Dynamic Symbolic Execution for Evolving Software

Cristian Cadar
Current and recent members

Anastasios Andronidis
Frank Busse
Manuel Carrasco
Karine Even-Mendoza
Martin Nowack

Cristian Cadar

Jordy Ruiz
Daniel Schemmel
Arindam Sharma
Bachir Bendrissou
Ahmed Zaki

http://srg.doc.ic.ac.uk
Current and recent projects

- 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

I am hiring! Let me know if you are interested in a PhD or postdoc in the group!
Do Developers Like Tests?

Test cases are valuable as:

- Quality assurance
- Documentation
- Bug Reports
- Debugging Aid
Do Developers Like Tests?

Line Coverage in Several Popular Open-Source Applications

Joint work with Marinescu and Hosek
Do Developers Like Tests? Writing

Fully-Covered Patches in Several Popular Open-Source Applications

12y development time across apps

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*

```c
if (x > 5)
    printf(“>5”);

if (x > 2)
    printf(“>2”);
```

Real programs: huge number of paths, huge formulas
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

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

Webpage: https://klee.github.io/
Code: https://github.com/klee/
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
Chopped Symbolic Execution

```
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\}

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

Joint work with Trabish, Mattavelli, Rinetzky
Dependent Loads

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

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

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

Program counter: line 2
Stack = [main]
Path constraints: {}
Memory: \{x = 0, y = 0, k = ...\}

```c
void main() {
    f();
    if (j > 0) {
        if (y)
            target1;
    }
    else target2;
}
```
Reaching Target – Ideal Case

\[ j, k = * \]

\[ j \leq 0 \]

```c
void main() {
    f();
    if (j > 0) {
        if (y)
            target1;
    }
    else target2;
}
```
Reaching Target – Ideal Case

\[ j, k = * \]

\[ j \leq 0 \]

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

j, k = *

j ≤ 0

j > 0

Reaching Target – Recovery Needed

dependent load

snapshot

Reaching Target – Recovery Needed

snapshot

j, k = *

j ≤ 0

j > 0

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

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

Recovery Process

j, k = *

create recovery state

snapshot

j ≤ 0

j > 0

dependent load
Recovery Process

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

\[ j, k = * \]

\[ j \leq 0 \]

\[ j > 0 \]

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

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

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

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

```c
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

```java
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;
}
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

![Graph showing time in minutes for different vulnerabilities with KLEE and Chopper, highlighting timeouts and out of memory issues.]

- CVE-2014-3467 (1)
- CVE-2014-3467 (2)
- CVE-2015-2806
- CVE-2014-3467 (3)
- CVE-2015-3622
- CVE-2012-1569

- KLEE
- Chopper

[random path search]

TIMEOUT (24h)  TIMEOUT (24h)  OUT OF MEMORY

Minutes

No recoveries!

Over 43k recoveries!
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**

```c
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:

<table>
<thead>
<tr>
<th>Old</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (x % 2 == 0)</td>
<td>if (x % 3 == 0)</td>
</tr>
<tr>
<td>. . .</td>
<td>. . .</td>
</tr>
</tbody>
</table>

No further uses of x

<table>
<thead>
<tr>
<th>x = 6</th>
<th>x = 7</th>
<th>x = 8</th>
<th>x = 9</th>
</tr>
</thead>
</table>

?
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

```
x = 6
```

```
x = 7
```

```
x = 8
```

```
x = 9
```

old → then
new → else

old → else
new → then
Shadow Symbolic Execution
Symbolic Execution on Both Versions Concurrently

Old

\[
\text{if } (x \mod 2 == 0) \\
\ldots
\]

\[
(x \mod 2 = 0) \land (x \mod 3 \neq 0)
\]

New

\[
\text{if } (x \mod 3 == 0) \\
\ldots
\]

\[
(x \mod 2 \neq 0) \land (x \mod 3 = 0)
\]

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

<table>
<thead>
<tr>
<th>Input</th>
<th>Old</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cut -c1-3,8- -output-d=: file</code> (file is “abcdefg”)</td>
<td><code>abc</code></td>
<td><code>abc + buffer overflow</code></td>
</tr>
<tr>
<td><code>cut -c1-7,8- --output-d=: file</code> file contains “abcdefg”</td>
<td><code>abcdef</code></td>
<td><code>abcdef + buffer overflow</code></td>
</tr>
<tr>
<td><code>cut -b0-2,2- --output-d=: file</code> file contains “abc”</td>
<td><code>abc</code></td>
<td><code>signal abort</code></td>
</tr>
<tr>
<td><code>cut -s -d: -f0- file</code> (file is “:::\n:1”)</td>
<td><code>:::\n:1</code></td>
<td><code>\n\n</code></td>
</tr>
<tr>
<td><code>cut -d: -f1,0- file</code> (file is “a:b:c”)</td>
<td><code>a:b:c</code></td>
<td><code>a</code></td>
</tr>
</tbody>
</table>
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

```python
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, ...

```cpp
int Fn;
if (n <= 1) Fn = 1;
else {
    int Fn_2 = 1, 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);
    }
}
```

<table>
<thead>
<tr>
<th>n</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>prev</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>curr</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

```
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);
}
```
“Do not hard-code ‘/’. Use IS_ABSOLUTE_FILE_NAME and dir_len instead. Use stpcpy/stpncpy in place of strncpy/strncpy.”

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;
“Do not hard-code ‘/’. Use IS_ABSOLUTE_FILE_NAME and dir_len instead. Use stpcpy/strcpy in place of strncpy/strcpy.”

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

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

if (linkbuf == NULL)
    return xstrdup (linkname);

size_t bufsiz = linkbuf - name;

Spec violation:
name = /a
linkname = x

char *p = xmalloc (bufsiz + strlen (linkname) + 1);

Spec violation:
name = /x//y
linkname = a

strncpy (p, name, bufsiz);
strcpy (p + bufsiz, linkname);
return p;

Spec violation:
name = /a
linkname = x

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

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

size_t prefix_len = dir_len (name);

Spec violation:
name = /x//y
linkname = a

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);

Spec violation:
name = /x//y
linkname = a

assert( strcmp(p, p_prev) == 0 );
return p;
```
“Do not hard-code ‘/’. Use IS_ABSOLUTE_FILE_NAME and dir_len instead. Use stpcpy/stpncpy in place of strncpy/strcpy.”

Spec violation:

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

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

Spec violation:

if (linkbuf == NULL)
    return xstrdup (linkname);

size_t bufsiz = linkbuf - name;

Spec violation:

char *p = xmalloc (bufsiz + strlen (linkname) + 1);

size_t prefix_len = dir_len (name);

if (prefix_len == 0)
    return xstrdup (linkname);

char *p = xmalloc (prefix_len + 1 + strlen (linkname) + 1);

strncpy (p, name, prefix_len);

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

Do Developers Like Tests? Writing

Line Coverage in Several Popular Open-Source Applications

<table>
<thead>
<tr>
<th>Software</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Biscuits</td>
<td>17%</td>
</tr>
<tr>
<td>Lighttpd</td>
<td>39%</td>
</tr>
<tr>
<td>Redis</td>
<td>60%</td>
</tr>
<tr>
<td>OMQ</td>
<td>67%</td>
</tr>
<tr>
<td>Memcached</td>
<td>73%</td>
</tr>
<tr>
<td>Git</td>
<td>81%</td>
</tr>
</tbody>
</table>

Chopped Symbolic Execution

```c
void main() {
    f();
    if (location) {
        if (x > 0) {
            x = 1;
        } else {
            y = 1;
        }
    } else {
        y = 0;
    }
}
```

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

Refl(main) = (j, y)

Mod(f) = (x, y)

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

“Do not hard-code ‘/’. Use ABSOLUTE FILE NAME and dir len instead. Use stpcpy/strncpy in place of strncpy/strncpy.”

Shadow Symbolic Execution
Symbolic Execution on Both Versions Concurrently

**Old**

```c
if (x % 2 == 0) {
    . .
}
```

**New**

```c
if (x % 3 == 0) {
    . .
}
```

(x % 2 = 0) ∧ (x % 3 ≠ 0)

TRUE

(x % 2 ≠ 0) ∧ (x % 3 = 0)

FALSE

x = 8

x = 9

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 = strtchr(name, '/');
if (linkbuf == NULL) return xstrdup(linkname);

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

if (!ISSLASH(name(prefix_len - 1))) ++prefix_len;
strncpy(strncpy(p, name, prefix_len), linkname);
assert(patheq(p, p_prev) == 0);
return p;