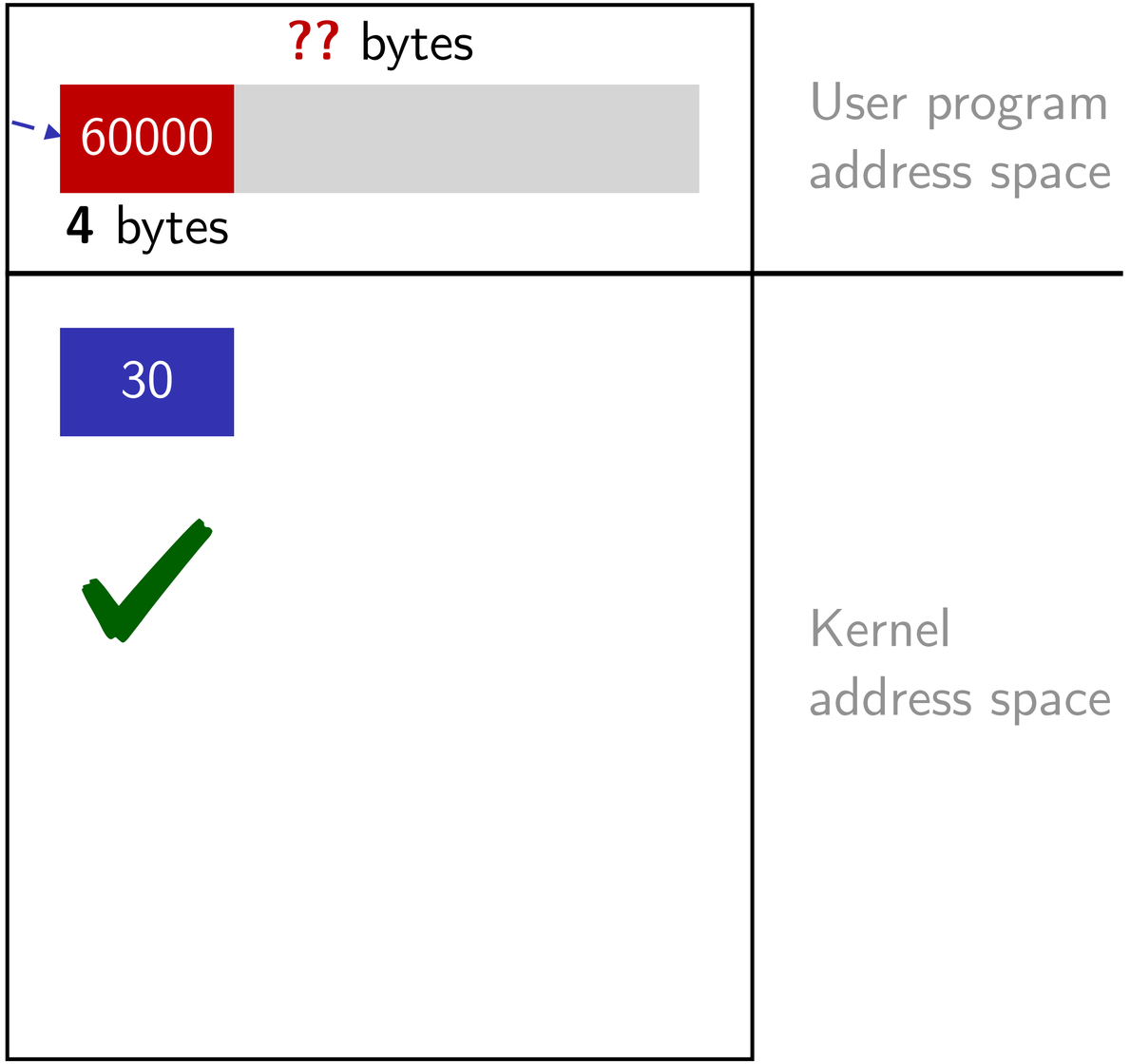
Up until the first fetch...

```
static int perf_copy_attr(
    struct perf_event_attr __user *uattr,
    struct perf_event_attr *kattr) {

  u32 size;

  /* first fetch */
  if (copy_from_user(&size, &uattr->size, 4))
    return -EFAULT;
```



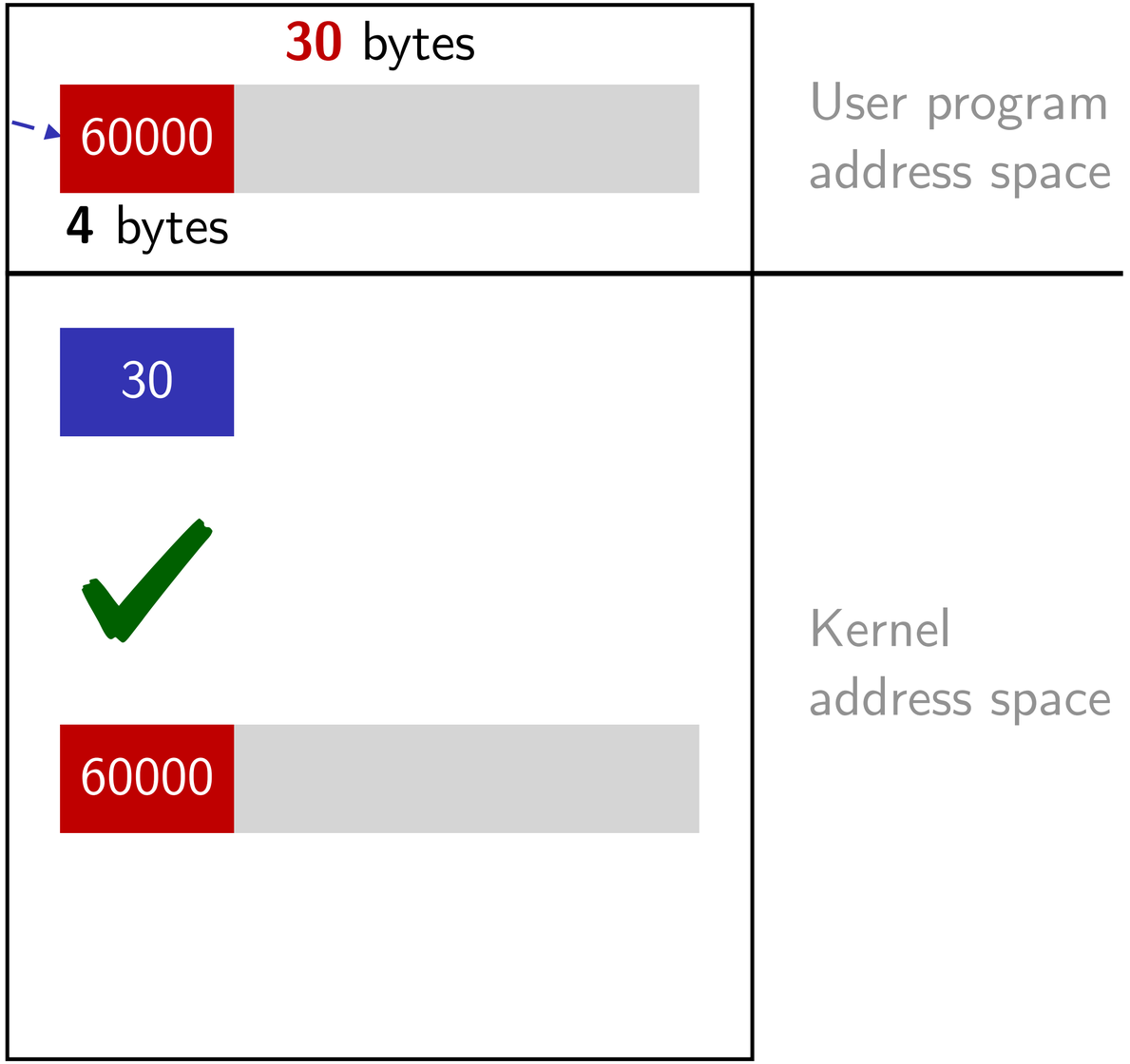
Wrong assumption: atomicity in syscall

```
static int perf_copy_attr(
    struct perf_event_attr __user *uattr,
    struct perf_event_attr *kattr) {

  u32 size;

  /* first fetch */
  if (copy_from_user(&size, &uattr->size, 4))
    return -EFAULT;

  /* sanity checks */
  if (size > PAGE_SIZE || size < PERF_ATTR_SIZE_VER0)
    return -EFAULT;
```



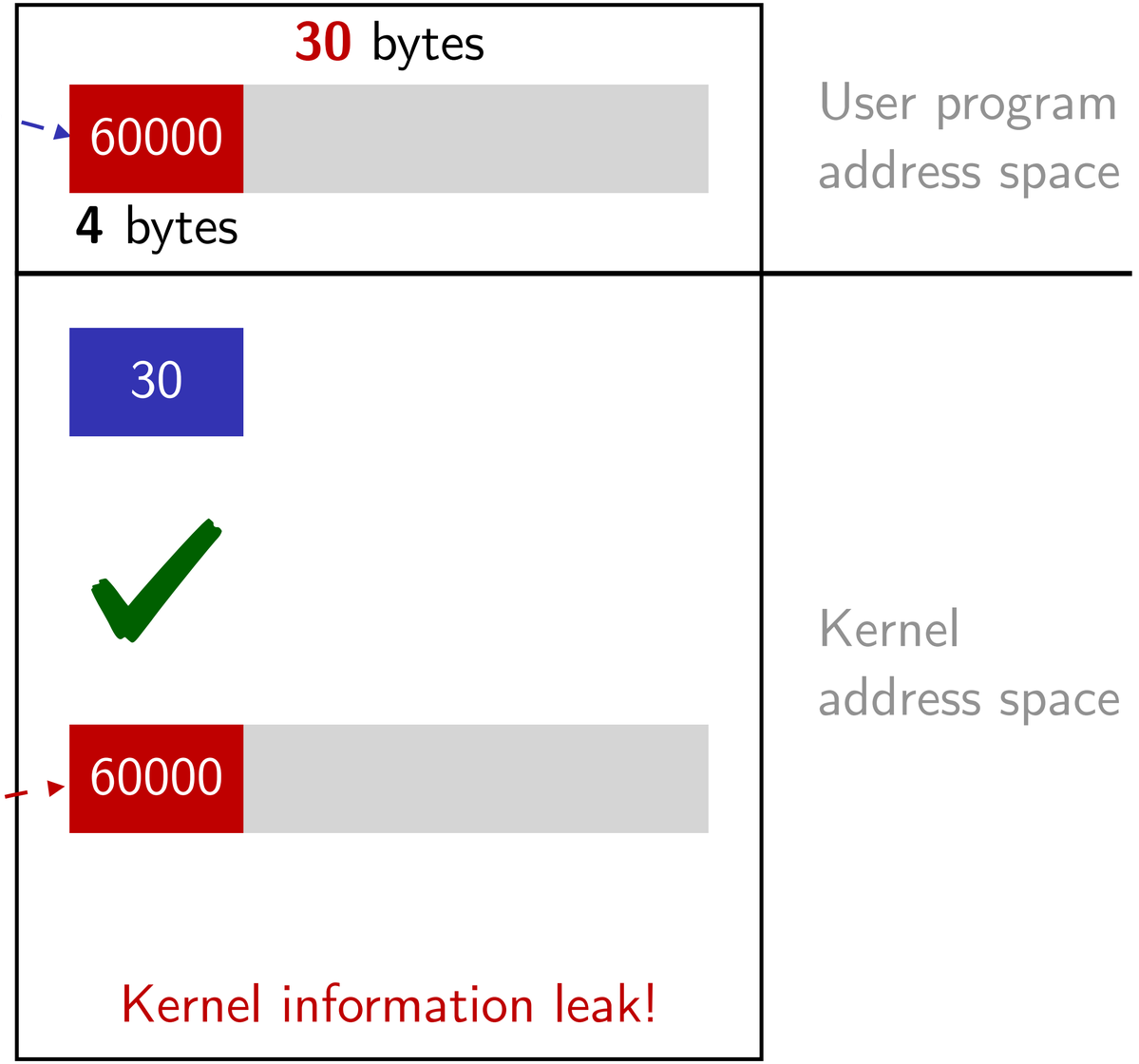
Wrong assumption: atomicity in syscall

```
static int perf_copy_attr(
    struct perf_event_attr __user *uattr,
    struct perf_event_attr *kattr) {

  u32 size;

  /* first fetch */
  if (copy_from_user(&size, &uattr->size, 4))
    return -EFAULT;

  /* sanity checks */
  if (size > PAGE_SIZE || size < PERF_ATTR_SIZE_VER0)
    return -EFAULT;

  /* second fetch */
  if (copy_from_user(kattr, uattr, size))
    return -EFAULT;
  ......
}
```



When the exploit happens

```
static int perf_copy_attr(
    struct perf_event_attr __user *uattr,
    struct perf_event_attr *kattr) {

  u32 size;

  /* first fetch */
  if (copy_from_user(&size, &uattr->size, 4))
    return -EFAULT;

  /* sanity checks */
  if (size > PAGE_SIZE || size < PERF_ATTR_SIZE_VER0)
    return -EFAULT;

  /* second fetch */
  if (copy_from_user(kattr, uattr, size))
    return -EFAULT;
  ......
}
/* BUG: when attr->size is used later */
copy_to_user(uattr, kattr, kattr->size))
```



Modeling a double-fetch bug

- **Fetch**: a pair (A, S), where
  - A - the starting address of the fetch
  - S - the size of memory copied into kernel

Modeling a double-fetch bug

- **Fetch**: a pair (A, S), where

  A - the starting address of the fetch

  S - the size of memory copied into kernel

- **Overlap**: two fetches, (A_1, S_1) and (A_2, S_2), that satisfy

  $$A_1 \leq A_2 < (A_1 + S_1) \quad || \quad A_2 \leq A_1 < (A_2 + S_2)$$

Modeling a double-fetch bug

- **Fetch**: a pair (A, S), where

  A - the starting address of the fetch

  S - the size of memory copied into kernel

- **Overlap**: two fetches, (A_1, S_1) and (A_2, S_2), that satisfy

  $A_1 \leq A_2 < (A_1 + S_1) \quad || \quad A_2 \leq A_1 < (A_2 + S_2)$

- **Dependency**: $\exists V \in \texttt{Overlap}$ such that V controls whether and how the second fetch might take place.

Modeling a double-fetch bug

- **Fetch**: a pair (A, S), where

  A - the starting address of the fetch

  S - the size of memory copied into kernel

- **Overlap**: two fetches, (A_1, S_1) and (A_2, S_2), that satisfy

  $A_1 \leq A_2 < (A_1 + S_1) \quad || \quad A_2 \leq A_1 < (A_2 + S_2)$

- **Dependency**: $\exists V \in$ `Overlap` such that V controls whether and how the second fetch might take place.

- **Precaution**: check that V' from the second fetch equals V from the first fetch.

Model illustration

```
static int perf_copy_attr(
    struct perf_event_attr __user *uattr,
    struct perf_event_attr *kattr) {

  u32 size;

  /* first fetch */
  if (copy_from_user(&size, &uattr->size, 4))
    return -EFAULT;

  /* sanity checks */
  if (size > PAGE_SIZE || size < PERF_ATTR_SIZE_VER0)
    return -EFAULT;

  /* second fetch */
  if (copy_from_user(kattr, uattr, size))
    return -EFAULT;
  ......
}
/* BUG: when attr->size is used later */
copy_to_user(uattr, kattr, kattr->size))
```

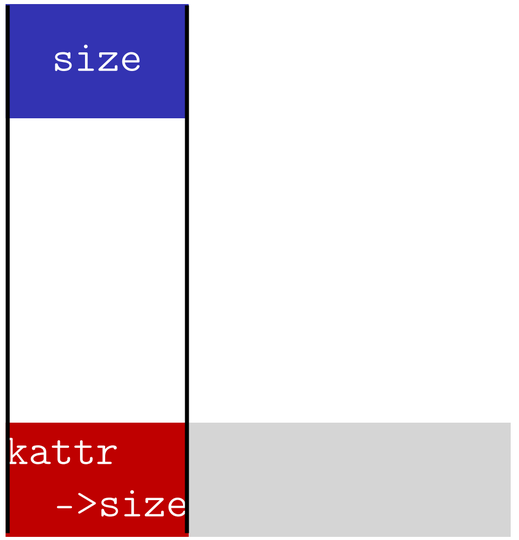
Model illustration

```
static int perf_copy_attr(
    struct perf_event_attr __user *uattr,
    struct perf_event_attr *kattr) {

  u32 size;

  /* first fetch */
  if (copy_from_user(&size, &uattr->size, 4))
    return -EFAULT;

  /* sanity checks */
  if (size > PAGE_SIZE || size < PERF_ATTR_SIZE_VER0)
    return -EFAULT;

  /* second fetch */
  if (copy_from_user(kattr, uattr, size))
    return -EFAULT;
  ......
}
/* BUG: when attr->size is used later */
copy_to_user(uattr, kattr, kattr->size))
```

size

Model illustration

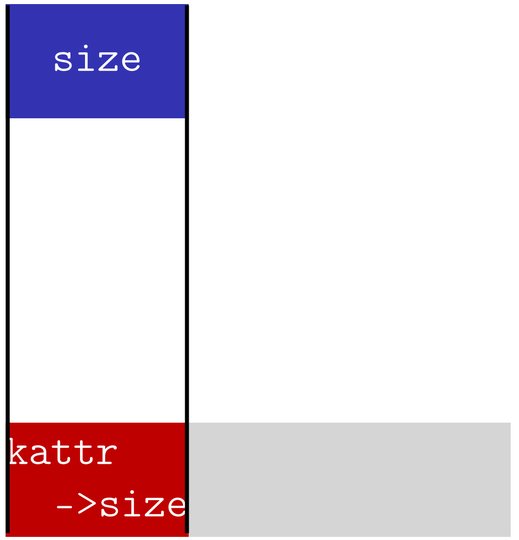
```
static int perf_copy_attr(
    struct perf_event_attr __user *uattr,
    struct perf_event_attr *kattr) {

  u32 size;

  /* first fetch */
  if (copy_from_user(&size, &uattr->size, 4))
    return -EFAULT;

  /* sanity checks */
  if (size > PAGE_SIZE || size < PERF_ATTR_SIZE_VER0)
    return -EFAULT;

  /* second fetch */
  if (copy_from_user(kattr, uattr, size))
    return -EFAULT;
  ......
}
/* BUG: when attr->size is used later */
copy_to_user(uattr, kattr, kattr->size))
```



Model illustration

```
static int perf_copy_attr(
    struct perf_event_attr __user *uattr,
    struct perf_event_attr *kattr) {

  u32 size;

  /* first fetch */
  if (copy_from_user(&size, &uattr->size, 4))
    return -EFAULT;

  /* sanity checks */
  if (size > PAGE_SIZE || size < PERF_ATTR_SIZE_VER0)
    return -EFAULT;

  /* second fetch */
  if (copy_from_user(kattr, uattr, size))
    return -EFAULT;
  ......
}
/* BUG: when attr->size is used later */
copy_to_user(uattr, kattr, kattr->size))
```

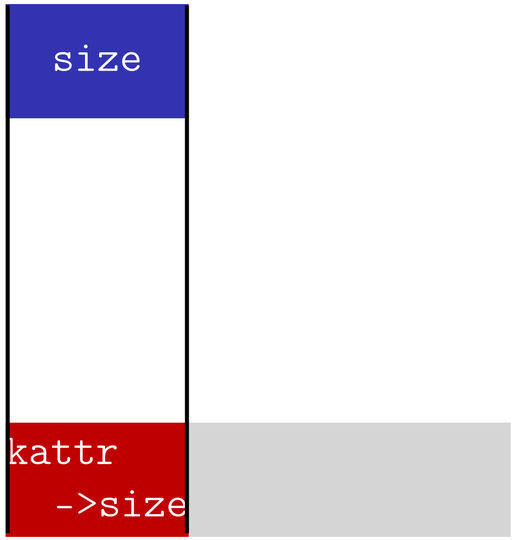
Control dependency on variable size



Model illustration

```
static int perf_copy_attr(
    struct perf_event_attr __user *uattr,
    struct perf_event_attr *kattr) {

  u32 size;

  /* first fetch */
  if (copy_from_user(&size, &uattr->size, 4))
    return -EFAULT;

  /* sanity checks */
  if (size > PAGE_SIZE || size < PERF_ATTR_SIZE_VER0)
    return -EFAULT;

  /* second fetch */
  if (copy_from_user(kattr, uattr, size))
    return -EFAULT;
  ......
}
/* BUG: when attr->size is used later */
copy_to_user(uattr, kattr, kattr->size))
```

Control dependency on variable size



Data dependency on variable size

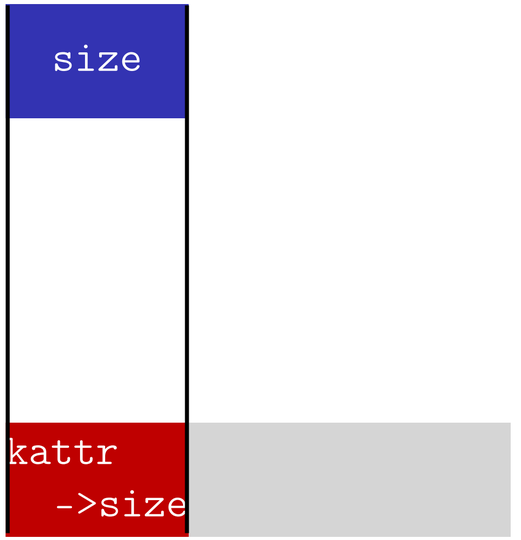
Model illustration

```
static int perf_copy_attr(
    struct perf_event_attr __user *uattr,
    struct perf_event_attr *kattr) {

  u32 size;

  /* first fetch */
  if (copy_from_user(&size, &uattr->size, 4))
    return -EFAULT;

  /* sanity checks */
  if (size > PAGE_SIZE || size < PERF_ATTR_SIZE_VER0)
    return -EFAULT;

  /* second fetch */
  if (copy_from_user(kattr, uattr, size))
    return -EFAULT;

}
/* BUG: when attr->size is used later */
copy_to_user(uattr, kattr, kattr->size))
```

Missing check: `kattr → size = size`

Control dependency on variable size

Data dependency on variable size



Symbolic checking

```
static int perf_copy_attr(
    struct perf_event_attr __user *uattr,
    struct perf_event_attr *kattr) {

  u32 size;

  /* first fetch */
  if (copy_from_user(&size, &uattr->size, 4))
    return -EFAULT;

  /* sanity checks */
  if (size > PAGE_SIZE || size < PERF_ATTR_SIZE_VER0)
    return -EFAULT;

  /* second fetch */
  if (copy_from_user(kattr, uattr, size))
    return -EFAULT;
  ......
}
/* BUG: when attr->size is used later */
copy_to_user(uattr, kattr, kattr->size))
```

Symbolic checking

```
static int perf_copy_attr(
    struct perf_event_attr __user *uattr,
    struct perf_event_attr *kattr) {

  u32 size;

  /* first fetch */
  if (copy_from_user(&size, &uattr->size, 4))
    return -EFAULT;

  /* sanity checks */
  if (size > PAGE_SIZE || size < PERF_ATTR_SIZE_VER0)
    return -EFAULT;

  /* second fetch */
  if (copy_from_user(kattr, uattr, size))
    return -EFAULT;
  ......
}
/* BUG: when attr->size is used later */
copy_to_user(uattr, kattr, kattr->size))
```

```
$1 = PARAM(uattr), @1 = USER_MEM(uattr)  // uattr
$2 = PARAM(kattr), @2 = KERN_MEM(kattr)  // kattr
```

Symbolic checking

```
static int perf_copy_attr(
    struct perf_event_attr __user *uattr,
    struct perf_event_attr *kattr) {

  u32 size;

  /* first fetch */
  if (copy_from_user(&size, &uattr->size, 4))
    return -EFAULT;

  /* sanity checks */
  if (size > PAGE_SIZE || size < PERF_ATTR_SIZE_VER0)
    return -EFAULT;

  /* second fetch */
  if (copy_from_user(kattr, uattr, size))
    return -EFAULT;
  ......
}
/* BUG: when attr->size is used later */
copy_to_user(uattr, kattr, kattr->size))
```

```
$1 = PARAM(uattr), @1 = USER_MEM(uattr)  // uattr
$2 = PARAM(kattr), @2 = KERN_MEM(kattr)  // kattr

/* first fetch */
fetch(F1): {A = $1, S = 4}
$3 ← @1(0, 4, U1), @3 = nil                // size
```

Symbolic checking

```
static int perf_copy_attr(
    struct perf_event_attr __user *uattr,
    struct perf_event_attr *kattr) {

  u32 size;

  /* first fetch */
  if (copy_from_user(&size, &uattr->size, 4))
    return -EFAULT;

  /* sanity checks */
  if (size > PAGE_SIZE || size < PERF_ATTR_SIZE_VER0)
    return -EFAULT;

  /* second fetch */
  if (copy_from_user(kattr, uattr, size))
    return -EFAULT;
  ......
}
/* BUG: when attr->size is used later */
copy_to_user(uattr, kattr, kattr->size))
```

```
$1 = PARAM(uattr), @1 = USER_MEM(uattr)  // uattr
$2 = PARAM(kattr), @2 = KERN_MEM(kattr)  // kattr

/* first fetch */
fetch(F1): {A = $1, S = 4}
$3 ← @1(0, 4, U1), @3 = nil                // size

/* sanity checks */
assert: $3 ≤ PAGE_SIZE AND $3 ≥ PERF_ATTR_SIZE_VER0
```

Symbolic checking

```
static int perf_copy_attr(
    struct perf_event_attr __user *uattr,
    struct perf_event_attr *kattr) {

  u32 size;

  /* first fetch */
  if (copy_from_user(&size, &uattr->size, 4))
    return -EFAULT;

  /* sanity checks */
  if (size > PAGE_SIZE || size < PERF_ATTR_SIZE_VER0)
    return -EFAULT;

  /* second fetch */
  if (copy_from_user(kattr, uattr, size))
    return -EFAULT;
  ......
}
/* BUG: when attr->size is used later */
copy_to_user(uattr, kattr, kattr->size))
```

```
$1 = PARAM(uattr), @1 = USER_MEM(uattr)  // uattr
$2 = PARAM(kattr), @2 = KERN_MEM(kattr)  // kattr

/* first fetch */
fetch(F1): {A = $1, S = 4}
$3 ← @1(0, 4, U1), @3 = nil                // size

/* sanity checks */
assert: $3 ≤ PAGE_SIZE AND $3 ≥ PERF_ATTR_SIZE_VER0

/* second fetch */
fetch(F2): {A = $1, S = $3}
@2(0, $2, K) ← @1(0, S2, U2)
```

Symbolic checking

```
static int perf_copy_attr(
    struct perf_event_attr __user *uattr,
    struct perf_event_attr *kattr) {

  u32 size;

  /* first fetch */
  if (copy_from_user(&size, &uattr->size, 4))
    return -EFAULT;

  /* sanity checks */
  if (size > PAGE_SIZE || size < PERF_ATTR_SIZE_VER0)
    return -EFAULT;

  /* second fetch */
  if (copy_from_user(kattr, uattr, size))
    return -EFAULT;
  ......
}
/* BUG: when attr->size is used later */
copy_to_user(uattr, kattr, kattr->size))
```

```
$1 = PARAM(uattr), @1 = USER_MEM(uattr)  // uattr
$2 = PARAM(kattr), @2 = KERN_MEM(kattr)  // kattr

/* first fetch */
fetch(F1): {A = $1, S = 4}
$3 ← @1(0, 4, U1), @3 = nil                // size

/* sanity checks */
assert: $3 ≤ PAGE_SIZE AND $3 ≥ PERF_ATTR_SIZE_VER0

/* second fetch */
fetch(F2): {A = $1, S = $3}
@2(0, $2, K) ← @1(0, S2, U2)

/* check overlap */
check: F2.A ≤ F1.A < (F2.A + F2.S)
     OR F1.A ≤ F2.A < (F1.A + F1.S)
[solve] → SAT with solution @1(0, 4, U)
```

Symbolic checking

```
static int perf_copy_attr(
    struct perf_event_attr __user *uattr,
    struct perf_event_attr *kattr) {

  u32 size;

  /* first fetch */
  if (copy_from_user(&size, &uattr->size, 4))
    return -EFAULT;

  /* sanity checks */
  if (size > PAGE_SIZE || size < PERF_ATTR_SIZE_VER0)
    return -EFAULT;

  /* second fetch */
  if (copy_from_user(kattr, uattr, size))
    return -EFAULT;
  ......
}
/* BUG: when attr->size is used later */
copy_to_user(uattr, kattr, kattr->size))
```

```
$1 = PARAM(uattr), @1 = USER_MEM(uattr)  // uattr
$2 = PARAM(kattr), @2 = KERN_MEM(kattr)  // kattr

/* first fetch */
fetch(F1): {A = $1, S = 4}
$3 ← @1(0, 4, U1), @3 = nil                // size

/* sanity checks */
assert: $3 ≤ PAGE_SIZE AND $3 ≥ PERF_ATTR_SIZE_VER0

/* second fetch */
fetch(F2): {A = $1, S = $3}
@2(0, $2, K) ← @1(0, S2, U2)

/* check overlap */
check: F2.A ≤ F1.A < (F2.A + F2.S)
     OR F1.A ≤ F2.A < (F1.A + F1.S)
[solve] → SAT with solution @1(0, 4, U)

/* check double-fetch bug */
[prove] @1(0, 4, U1) == @1(0, 4, U2) → FAIL
```

(Partial) symbolic model for concurrent memory access

	Sequential representation
Initialize	`(declare mem (Array[Int->Int]))` `(declare i Int)`
store	`(store mem i 30)`
1st load	`(load mem i) => 30`
2nd load	`(load mem i) => 30`

(Partial) symbolic model for concurrent memory access

	Sequential representation	**Concurrent representation**
Initialize	`(declare mem (Array[Int->Int]))`	`(declare mem (Array[(Int Int)->Int]))`
	`(declare i Int)`	`(declare i Int)`
		`(declare v Int)`
store	`(store mem i 30)`	`(store mem (i 0) 30)`
1st load	`(load mem i) => 30`	`(load mem (i 0)) => 30`
2nd load	`(load mem i) => 30`	`(load mem (i 1)) => <unknown>`

Back to our toy program

`count = 0`

Thread 1	Thread 2
`count++;`	`count++;`

`count == ??`

*Assume sequential consistency.

Back to our toy program

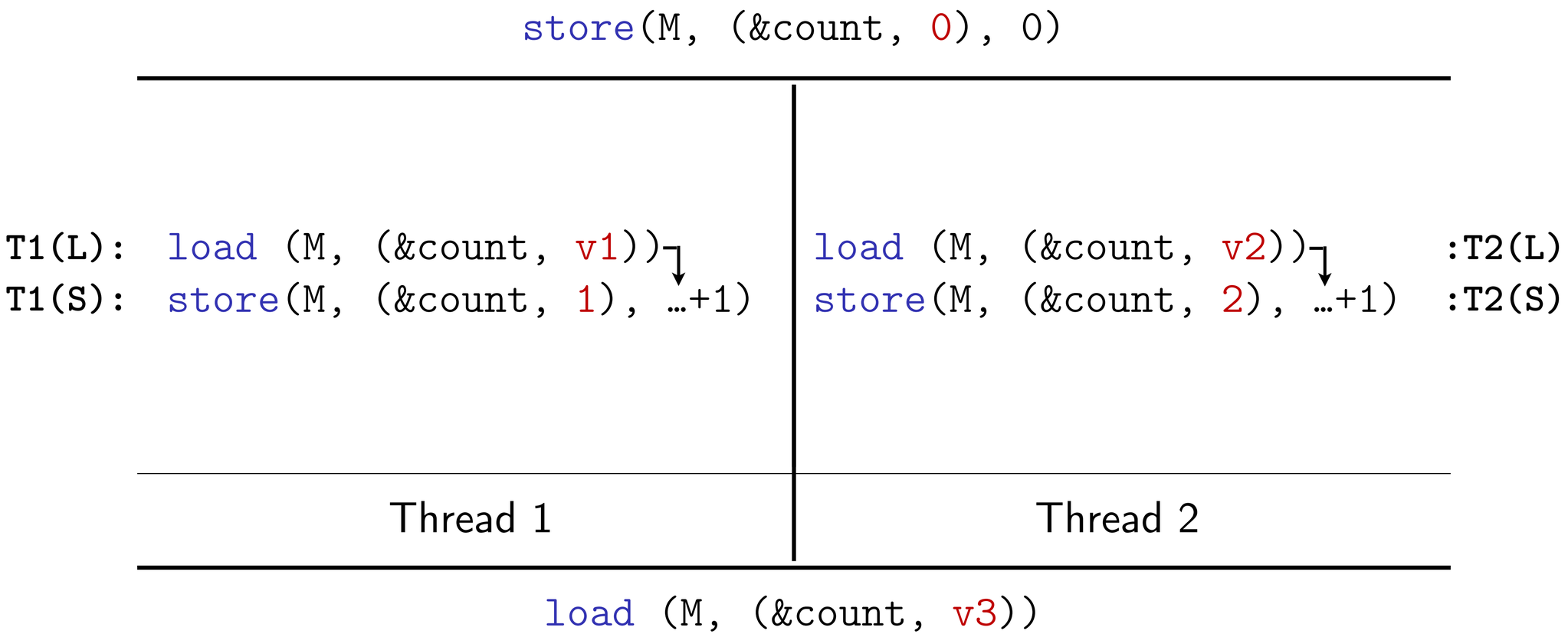
`count = 0`

Thread 1	Thread 2
`temp = load(&count);` `store(&count, temp + 1);`	`temp = load(&count);` `store(&count, temp + 1);`

`load(&count) == ??`

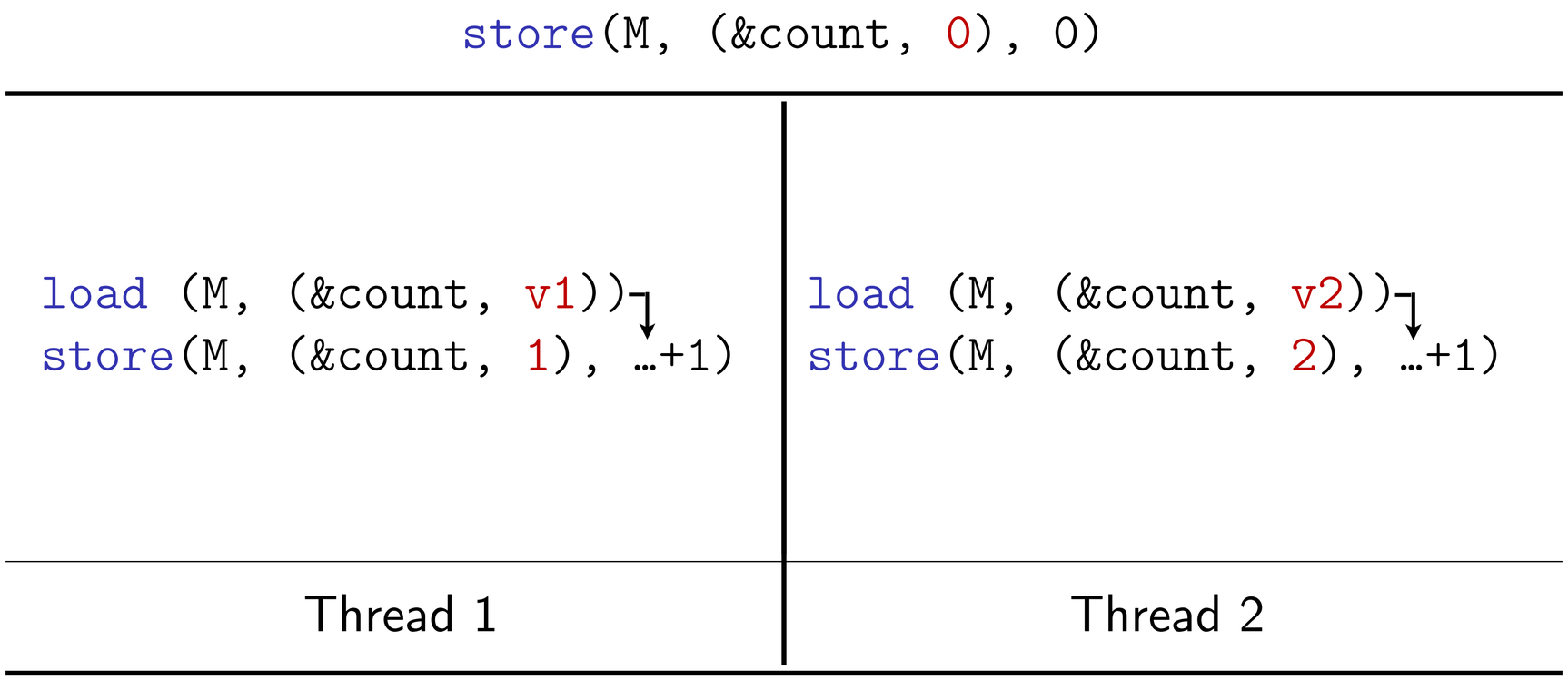
*Assume sequential consistency.

Modeling memory accesses with concurrency versioning



*Assume sequential consistency.

Modeling memory accesses with concurrency versioning



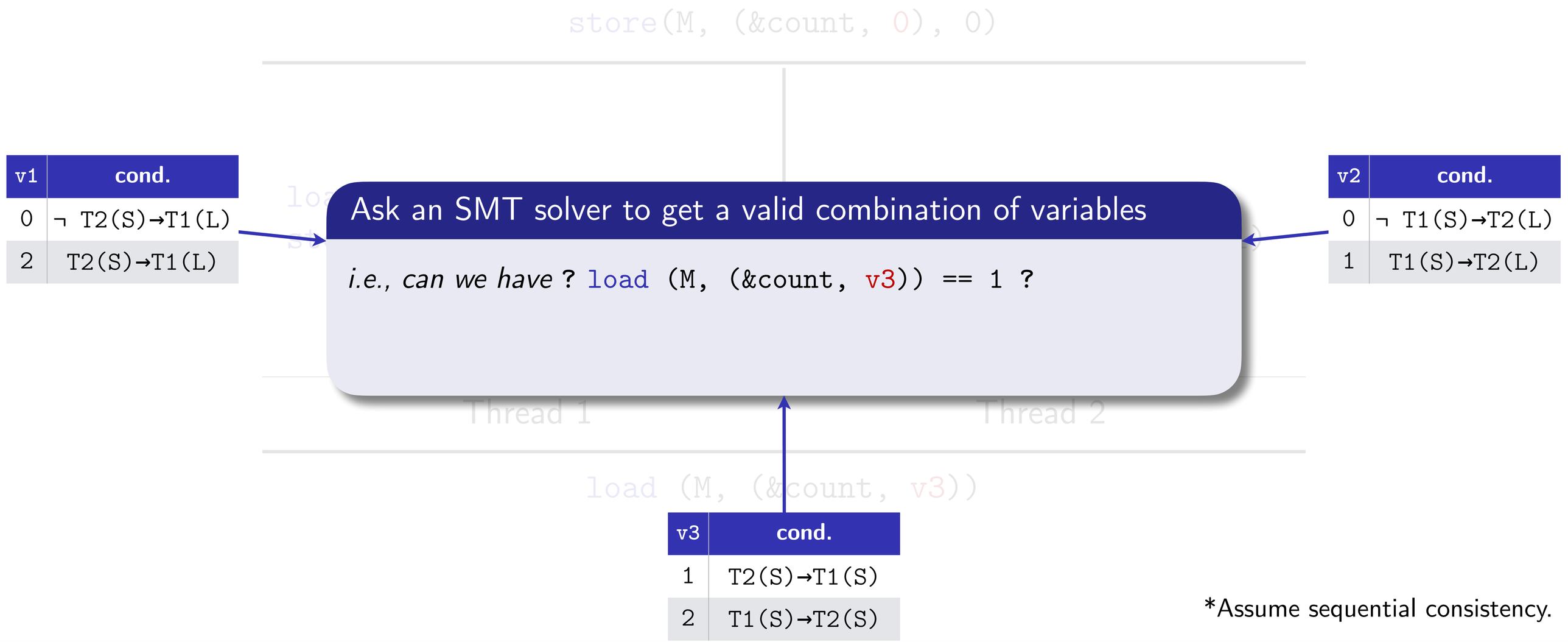
v1	cond.
0	¬ T2(S)→T1(L)
2	T2(S)→T1(L)

v2	cond.
0	¬ T1(S)→T2(L)
1	T1(S)→T2(L)

v3	cond.
1	T2(S)→T1(S)
2	T1(S)→T2(S)

*Assume sequential consistency.

Modeling memory accesses with concurrency versioning



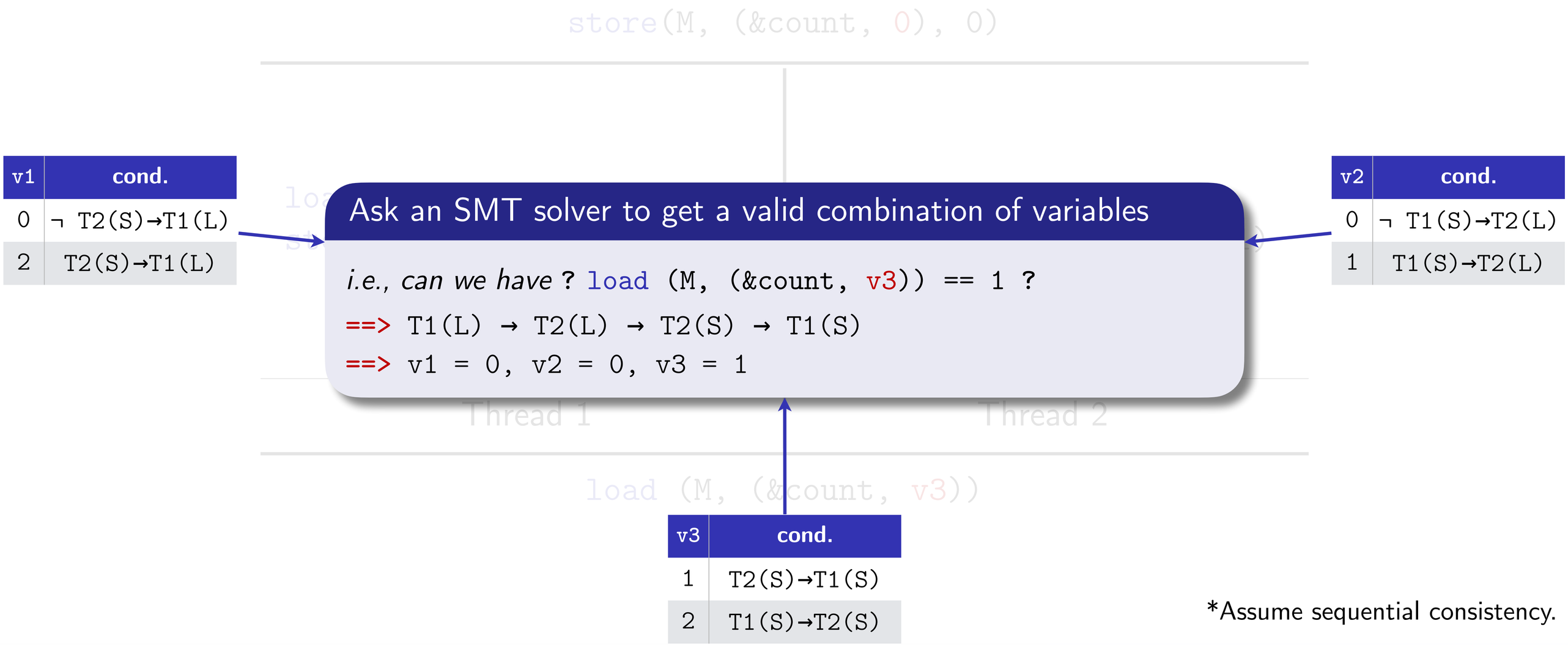
v1	cond.
0	¬ T2(S)→T1(L)
2	T2(S)→T1(L)

v2	cond.
0	¬ T1(S)→T2(L)
1	T1(S)→T2(L)

v3	cond.
1	T2(S)→T1(S)
2	T1(S)→T2(S)

*Assume sequential consistency.

Modeling memory accesses with concurrency versioning



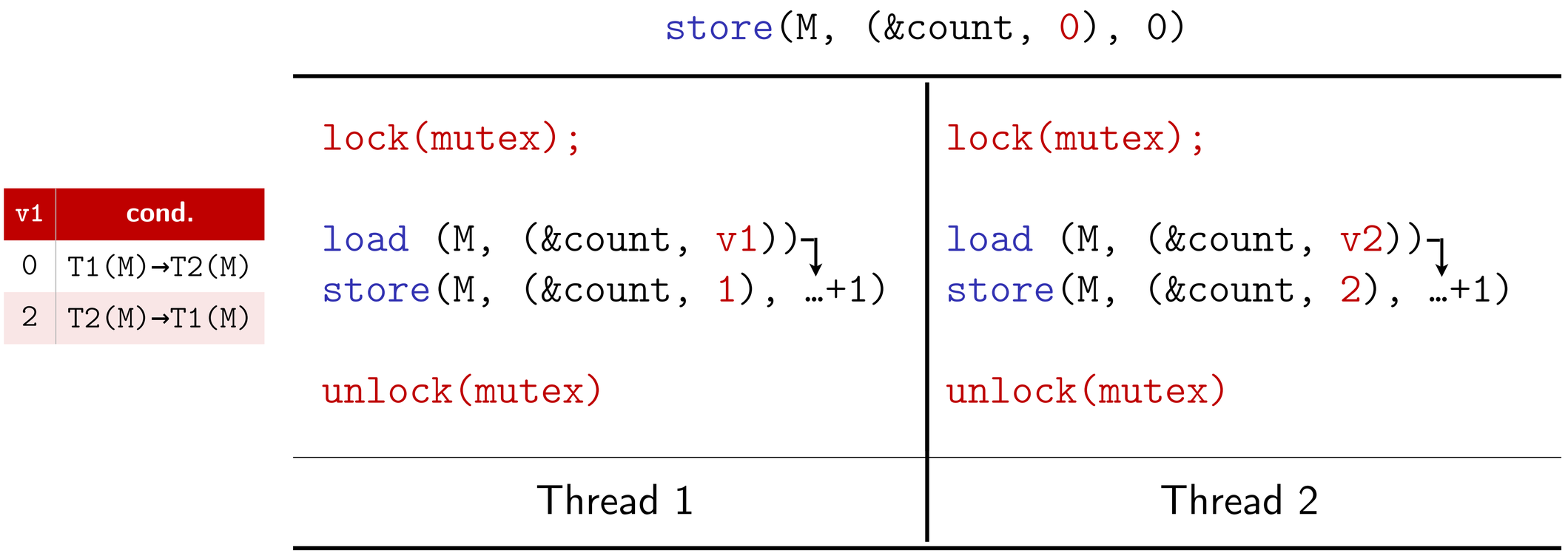
v1	cond.
0	¬ T2(S)→T1(L)
2	T2(S)→T1(L)

v2	cond.
0	¬ T1(S)→T2(L)
1	T1(S)→T2(L)

v3	cond.
1	T2(S)→T1(S)
2	T1(S)→T2(S)

*Assume sequential consistency.

Modeling locks as additional constraints



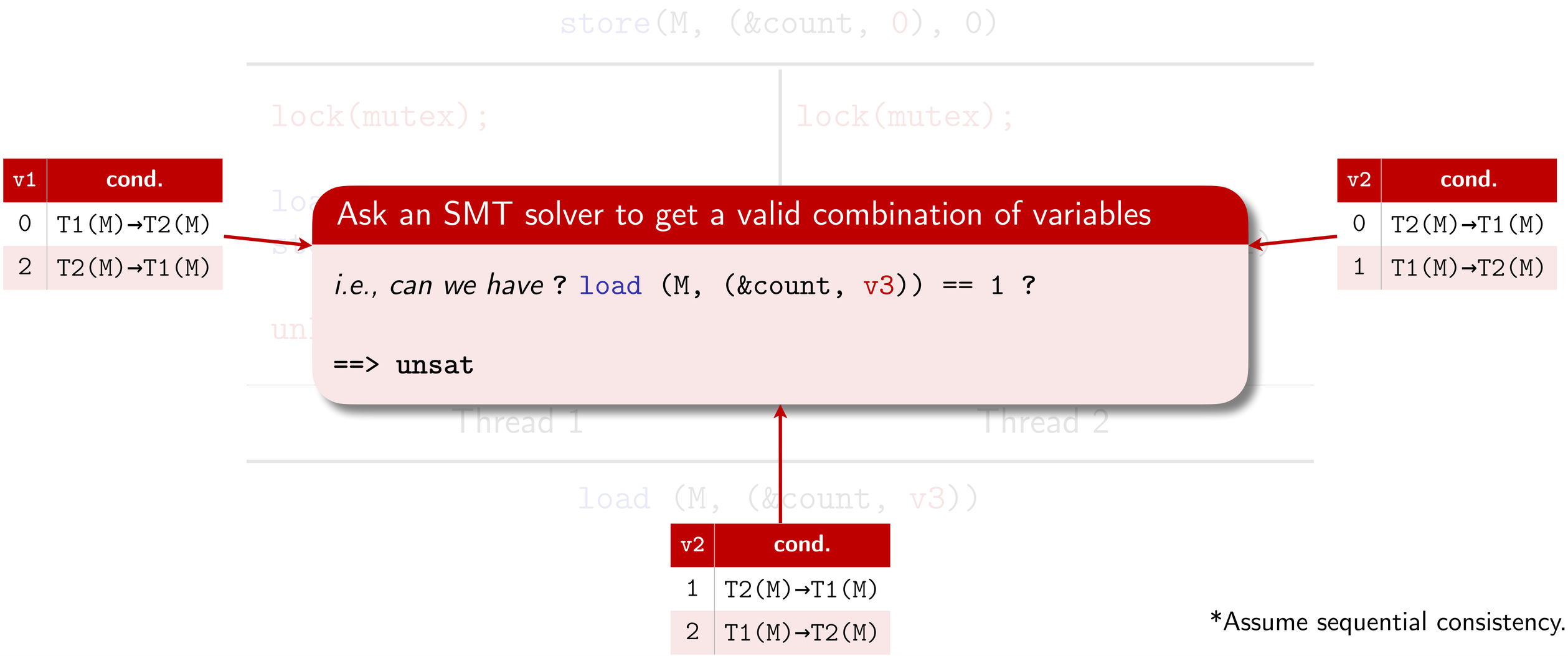
v1	cond.
0	T1(M)→T2(M)
2	T2(M)→T1(M)

v2	cond.
0	T2(M)→T1(M)
1	T1(M)→T2(M)

v2	cond.
1	T2(M)→T1(M)
2	T1(M)→T2(M)

*Assume sequential consistency.

Modeling locks as additional constraints



*Assume sequential consistency.

The challenges faced in extending our SP'18 work

1. Path explosion due to the number of branches
   - e.g., a typical filesystem-related syscall sees 60+ branches $\Rightarrow 2^{60}$ states
2. Handling of unbounded loops
   - e.g., 79% of loops in the Btrfs filesystem (4052 / 5124) are unbounded
3. Memory operations and pointer arithmetics
   - e.g., malloc(<symbolic-size>), memset(..., <symbolic-size>), ...
4. A large and diverse vocabulary of kernel synchronization primitives
   - e.g., sequence locks, RCUs, barriers, etc.

The gap between kernel code and the toy program...

1. Path explosion due to the number of branches
   - e.g., a typical filesystem-related syscall sees 60+ branches $\Rightarrow 2^{60}$ states
2. Han
   - e
3. Memory operations and pointer arithmetics
   - e.g., malloc(<symbolic-size>), memset(..., <symbolic-size>), ...
4. A large and diverse vocabulary of kernel synchronization primitives
   - e.g., sequence locks, RCUs, barriers, etc.

And yet we have to solve all these challenges, ...

If we were to run a symbolic checking on a whole kernel module (e.g., a filesystem)

Whole-program lossless symbolic representation

Translating bug description — focus of the SP'18 paper

Symbolizing program — focus of the ongoing CAV'21 paper

Will variable "*s*" overflow in the program?

```
int loop(int x) {
  int s = 1;
  for (int i=1; i<=x; i++) {
    s *= i;
  }
  return s;
}
```

???

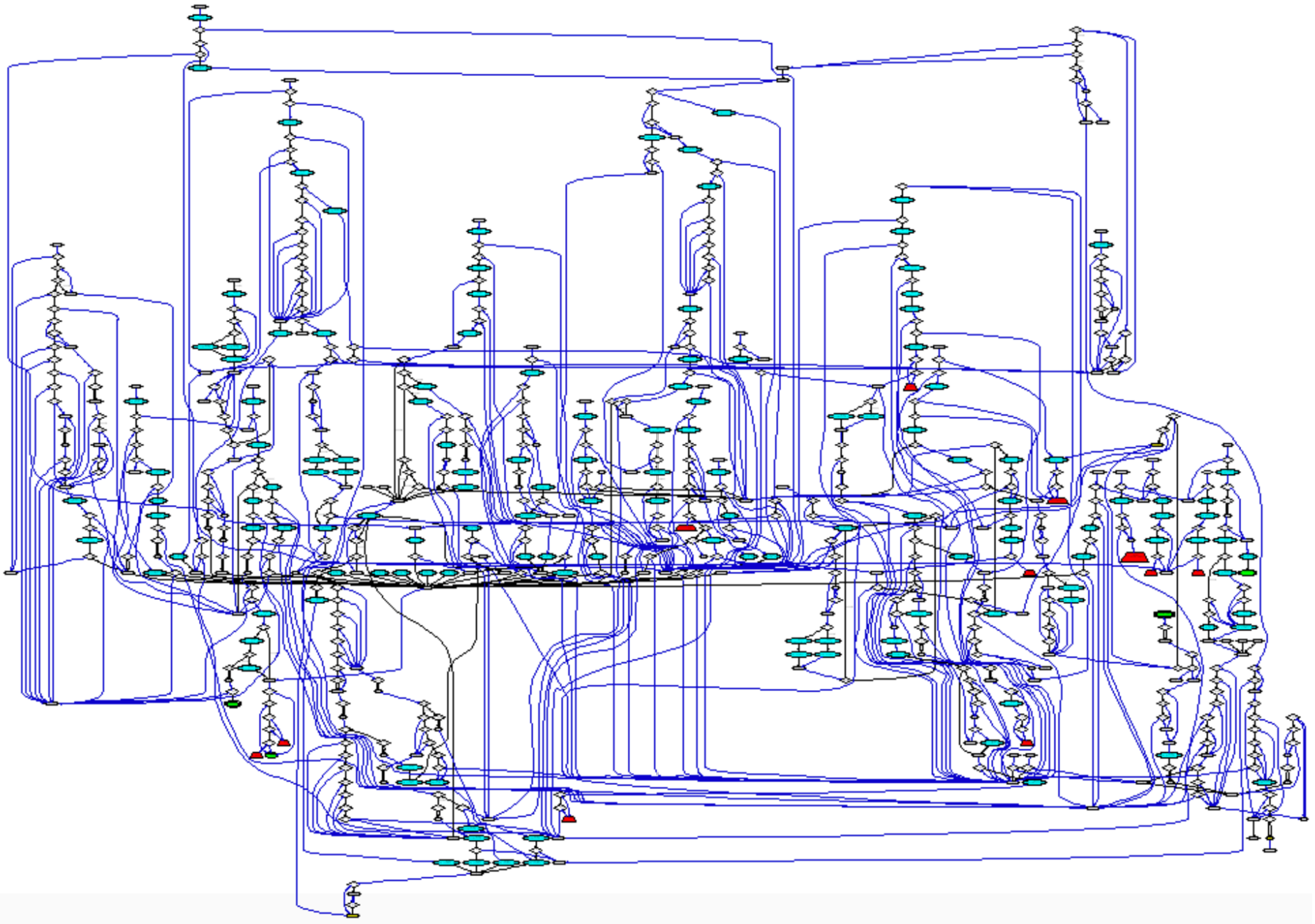
???

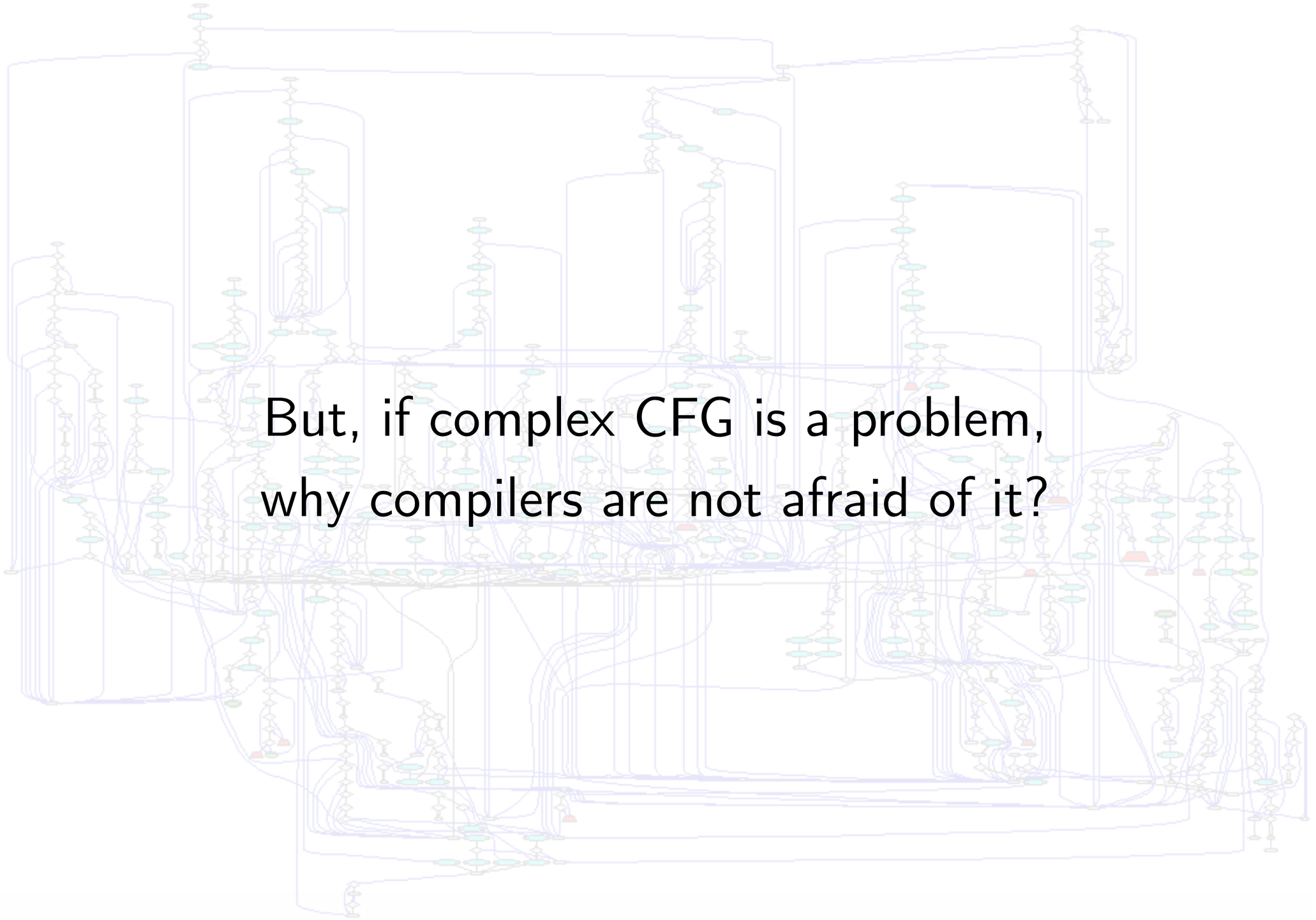
The problem we have in mind

The SMT script we formulated

The answer given by Z3 SMT solver

Problem 1: Path explosion due to branching

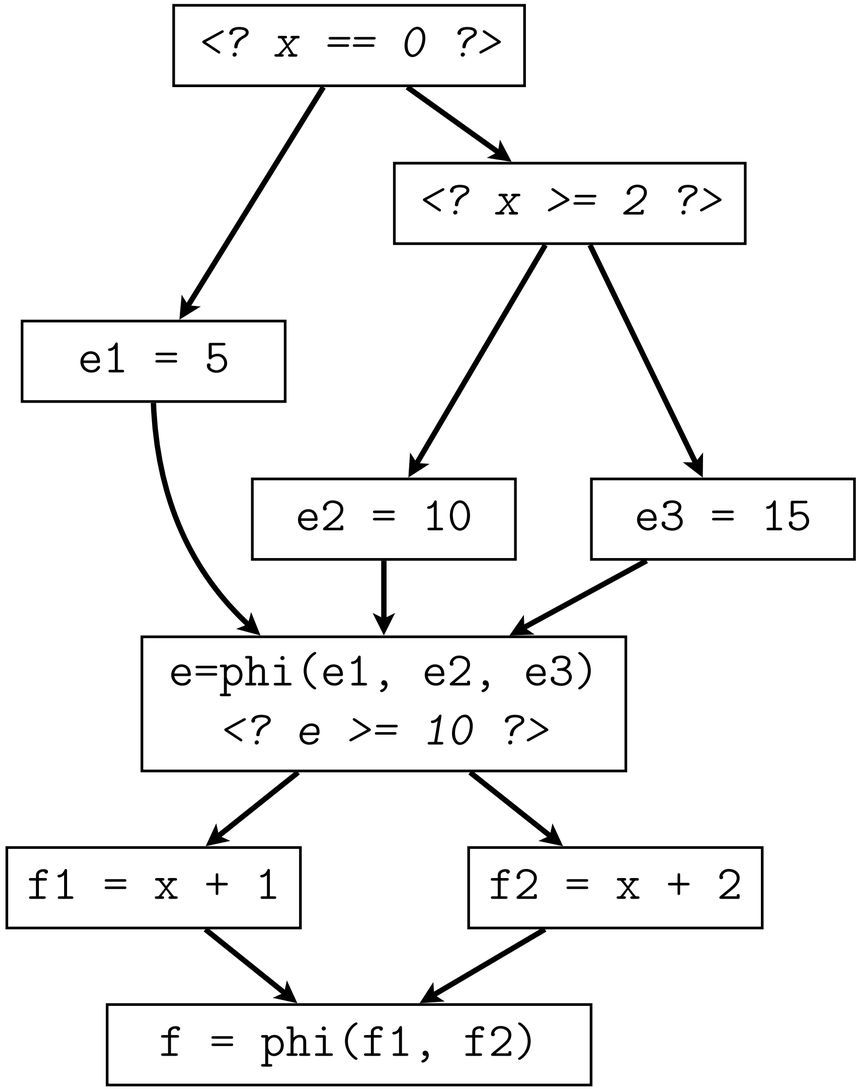




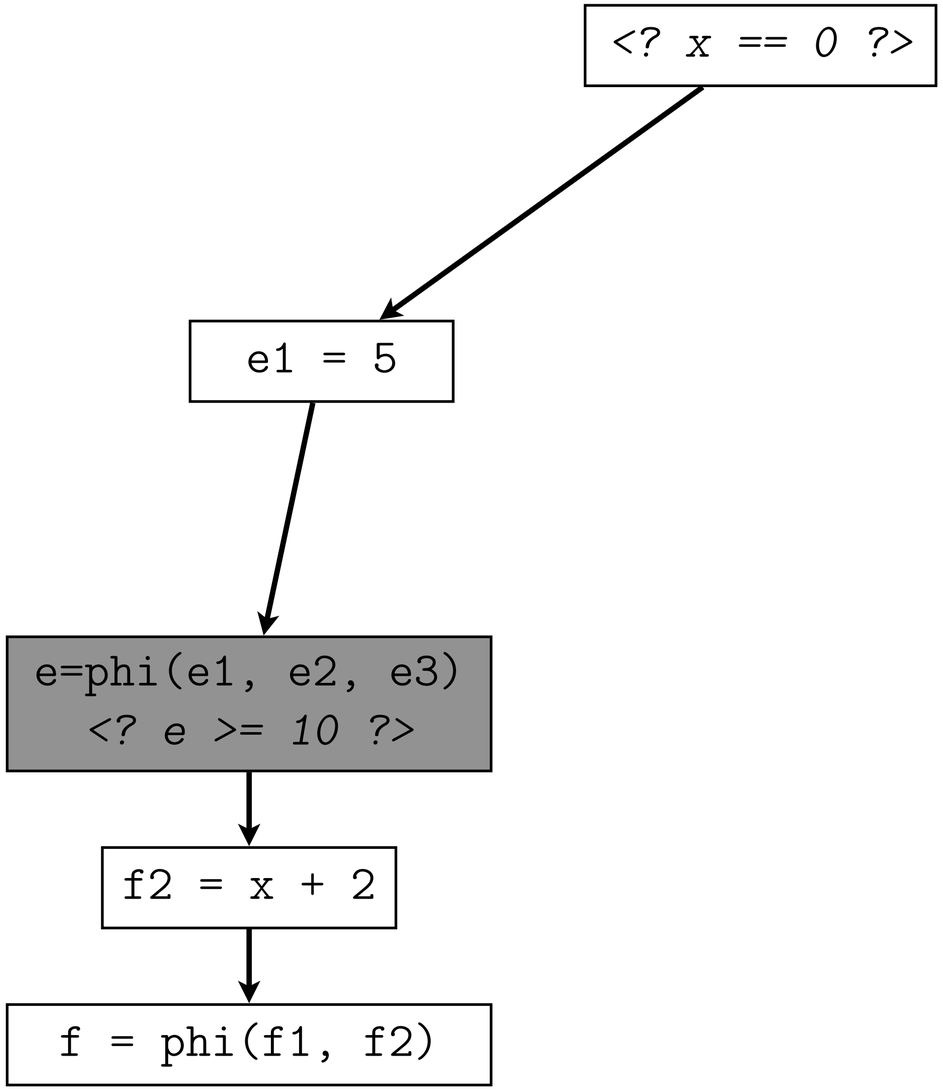
But, if complex CFG is a problem,
why compilers are not afraid of it?

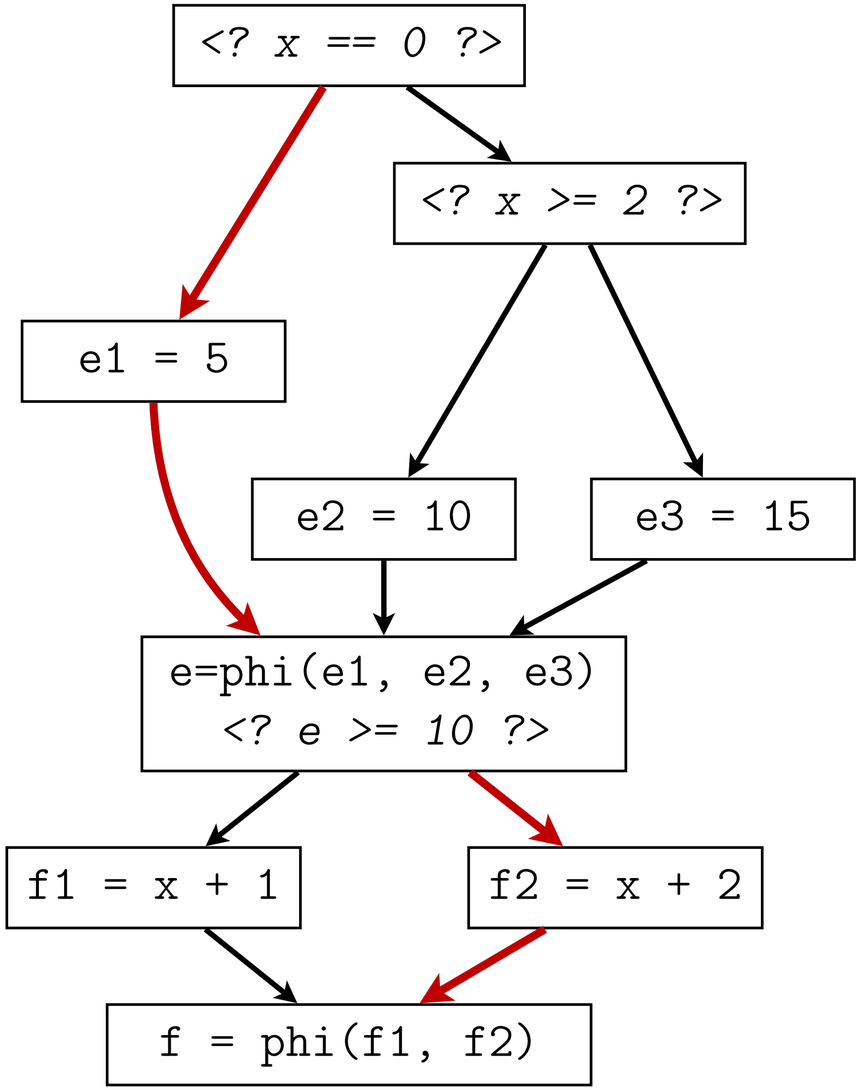
Problem 1: Path explosion due to branching

```
unsigned func(unsigned x) {
  unsigned e;
  if (x == 0) {
    e = 5;
  } else {
    if (x >= 2) {
      e = 10;
    } else {
      e = 15;
    }
  }
  unsigned f;
  if (e >= 10) {
    f = x + 1;
  } else {
    f = x + 2;
  }
  return f;
}
```

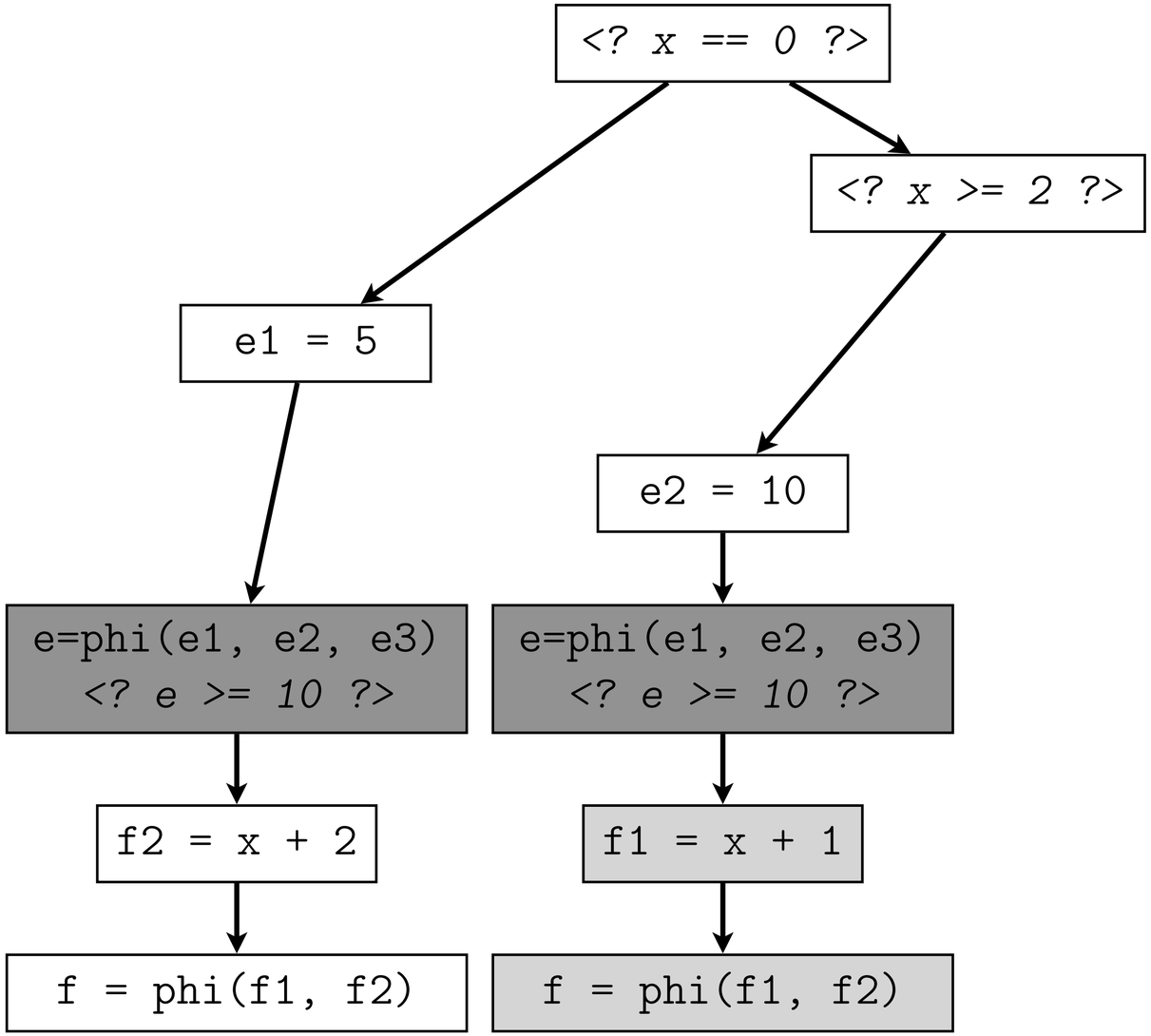


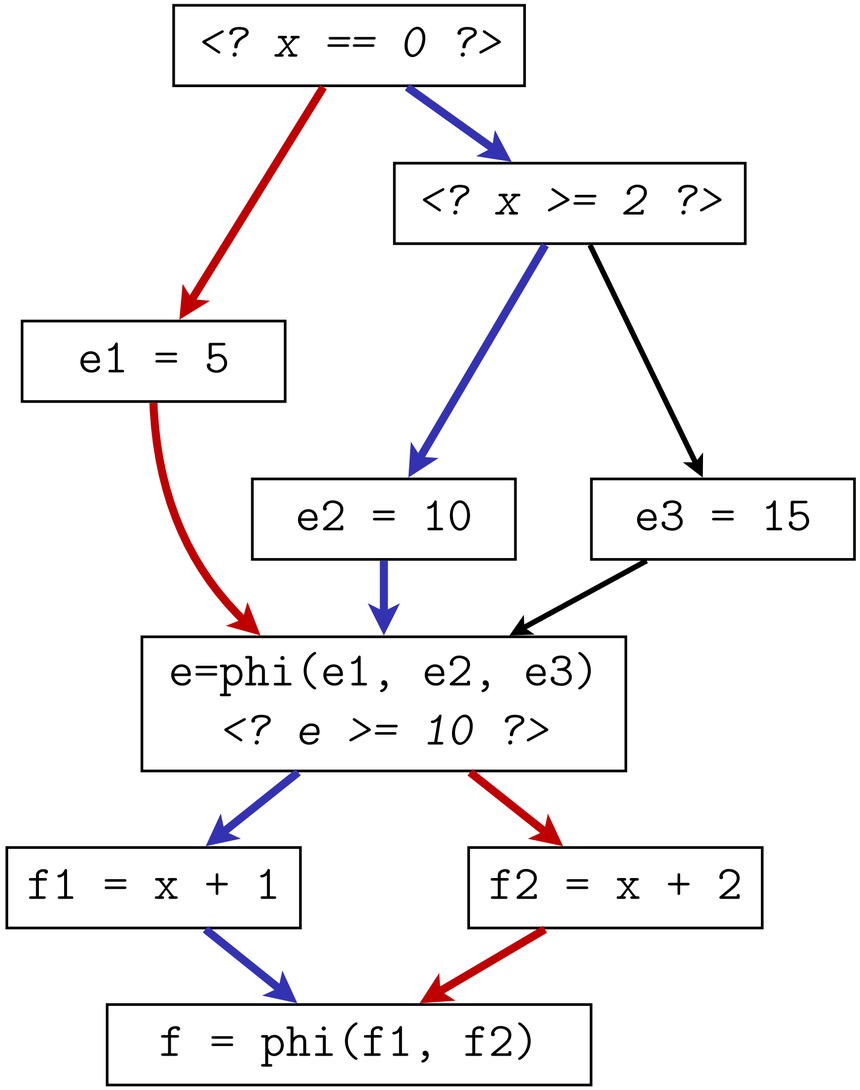
Problem 1: Path explosion due to branching



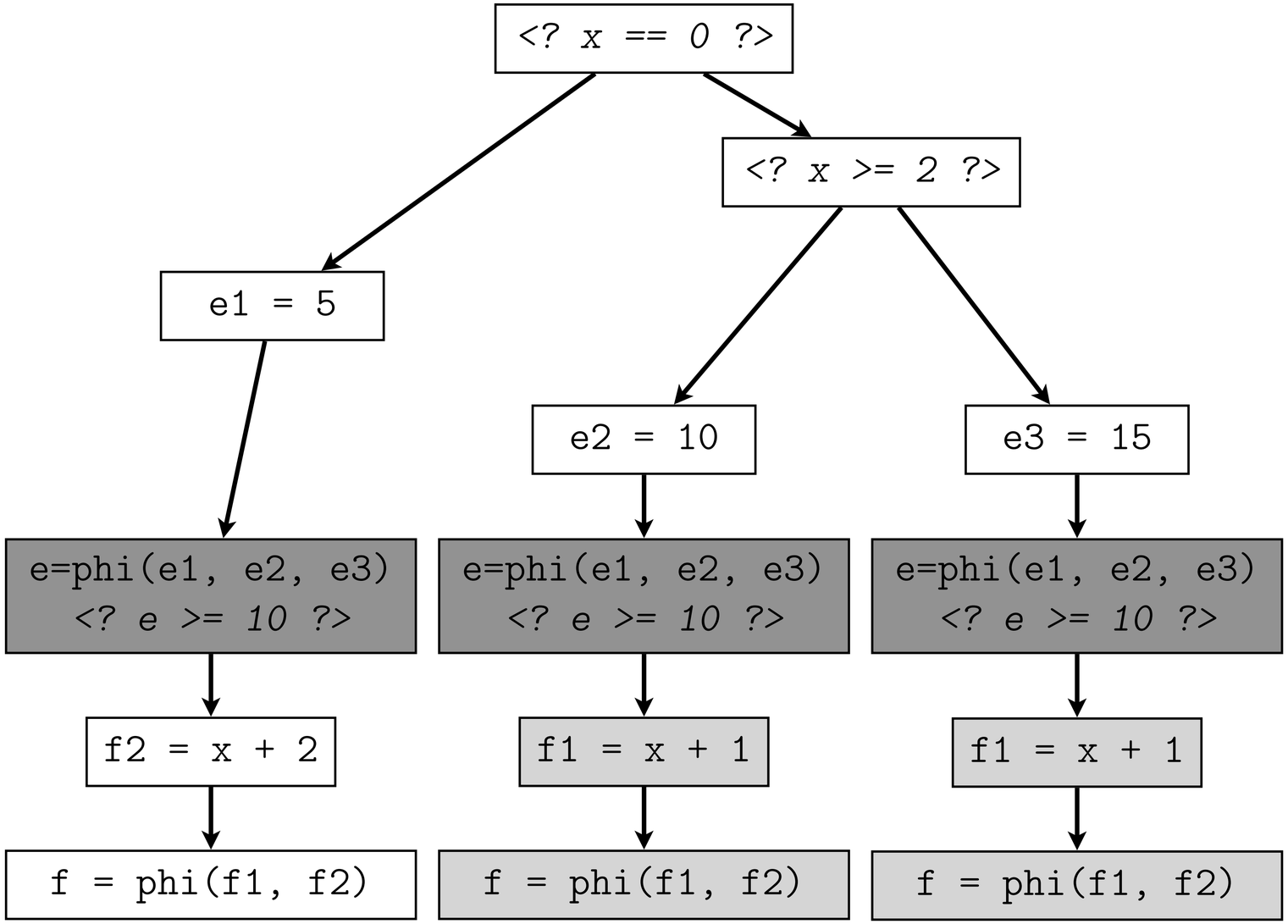


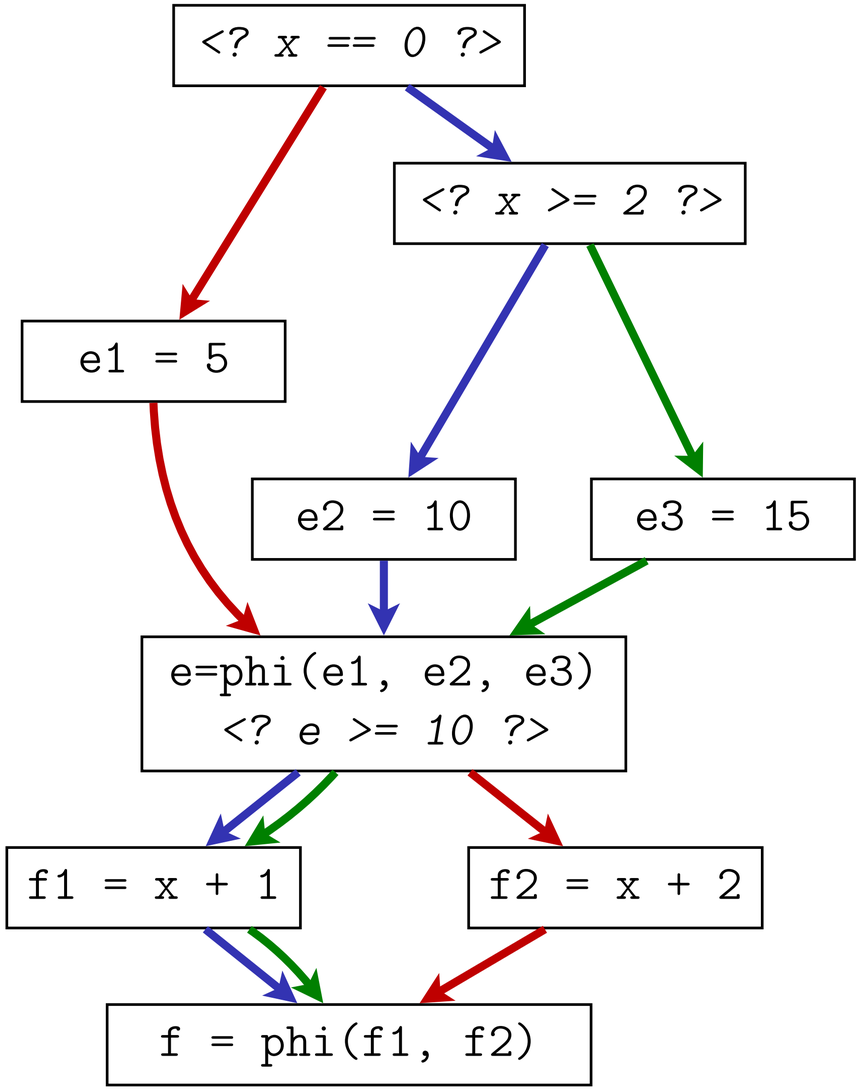
Problem 1: Path explosion due to branching



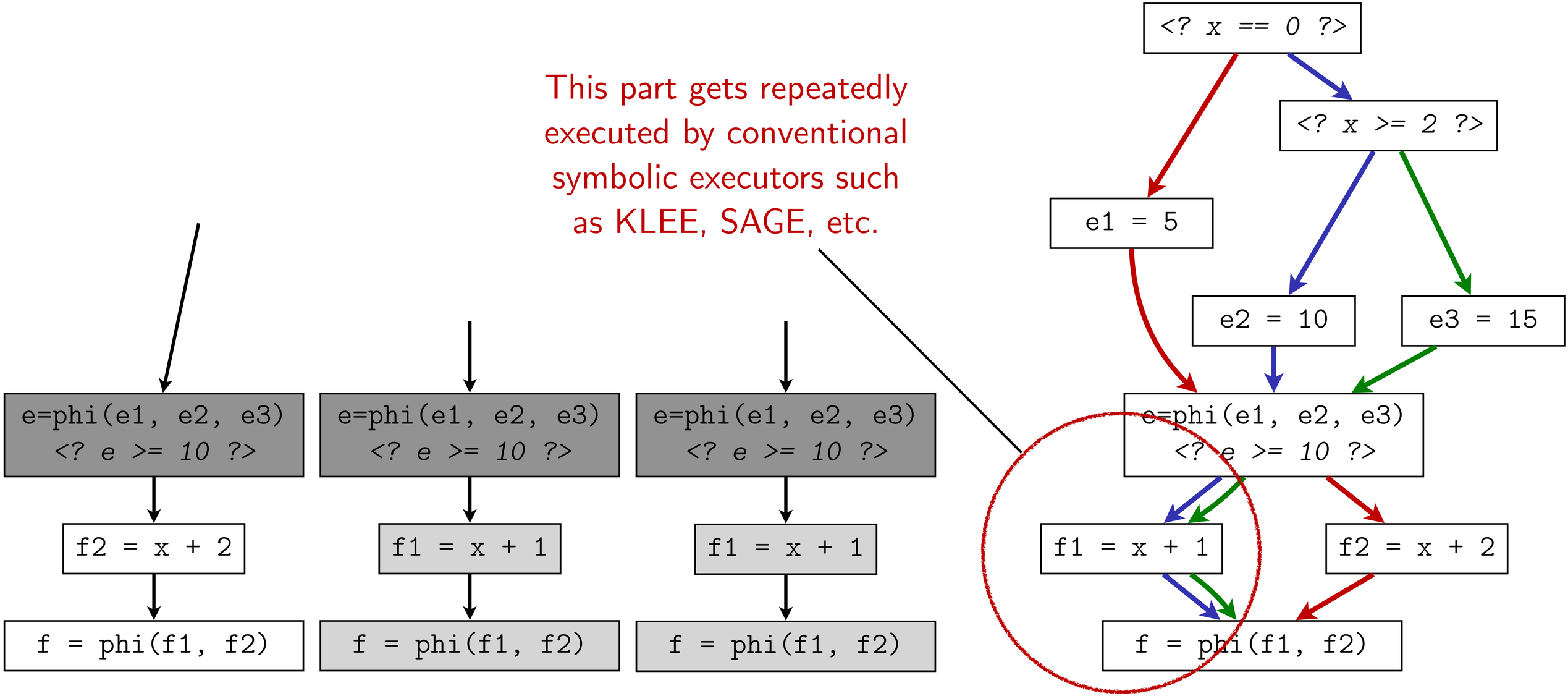


Problem 1: Path explosion due to branching

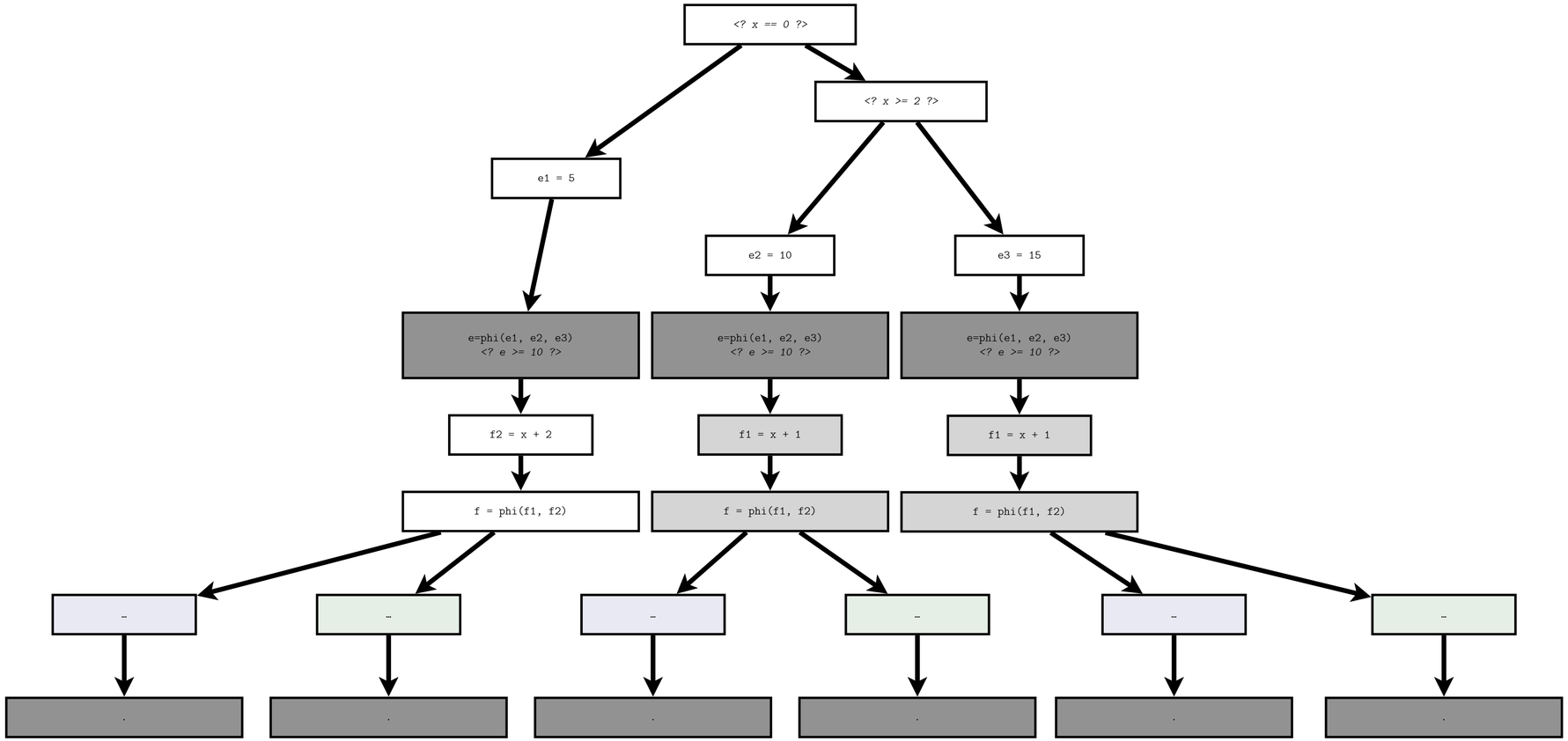


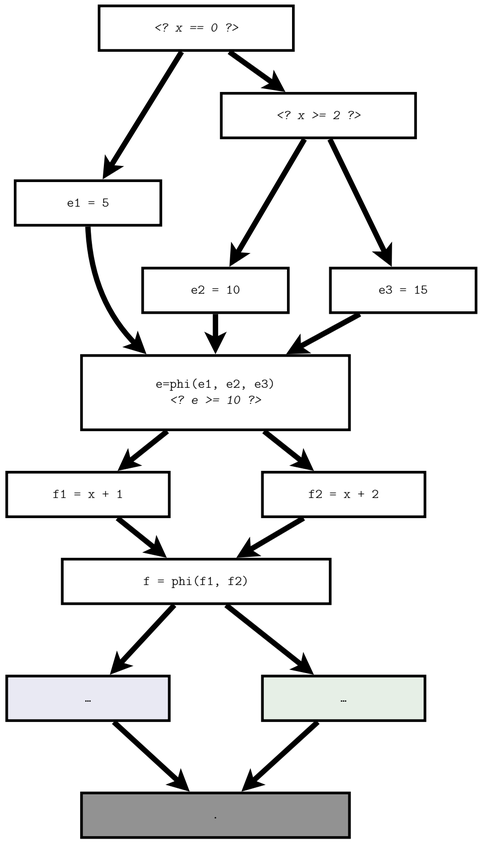


Problem 1: Path explosion due to branching



Problem 1: Path explosion due to branching





This is path explosion although there is only a small change at the bottom of the CFG

Problem 1: Path explosion due to branching

- Is there a way to avoid state forking?
  - Yes, as long as we do not try to enumerate all paths!

- Can we faithfully summarize a program without enumerating all paths?
  - Yes!
  - Besides depth-first exploration, there is breadth-first search for graphs.

Solution 1: Guarded symbolic representation

{{ pre-condition }}

[

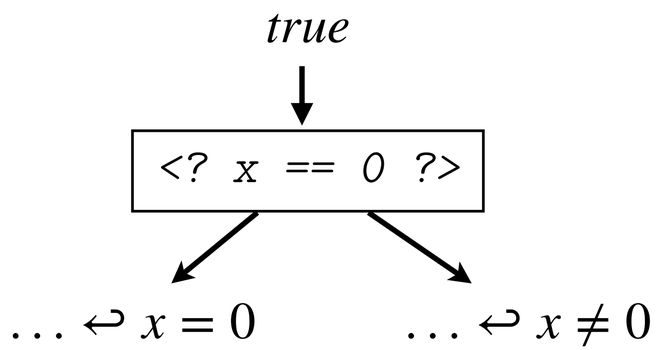
...

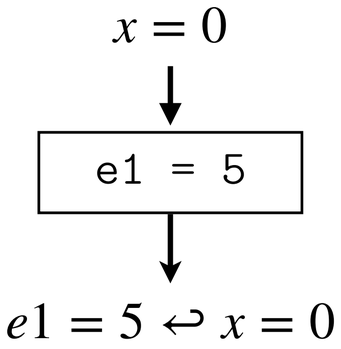
... <basic-block>

...

]

{{ post-condition }}





Solution 1: Guarded symbolic representation

{{ pre-condition }}

[

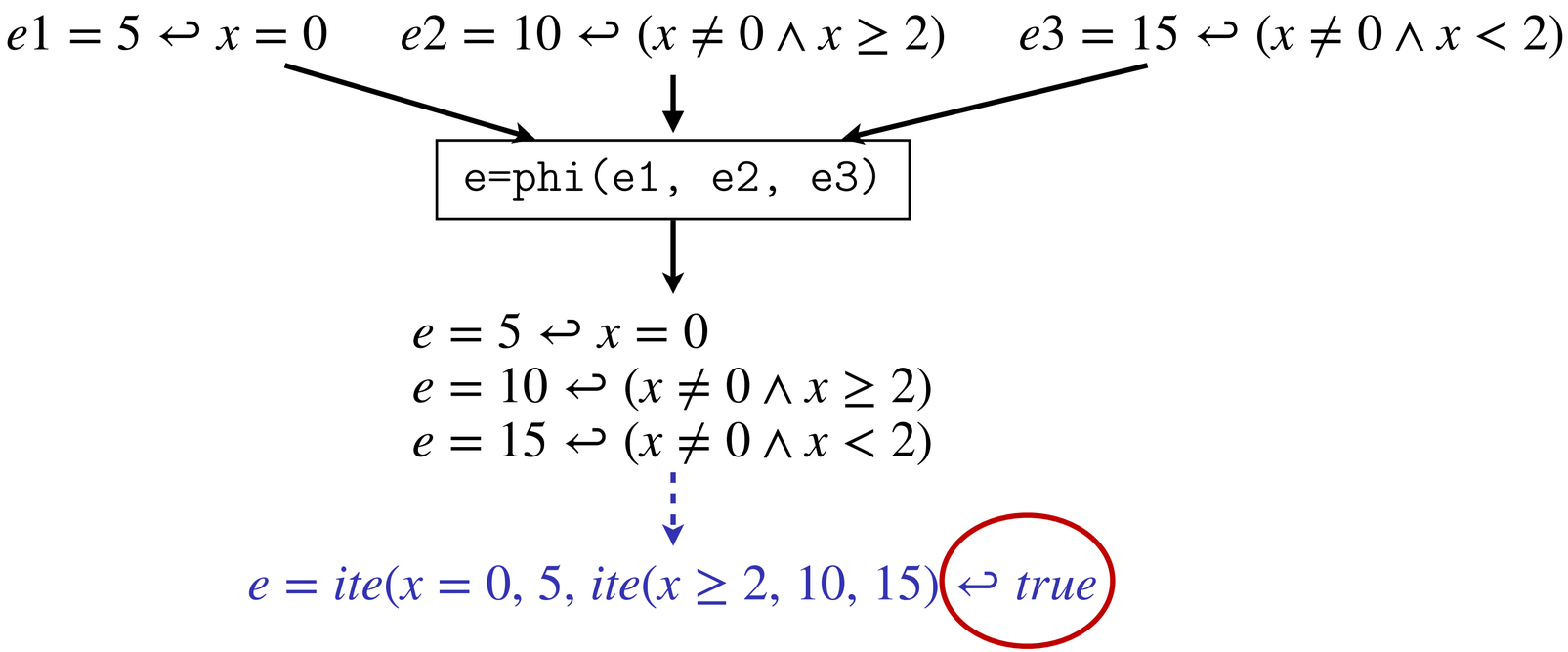
...

... <basic-block>

...

]

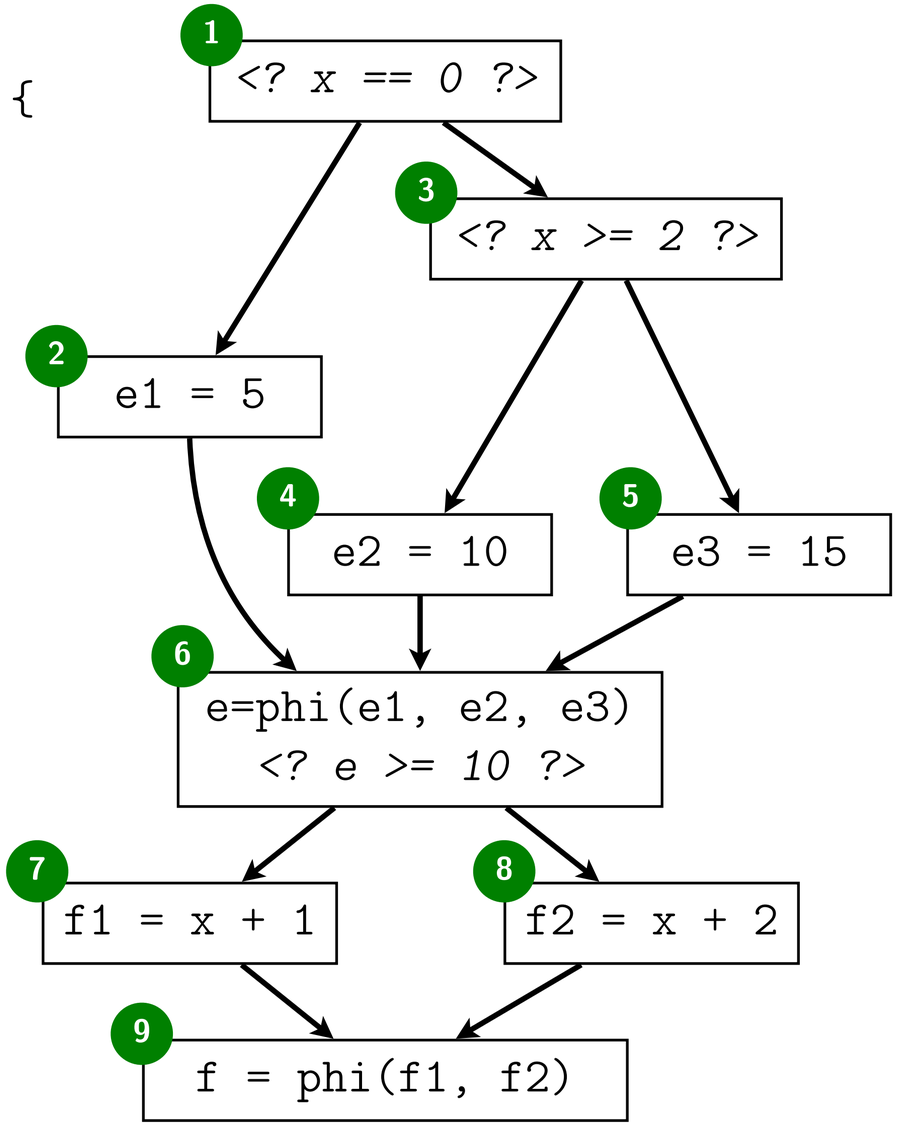
{{ post-condition }}



Path is joined here!

Solution 1: Guarded symbolic representation

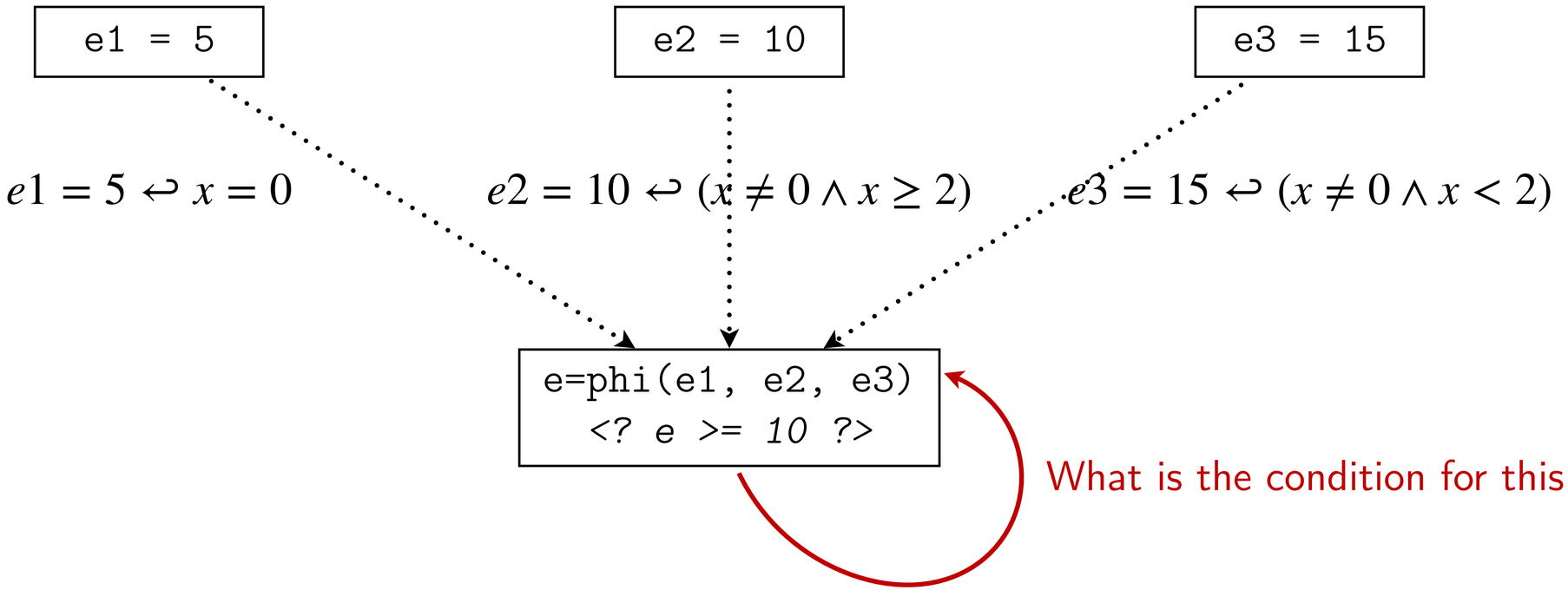
```
unsigned func(unsigned x) {
  unsigned e;
  if (x == 0) {
    e = 5;
  } else {
    if (x >= 2) {
      e = 10;
    } else {
      e = 15;
    }
  }
  unsigned f;
  if (e >= 10) {
    f = x + 1;
  } else {
    f = x + 2;
  }
  return f;
}
```



$$f = \begin{cases} x+1 \hookleftarrow x = 0 \\ x+2 \hookleftarrow x \neq 0 \end{cases}$$

$$f = ite(x = 0, x+1\ x+2) \hookleftarrow true$$

Problem 2: Loops



Problem 2: Unbounded loops

```
int bar_simple(void) {
  int s = 0;
  for (int i = 0; i < 100; i++) {
    s += i;
  }
  return s;
}
```

Bounded loops: know the number of iteration statically

Problem 2: Unbounded loops

```
int bar_simple(void) {
  int s = 0;
  for (int i = 0; i < 100; i++) {
    s += i;
  }
  return s;
}
```

Bounded loops: know the number of iteration statically

```
int bar(int x) {
  int s = 1;
  for (int i = 1; i <= x; i++) {
    s *= i;
    if (s > 1000) {
      break;
    }
  }
  return s;
}
```

Unbounded loops: the number of loop iteration is unknown

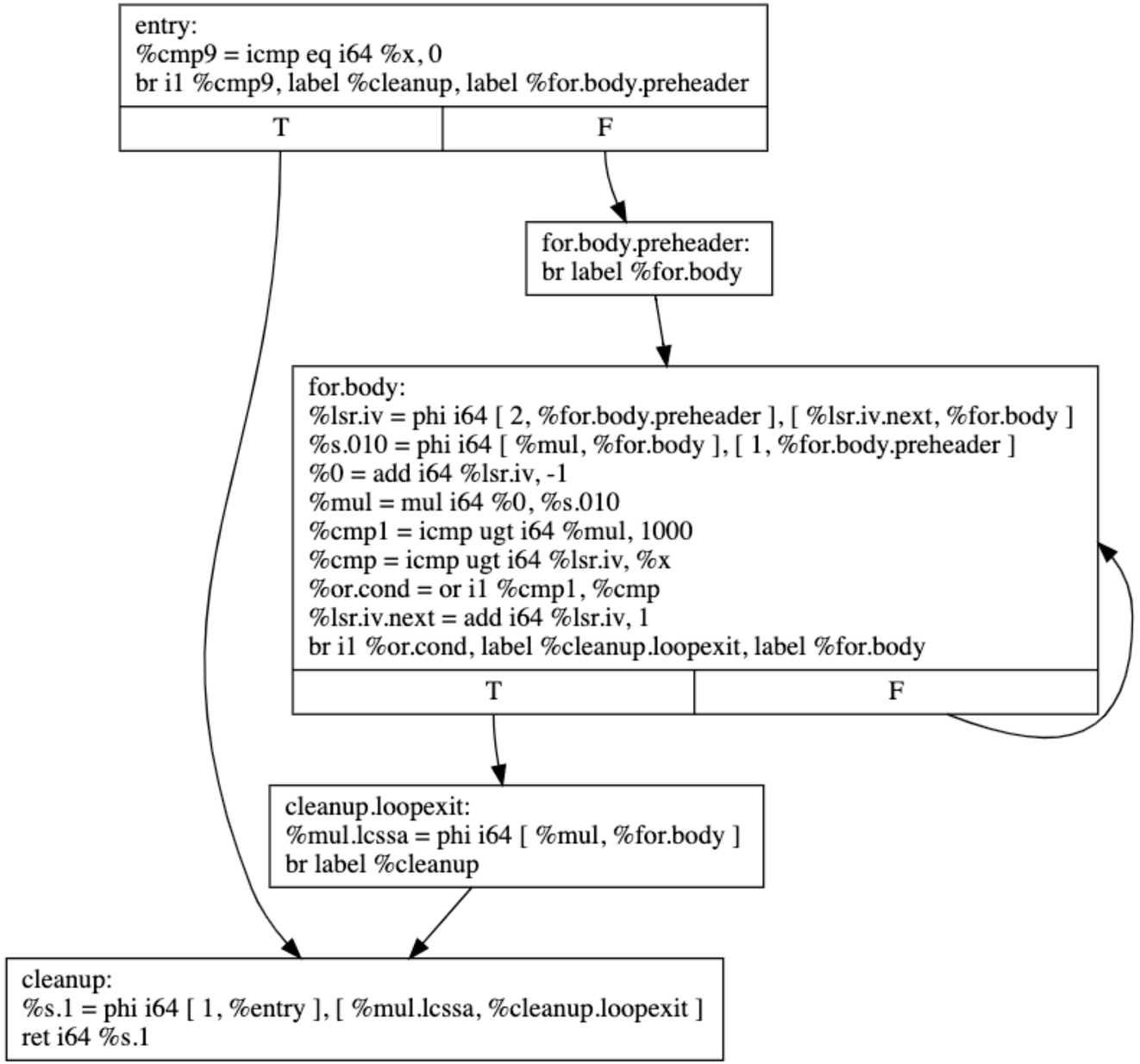
Solution 2: Loop modeling with recursive functions

Every loop can be converted to a recursion in a lossless way,

And SMT solvers like Z3 are capable of handling recursions!

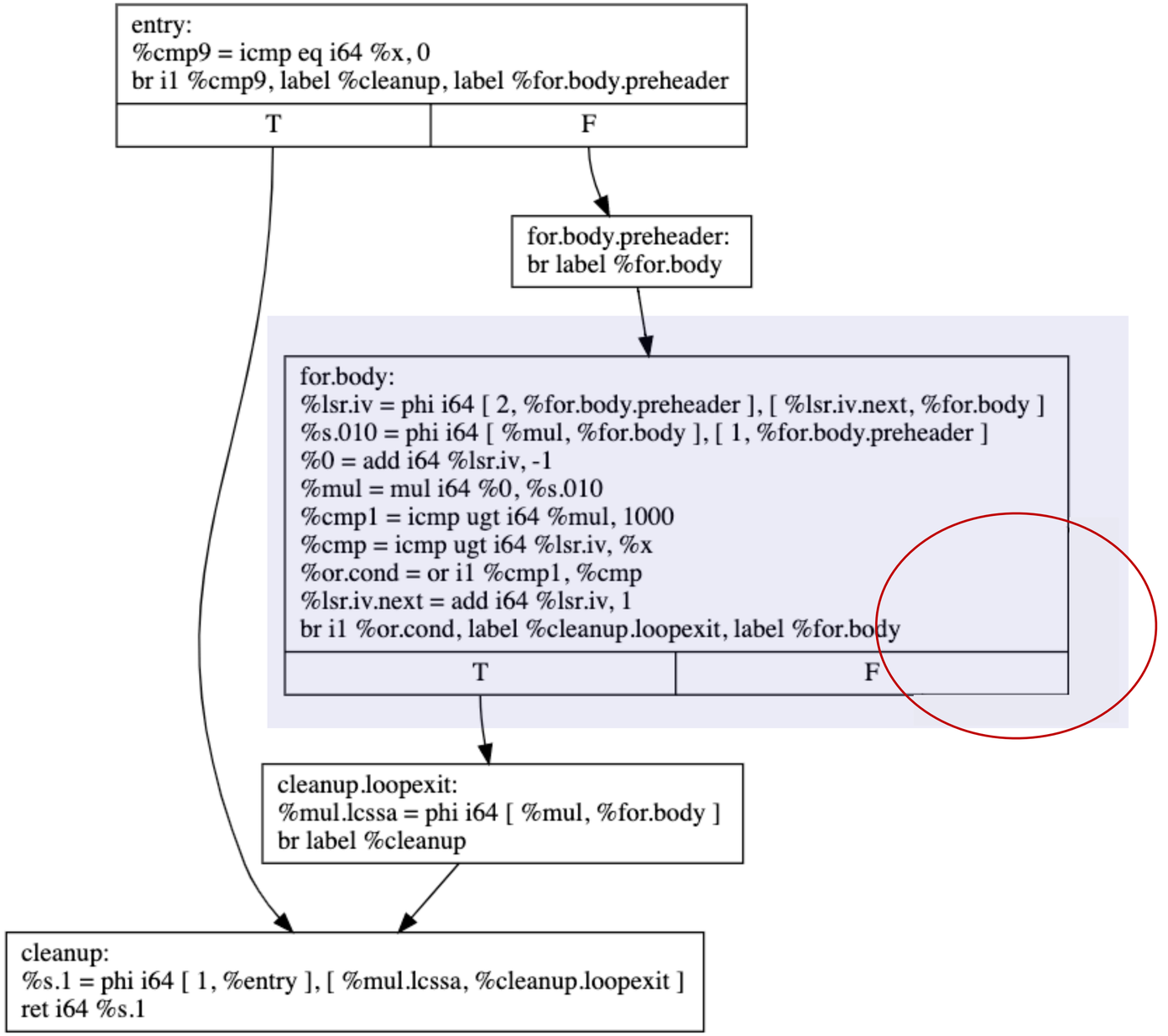
Solution 2: Loop modeling

```
int bar(int x) {
  int s = 1;
  for (int i=1; i<=x; i++) {
    s *= i;
    if (s > 1000) {
      break;
    }
  }
  return s;
}
```



Solution 2: Loop modeling

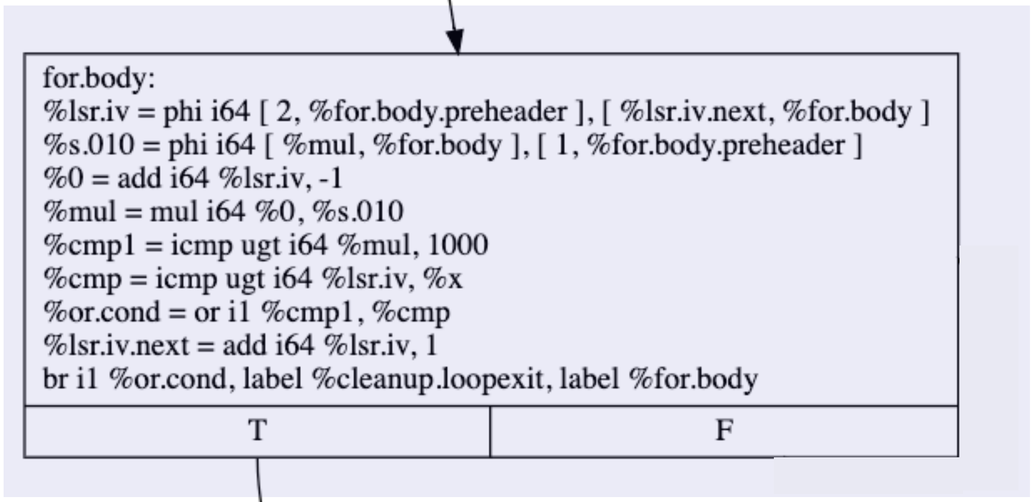
```
int bar(int x) {
  int s = 1;
  for (int i=1; i<=x; i++) {
    s *= i;
    if (s > 1000) {
      break;
    }
  }
  return s;
}
```



Solution 2: Loop modeling

```
int bar(int x) {
  int s = 1;
  for (int i=1; i<=x; i++) {
    s *= i;
    if (s > 1000) {
      break;
    }
  }
  return s;
}
```

$$s_0 = 1 \hookleftarrow x > 0$$

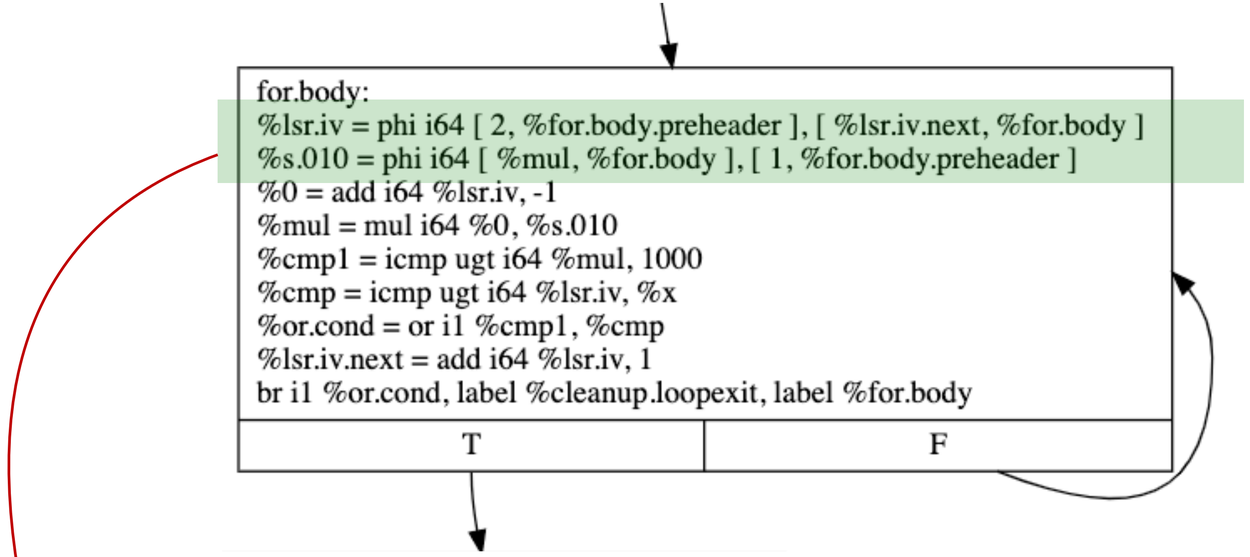


$$s_{phi} = \begin{cases} 5040 \hookleftarrow x \geq 8 \\ x! \quad\quad \hookleftarrow 0 < x < 8 \end{cases}$$

How do we get this?

Solution 2: Loop modeling

```
int bar(int x) {
  int s = 1;
  for (int i=1; i<=x; i++) {
    s *= i;
    if (s > 1000) {
      break;
    }
  }
  return s;
}
```



These are the variables we should recurse on

Solution 2: Loop modeling with recursive functions

```
int bar(int x) {
  int s = 1;
  for (int i=1; i<=x; i++) {
    s *= i;
    if (s > 1000) {
      break;
    }
  }
  return s;
}
```

Suppose we are in k-th iteration

$$f_i(k) = \begin{cases} 1 & \text{if } k = 0 \\ f_i(k-1) + 1 & \text{if } k > 0 \end{cases}$$

Solution 2: Loop modeling with recursive functions

```
int bar(int x) {
  int s = 1;
  for (int i=1; i<=x; i++) {
    s *= i;
    if (s > 1000) {
      break;
    }
  }
  return s;
}
```

Suppose we are in k-th iteration

$$f_i(k) = \begin{cases} 1 & \text{if } k = 0 \\ f_i(k-1) + 1 & \text{if } k > 0 \end{cases}$$

$$f_s(k) = \begin{cases} 1 & \text{if } k = 0 \\ f_s(k-1) \times f_i(k-1) & \text{if } k > 0 \end{cases}$$

Solution 2: Loop modeling with recursive functions

```
int bar(int x) {
  int s = 1;
  for (int i=1; i<=x; i++) {
    s *= i;
    if (s > 1000) {
      break;
    }
  }
  return s;
}
```

Suppose we are in k-th iteration

$$f_i(k) = \begin{cases} 1 & \text{if } k = 0 \\ f_i(k-1) + 1 & \text{if } k > 0 \end{cases}$$

$$f_s(k) = \begin{cases} 1 & \text{if } k = 0 \\ f_s(k-1) \times f_i(k-1) & \text{if } k > 0 \end{cases}$$

$$f_{loop}(k) = \begin{cases} True & \text{if } k = 0 \\ f_{loop}(k-1) \wedge (f_i(k-1) \leq x) \wedge (f_s(k-1) \leq 1000) & \text{if } k > 0 \end{cases}$$

Solution 2: Loop modeling with recursive functions

```
int bar(int x) {
  int s = 1;
  for (int i=1; i<=x; i++) {
    s *= i;
    if (s > 1000) {
      break;
    }
  }
  return s;
}
```

Suppose we are in k-th iteration

$$f_i(k) = \begin{cases} 1 & \text{if } k = 0 \\ f_i(k-1) \ + \ 1 & \text{if } k > 0 \end{cases}$$

$$f_s(k) = \begin{cases} 1 & \text{if } k = 0 \\ f_s(k-1) \ \times \ f_i(k-1) & \text{if } k > 0 \end{cases}$$

$$f_{loop}(k) = \begin{cases} True & \text{if } k = 0 \\ f_{loop}(k-1) \ \wedge \ (f_i(k-1) \leq x) \ \wedge \ (f_s(k-1) \leq 1000) & \text{if } k > 0 \end{cases}$$

Suppose we exited after m-th iteration

$s_{phi} = f_s(m) \hookleftarrow (f_i(m) > x \wedge f_{loop}(m))$ ← Exited through `i > x`

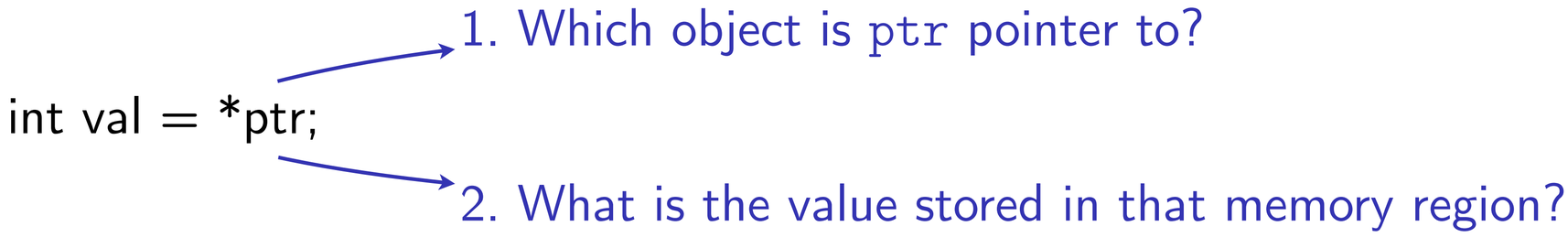
$s_{phi} = f_s(m) \hookleftarrow (f_s(m) > 1000 \wedge f_{loop}(m))$ ← Exited through `s > 1000`

Solution 2: Loop modeling with recursive functions

```
int bar(int x) {
  int s = 1;
  for (int i=1; i<=x; i++) {
    s *= i;
    if (s > 1000) {
      break;
    }
  }
  return s;
}
```

$$s_{phi} = \begin{cases} 5040 & \hookleftarrow x \geq 8 \\ x! & \hookleftarrow 0 < x < 8 \end{cases}$$

Problem 3: Symbolic memory



Problem 3: Symbolic memory

```
if (x == 0) {
  e = 5;
} else {
  if (x >= 2) {
    e = 10;
  } else {
    e = 15;
  }
}
```



```
void *p = malloc(e);
```

Malloc with symbolic size

Problem 3: Symbolic memory

```
if (x == 0) {
  e = 5;
} else {
  if (x >= 2) {
    e = 10;
  } else {
    e = 15;
  }
}
```



```
void *p = malloc(e);
```

```
char *h = malloc(8);
h[0:7] = 1;

if (x > 100) {
  h[0:5] = 2;
} else if (x < 100) {
  h[2:7] = 3;
} else {
  h[3:4] = 4;
}

if (x >= 100) {
  h[1:6] = 5;
}
```

Malloc with symbolic size

Partial override with conditions

Problem 3: Symbolic memory

```
if (x == 0) {
  e = 5;
} else {
  if (x >= 2) {
    e = 10;
  } else {
    e = 15;
  }
}
```



```
void *p = malloc(e);
```

Malloc with symbolic size

```
char *h = malloc(8);
h[0:7] = 1;

if (x > 100) {
  h[0:5] = 2;
} else if (x < 100) {
  h[2:7] = 3;
} else {
  h[3:4] = 4;
}

if (x >= 100) {
  h[1:6] = 5;
}
```

Partial override with conditions

```
char *g = malloc(128);
memset(g, 42, 20);
memset(g, 66, x);
```

Memset with symbolic length

Solution 3: Object-chunk memory model

```
char *p = malloc(e);
```

Obj.	Pointer	Size	Condition
1	0x0001_0000	5	x == 0
2	0x0002_0000	10	x >= 2
3	0x0003_0000	15	x == 1

Solution 3: Object-chunk memory model

```
char *p = malloc(e);
```

Obj.	Pointer	Size	Condition
1	0x0001_0000	5	x == 0
2	0x0002_0000	10	x >= 2
3	0x0003_0000	15	x == 1

Chunk	Offset	Length	Value	Cond.	Blob	Live

Chunk	Offset	Length	Value	Cond.	Blob	Live

Chunk	Offset	Length	Value	Cond.	Blob	Live

Solution 3: Object-chunk memory model

```
char *p = malloc(e);

p[2] = 42;
```

Obj.	Pointer	Size	Condition
1	0x0001_0000	5	x == 0
2	0x0002_0000	10	x >= 2
3	0x0003_0000	15	x == 1

Chunk	Offset	Length	Value	Cond.	Blob	Live
1	2	1	42	TRUE	store (2, 42)	TRUE

Chunk	Offset	Length	Value	Cond.	Blob	Live
1	2	1	42	TRUE	store (2, 42)	TRUE

Chunk	Offset	Length	Value	Cond.	Blob	Live
1	2	1	42	TRUE	store (2, 42)	TRUE

Solution 3: Object-chunk memory model

```
char *p = malloc(e);
p[2] = 42;
p[2] = 0;
```

Obj.	Pointer	Size	Condition
1	0x0001_0000	5	x == 0
2	0x0002_0000	10	x >= 2
3	0x0003_0000	15	x == 1

Chunk	Offset	Length	Value	Cond.	Blob	Live
1	2	1	42	TRUE	store (2, 42)	FALSE
2	2	1	42	TRUE	store (2, 0)	TRUE

Chunk	Offset	Length	Value	Cond.	Blob	Live
1	2	1	42	TRUE	store (2, 42)	FALSE
2	2	1	42	TRUE	store (2, 0)	TRUE

Chunk	Offset	Length	Value	Cond.	Blob	Live
1	2	1	42	TRUE	store (2, 42)	FALSE
2	2	1	42	TRUE	store (2, 0)	TRUE

Solution 3: Object-chunk memory model

```
char *h = malloc(8);
h[0:7] = 1;

if (x > 100) {
  h[0:5] = 2;
} else if (x < 100) {
  h[2:7] = 3;
} else {
  h[3:4] = 4;
}

if (x >= 100) {
  h[1:6] = 5;
}
```

Chunk	Offset	Length	Value	Cond.	Blob	Live

Solution 3: Object-chunk memory model

```
char *h = malloc(8);
h[0:7] = 1;

if (x > 100) {
  h[0:5] = 2;
} else if (x < 100) {
  h[2:7] = 3;
} else {
  h[3:4] = 4;
}

if (x >= 100) {
  h[1:6] = 5;
}
```

Chunk	Offset	Length	Value	Cond.	Blob	Live
1	0	8	1	TRUE	11111111	TRUE

Solution 3: Object-chunk memory model

```
char *h = malloc(8);
h[0:7] = 1;

if (x > 100) {
  h[0:5] = 2;
} else if (x < 100) {
  h[2:7] = 3;
} else {
  h[3:4] = 4;
}

if (x >= 100) {
  h[1:6] = 5;
}
```

Chunk	Offset	Length	Value	Cond.	Blob	Live
1	0	8	1	TRUE	11111111	x <= 100
2	0	6	2	x > 100	22222211	x > 100

Solution 3: Object-chunk memory model

```
char *h = malloc(8);
h[0:7] = 1;

if (x > 100) {
  h[0:5] = 2;
} else if (x < 100) {
  h[2:7] = 3;
} else {
  h[3:4] = 4;
}

if (x >= 100) {
  h[1:6] = 5;
}
```

Chunk	Offset	Length	Value	Cond.	Blob	Live
1	0	8	1	TRUE	11111111	x == 100
2	0	6	2	x > 100	22222211	x > 100
3	2	6	3	x < 100	11333333	x < 100

Solution 3: Object-chunk memory model

```
char *h = malloc(8);
h[0:7] = 1;

if (x > 100) {
  h[0:5] = 2;
} else if (x < 100) {
  h[2:7] = 3;
} else {
  h[3:4] = 4;
}

if (x >= 100) {
  h[1:6] = 5;
}
```

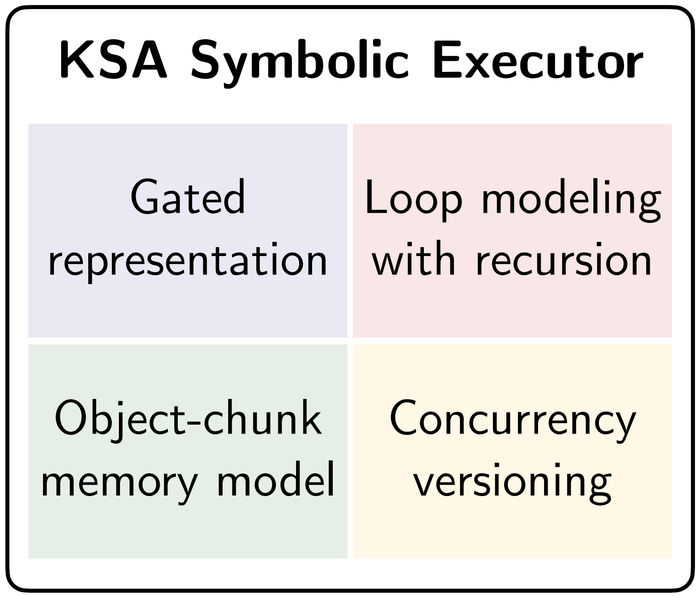
Chunk	Offset	Length	Value	Cond.	Blob	Live
1	0	8	1	TRUE	11111111	FALSE
2	0	6	2	x > 100	22222211	x > 100
3	2	6	3	x < 100	11333333	x < 100
4	3	2	4	x == 100	11144111	x == 100

Solution 3: Object-chunk memory model

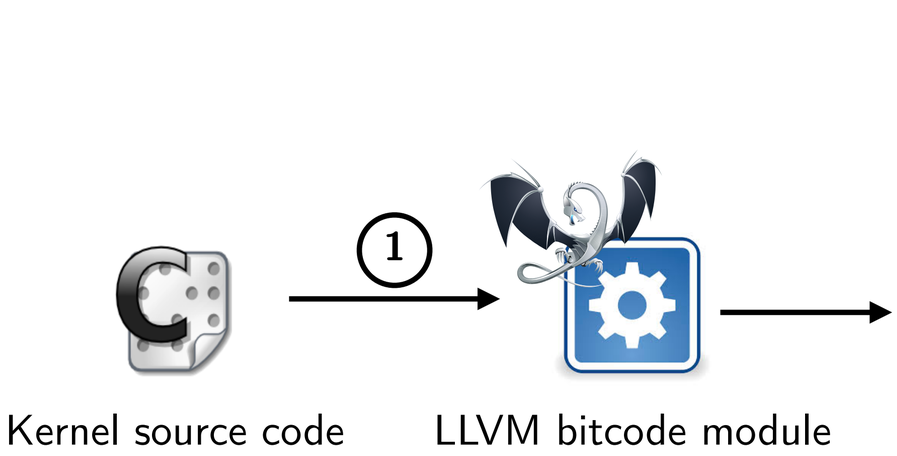
```
char *h = malloc(8);
h[0:7] = 1;

if (x > 100) {
  h[0:5] = 2;
} else if (x < 100) {
  h[2:7] = 3;
} else {
  h[3:4] = 4;
}

if (x >= 100) {
  h[1:6] = 5;
}
```

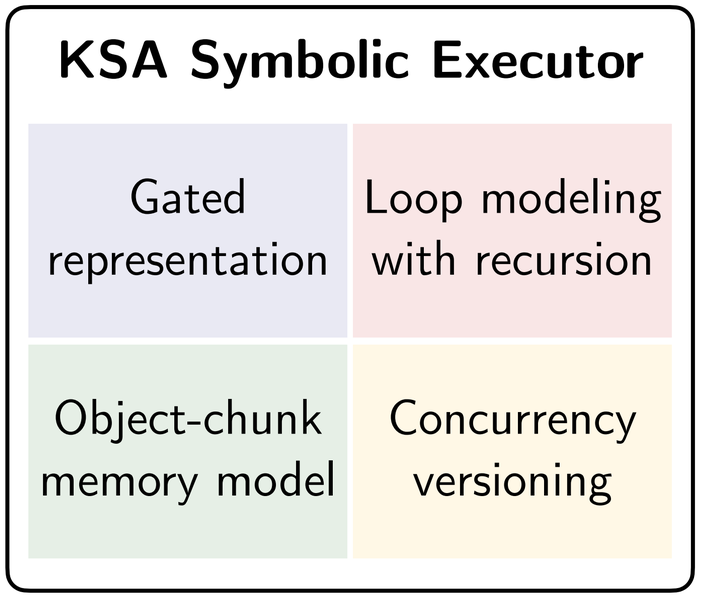
Chunk	Offset	Length	Value	Cond.	Blob	Live
1	0	8	1	TRUE	11111111	FALSE
2	0	6	2	x > 100	22222211	FALSE
3	2	6	3	x < 100	11333333	x < 100
4	3	2	4	x == 100	11144111	FALSE
5	1	6	5	x > 100	25555551	x > 100
6	1	6	5	x == 100	15555551	x == 100

A new design for kernel symbolic execution

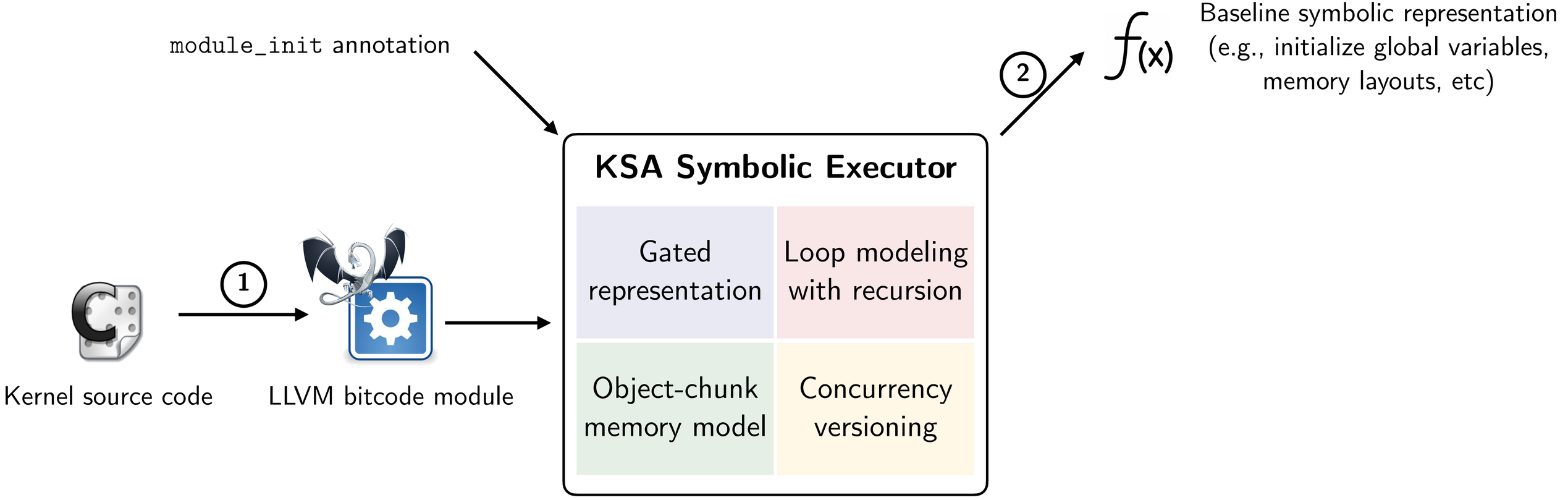


A new design for kernel symbolic execution

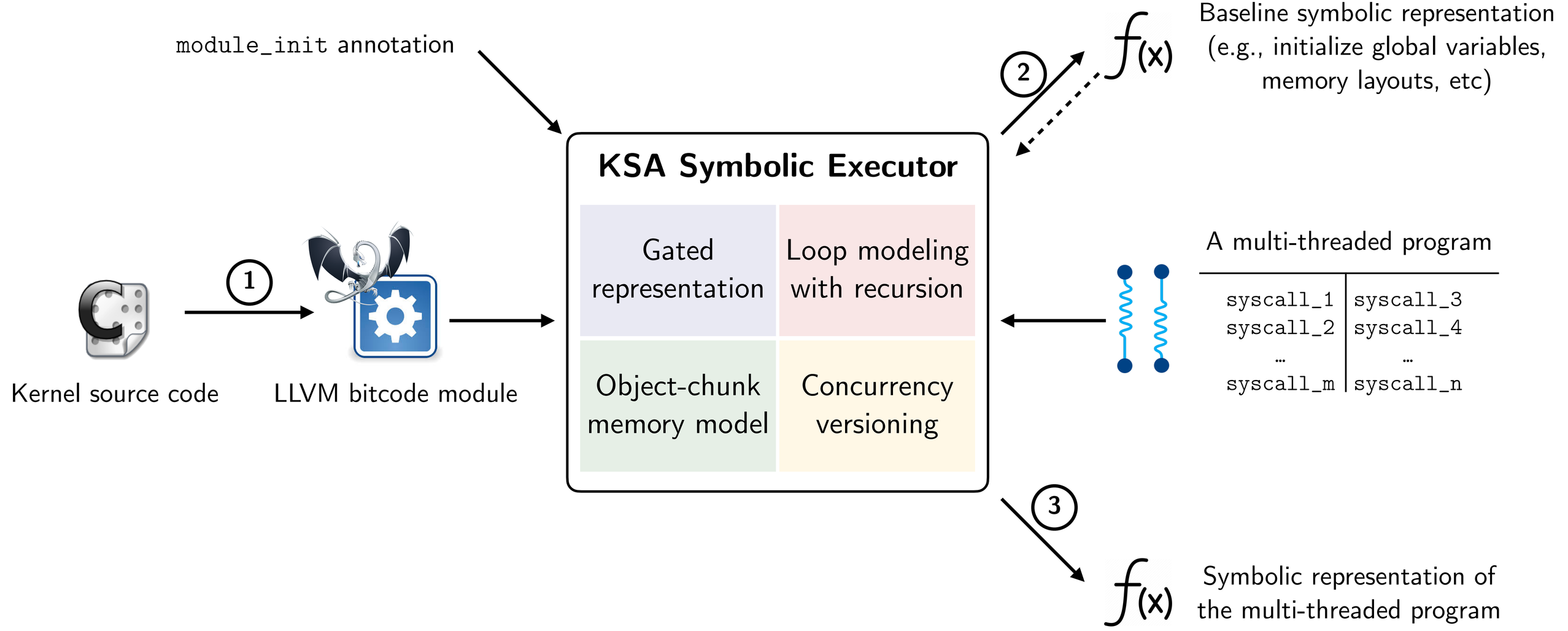




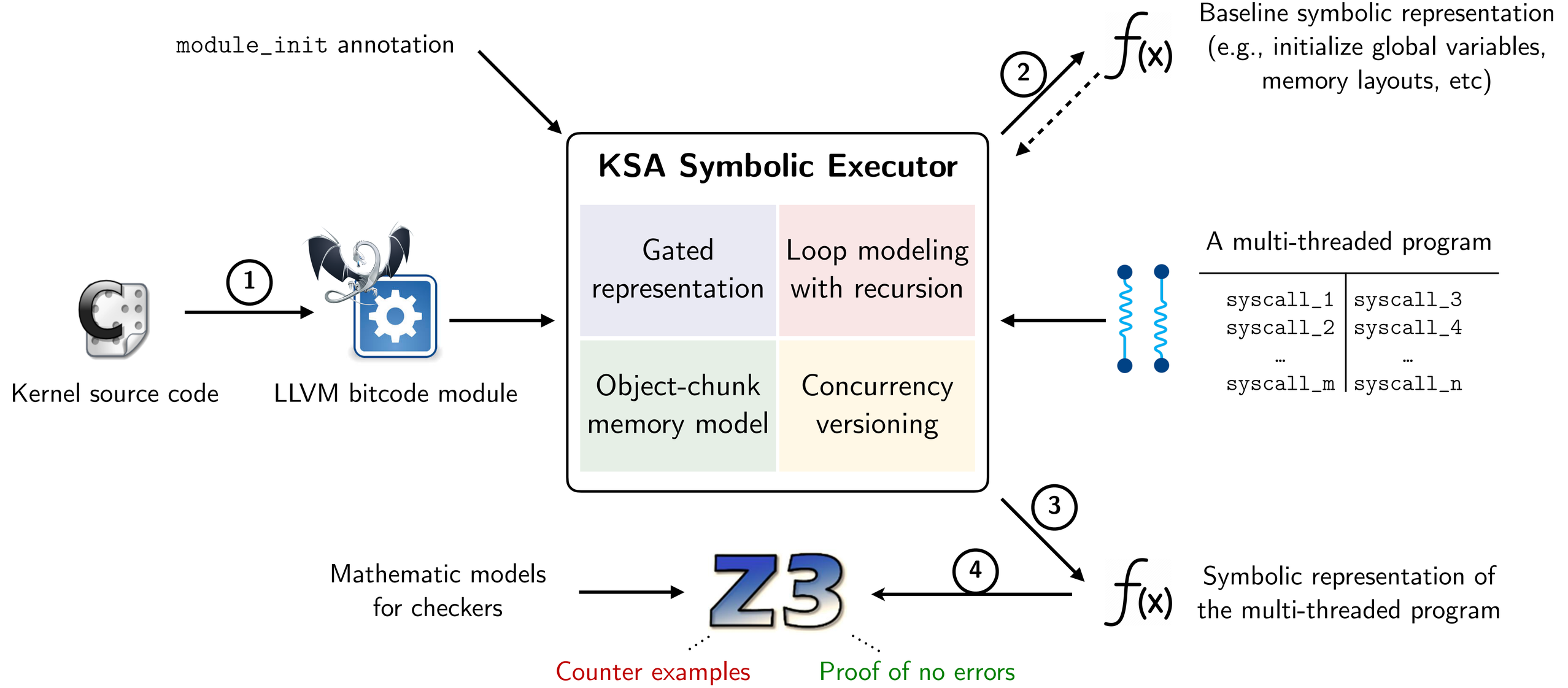
A new design for kernel symbolic execution



A new design for kernel symbolic execution



A new design for kernel symbolic execution



Ongoing research on the KSA framework

- Integration with bug definitions
  - Memory errors
  - Data races
  - Incorrect locking
- Integration with SMT backends
  - Expression simplifier
  - ML-based constraint solver
  - Conditional abstraction

Agenda

1. What are race conditions?

2. Finding their presence with **fuzzing**?
   - [SP'20] Data races in file systems

3. Towards a more **systematic** methodology?
   - [SP'18] Symbolic race checking

4. Up to the extreme of **completeness and soundness**?
   - [WIP (CAV'21)] C to SMT transpilation

Acknowledgement

Taesoo Kim
Wenke Lee
Alessandro Orso
Brendan Saltaformaggio
Marcus Peinado
Michael Backes
Xinyu Xing
Byoungyoung Lee
Chengyu Song
Kangjie Lu
Yeongjin Jang
Sangho Lee
Yang Ji
Changwoo Min
Hanqing Zhao
Jungyeon Yoon
Steffen Maass
Mohan Kumar
Chenxiong Qian
Ruian Duan
Seulbae Kim
Fan Sang
Ren Ding
Wen Xu
Ming-Wei Shih
Insu Yun
Sanidhya Kashap
Daehee Jang
Hong Hu
Paul England
Manuel Huber
Zhichuang Sun

Georgia Tech

CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

Microsoft® Research

facebook

Summary

- **Concept**
  - Alias coverage
  - Formal bug definitions
  - Lossless C to SMT transpilation
- **Impact**
  - 50+ bugs found and reported
  - All tools open-sourced
- **Future work**
  - Working with the SMT community to solve the constraints generated by KSA.
  - Extending the techniques for checking properties on neural networks.
  - Symbolic representation of re-entrant programs (e.g., protocol-ed programs).