Chopping Code for More Modular and Scalable Symbolic Execution

Cristian Cadar
Department of Computing
Imperial College London

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

[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  JDart
CATG  CiVL  Mayhem  S²E
• Active user and developer base with over 300 subscribers on the mailing list and over 50 contributors listed on GitHub

• Academic impact:
  • Over 2K citations to original KLEE paper
  • From many different research communities: testing, verification, systems, software engineering, security, etc.
  • Several influential systems using KLEE: AEG, Angelix, BugRedux, Cloud9, GKLEE, KleeNet, KLEE-UC, S2E, SemFix, etc.

• Growing impact in industry:
  • **Fujitsu**: [PPoPP 2012], [CAV 2013], [ICST 2015], [IEEE Software 2017], [KLEE Workshop 2018], **Hitachi**: [CPSNA 2014], [ISPA 2015], [EUC 2016], **Trail of Bits**: https://blog.trailofbits.com/, **Intel**: [WOOT 2015], **NASA Ames**: [NFM 2014], **Baidu**: [KLEE Workshop 2018]
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

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 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;
}
```
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;
}

Ref(main) = {j, y}
Mod(f) = {x, y}
```
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;
        }
    } else {
        target2;
    }
}

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

Dependent load
Chopped Symbolic Execution

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

\[ j, k = \ast \]
Chopped Symbolic Execution

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

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

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

```c
void main()
```

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

Chopped Symbolic Execution

\[ j, k = * \]

\[ j \leq 0 \quad \rightarrow \quad j > 0 \]

create recovery state

snapshot

dependent load
void f() {
    if (k > 0)
        x = 1;
    else
        if (j > 0)
            y = 1;
        else
            y = 0;
}
Chopped Symbolic Execution

\[ j, k = * \]

\[ j \leq 0 \quad \text{and} \quad j > 0 \]

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

removed by static slicing

void f() {
    if (k > 0)
        // \( x = 1 \);
    else
        if (j > 0)
            y = 1;
        else
            y = 0;
}
Chopped Symbolic Execution

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

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

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

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

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

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

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

```c
void main() {
    f();
    if (j > 0) {
        if (y)
            target1;
    } else target2;
}
```
```c
void main() {
    f();
    if (j > 0) {
        if (y)
            bug1();
    } else bug2();
}

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

**Execution Trees**

**Standard SE**

**Chopped SE**
Implementation: Chopper

- Symbolic execution based on KLEE
- Mod-ref analysis based on SVF
- Static slicing based on DG
Experiments

Security Vulnerability Reproduction

Test Suite Augmentation

Patch Testing

void Executor::resolveExact(ExecutionState &state, ref<Expr> p, ExactResolutionList &results, const std::string &name) {
    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(); it != rl.end(); ++it) {
        ref<Expr> inBounds = EqExpr::create(p, it->first);
        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

![Bar chart showing the minutes taken by KLEE and Chopper to reproduce security vulnerabilities. The chart includes CVE-2014-3467 (1), CVE-2014-3467 (2), CVE-2015-2806, CVE-2014-3467 (3), CVE-2015-3622, and CVE-2012-1569.]
Effectiveness of Chopped Symbolic Execution

• Choice of code to skip
  • Application-specific, still work in progress
  • Some scenarios are easier to automate than others
  • Can always make different guesses and try them in parallel

• Precision of points-to analysis
  • Currently we use a flow-insensitive, context-insensitive and field-sensitive points-to analysis
  • Currently a single points-to analysis, in the beginning
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?

Opportunities of more localized tasks:
  • Prune a large part of the search space
  • Perform incremental reasoning/analysis
  • Use previous/correct version as an oracle
  • etc.

Experiments

SECURITY VULNERABILITY REPRODUCTION

TEST SUITE AUGMENTATION

PATCH TESTING

Chopped Symbolic Execution

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

Chopped Symbolic Execution [ICSE 2018]

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

ABSTRACT

Symbolic execution is a powerful program analysis technique that systematically explores multiple program paths. However, despite important technical advances, symbolic execution often struggles to reach deep parts of the code due to the well-known path explosion problem and constraint solving limitations.

In this paper, we propose chopped symbolic execution, a novel form of symbolic execution that allows users to specify uninteresting parts of the code to exclude during the analysis, thus only targeting the exploration to paths of importance. However, the excluded parts are not summarily ignored, as this may lead to both false positives and false negatives. Instead, they are executed lazily, when their effect may be observable by code under analysis. Chopped symbolic execution leverages various on-demand static analyses at runtime to statically exclude code fragments

the code with symbolic values instead of concrete ones. Symbolic execution engines thus replace concrete program operations with ones that manipulate symbols, and add appropriate constraints on the symbolic values. In particular, whenever the symbolic executor reaches a branch condition that depends on the symbolic inputs, it determines the feasibility of both sides of the branch, and creates two new independent symbolic states which are added to a worklist to follow each feasible side separately. This process, referred to as forking, refines the conditions with values by adding appropriate constraints on each path according to the conditions on the branch. Test cases are generated by finding concrete values for the symbolic inputs that satisfy the path conditions. To both determine the feasibility of path conditions and generate concrete solutions that satisfies them, symbolic execution engines employ satisfiability-modulo-theory (SMT) constraint solvers [19].