Toward More Scalable Symbolic Execution via Code Chopping

Cristian Cadar
Department of Computing

Joint work with
David Trabish and Noam Rinetzky (Tel Aviv University)
Timotej Kapus and Andrea Mattavelli (Imperial College London)
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

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.*

[Cadar and Sen, CACM 2013]
Dynamic Symbolic Execution

PyExZ3  SymDroid  KLOVER  jCUTE
SAGE  Otter  PathGrind  SymJS
Jalangi2  BinSE  CREST  Miasm
angr  Symbolic PathFinder  CUTE  Kite
Pex  Rubyx  DART  LDSE
CATG  CiVL  JDart  S²E
angr  Mayhem
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.

Which one is easier?

Opportunity of more localized tasks:

*Prune a large part of the search space*
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

```c
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;
}
```

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

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

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;
}
```

Dependent load
Chopped Symbolic Execution

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

j, k = *

Chopped Symbolic Execution
Taking Snapshots

\[ j, k = * \]

```
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 = ...}

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

```c
void main() {
    f();
    if (j > 0) {
        if (y)
            target1;
    }
    else target2;
}
```
void main() {
    f();
    if (j > 0) {
        if (y) {
            target1;
        }
        else target2;
    }
}
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
void f() {
    if (k > 0)
        x = 1;
    else
        if (j > 0)
            y = 1;
        else
            y = 0;
}
Let $j, k = *$

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

removed by static slicing
Recovery Process

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

\[
\begin{align*}
    j, k &= * \\
    \text{if } (k > 0) &\quad \text{// } x = 1; \\
    \text{else if } (j > 0) &\quad y = 1; \\
    \text{else} &\quad y = 0; \\
\end{align*}
\]
Recovery Process

```c
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

```java
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

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

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

```cpp
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 ofalloca", Free, NULL,
                             getAddressInfo(it->second, address));
    } else if (mo->isGlobal) {
      terminateStateOnError(it->second, "free of global", Free,
                             getAddressInfo(it->second, address));
    } else {
      it->second->addressSpace.unbindObject(mo);
      if (target)
        bindLocal(target, *it->second, Expr::createPointer(0));
    }
  }
}
void Executor::resolveExact(ExecutionState &state,
                            ref<Expr> p,
                            ExactResolutionList &results,
                            const std::string &name) {
  p = optimizer.optimizeExpr(p, true);
  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;
  }
  // XXX we may want to be capping this?
}
```
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

![Graph showing the comparison between KLEE and Chopper]

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

- **KLEE**
- **Chopper**

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

[random path search]

- Over 43k recoveries!
- No recoveries!
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

```c
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

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

All objects allocated in the loop have same allocation site

\[ \downarrow \]

Cannot distinguish between \texttt{objs[0]} and \texttt{objs[1]}

\[ \downarrow \]

\texttt{o->d} may point to \((B,0)\)

\[ \downarrow \]

\texttt{Mod(f) = \{(B,0)\}}

Unnecessary recovery!

False dependency!

```
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)
  ...
```
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

```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)
    ...
```
Unique Allocation Sites

During symbolic execution:
• Allocated objects are associated with unique allocation sites

```c
for (int i = 0; i < N; i++)
    objs[i] = calloc(...); // AS: B
```
During symbolic execution:
- Allocated objects are associated with unique allocation sites

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

**Unique Allocation Sites**

![Diagram showing allocation sites and objects](image)

- `objs[0]`: Attached to `mo_0` and `B_0`
- `objs[1]`: Attached to `mo_1` and `B_1`
- ...
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**

```c
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**

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

obj_t *p = malloc(...); // AS: C

foo(objs[1]);
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:

```c
void foo(obj_t *o) {
    if (o->p)
        o->d = 7;
}
```
Analyze `foo` from the initial abstract state:

```c
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
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

<table>
<thead>
<tr>
<th></th>
<th>KLEE</th>
<th>Static</th>
<th>PSPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>libosip</td>
<td>33:10</td>
<td>Timeout</td>
<td>04:16</td>
</tr>
<tr>
<td>libtasn1</td>
<td>41:29</td>
<td>Timeout</td>
<td>02:12</td>
</tr>
<tr>
<td>libtiff</td>
<td>32:40</td>
<td>Timeout</td>
<td>10:02</td>
</tr>
</tbody>
</table>

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.
### Chopped Symbolic Execution

<table>
<thead>
<tr>
<th>Name</th>
<th>Institution</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>David Trabish</td>
<td>Tel Aviv University</td>
<td><a href="mailto:davivtra@post.tau.ac.il">davivtra@post.tau.ac.il</a></td>
</tr>
<tr>
<td>Andrea Mattavelli</td>
<td>Imperial College London</td>
<td><a href="mailto:amattave@imperial.ac.uk">amattave@imperial.ac.uk</a></td>
</tr>
<tr>
<td>Noam Rinetzky</td>
<td>Tel Aviv University</td>
<td><a href="mailto:maon@cs.tau.ac.il">maon@cs.tau.ac.il</a></td>
</tr>
<tr>
<td>Cristian Cadar</td>
<td>Imperial College London</td>
<td><a href="mailto:c.cadar@imperial.ac.uk">c.cadar@imperial.ac.uk</a></td>
</tr>
</tbody>
</table>

Prototypes are open source: [https://srg.doc.ic.ac.uk/projects/](https://srg.doc.ic.ac.uk/projects/)

### Past-Sensitive Pointer Analysis for Symbolic Execution

<table>
<thead>
<tr>
<th>Name</th>
<th>Institution</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>David Trabish</td>
<td>Tel Aviv University</td>
<td><a href="mailto:davivtra@post.tau.ac.il">davivtra@post.tau.ac.il</a></td>
</tr>
<tr>
<td>Timotej Kapus</td>
<td>Imperial College London</td>
<td><a href="mailto:t.kapus@imperial.ac.uk">t.kapus@imperial.ac.uk</a></td>
</tr>
<tr>
<td>Noam Rinetzky</td>
<td>Tel Aviv University</td>
<td><a href="mailto:maon@cs.tau.ac.il">maon@cs.tau.ac.il</a></td>
</tr>
<tr>
<td>Cristian Cadar</td>
<td>Imperial College London</td>
<td><a href="mailto:c.cadar@imperial.ac.uk">c.cadar@imperial.ac.uk</a></td>
</tr>
</tbody>
</table>
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