Toward More Scalable Symbolic Execution via Code Chopping

Cristian Cadar

Department of Computing

Imperial College London

SOFTWARE RELIABILITY GROUP

Joint work with

David Trabish and Noam Rinetzky (Tel Aviv University)

Timotej Kapus and Andrea Mattavelli (Imperial College London)

Generously sponsored by

UKRI Engineering and Physical Sciences Research Council

erc European Research Council Established by the European Commission

TAPAS Workshop
19 November 2020

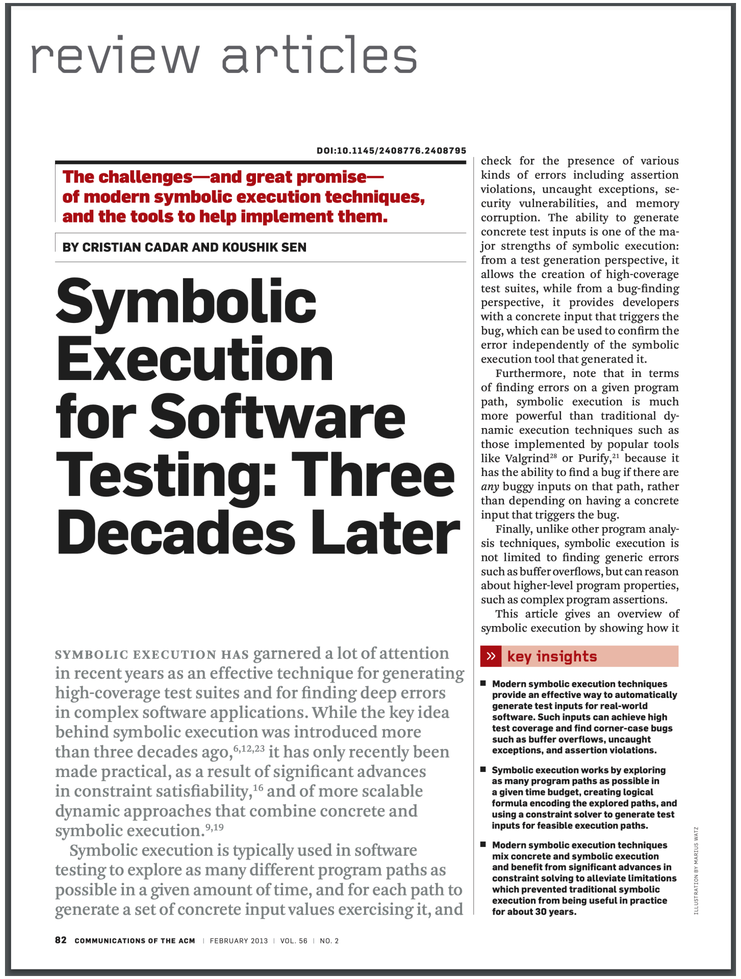
Symbolic Execution
or Dynamic Symbolic Execution (DSE)

Program analysis technique for ***automatically exploring paths*** through a program

Applications in:

- Bug finding
- Test generation
- Vulnerability detection and exploitation
- Equivalence checking
- Debugging
- Program repair
- etc. etc.

Modern Symbolic Execution


review articles

DOI:10.1145/2408776.2408795

The challenges—and great promise—of modern symbolic execution techniques, and the tools to help implement them.

BY CRISTIAN CADAR AND KOUSHIK SEN

Symbolic Execution for Software Testing: Three Decades Later

SYMBOLIC EXECUTION HAS garnered a lot of attention in recent years as an effective technique for generating high-coverage test suites and for finding deep errors in complex software applications. While the key idea behind symbolic execution was introduced more than three decades ago,[6,12,23] it has only recently been made practical, as a result of significant advances in constraint satisfiability,[16] and of more scalable dynamic approaches that combine concrete and symbolic execution.[9,19]

Symbolic execution is typically used in software testing to explore as many different program paths as possible in a given amount of time, and for each path to generate a set of concrete input values exercising it, and check for the presence of various kinds of errors including assertion violations, uncaught exceptions, security vulnerabilities, and memory corruption. The ability to generate concrete test inputs is one of the major strengths of symbolic execution: from a test generation perspective, it allows the creation of high-coverage test suites, while from a bug-finding perspective, it provides developers with a concrete input that triggers the bug, which can be used to confirm the error independently of the symbolic execution tool that generated it.

Furthermore, note that in terms of finding errors on a given program path, symbolic execution is much more powerful than traditional dynamic execution techniques such as those implemented by popular tools like Valgrind[28] or Purify,[21] because it has the ability to find a bug if there are *any* buggy inputs on that path, rather than depending on having a concrete input that triggers the bug.

Finally, unlike other program analysis techniques, symbolic execution is not limited to finding generic errors such as buffer overflows, but can reason about higher-level program properties, such as complex program assertions.

This article gives an overview of symbolic execution by showing how it

» key insights

- **Modern symbolic execution techniques provide an effective way to automatically generate test inputs for real-world software. Such inputs can achieve high test coverage and find corner-case bugs such as buffer overflows, uncaught exceptions, and assertion violations.**
- **Symbolic execution works by exploring as many program paths as possible in a given time budget, creating logical formula encoding the explored paths, and using a constraint solver to generate test inputs for feasible execution paths.**
- **Modern symbolic execution techniques mix concrete and symbolic execution and benefit from significant advances in constraint solving to alleviate limitations which prevented traditional symbolic execution from being useful in practice for about 30 years.**

ILLUSTRATION BY MARIUS WATZ

82 COMMUNICATIONS OF THE ACM | FEBRUARY 2013 | VOL. 56 | NO. 2

[Cadar and Sen, CACM 2013]

Symbolic execution introduced in 1970s
- *Boyer, Elspass, Levitt (SRI)*
- *Clarke (UMass Amherst)*
- *King (IBM Research)*

Revived in 2005 in the form of *dynamic symbolic execution**
- *DART system (Bell Labs)*
- *EGT system (Stanford)*

**aka concolic execution, whitebox fuzzing, etc.*

Dynamic Symbolic Execution

PyExZ3

SymDroid

KLOVER

jCUTE

SAGE

The KeY Project

Otter

PathGrind

SymJS

BinSE

CREST

Miasm

Jalangi2

Symbolic PathFinder

CUTE

angr

KLEE

DART

Kite

LDSE

Pex

Rubyx

JDart

S2E

CATG

CiVL

Mayhem

KLEE

Webpage: **https://klee.github.io/**

Code: **https://github.com/klee/**

Active user and developer base with over 300 subscribers on the mailing list and over 70 contributors listed on GitHub

Academic impact:

- SIGOPS Hall of Fame Award (KLEE paper) and ACM CCS Test of Time Award (EXE paper)
- Around 3K citations to original KLEE paper (OSDI 2008)
- From many different research communities: testing, verification, systems, software engineering, programming languages, security, etc.
- Many different systems using KLEE: AEG, Angelix , BugRedux , Cloud9, GKLEE, KleeNet, KLEE-UC, S2E, SemFix, etc.

Growing impact in industry:

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

From Whole-Program Analysis…
…To More Localized Tasks

Most work on modern symbolic execution:

- Whole-program test generation
- Whole-program bug-finding

More recently attention shifted to more localized tasks:

- Patch testing
- Debugging
- Bug reproduction
- Program repair
- etc.

Opportunity of more localized tasks:

Prune a large part of the search space

Which one is easier?

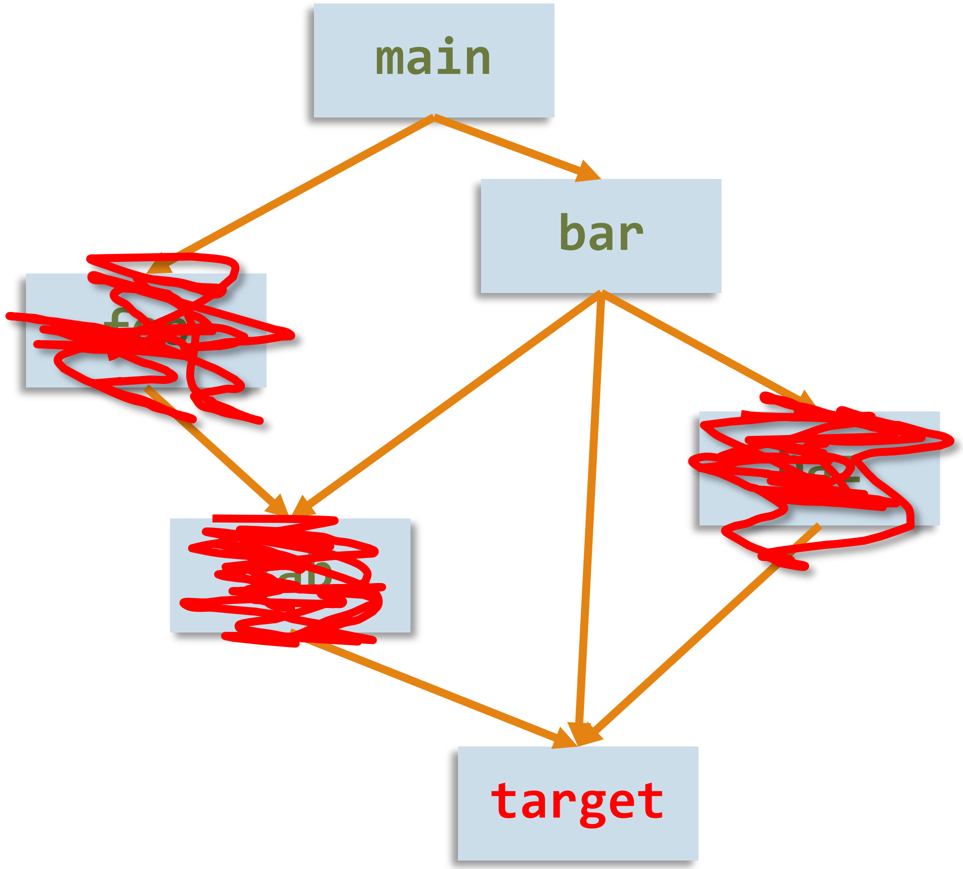
Chopped Symbolic Execution

Some code fragments are unrelated to certain tasks

- 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

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}**

Dependent Loads

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

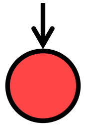
```
void main() {
  f();
  if (j > 0) {
    if (y)
      target1;
  }
  else target2;
}
```

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

Dependent load

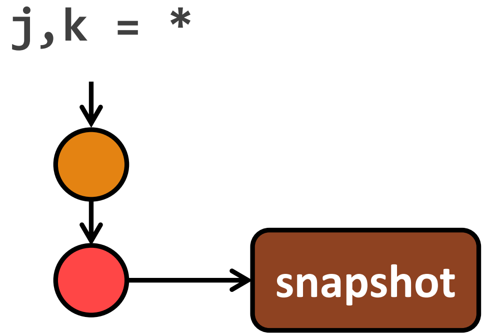
Chopped Symbolic Execution

j,k = *



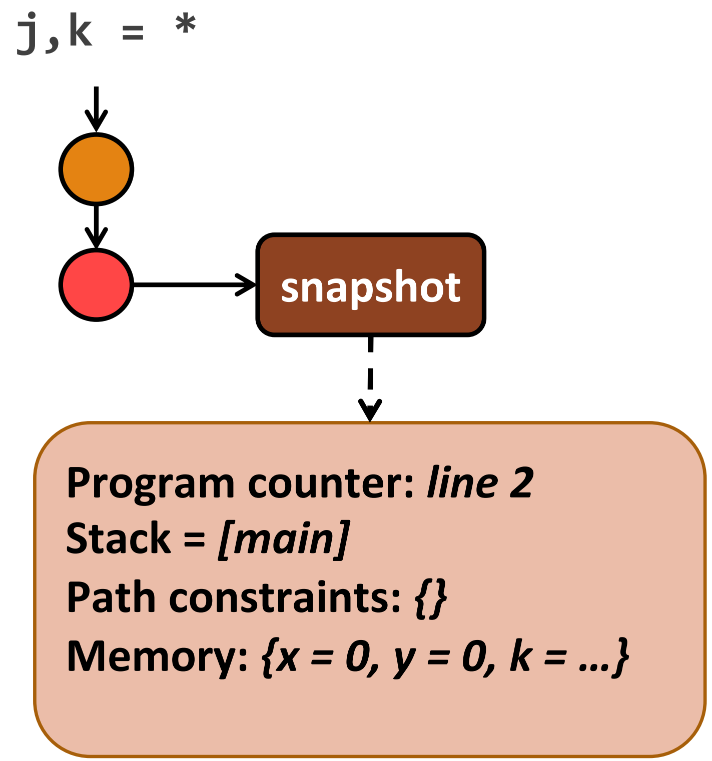
```
void main() {
  f();
  if (j > 0) {
    if (y)
      target1;
  }
  else target2;
}
```

Taking Snapshots



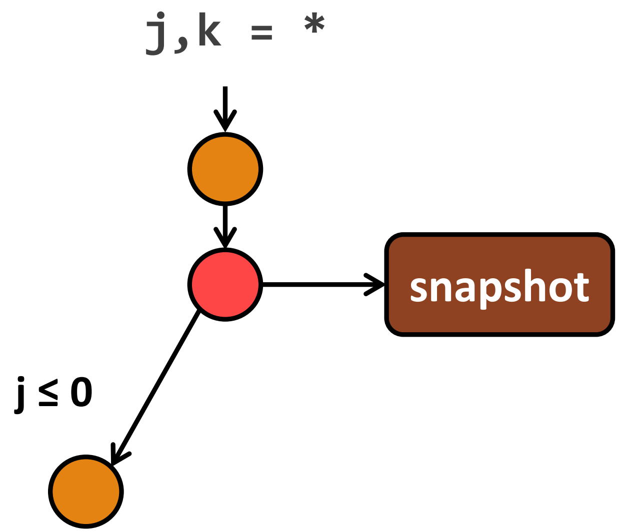

```
void main() {
  f();
  if (j > 0) {
    if (y)
      target1;
  }
  else target2;
}
```

Taking Snapshots



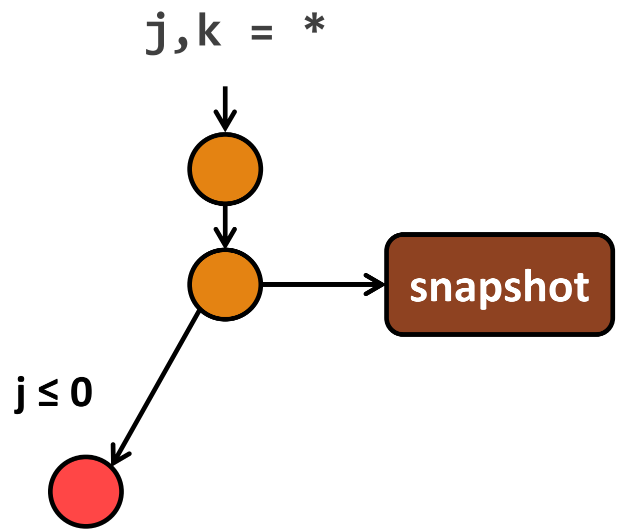

```
void main() {
  f();
  if (j > 0) {
    if (y)
      target1;
  }
  else target2;
}
```

Reaching Target – Ideal Case



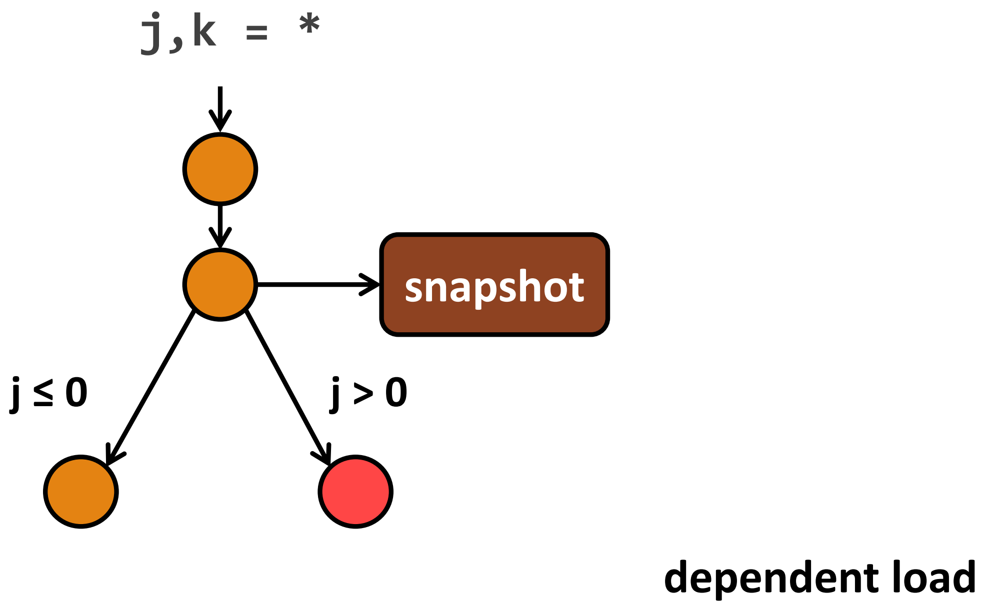

```
void main() {
  f();
  if (j > 0) {
    if (y)
      target1;
  }
  else target2;
}
```

Reaching Target – Ideal Case



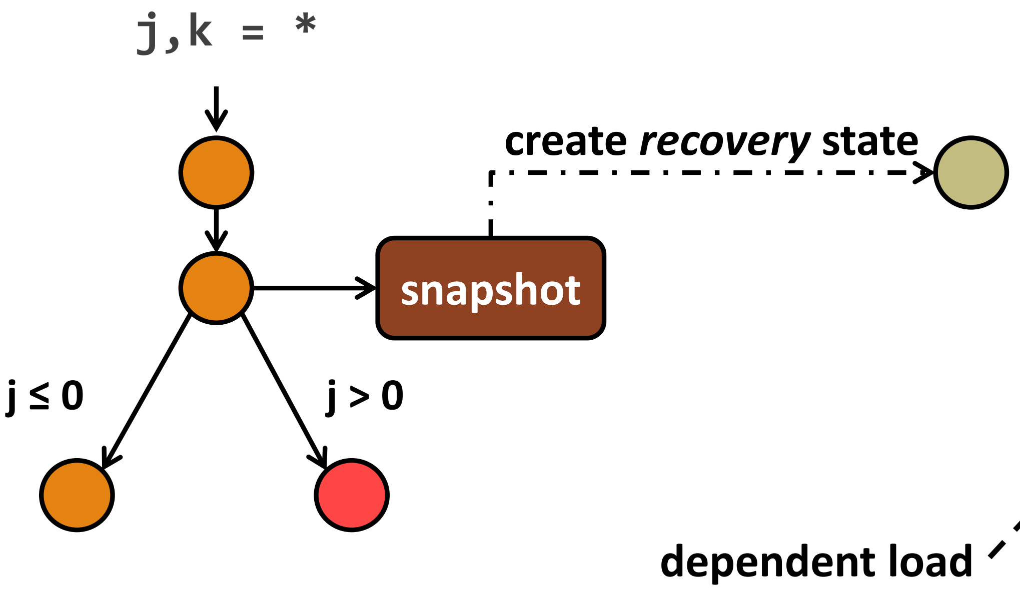

```
void main() {
  f();
  if (j > 0) {
    if (y)
      target1;
  }
  else target2;
}
```

Reaching Target – Recovery Needed



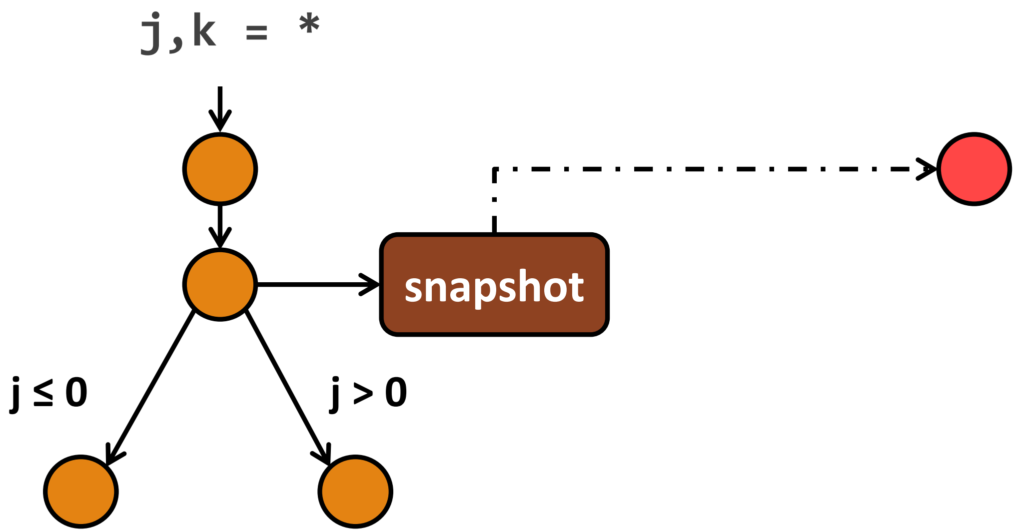

```
void main() {
  f();
  if (j > 0) {
    if (y)
      target1;
  }
  else target2;
}
```

Recovery Process



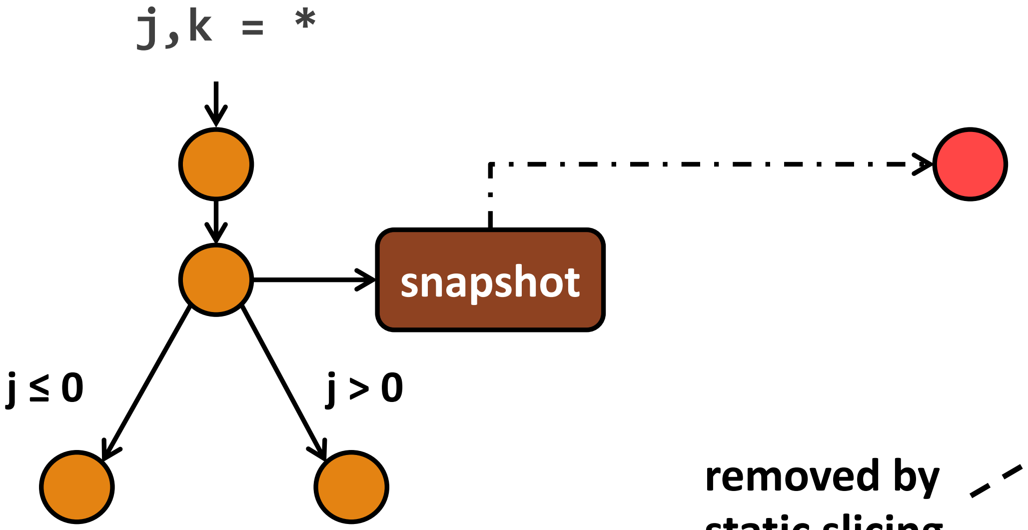

```
void main() {
  f();
  if (j > 0) {
    if (y)
      target1;
  }
  else target2;
}
```

Recovery Process




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

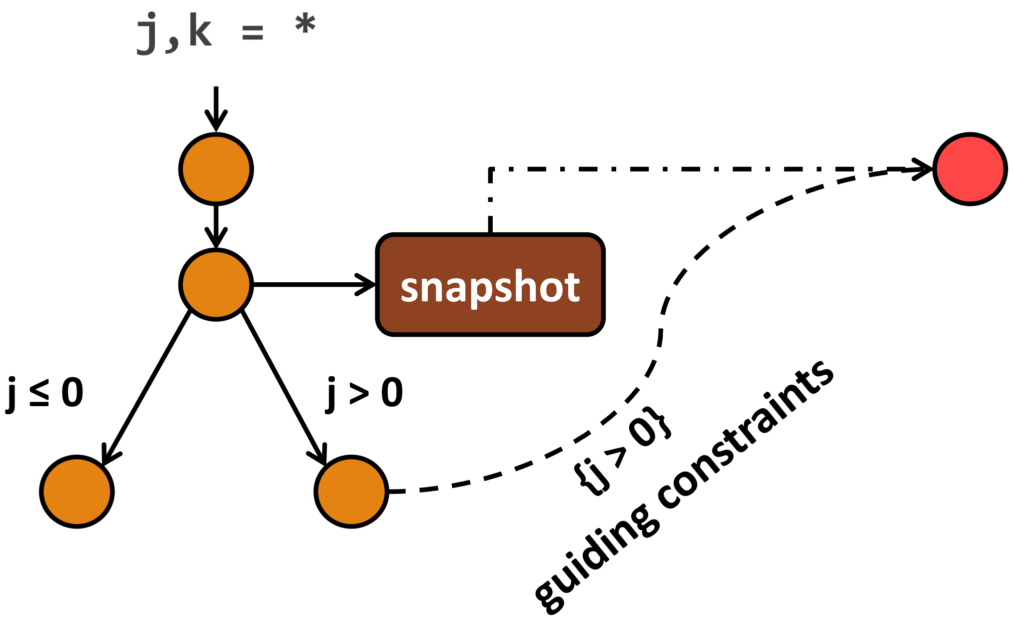
Static Slicing



removed by
static slicing

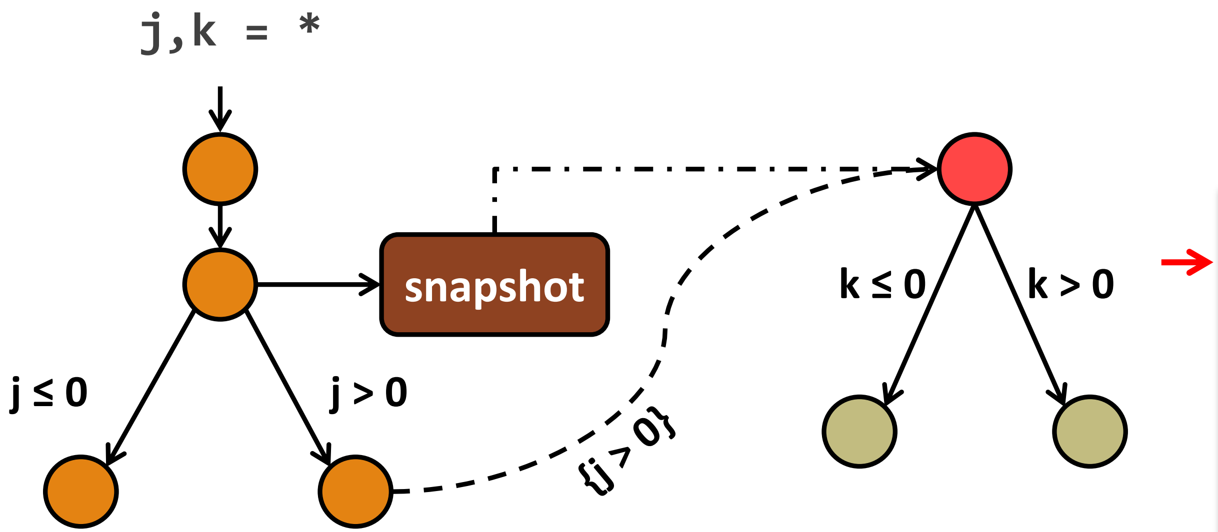
```
void f() {
  if (k > 0)
    // x = 1;
  else
    if (j > 0)
      y = 1;
    else
      y = 0;
}
```

Recovery Process



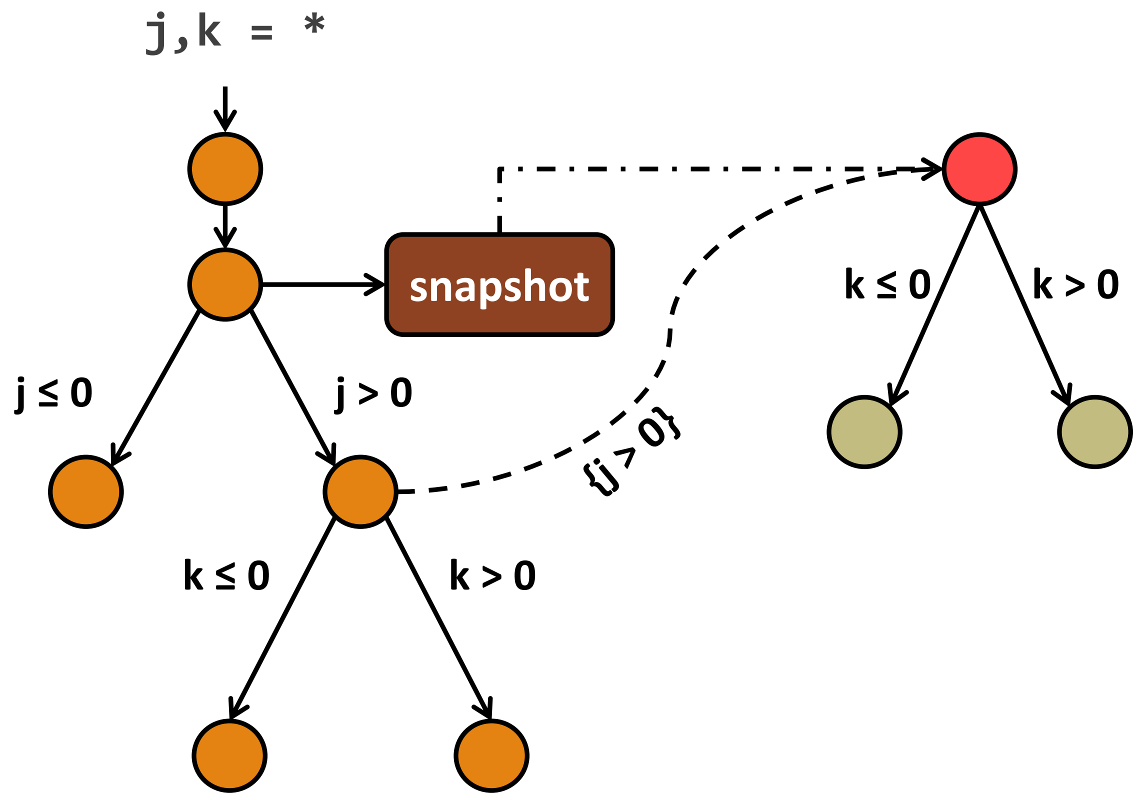

```
void f() {
  if (k > 0)
    // x = 1;
  else
    if (j > 0)
      y = 1;
    else
      y = 0;
}
```

Recovery Process



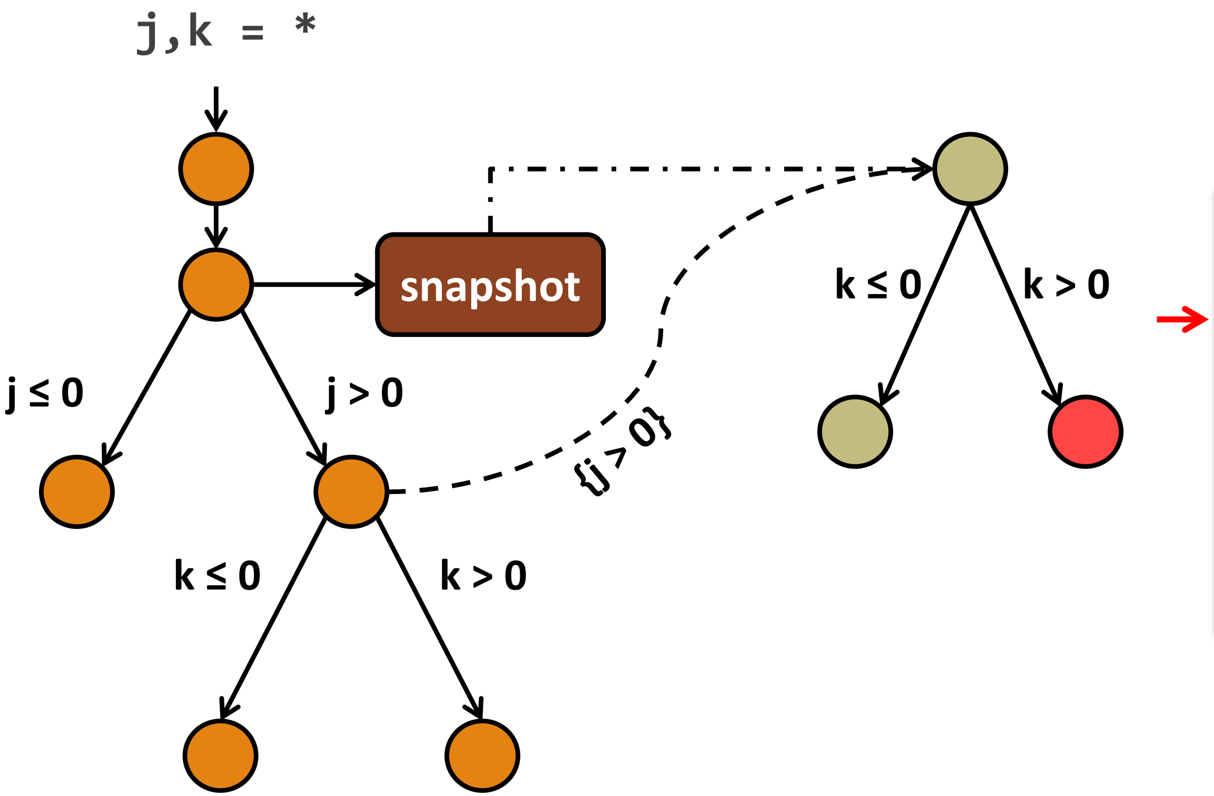

```
void f() {
  if (k > 0)
    // x = 1;
  else
    if (j > 0)
      y = 1;
    else
      y = 0;
}
```

Recovery Process



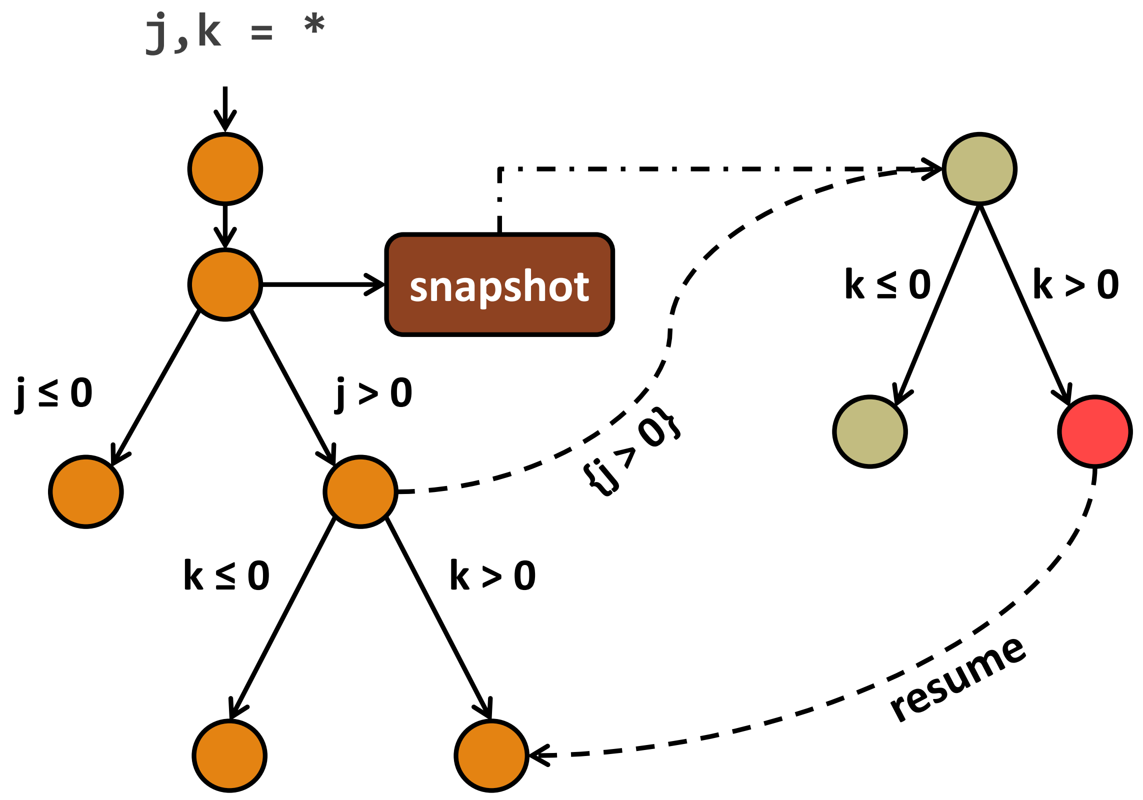

```
void f() {
  if (k > 0)
    // x = 1;
  else
    if (j > 0)
      y = 1;
    else
      y = 0;
}
```

Recovery Process



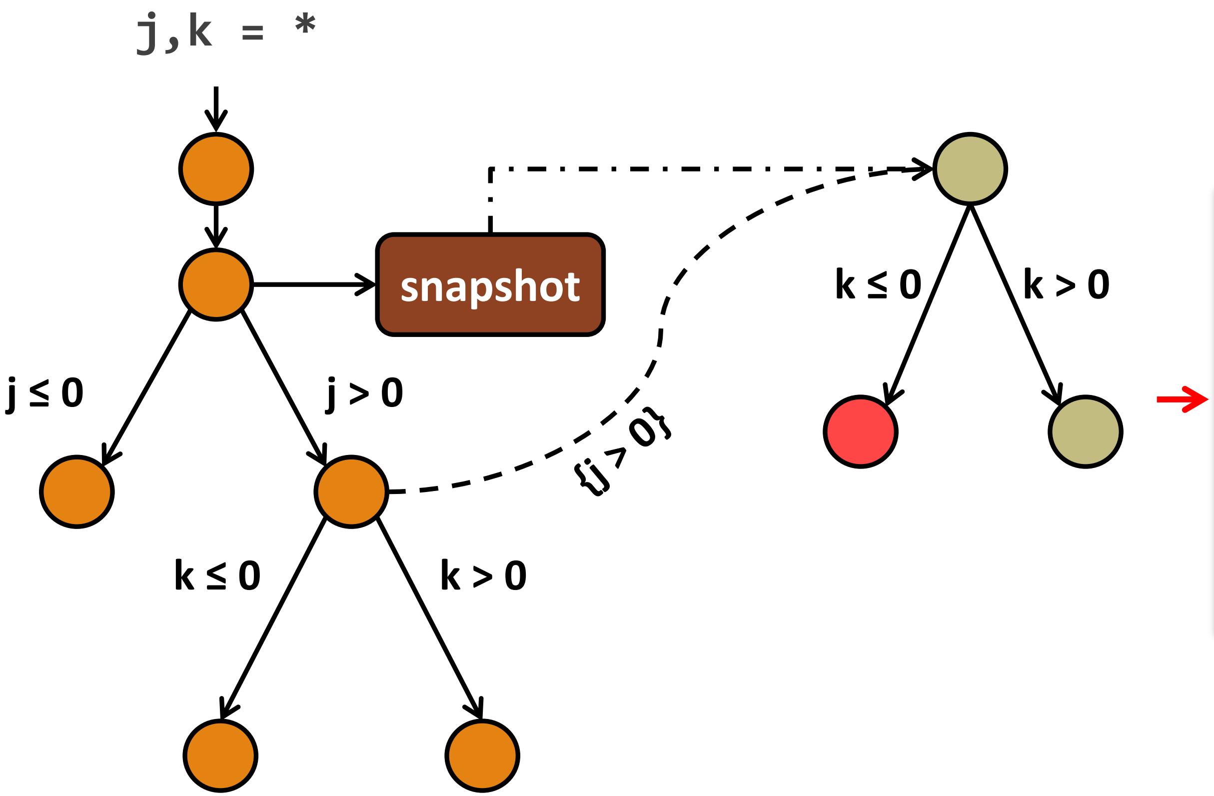

```
void f() {
  if (k > 0)
    // x = 1;
  else
    if (j > 0)
      y = 1;
    else
      y = 0;
}
```

Recovery Process



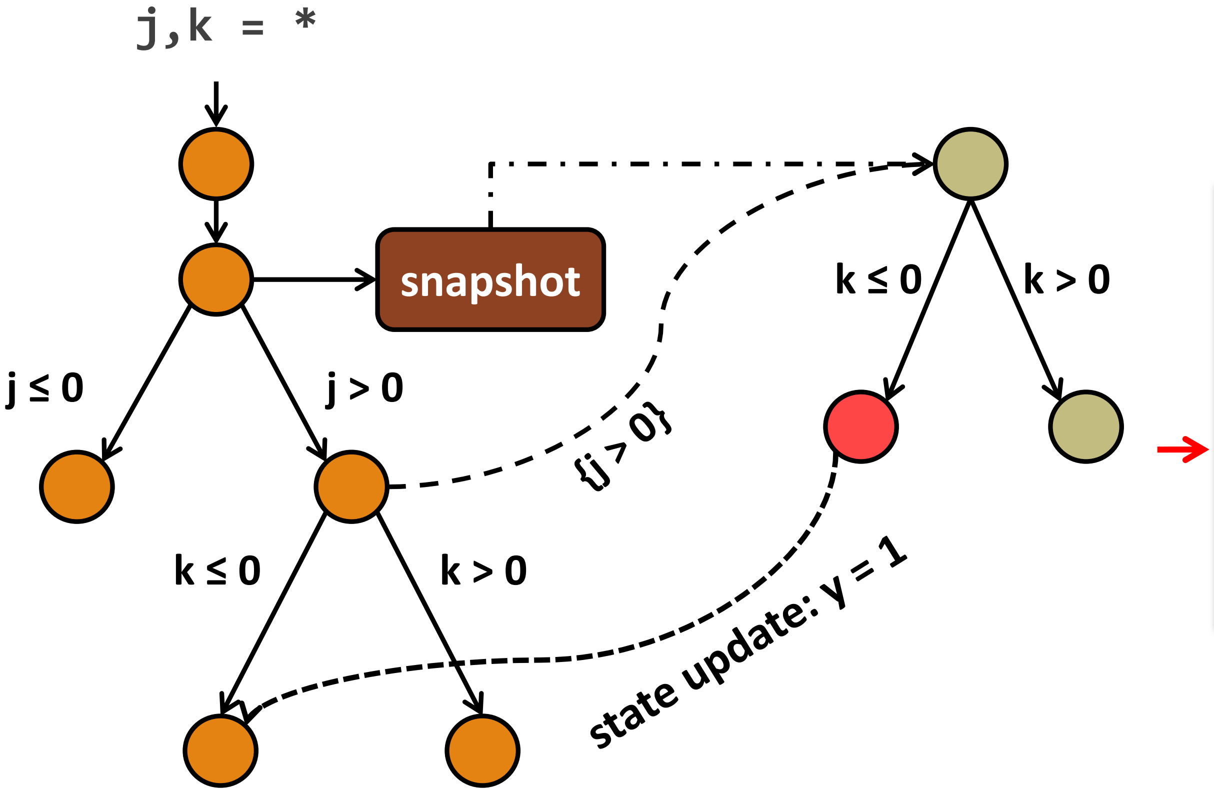

```
void f() {
  if (k > 0)
    // x = 1;
  else
    if (j > 0)
      y = 1;
    else
      y = 0;
}
```

Recovery Process



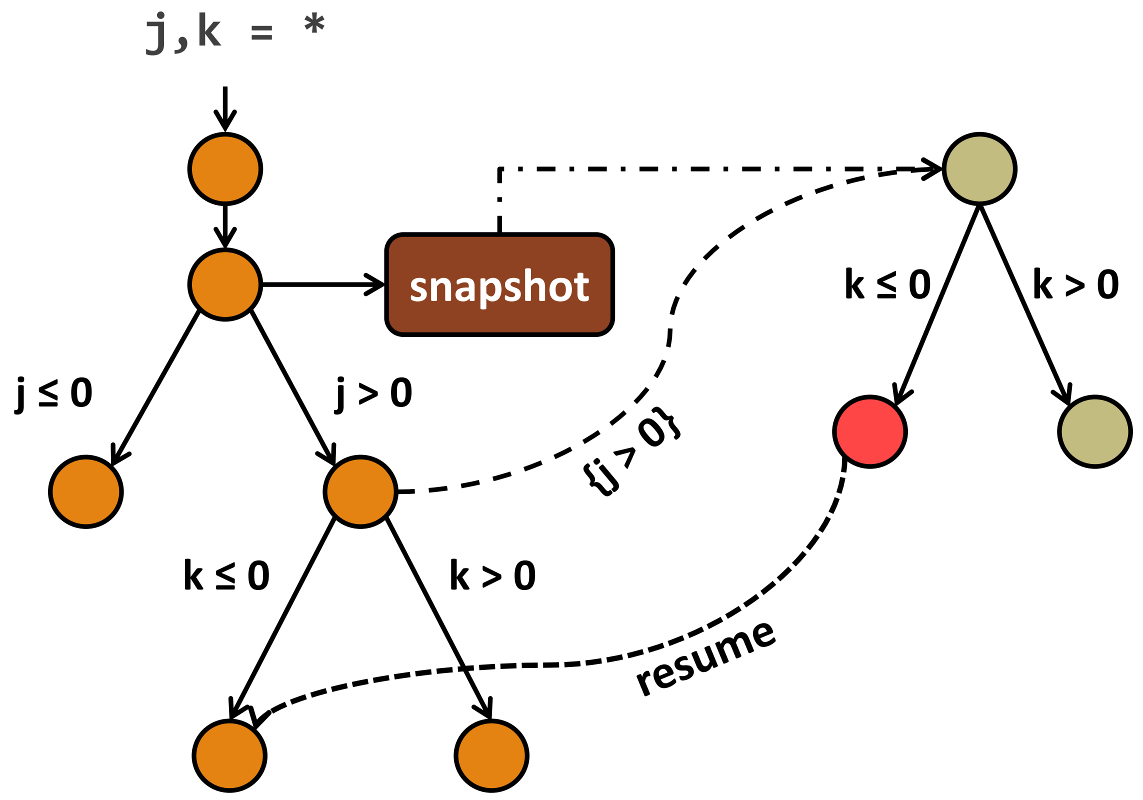

```
void f() {
  if (k > 0)
    // x = 1;
  else
    if (j > 0)
      y = 1;
    else
      y = 0;
}
```

Recovery Process



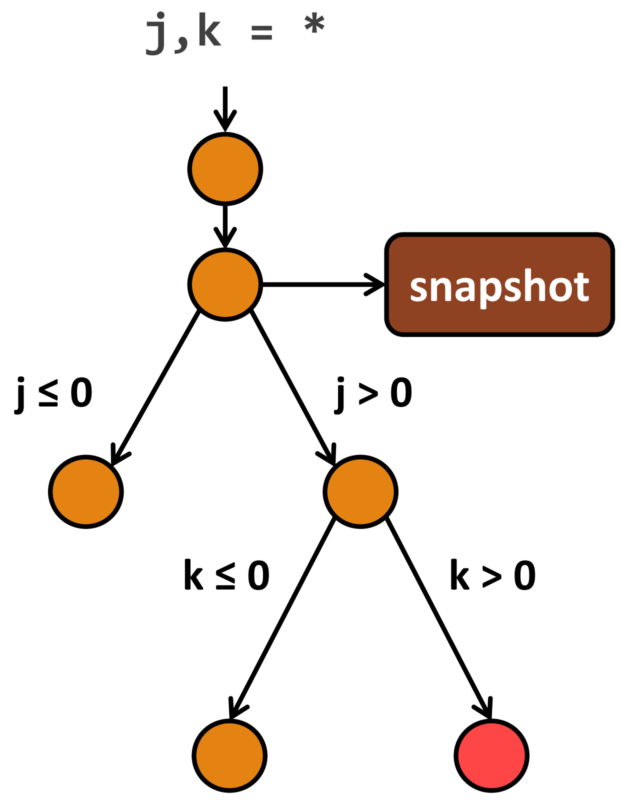

```
void f() {
  if (k > 0)
    // x = 1;
  else
    if (j > 0)
      y = 1;
    else
      y = 0;
}
```

Recovery Process




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

Recovery Process

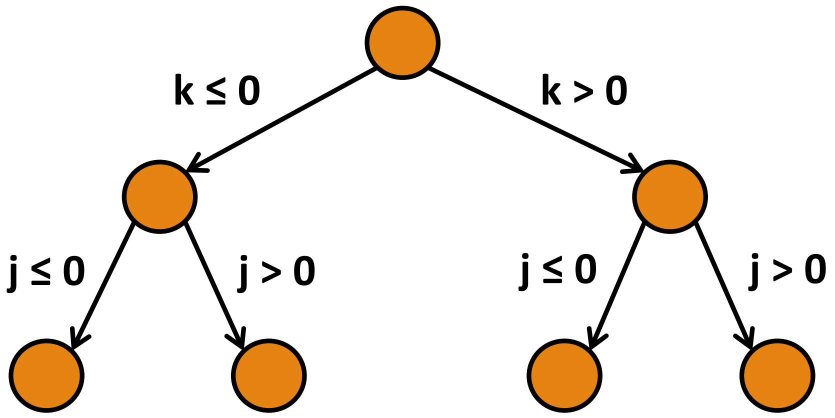



```
void main() {
  f();
  if (j > 0) {
    if (y)
      target1;
  }
  else target2;
}
```

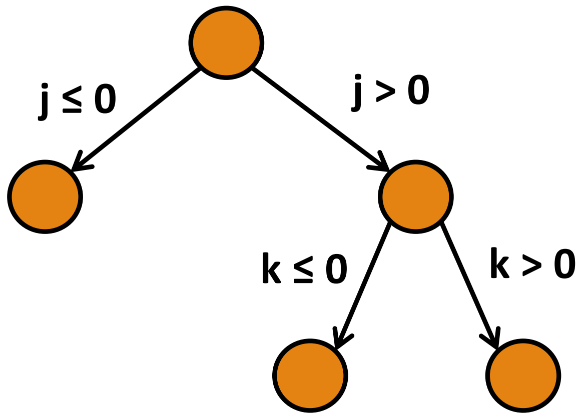
Execution Trees

```
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;
}
```



Standard SE



Chopped SE

Implementation: Chopper

Chopped Symbolic Execution for **C code**

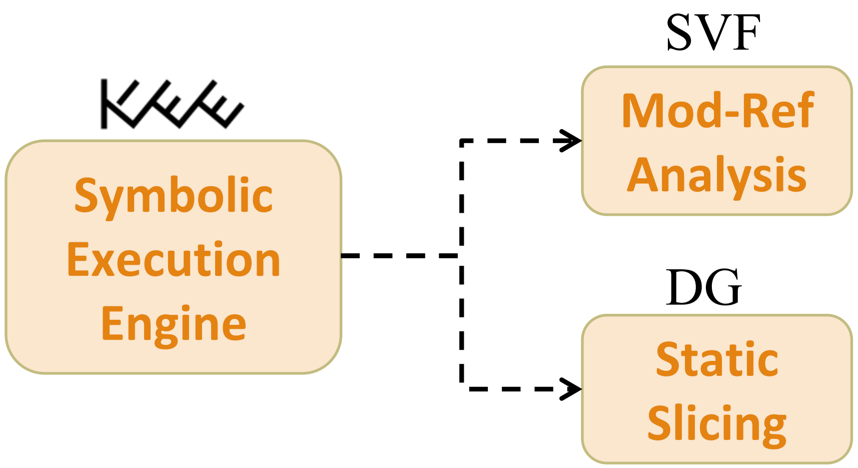
- Implemented at the LLVM bitcode level

Symbolic execution based on **KLEE** [https://klee.github.io/]

Mod-ref analysis based on **SVF** [Yulei Sui and Jingling Xue, https://svf-tools.github.io/SVF/]

- we use a flow-insensitive, context-insensitive, field-sensitive analysis

Static slicing based on **DG** [Marek Chalupa, https://github.com/mchalupa/dg/]



Experiments

Security Vulnerability Reproduction

Coverage Augmentation

Patch Testing

```
address = optimizer.optimizeExpr(address, true);
 StatePair zeroPointer = fork(state, Expr::createIsZero(address), true);
 if (zeroPointer.first) {
   if (target)
     bindLocal(target, *zeroPointer.first, Expr::createPointer(0));
 }
 if (zeroPointer.second) { // address != 0
   ExactResolutionList rl;
   resolveExact(*zeroPointer.second, address, rl, "free");

   for (Executor::ExactResolutionList::iterator it = rl.begin(),
          ie = rl.end(); it != ie; ++it) {
     const MemoryObject *mo = it->first.first;
     if (mo->isLocal) {
       terminateStateOnError(*it->second, "free of alloca", Fr
                             getAddressInfo(*it->second, add
     } else if (mo->isGlobal) {
       terminateStateOnError(*it->second, "free of globa
                             getAddressInfo(*it->secon
     } else {
       it->second->addressSpace.unbindObject(mo)
       if (target)
         bindLocal(target, *it->second, Expr
     }
   }
 }
}

void Executor::resolveExact(Executi
                            ref<Expr>
                            ExactRes                  lts,
                            const st                ) {
 p = optimizer.optimizeExpr(p, true);
 // XXX we may want to be capping this
 ResolutionList rl;
 state.addressSpace.resolve(state, solver, p, rl);

 ExecutionState *unbound = &state;
 for (ResolutionList::iterator it = rl.begin(), ie = rl.end();
      it != ie; ++it) {
   ref<Expr> inBounds = EqExpr::create(p, it->first->getBaseExpr());

   StatePair branches = fork(*unbound, inBounds, true);

   if (branches.first)
     results.push_back(std::make_pair(*it, branches.first));

   unbound = branches.second;
   if (!unbound) // Fork failure
     break;
```

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)

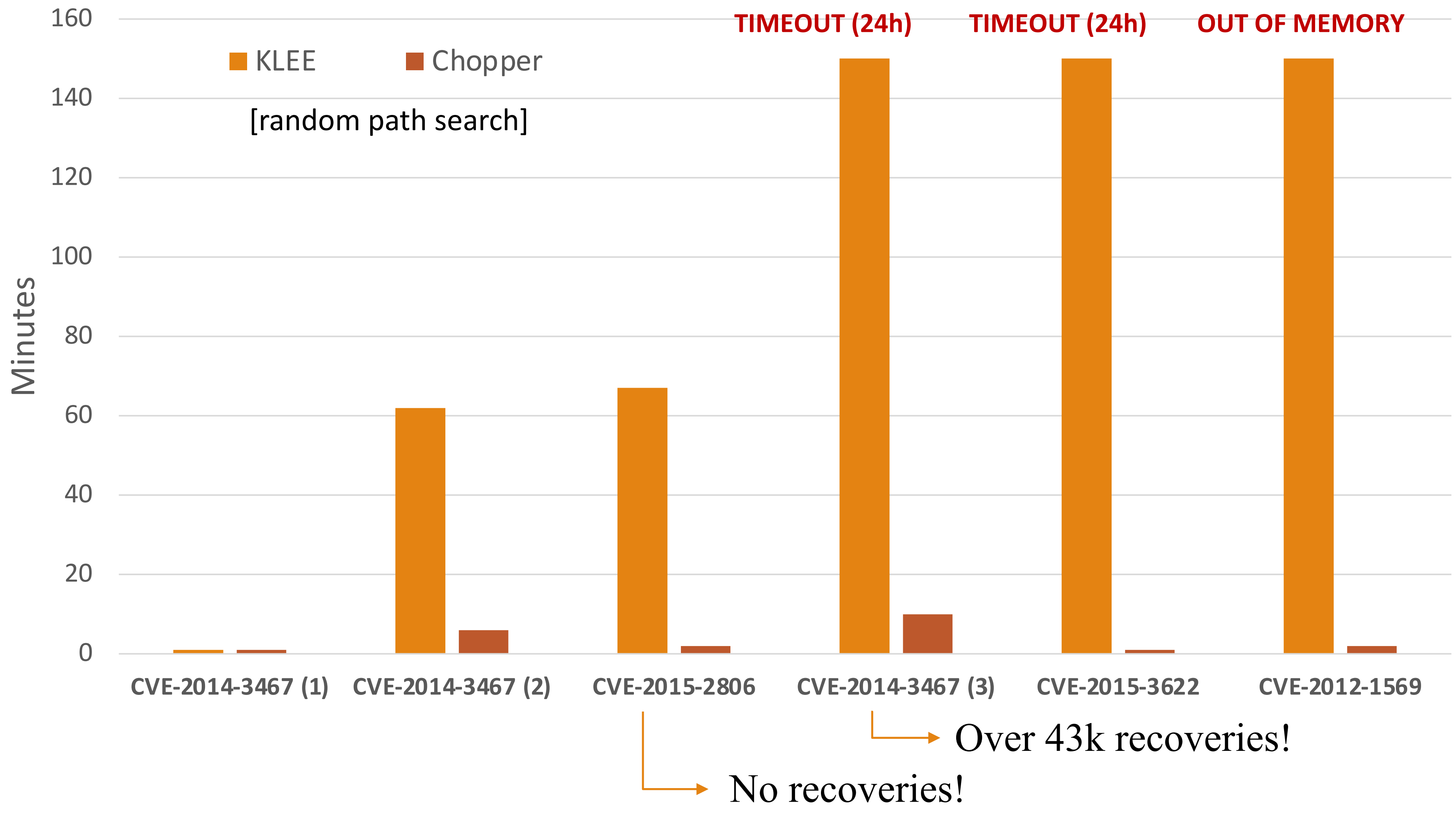
Goal:

- Starting from the CVE report, generate inputs that trigger out-of-bounds accesses at the vulnerable locations

Methodology:

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

Reproducing Security Vulnerabilities



Effectiveness of Chopped Symbolic Execution

Choice of code to skip [ongoing work]

- Task-specific, some scenarios are easier to automate than others
- Can always make different guesses and try them in parallel

Precision of pointer analysis

- Currently a single pointer analysis, in the beginning, where we compute all mod/ref sets
- IDEA: run pointer analysis on demand, just before skipping a function

Motivating Example

```
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)
   ...
```

Imprecision of Pointer Analysis

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

All objects allocated in the loop have **same** allocation site

Cannot distinguish between objs[0] and objs[1]

Imprecision of Pointer Analysis

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

All objects allocated in the loop have **same** allocation site

Cannot distinguish between objs[0] and objs[1]

o->d may point to *(B, 0)*

Imprecision of Pointer Analysis

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

All objects allocated in the loop have **same** allocation site

Cannot distinguish between objs[0] and objs[1]

o->d may point to (*B*, *0*)

Mod(f) = {(*B*, *0*)}

Unnecessary Recoveries

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

Unnecessary recovery!

All objects allocated in the loop have **same** allocation site

$\Downarrow$

Cannot distinguish between objs[0] and objs[1]

$\Downarrow$

o->d may point to (*B*, *0*)

$\Downarrow$

Mod(f) = {(*B*, *0*)}

False dependency!

Past-Sensitive Pointer Analysis (PSPA)

- Run pointer analysis on-demand, not ahead of time:
  - From a specific **symbolic state**
  - On a specific function, **locally**
- Distinguish between past and future:
  - Objects that were *already allocated*
  - 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)
   ...
```

Unique Allocation Sites

During symbolic execution:

- Allocated objects are associated with unique allocation sites

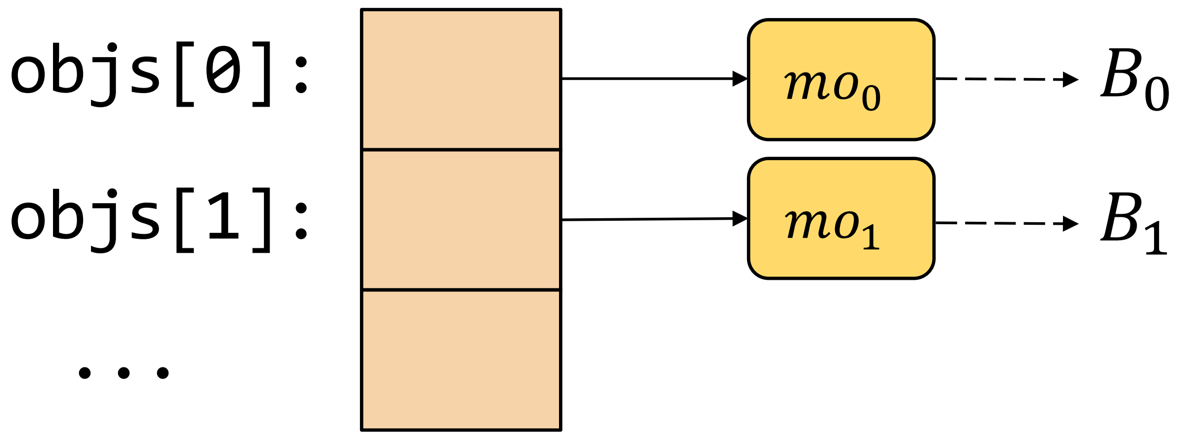
```
for (int i = 0; i < N; i++)
  objs[i] = calloc(...); // AS: B
```

Unique Allocation Sites

During symbolic execution:

- Allocated objects are associated with unique allocation sites

```
for (int i = 0; i < N; i++)
  objs[i] = calloc(...); // AS: B
```

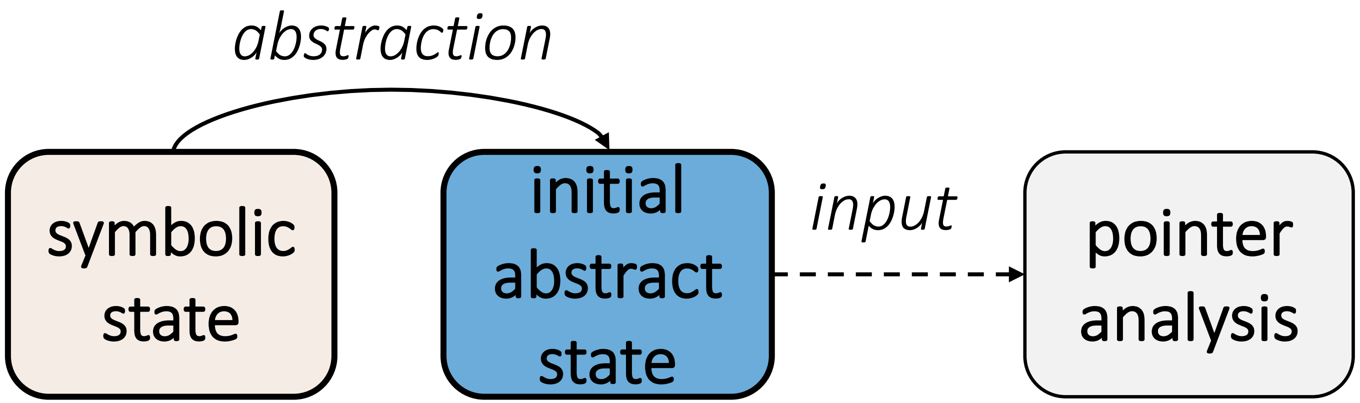


Past-Sensitive Pointer Analysis

When a symbolic state reaches a function call to be skipped:

- Compute a **path-specific abstraction**
- Run pointer analysis from the **initial abstract state**

```
void foo(obj_t *o) {
  if (o->p)
    o->d = 7;
}
...
...
objs[0]->p = malloc(...);
foo(objs[1]);
```

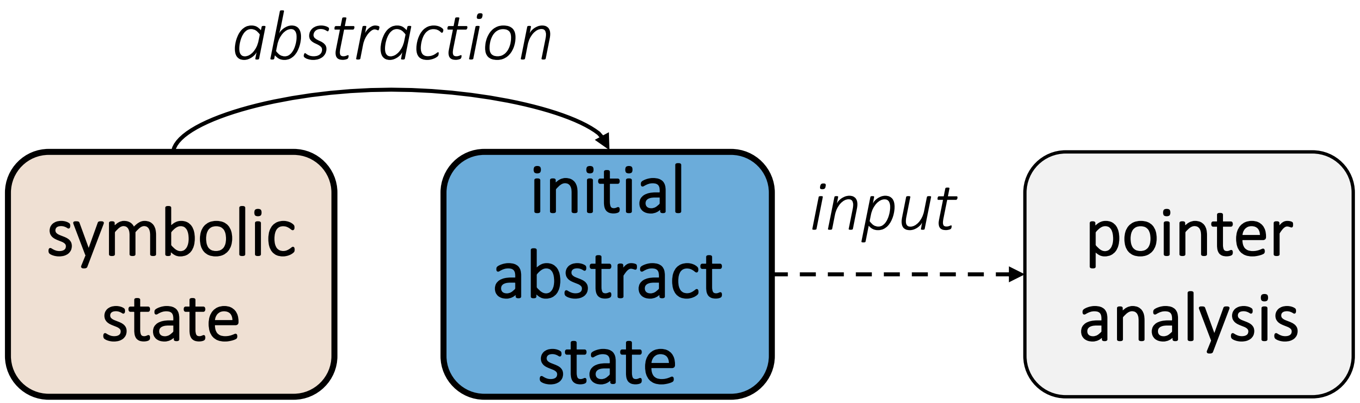


Initial Abstract State

Use current **symbolic state** to construct the **initial abstract state**:

- Traverse function parameters and global variables
- Translate to **points-to graph**

```
void foo(obj_t *o) {
  if (o->p)
    o->d = 7;
}
...
...
objs[0]->p = malloc(...);
foo(objs[1]);
```



Initial Abstract State

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

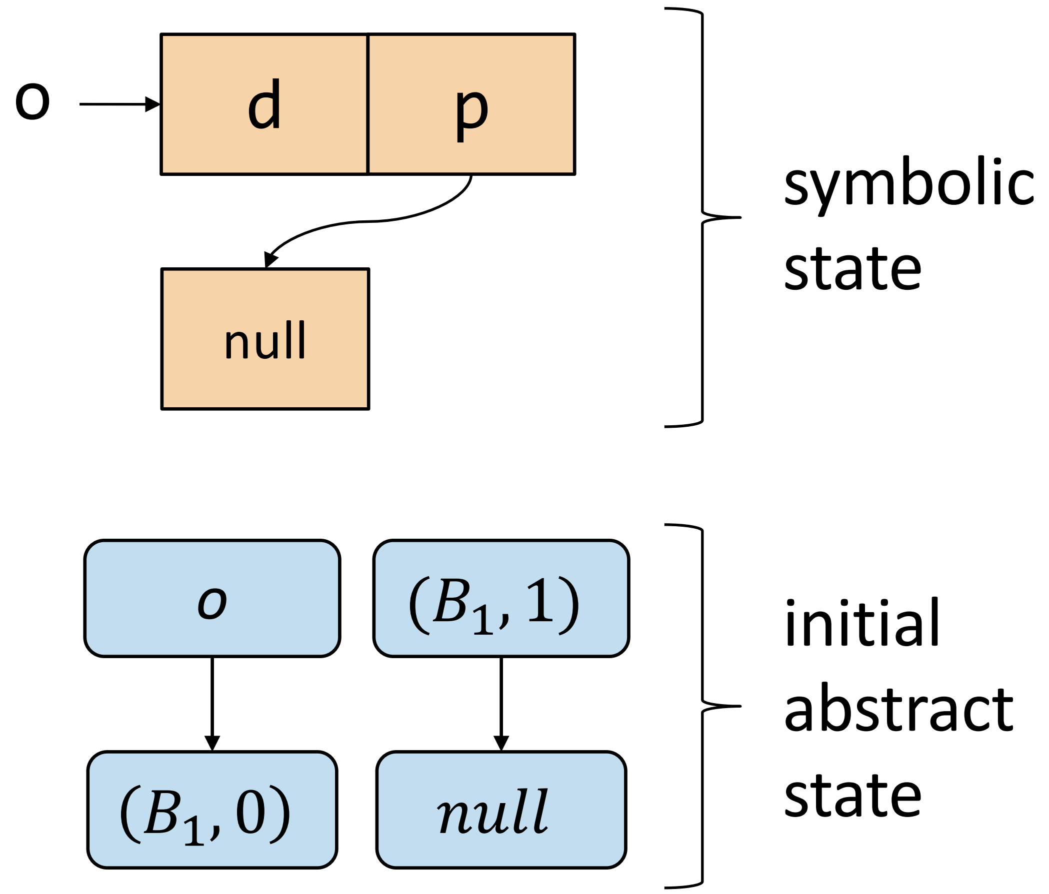
formal parameter

o

symbolic state

Initial Abstract State

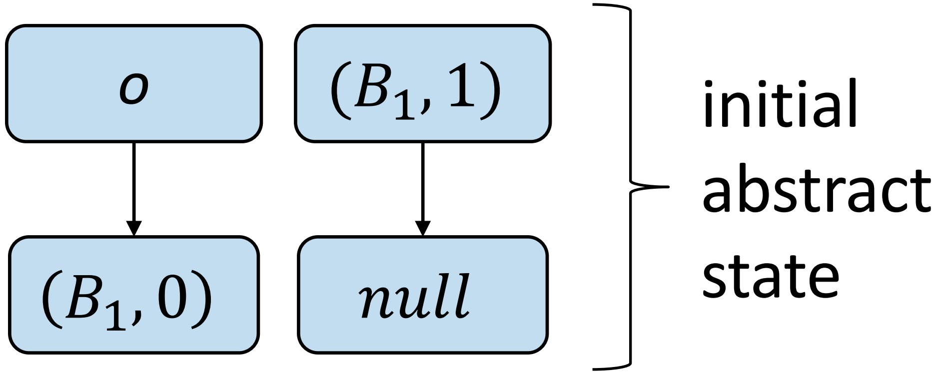
```c
typedef struct { int x, *p; } obj_t;
void foo(obj_t *o) {
  if (o->p)
    o->d = 7;
}
...
obj_t objs[N]; // AS: A
for (int i = 0; i < N; i++)
  objs[i] = calloc(...); // AS: B
...
objs[0]->p = malloc(...); // AS: C
foo(objs[1]);
```



Initial Abstract State

Analyze **foo** from the initial abstract state:

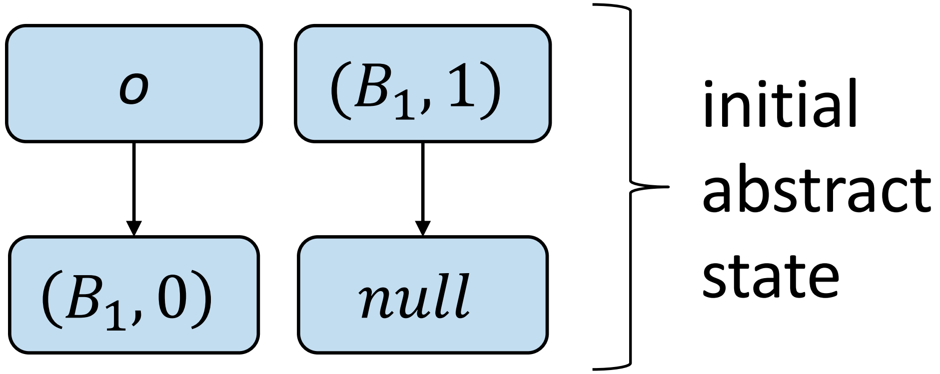
```
void foo(obj_t *o) {
  if (o->p)
    o->d = 7;
}
```



Initial Abstract State

Analyze **foo** from the initial abstract state:

```
void foo(obj_t *o) { // Mod(foo) = {(B₁,0)}
  if (o->p)
    o->d = 7; // pts: (B₁,0)
}
```



No false positives!

No unnecessary recoveries!

Reusing Summaries

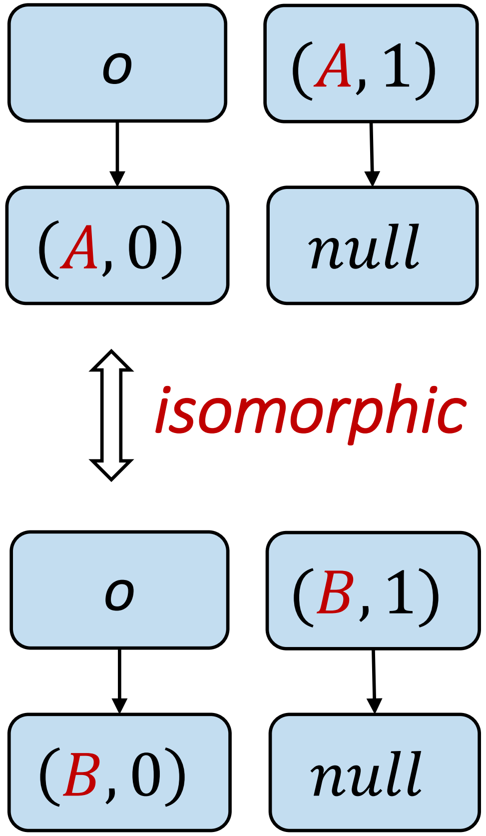
- Number of analyzed functions can be high
  - Running pointer analysis from scratch is **expensive**
- Empirical observation
  - Initial abstract states are **often isomorphic**

Reusing Summaries

```
void foo(obj_t *o) {
  if (o->p)
    o->d = 7;
}
...
foo(o1);
...
foo(o2);
```

initial abstract state

mod-set



$\{(A, 0)\}$

$\{(B, 0)\}$

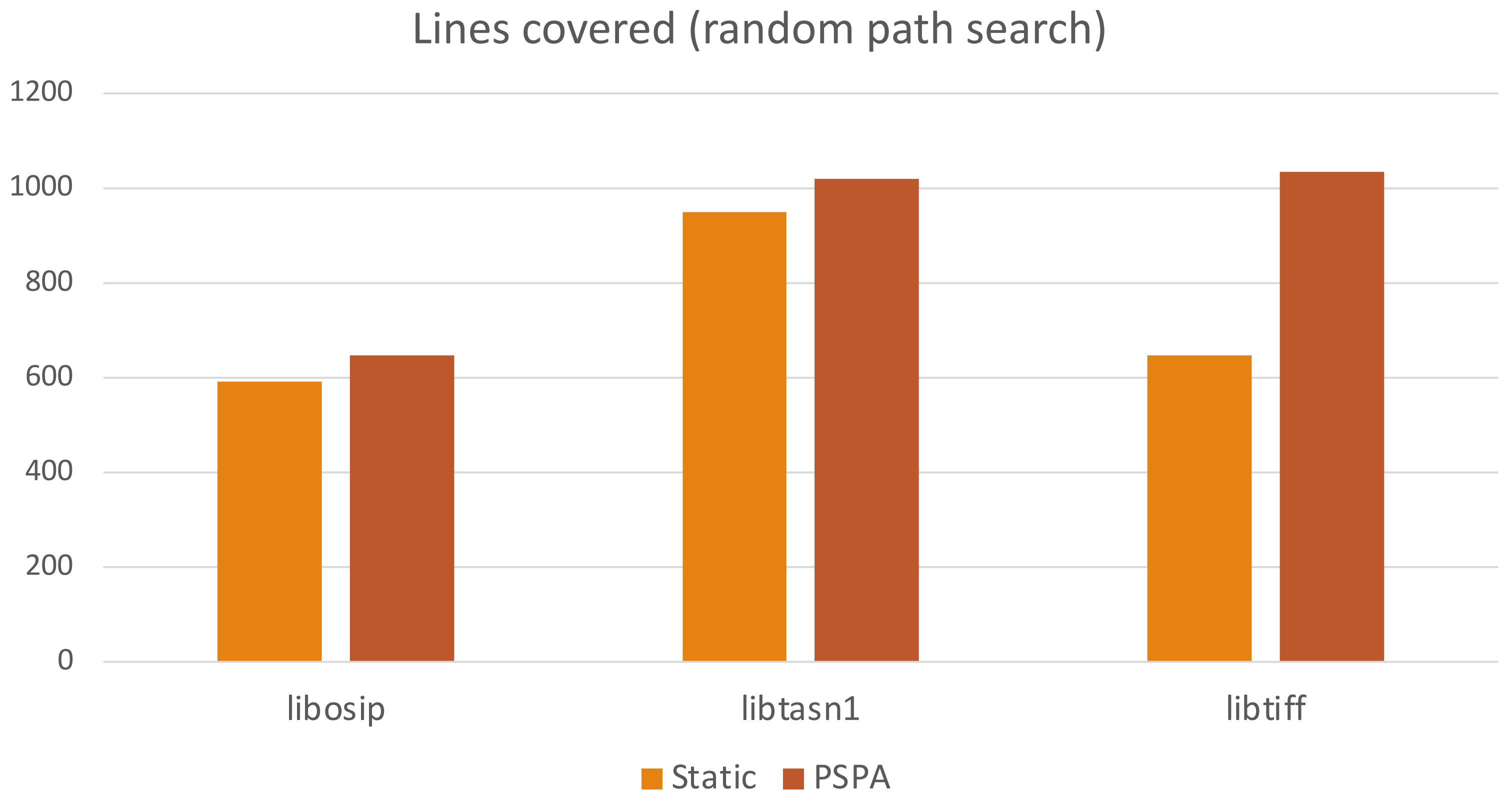
PSPA Impact

Compare Chopper with **static** vs **past-sensitive** pointer analysis for:

- Augmenting coverage
- Reducing recoveries

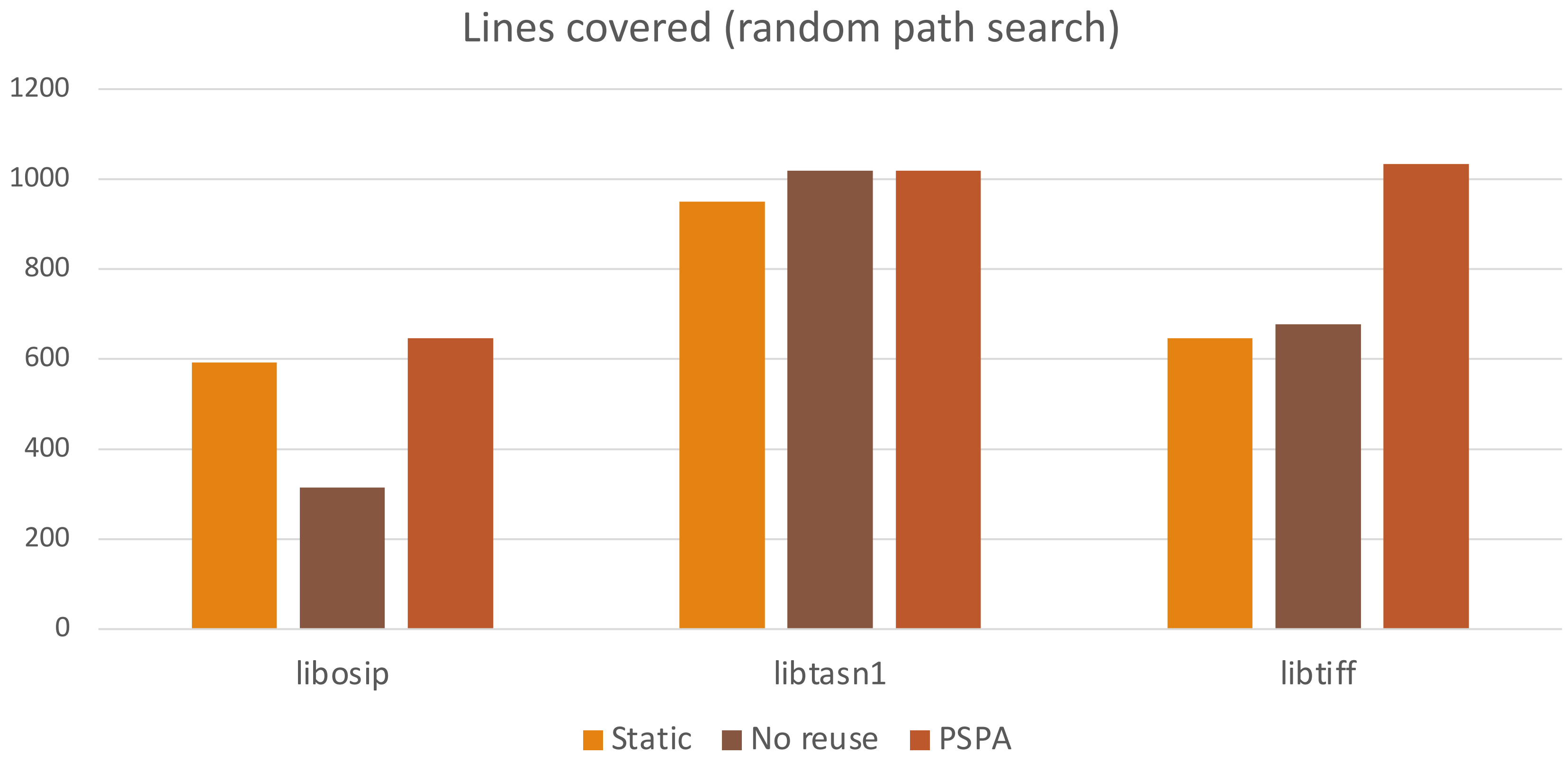
Augmenting Code Coverage

- Manually select skipped functions
  - E.g., In libtiff, we skip logging functions which create many redundant forks
- Run each configuration for 1h
- Measure lines covered



Augmenting Code Coverage: Reuse Impact

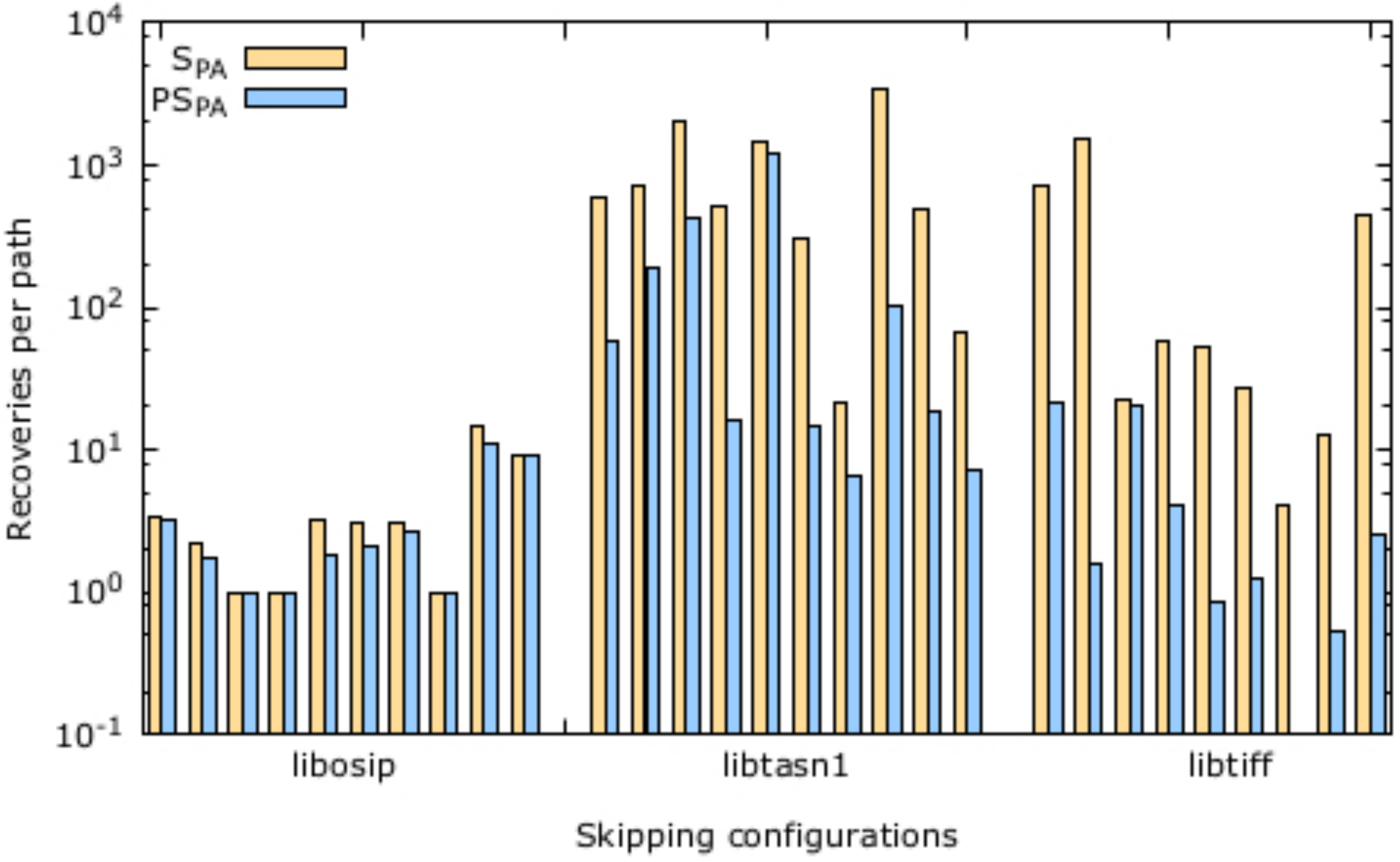
- Run additional configuration with reuse disabled



Reducing Recoveries [no manual selection]

For each benchmark, 10 configurations of 10 randomly-skipped functions

Runs of 10 minutes per configuration



All-path Exploration [search mostly irrelevant]

Construct drivers for our benchmarks that ensure KLEE terminates in <1h
Skip the same functions as before
Run each configuration with a timeout of 1h

	KLEE	Static	PSPA
libosip	33:10	Timeout	04:16
libtasn1	41:29	Timeout	02:12
libtiff	32:40	Timeout	10:02

Excessive number of recoveries

Summary

- Chopped Symbolic Execution enhanced with Past-Sensitive Pointer Analysis can make it possible to skip code irrelevant to a certain task
- At a high-level, we conservatively compute the side effects of the skipped code and if and only if the side effects are ever used, we start a recovery process which goes back and executes missing paths in the skipped code
- Computing the side effects on demand via past-sensitive pointer analysis can significantly reduce the size of the side effects (mod-sets) and thus the number of unnecessary recoveries
- Preliminary results are promising, showing high gains compared to standard symbolic execution for tasks such as bug reproduction and coverage augmentation

[ICSE 2018]

Chopped Symbolic Execution

David Trabish
Tel Aviv University
Israel
davivtra@post.tau.ac.il

Andrea Mattavelli
Imperial College London
United Kingdom
amattave@imperial.ac.uk

Noam Rinetzky
Tel Aviv University
Israel
maon@cs.tau.ac.il

Cristian Cadar
Imperial College London
United Kingdom
c.cadar@imperial.ac.uk

Prototypes are open source: https://srg.doc.ic.ac.uk/projects/

[ESEC/FSE 2020]

Past-Sensitive Pointer Analysis for Symbolic Execution

David Trabish
Tel Aviv University
Israel
davivtra@post.tau.ac.il

Timotej Kapus
Imperial College London
United Kingdom
t.kapus@imperial.ac.uk

Noam Rinetzky
Tel Aviv University
Israel
maon@cs.tau.ac.il

Cristian Cadar
Imperial College London
United Kingdom
c.cadar@imperial.ac.uk

Future Directions

- Skipping arbitrary code fragments
  - Chopper can currently only skip functions
  - What is the right unit of skipped code?
- Better integration with static pointer analysis
  - What is the right underlying pointer analysis to use?
  - What are the performance/precision trade-offs?
- In-depth exploration of scenarios that can benefit from chopping
  - Coverage augmentation, bug reproduction, debugging, program repair, etc.
- Automatic way of selecting code to be skipped
  - Ongoing work on patch testing