Dynamic Symbolic Execution for Evolving Software

Cristian Cadar

SOFTWARE RELIABILITY GROUP

Imperial College London

Research sponsored by

European Research Council
Established by the European Commission

Engineering and Physical Sciences Research Council

University of Edinburgh
10 March 2023

SOFTWARE RELIABILITY GROUP

http://srg.doc.ic.ac.uk

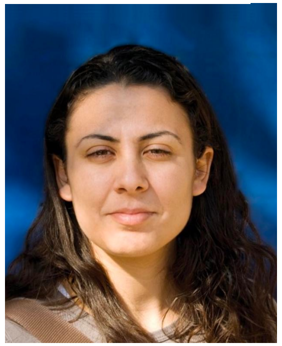
Current and recent members

Cristian Cadar

Anastasios Andronidis

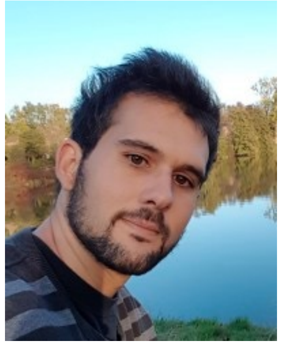
Frank Busse

Manuel Carrasco



Karine Even-Mendoza

Martin Nowack



Jordy Ruiz

Daniel Schemmel

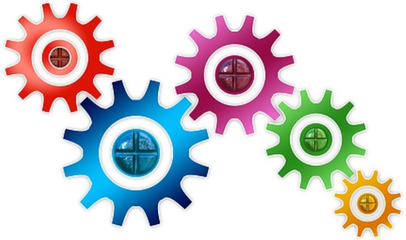
Arindam Sharma

Bachir Bendrissou

Ahmed Zaki

SOFTWARE RELIABILITY GROUP

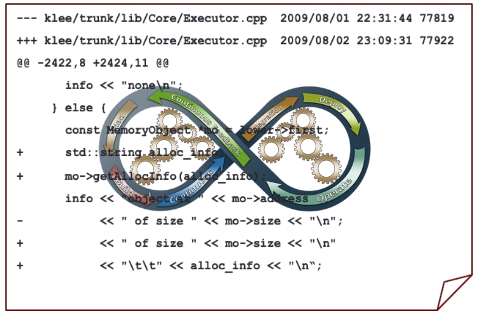
Program analysis techniques for improving the reliability and security of software systems

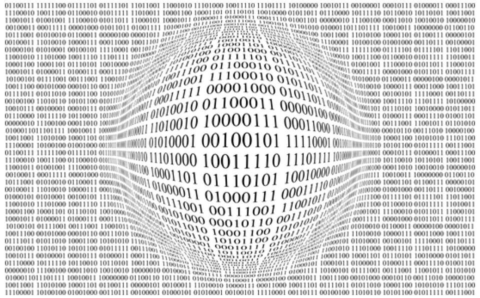


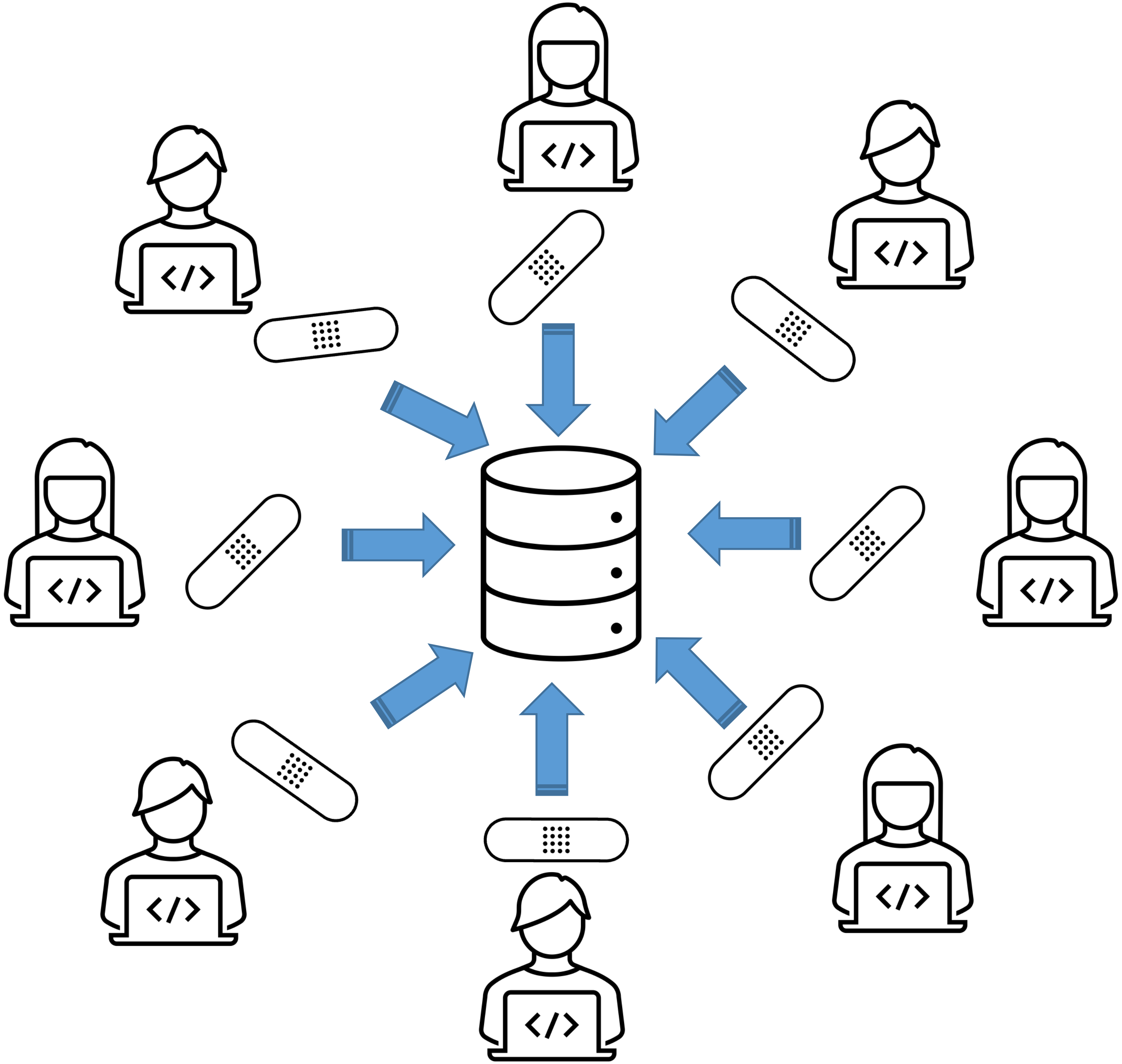
Current and recent projects

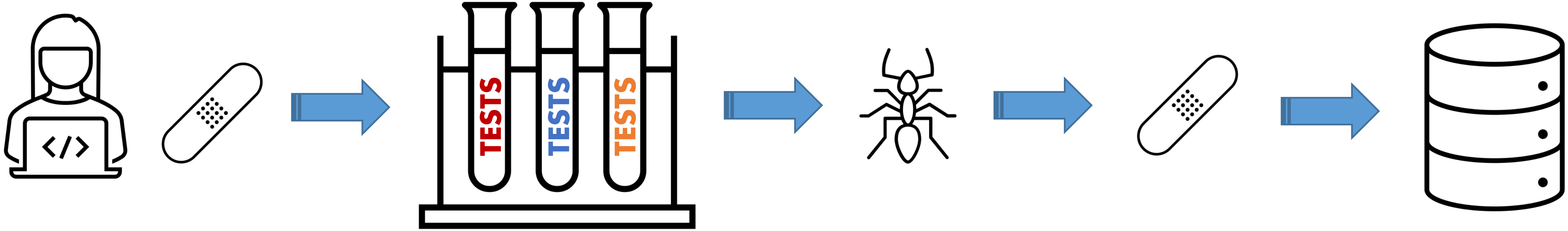
I am hiring! Let me know if you are interested in a PhD or postdoc in the group!

- Program analysis for evolving software
- Understanding, detecting and preventing compiler bugs
- Automatic improvement of program test suites
- Automatic generation of test drivers
- Fuzzing of network protocol implementation
- Selective binary rewriting for fuzzing and debugging
- Multi-variant execution for improving reliability & security
- Code refactoring
- Confirming static analysis reports
- Constraint solving and sampling








</>
TESTS
TESTS
TESTS

Do Developers Like Tests?

YES!

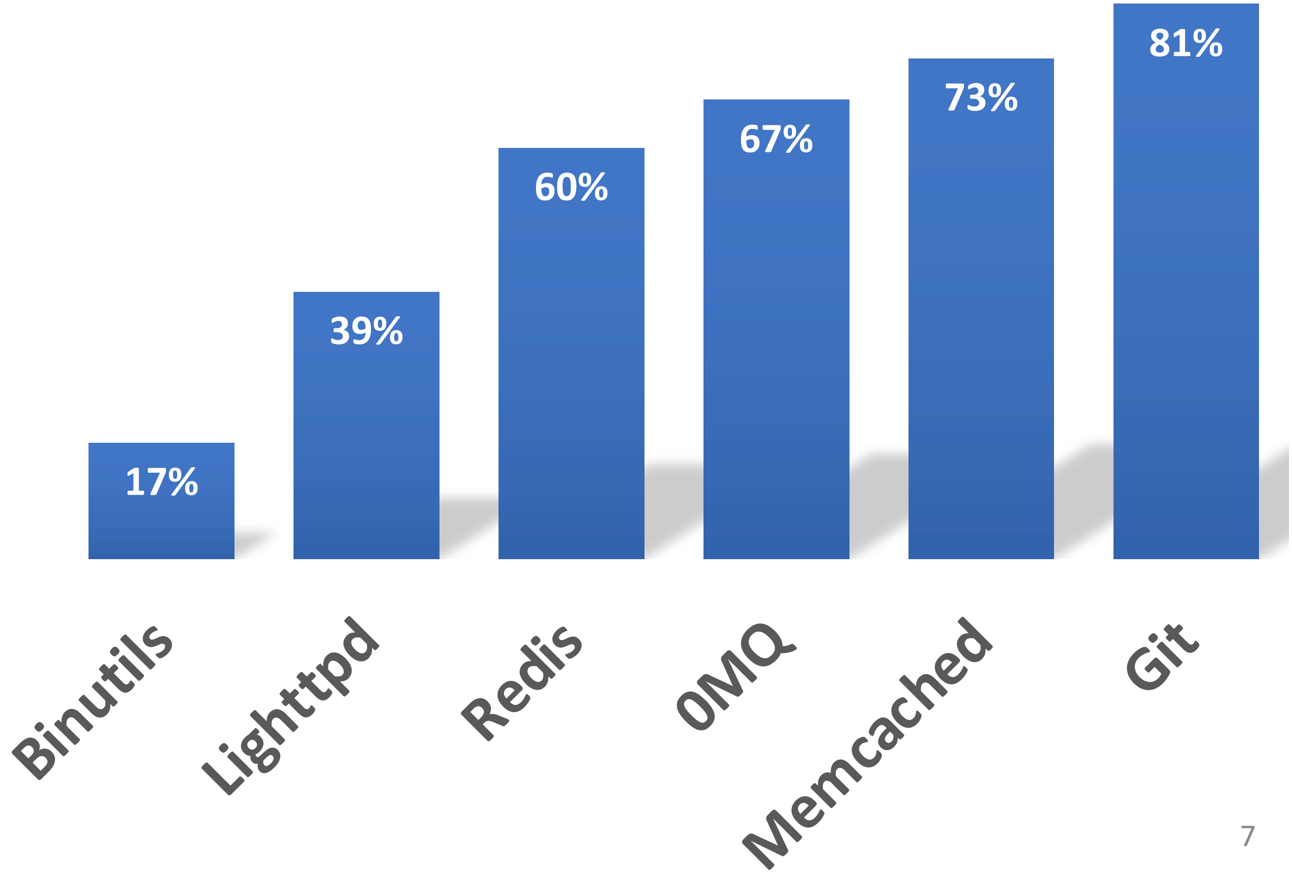
Test cases are valuable as:

- *Quality ensurance*
- *Documentation*
- *Bug Reports*
- *Debugging Aid*

Line Coverage in Several Popular Open-Source Applications

Do Developers Like Tests?

Writing



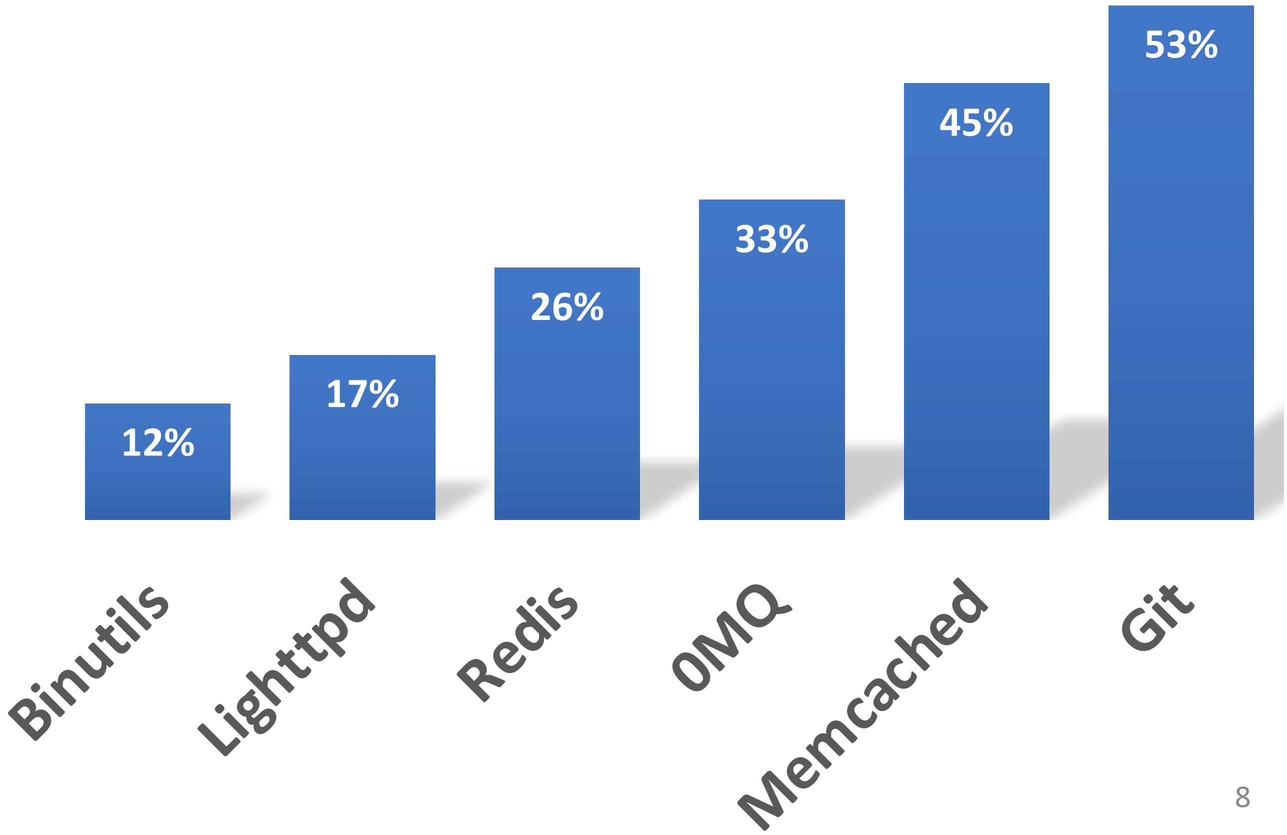
Joint work with Marinescu and Hosek

Fully-Covered Patches in Several Popular Open-Source Applications

12y development time across apps

Do Developers Like Tests?

Writing



Between ≈ 5% and 50% of patches are not covered AT ALL

Joint work with Marinescu and Hosek

Automatic Patch Testing

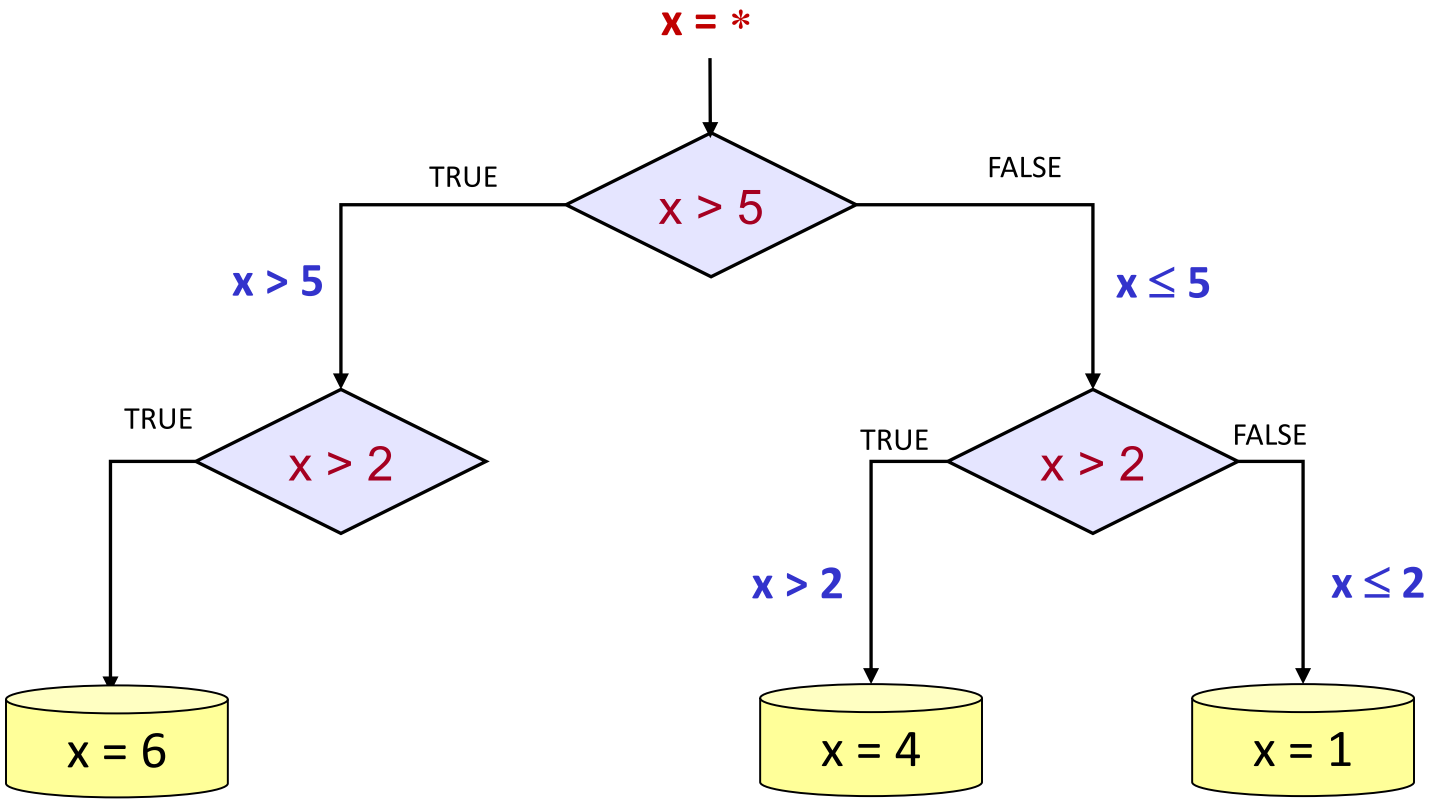
Objective: Generate tests that exercise the patch code, FAST

Approach: Explore program paths using **dynamic symbolic execution**

```
if (x > 5)
   printf(“>5”);

if (x > 2)
   printf(“>2”);
```

Real programs: huge number of paths, huge formulas



KLEE

Webpage: https://klee.github.io/
Code: https://github.com/klee/

Popular symbolic executor primarily developed and maintained at Imperial

Active user and developer base:

- 100+ contributors KLEE and subprojects, 500+ forks, 2000+ stars, 400+ mailing list subscribers

Academic impact:

- ACM SIGOPS Hall of Fame Award and ACM CCS Test of Time Award
- 3.5K+ citations to original KLEE paper (OSDI 2008)
- From many different research communities: testing, verification, systems, software engineering, programming languages, security, etc.

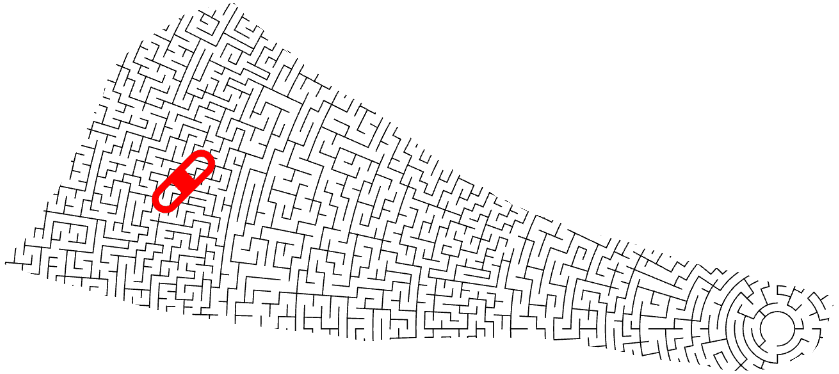
Growing impact in industry:

- **Baidu, Bloomberg, Fujitsu, Google, Huawei, Qualcomm, Samsung, Trail of Bits** as sponsors of KLEE workshops
- **Baidu**: [KLEE-W 2018], **Fujitsu**: [PPoPP 2012], [CAV 2013], [ICST 2015], [IEEE Software 2017], [KLEE-W 2018], **Google**: [2x KLEE-W 2021], **Hitachi**: [CPSNA 2014], [ISPA 2015], [EUC 2016], [KLEE-W 2021], **Intel**: [WOOT 2015], **NASA Ames**: [NFM 2014], **Samsung**: [2x KLEE-W 2018], **Trail of Bits** [https://blog.trailofbits.com/], **etc.**

400+ participants to KLEE Workshops, with good mix of academia and industry

From Whole-Program Analysis …To More Localized Tasks

- Most work on modern symbolic execution on **whole-program** analysis (test generation, bug finding, etc.)

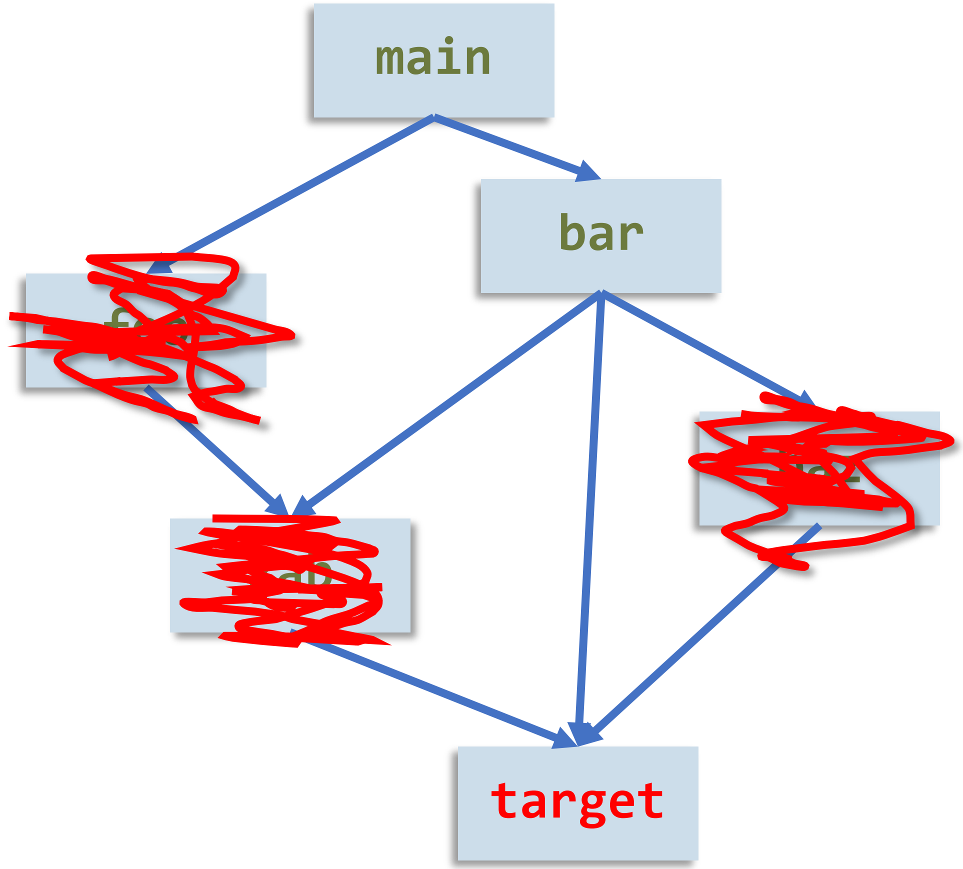


- How does it compare to **patch-targeted** analysis?
- Which one is easier?

Opportunities for patch testing:
1) Reuse the results of the analysis (see MoKLEE [Busse et al, ISSTA'21])
2) Prune the (large) part of the search space unrelated to the patch

Prune Search Space Unrelated to Patch

- Many code fragments are unrelated to the patch
  - But symbolic execution can spend lots of time unnecessarily analyzing them
- Determining precisely if a part of the code is unrelated is hard
  - Often, most computation in a code fragment is unrelated, but not all



Chopped Symbolic Execution

IDEA:

1) Guess unrelated code fragments (manually or via lightweight analysis)
2) Speculatively skip these code fragments
3) If their side effects are ever needed, execute relevant skipped paths only

Joint work with Trabish, Mattavelli, Rinetzky

Chopped Symbolic Execution

Note that in general, we need to use a pointer alias analysis to compute the ref/mod sets.

```
int j; // symbolic
int k; // symbolic
int x = 0;
int y = 0;
```

```
void main() {
  f();
  if (j > 0) {
   if (y)
       target1;
  }
  else target2;
}
```

```
void f() {
  if (k > 0)
    x = 1;
  else
    if (j > 0)
      y = 1;
    else
      y = 0;
}
```

Ref(main) = **{j, y}**

Mod(f) = **{x, y}**

Joint work with Trabish, Mattavelli, Rinetzky

Dependent Loads

```
int j; // symbolic
int k; // symbolic
int x = 0;
int y = 0;
```

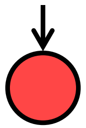
```
void main() {
  f();
  if (j > 0) {
    if (y)
      target1;
  }
  else target2;
}
```

```
void f() {
  if (k > 0)
    x = 1;
  else
    if (j > 0)
      y = 1;
    else
      y = 0;
}
```

Dependent load

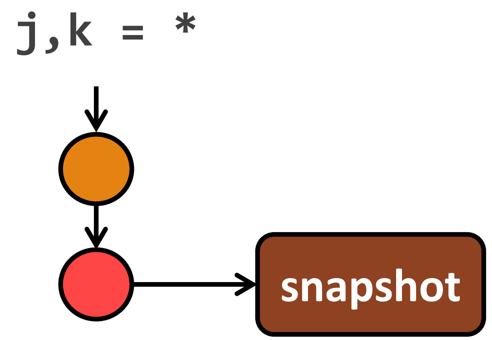
Chopped Symbolic Execution

j,k = *



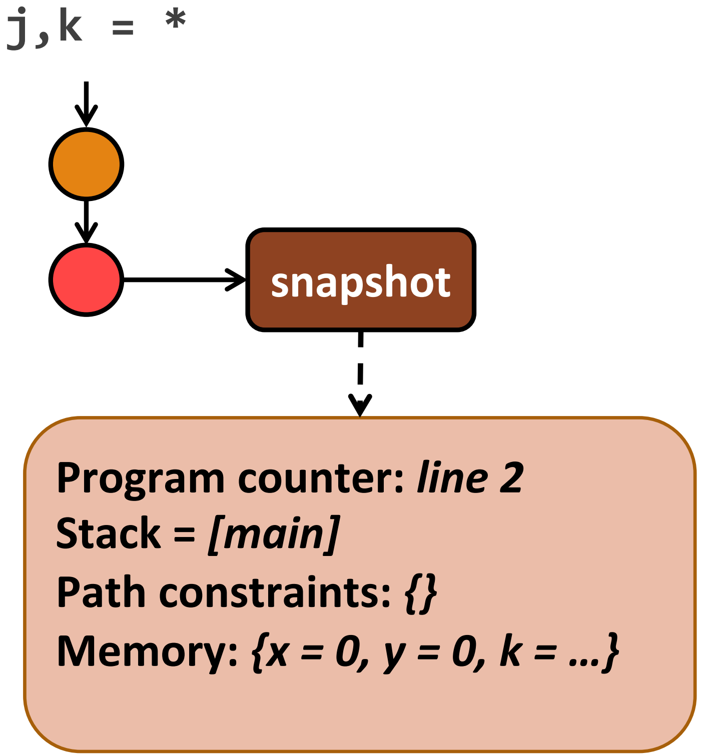
```
void main() {
  f();
  if (j > 0) {
    if (y)
      target1;
  }
  else target2;
}
```

Taking Snapshots



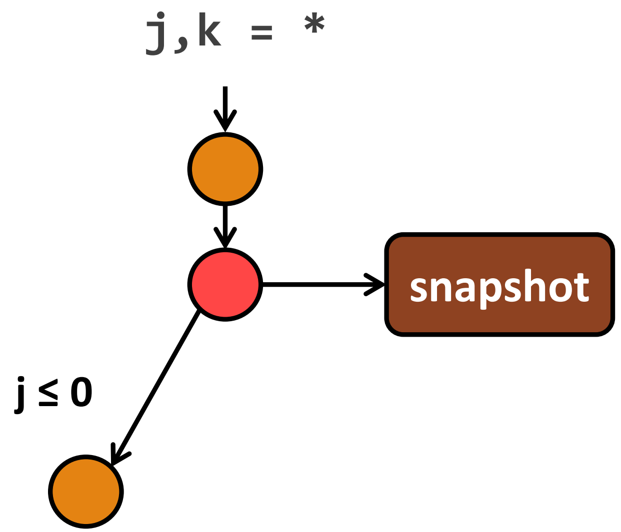

```
void main() {
  f();
  if (j > 0) {
    if (y)
      target1;
  }
  else target2;
}
```

Taking Snapshots



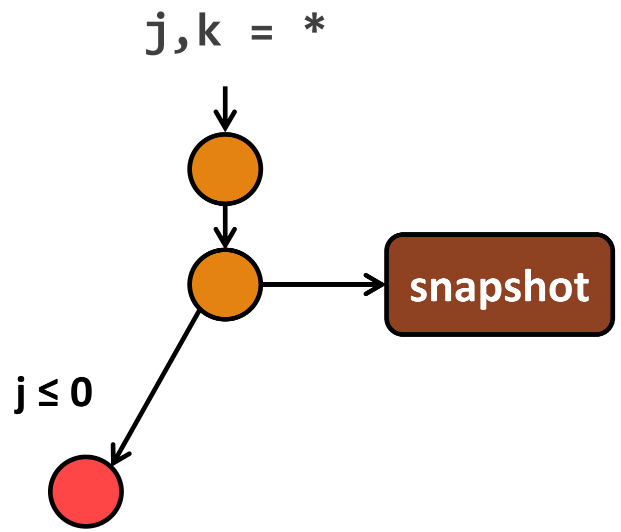

```
void main() {
  f();
  if (j > 0) {
    if (y)
      target1;
  }
  else target2;
}
```

Reaching Target – Ideal Case



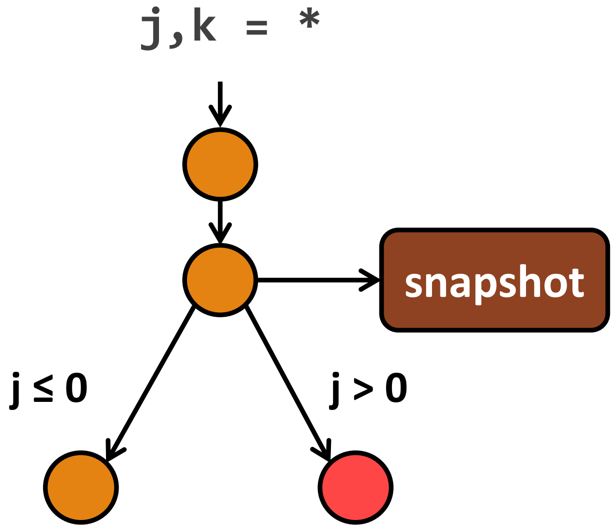

```
void main() {
  f();
  if (j > 0) {
    if (y)
      target1;
  }
  else target2;
}
```

Reaching Target – Ideal Case




```
void main() {
  f();
  if (j > 0) {
    if (y)
      target1;
  }
  else target2;
}
```

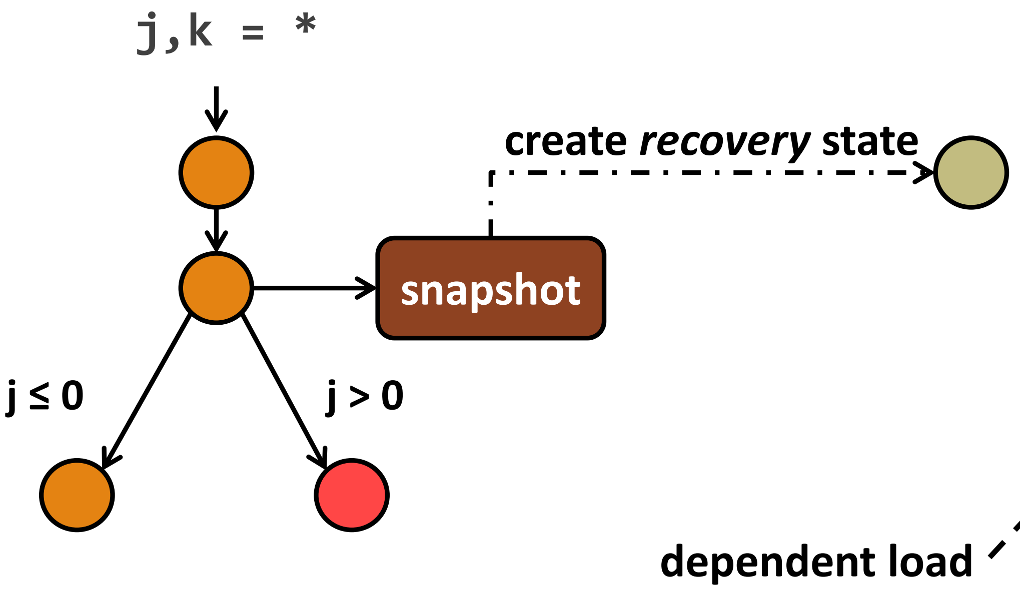
Reaching Target – Recovery Needed



dependent load

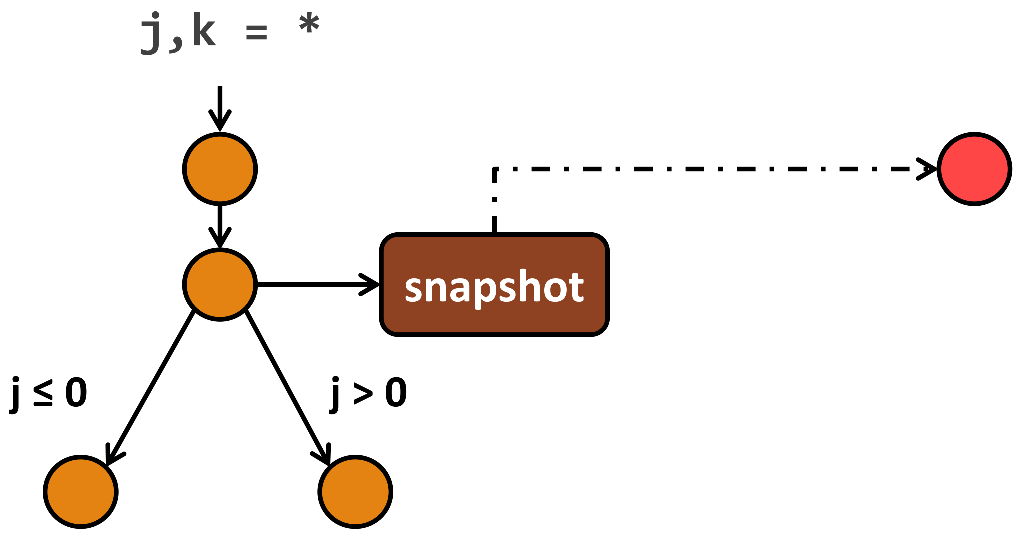
```
void main() {
  f();
  if (j > 0) {
    if (y)
      target1;
  }
  else target2;
}
```

Recovery Process



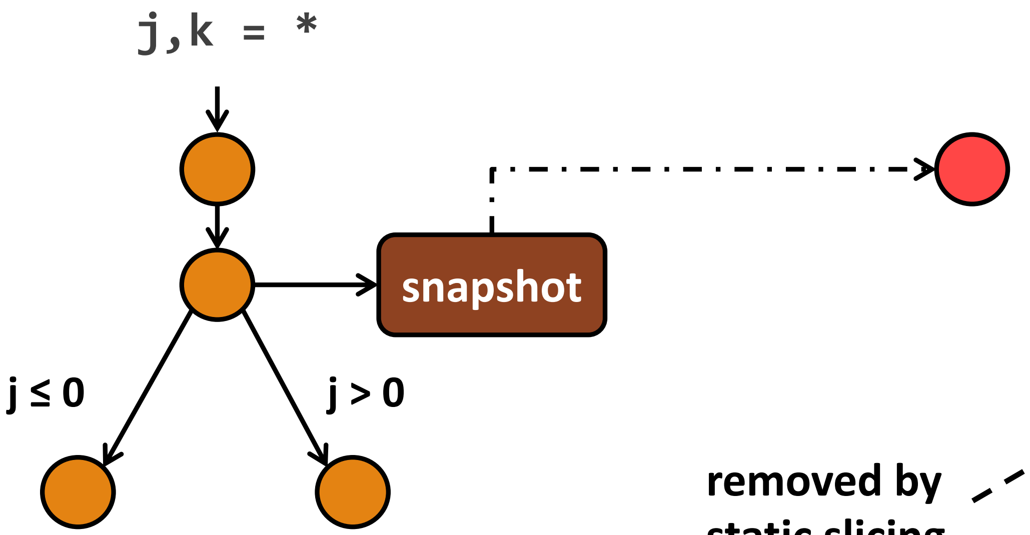

```
void main() {
  f();
  if (j > 0) {
    if (y)
      target1;
  }
  else target2;
}
```

Recovery Process




```
void f() {
  if (k > 0)
    x = 1;
  else
    if (j > 0)
      y = 1;
    else
      y = 0;
}
```

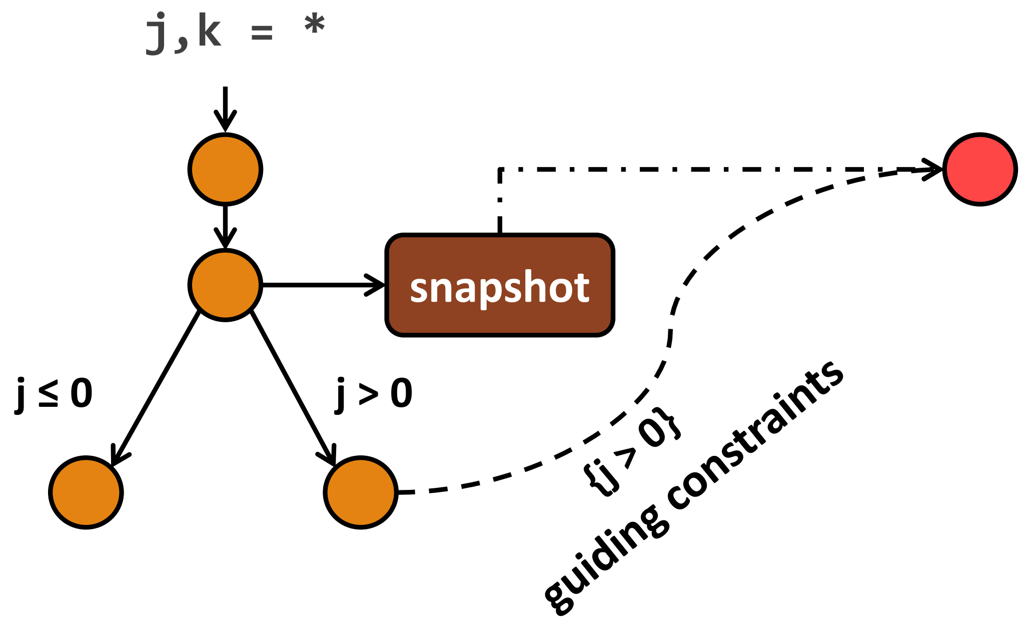
Static Slicing



removed by static slicing

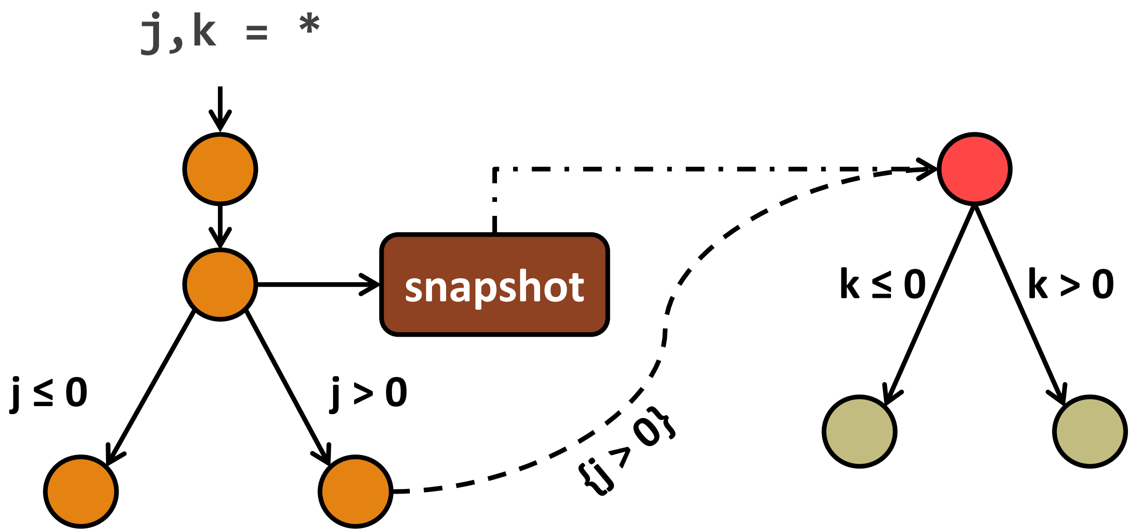
```
void f() {
  if (k > 0)
    // x = 1;
  else
    if (j > 0)
      y = 1;
    else
      y = 0;
}
```

Recovery Process



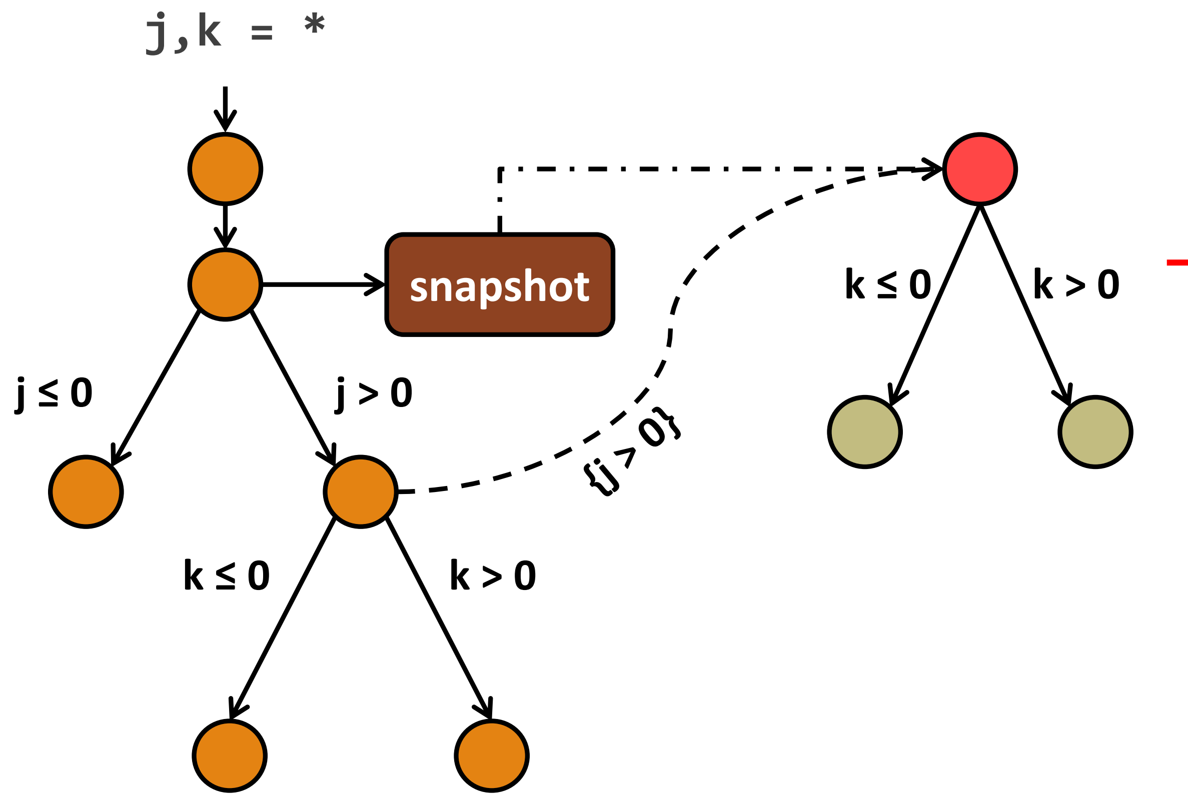

```
void f() {
  if (k > 0)
    // x = 1;
  else
    if (j > 0)
      y = 1;
    else
      y = 0;
}
```

Recovery Process



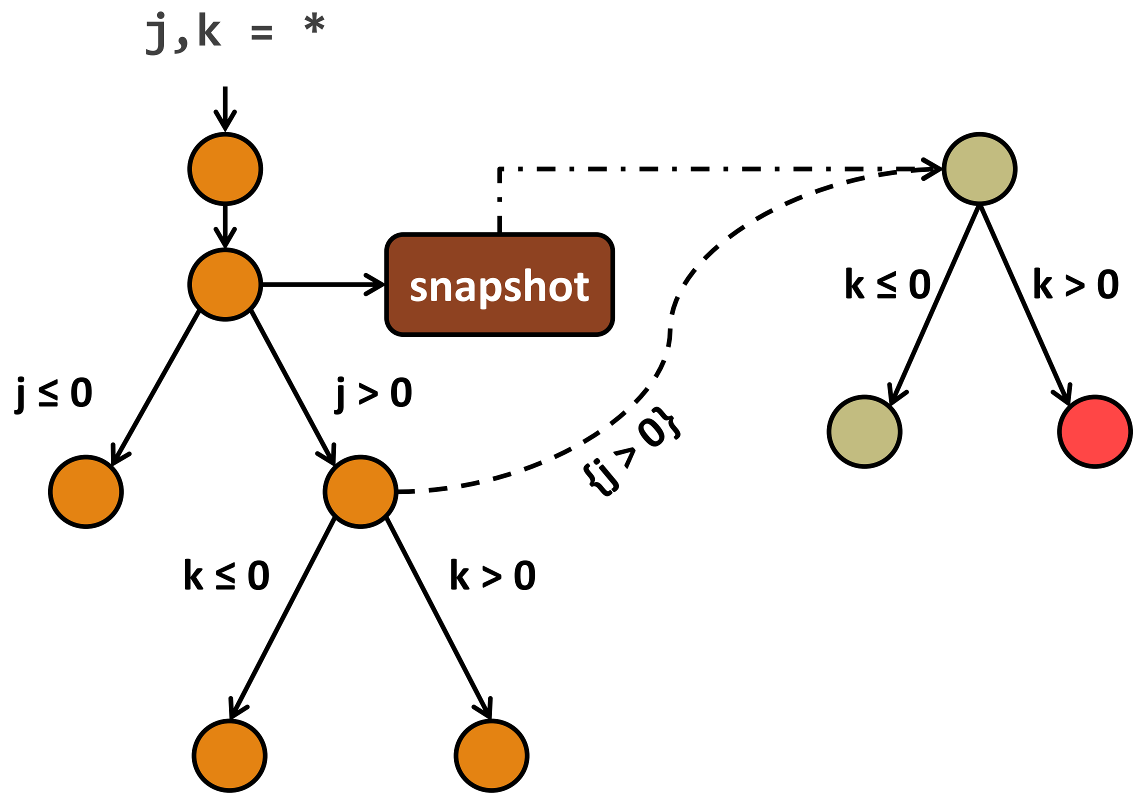

```
void f() {
  if (k > 0)
    // x = 1;
  else
    if (j > 0)
      y = 1;
    else
      y = 0;
}
```

Recovery Process



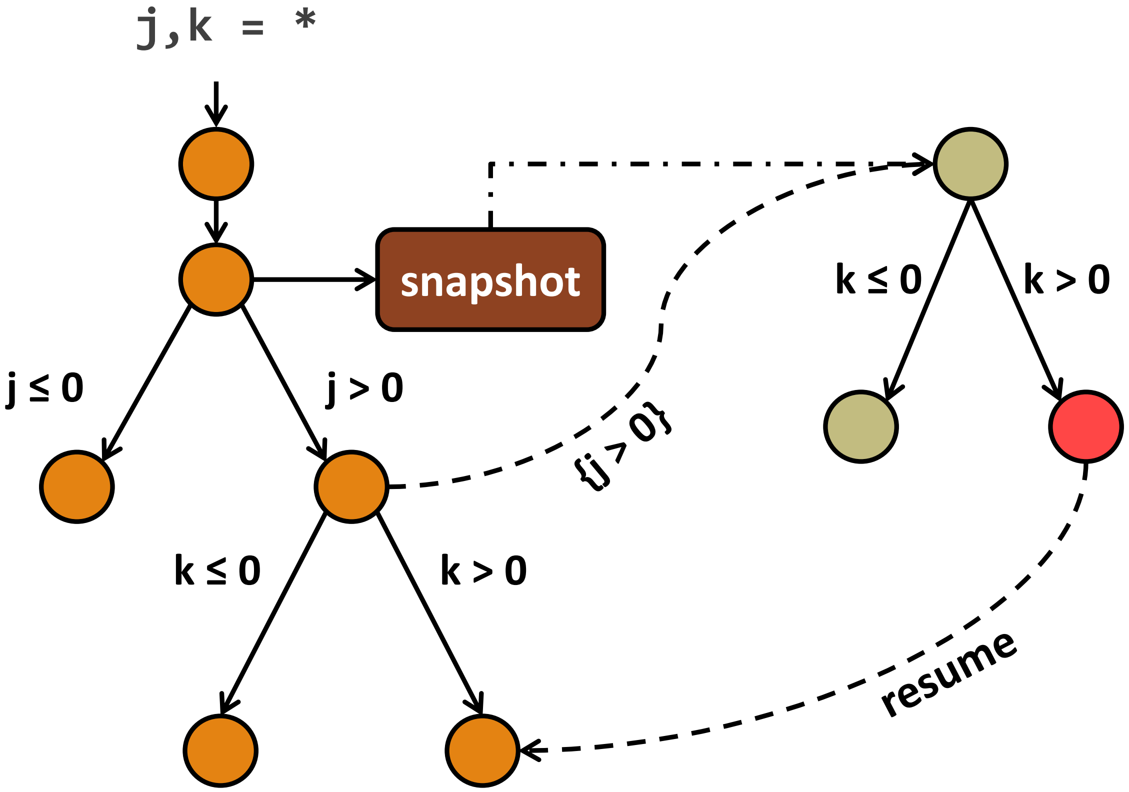

```
void f() {
  if (k > 0)
    // x = 1;
  else
    if (j > 0)
      y = 1;
    else
      y = 0;
}
```

Recovery Process



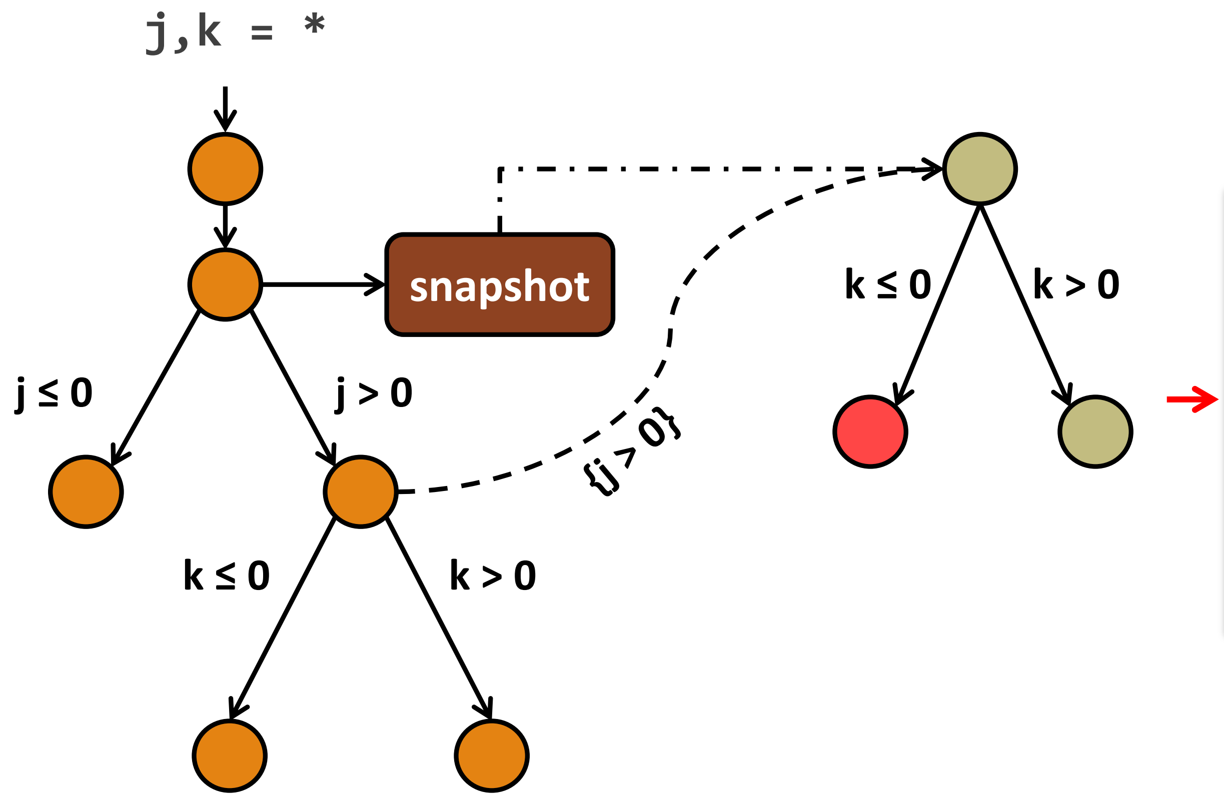

```
void f() {
  if (k > 0)
    // x = 1;
  else
    if (j > 0)
      y = 1;
    else
      y = 0;
}
```

Recovery Process



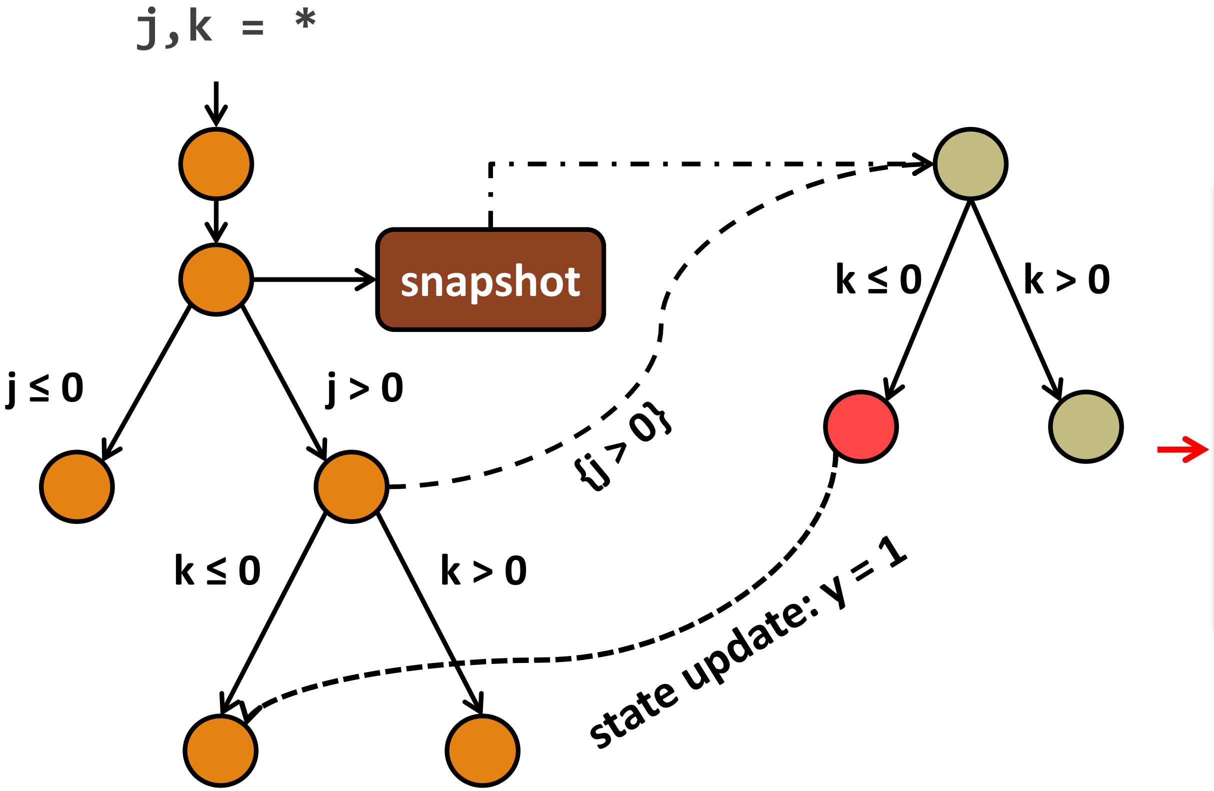

```
void f() {
  if (k > 0)
    // x = 1;
  else
    if (j > 0)
      y = 1;
    else
      y = 0;
}
```

Recovery Process



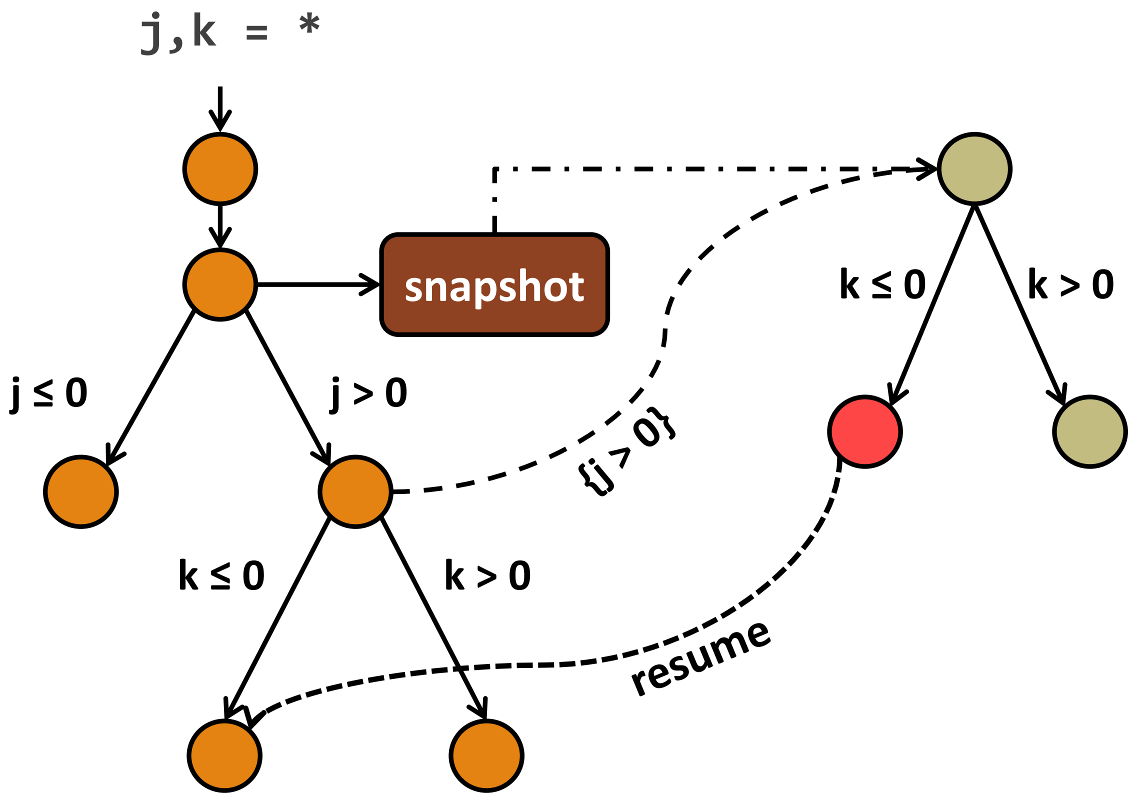

```
void f() {
  if (k > 0)
    // x = 1;
  else
    if (j > 0)
      y = 1;
    else
      y = 0;
}
```

Recovery Process



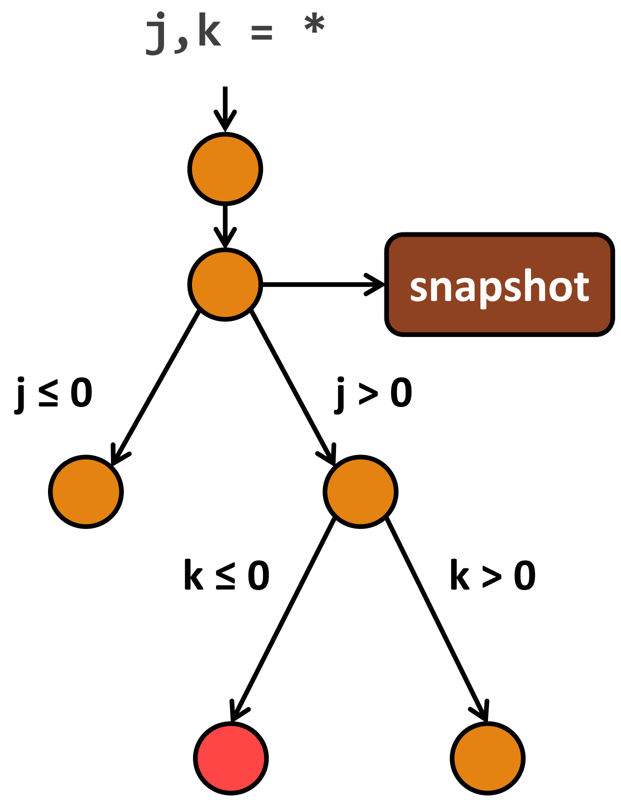

```
void f() {
  if (k > 0)
    // x = 1;
  else
    if (j > 0)
      y = 1;
    else
      y = 0;
}
```

Recovery Process




```
void f() {
  if (k > 0)
    // x = 1;
  else
    if (j > 0)
      y = 1;
    else
      y = 0;
}
```

Recovery Process




```
void main() {
  f();
  if (j > 0) {
    if (y)
      target1;
  }
  else target2;
}
```

Preliminary Experience:

Reproducing Security Vulnerabilities

Benchmark: GNU libtasn1

- ASN.1 protocol used in many networking and cryptographic applications, such as for public key certificates and e-mail
- Considered 4 CVE security vulnerabilities, with a total of 6 vulnerable locations (out-of-bounds accesses)

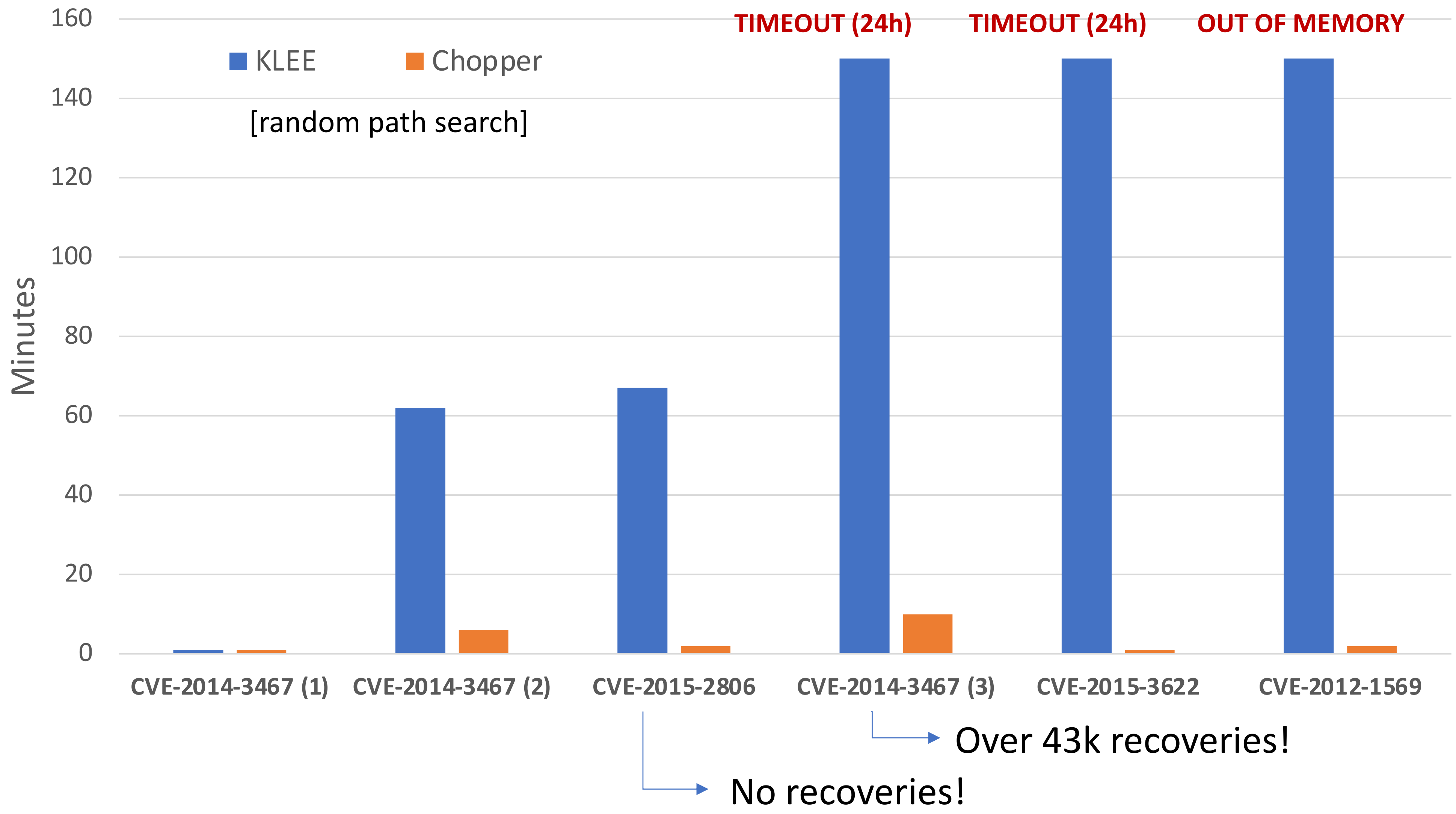
Goal:

- Starting from the CVE report, generate inputs that trigger OOB accesses at the vulnerable locations

Methodology:

- Manually identified the irrelevant functions to skip
- Time limit 24 hours, memory limit 4 GB

Reproducing Security Vulnerabilities



Challenges of Chopped Symbolic Execution

Code to skip [ongoing work with Nowack, Ruiz, Zaki]

- Idea: skip all function calls not on the shortest path to the patch
  - Can always make different guesses and try them in parallel
- Idea: dynamically adjust list of skipped functions
  - E.g., remove those that trigger many recoveries

Precision of pointer analysis

- Initially a single pointer analysis, in the beginning, where we compute all mod/ref sets
- Run pointer analysis on demand, just before skipping a function

Past-Sensitive Pointer Analysis (PSPA)

- Run pointer analysis **on-demand**, not **ahead of time**:
  - From a specific **symbolic state**
- Distinguish between **past** and **future**:
  - Objects that were *already allocated*
    - Allocated objects are associated with unique allocation sites
  - Objects that might be *allocated during pointer analysis*

```
typedef struct { int d, *p; } obj_t;
void foo(obj_t *o) {
  if (o->p)
    o->d = 7;
}
...
obj_t* objs[N];
for (int i = 0; i < N; i++)
  objs[i] = calloc(...);
...
objs[0]->p = malloc(...);
foo(objs[1]);
if (objs[0]->d)
   ...
```

Joint work with Trabish, Kapus, Rinetzky

Are All Inputs the Same?

Consider the patch:

Old

```
if (x % 2 == 0)
    . . .
```

No further uses of x

New

```
if (x % 3 == 0)
    . . .
```

No further uses of x

?

x = 6

?

x = 7

?

x = 8

?

x = 9

Are All Inputs the Same?

Consider the patch:

Old

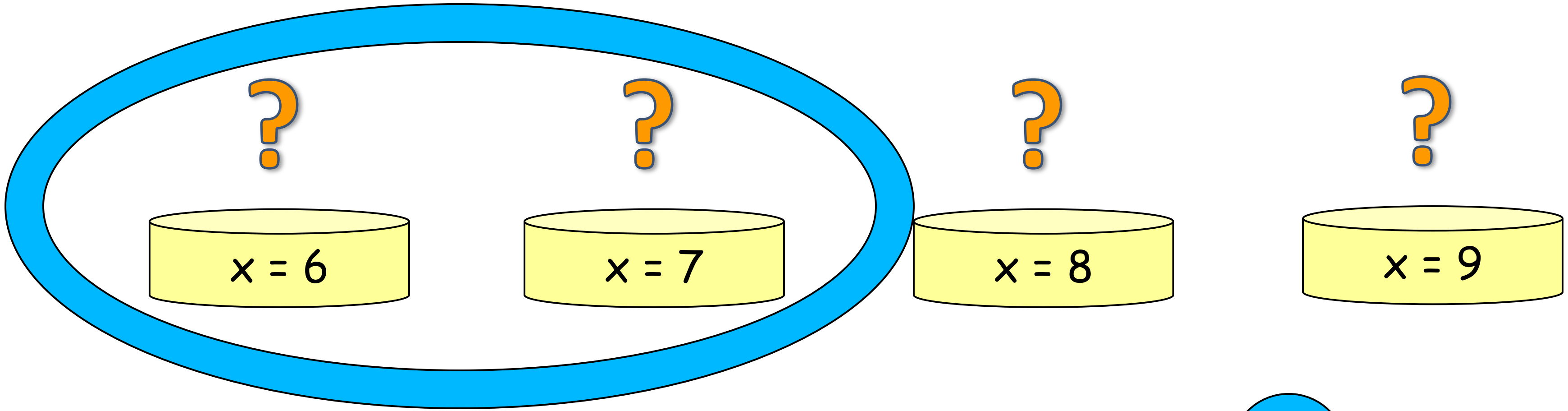
```
if (x % 2 == 0)
   . . .
```

No further uses of x

New

```
if (x % 3 == 0)
   . . .
```

No further uses of x



Full branch coverage in the new version

Are All Inputs the Same?

Consider the patch:

Old

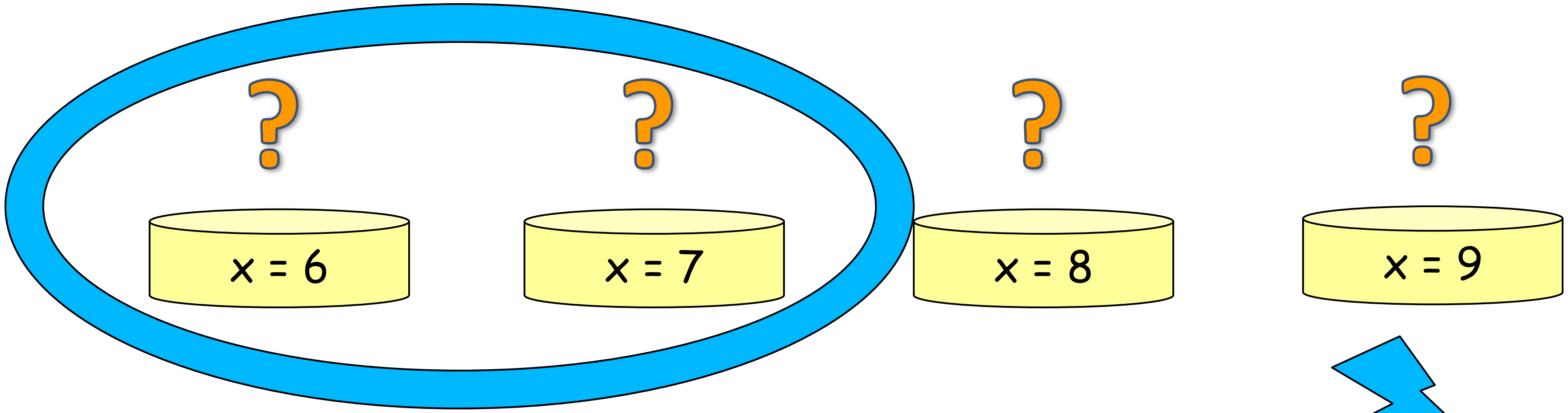
```
if (x % 2 == 0)
   . . .
```

No further uses of x

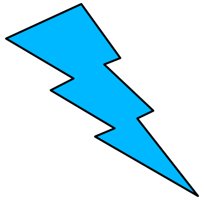
New

```
if (x % 3 == 0)
   . . .
```

No further uses of x



However, totally useless for testing the patch!



Are All Inputs the Same?

Consider the patch:

Old

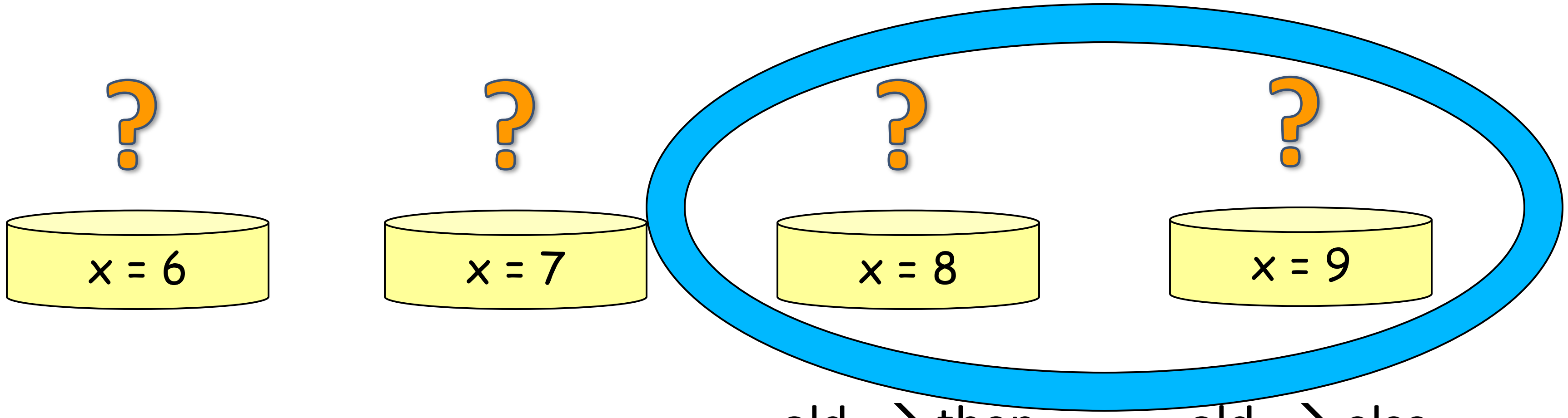
```
if (x % 2 == 0)
   . . .
```

No further uses of x

New

```
if (x % 3 == 0)
   . . .
```

No further uses of x



old → then
new → else

old → else
new → then

Shadow Symbolic Execution

Symbolic Execution on Both Versions Concurrently

Old

```
if (x % 2 == 0)
   . . .
```

New

```
if (x % 3 == 0)
   . . .
```

TRUE

FALSE

$(x \% 2 = 0) \land (x \% 3 \neq 0)$

$(x \% 2 \neq 0) \land (x \% 3 = 0)$

x = 8

x = 9

Joint work with Kuchta and Palikareva

Shadow Symbolic Execution

Automatically generate inputs that trigger different behaviors in the two versions

Run the two versions together, in the same symbolic execution instance:

- Can prune large parts of the search space, for which the two versions behave identically
- Provides the ability to reason about specific values leading to simpler path constraints
- Is memory-efficient by sharing large parts of the symbolic constraints
- Does not execute unchanged computations twice

Case Study: cut

Input	**Old**	**New**
cut –c1-3,8- -output-d=: file (file is “abcdefg”)	abc	abc + *buffer overflow*
cut -c1-7,8- --output-d=: file file contains “abcdefg”	abcdef	abcdef + *buffer overflow*
cut -b0-2,2- --output-d=: file file contains “abc”	abc	signal abort
cut -s -d: -f0- file (file is “:::\n:1”)	:::\n:1	\n\n
cut –d: -f1,0- file (file is “a:b:c”)	a:b:c	a

Test Cases as Documentation!

Beyond Generic Errors

- Symbolic execution can precisely reason about arbitrary properties
  - Paths are modeled as a mathematical constraints
- But specifications are notoriously hard to write
  - Often significantly bigger than the code itself and complicated to write
- What about patch specs?

Patch Specifications

- Specifications, potentially incomplete, encoding cross-patch properties

```
assert(out == out_prev + 1)
```

- We need a way to make the state of both versions available to the analyser

Product Programs

Used to reason about hyperproperties in a security context

- Particularly non-interference
- Product program of program P with itself

1) **Can product programs work for multiple versions of a program?**
2) **Can they be constructed automatically for large programs?**
3) **Can they facilitate the writing of patch specifications?**

Joint work with Sharma and Schemmel

Toy Example

Fn = …. 8, 13, 21, 34, …

```
int Fn;
if (n <= 1)  Fn = ~~1;~~ n
else {
  int Fn_2 = ~~1~~ 0,  Fn_1 = 1;
  Fn = Fn_1 + Fn_2;
  for (int i = 2; i < n; i++) {
    Fn_2 = Fn_1;  Fn_1 = Fn;
    Fn = Fn_1 + Fn_2;
  assert(Fn = Fn_1_prev);
}
```

n	0	1	2	3	4
prev	1	1	2	3	5
curr	0	1	1	2	3

```
int Fn_prev, Fn;
if (n <= 1) { Fn_prev = 1; Fn = n; }
else {
  int Fn_2_prev = 1; int Fn_2 = 0;
  int Fn_1_prev = 1; int Fn_1 = 1;

  Fn_prev = Fn_1_prev + Fn_2_prev;
  Fn = Fn_1 + Fn_2;

  for (int i = 2; i < n; i++) {
    Fn_2_prev = Fn_1_prev; Fn_2 = Fn_1;
    Fn_1_prev = Fn_prev; Fn_1 = Fn;
    Fn_prev = Fn_1_prev + Fn_2_prev;
    Fn = Fn_1 + Fn_2;
  }
  assert(Fn == Fn_1_prev);
}
```

ls

"Do not hard-code '/'. Use IS_ABSOLUTE_FILE_NAME and dir_len instead. Use stpcpy/stpncpy in place of strncpy/strcpy."

Spec violation:

name = /a

linkname = x

```
if (*linkname == '/')
   return xstrdup (linkname);

char const *linkbuf = strrchr (name, '/');

if (linkbuf == NULL)
   return xstrdup (linkname);
size_t bufsiz = linkbuf - name + 1;
char *p = xmalloc (bufsiz + strlen (linkname) + 1);
strncpy (p, name, bufsiz);
strcpy (p + bufsiz, linkname);
return p;
```

```
assert((IS_ABSOLUTE_FILE_NAME (linkname))
          == (*linkname_prev == '/'));

if (IS_ABSOLUTE_FILE_NAME (linkname))
   return xstrdup (linkname);

size_t prefix_len = dir_len (name);
assert((prefix_len == 0) == (linkbuf_prev == NULL));
if (prefix_len == 0)
   return xstrdup (linkname);

char *p = xmalloc (prefix_len + 1 + strlen (linkname) + 1);
stpcpy (stpncpy (p, name, prefix_len + 1), linkname);
assert( strcmp(p, p_prev) == 0 );
return p;
```

ls

"Do not hard-code '/'. Use IS_ABSOLUTE_FILE_NAME and dir_len instead. Use stpcpy/stpncpy in place of strncpy/strcpy."

~~**Spec violation:**~~

~~name = /a~~

~~linkname = x~~

```
if (*linkname == '/')
  return xstrdup (linkname);

char const *linkbuf = strrchr (name, '/');

if (linkbuf == NULL)
  return xstrdup (linkname);
size_t bufsiz = linkbuf - name
char *p = xmalloc (bufsiz + strlen (linkname) + 1);
strncpy (p, name, bufsiz);
strcpy (p + bufsiz, linkname);
return p;
```

Spec violation:

name = /x//y

linkname = a

```
assert((IS_ABSOLUTE_FILE_NAME (linkname))
        == (*linkname_prev == '/'));

if (IS_ABSOLUTE_FILE_NAME (linkname))
  return xstrdup (linkname);

size_t prefix_len = dir_len (name);
assert((prefix_len == 0) == (linkbuf_prev == NULL));
if (prefix_len == 0)
  return xstrdup (linkname);
char *p = xmalloc (prefix_len + 1 + strlen (linkname) + 1);

if ( ! ISSLASH (name[prefix_len - 1]))  ++prefix_len;
stpcpy (stpncpy (p, name, prefix_len), linkname);

assert( strcmp(p, p_prev) == 0 );
return p;
```

ls

"Do not hard-code '/'. Use IS_ABSOLUTE_FILE_NAME and dir_len instead. Use stpcpy/stpncpy in place of strncpy/strcpy."

```
if (*linkname == '/')
   return xstrdup (linkname);

char const *linkbuf = strrchr (name, '/');

if (linkbuf == NULL)
   return xstrdup (linkname);
size_t bufsiz = linkbuf - name
char *p = xmalloc (bufsiz + strlen (linkname) + 1);
strncpy (p, name, bufsiz);
strcpy (p + bufsiz, linkname);
return p;
```

~~Spec violation:~~

~~name = /a~~

~~linkname = x~~

~~Spec violation:~~

~~name = /x//y~~

~~linkname = a~~

```
assert((IS_ABSOLUTE_FILE_NAME (linkname))
        == (*linkname_prev == '/'));

if (IS_ABSOLUTE_FILE_NAME (linkname))
   return xstrdup (linkname);

size_t prefix_len = dir_len (name);
assert((prefix_len == 0) == (linkbuf_prev == NULL));
if (prefix_len == 0)
   return xstrdup (linkname);
char *p = xmalloc (prefix_len + 1 + strlen (linkname) + 1);
if ( ! ISSLASH (name[prefix_len - 1]))  ++prefix_len;
stpcpy (stpncpy (p, name, prefix_len), linkname);
assert( patheq(p, p_prev) == 0 );
return p;
```

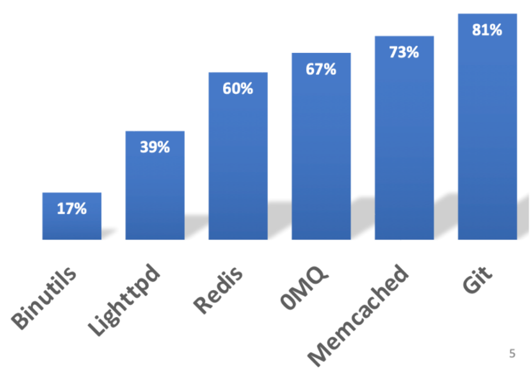
Dynamic Symbolic Execution for Evolving Software

SOFTWARE RELIABILITY GROUP

Imperial College London

Line Coverage in Several Popular Open-Source Applications

Do Developers Like ~~Tests~~? Writing



5

Chopped Symbolic Execution

Note that in general, we need to use a pointer alias analysis to compute the ref/mod sets.

```
int j; // symbolic
int k; // symbolic
int x = 0;
int y = 0;
```

```
void main() {
  f();
  if (j > 0) {
    if (y)
      target1;
  }
  else target2;
}
```

Ref(main) = {j, y}

```
void f() {
  if (k > 0)
    x = 1;
  else
    if (j > 0)
      y = 1;
    else
      y = 0;
}
```

Mod(f) = {x, y}

Shadow Symbolic Execution

Symbolic Execution on Both Versions Concurrently

Old

```
if (x % 2 == 0)
  . . .
```

New

```
if (x % 3 == 0)
  . . .
```

TRUE: $(x \% 2 = 0) \land (x \% 3 \neq 0)$ — x = 8

FALSE: $(x \% 2 \neq 0) \land (x \% 3 = 0)$ — x = 9

41

"Do not hard-code '/'. Use IS ABSOLUTE FILE NAME and dir len instead. Use stpcpy/stpnpcy in place of strncpy/strcpy."

if (*linkname == '/')
return xstrdup (linkname);
	~~**Spec violation:** name = /a linkname = x~~	assert((IS_ABSOLUTE_FILE_NAME (linkname)) == (*linkname_prev == '/'));
		if (IS_ABSOLUTE_FILE_NAME (linkname))
		return xstrdup (linkname);

char const *linkbuf = strrchr (name, '/');
if (linkbuf == NULL)
return xstrdup (linkname);
	~~**Spec violation:** name = /x//y linkname = a~~	size_t prefix_len = dir_len (name);
		assert((prefix_len == 0) == (linkbuf_prev == NULL));
		if (prefix_len == 0)
		return xstrdup (linkname);

size_t bufsiz = linkbuf - name
char *p = xmalloc (bufsiz + strlen (linkname) + 1);
strncpy (p, name, bufsiz);
strcpy (p + bufsiz, linkname);
return p;
		if (! ISSLASH (name[prefix_len - 1])) ++prefix_len;
		stpcpy (stpncpy (p, name, prefix_len), linkname);
		assert(patheq(p, p_prev) == 0);
		return p;