Efficient Leveraging of Symbolic Execution to Advanced Coverage Criteria

Sébastien Bardin, Nikolai Kosmatov and François Cheynier

CEA LIST, Software Safety Lab
(Paris-Saclay, France)
Dynamic Symbolic Execution [dart, cute, exe, sage, pex, klee, ...]
✓ very powerful approach to (white box) test generation
✓ many tools and many successful case-studies since mid 2000’s
✓ arguably one of the most wide-spread use of formal methods in “common software” [SAGE at Microsoft]
Dynamic Symbolic Execution [dart, cute, exe, sage, pex, klee, …]
✓ very powerful approach to (white box) test generation
✓ many tools and many successful case-studies since mid 2000’s
✓ arguably one of the most wide-spread use of formal methods in “common software” [SAGE at Microsoft]

Symbolic Execution [King 70’s]
- consider a program P on input v, and a given path σ
- a path predicate \( \varphi_\sigma \) for \( \sigma \) is a formula s.t.
  \[ v \models \varphi_\sigma \Rightarrow P(v) \text{ follows } \sigma \]
- can be used for bounded-path testing!
- old idea, recent renew interest [requires powerful solvers]
Dynamic Symbolic Execution [dart, cute, exe, sage, pex, klee, ...]
✓ very powerful approach to (white box) test generation
✓ many tools and many successful case-studies since mid 2000’s
✓ arguably one of the most wide-spread use of formal methods in “common software” [SAGE at Microsoft]

Symbolic Execution [King 70’s]
- consider a program $P$ on input $v$, and a given path $\sigma$
- a path predicate $\varphi_\sigma$ for $\sigma$ is a formula s.t.
  $$v \models \varphi_\sigma \Rightarrow P(v) \text{ follows } \sigma$$
- can be used for bounded-path testing!
- old idea, recent renew interest [requires powerful solvers]

Dynamic Symbolic Execution [Korel+, Williams+, Godefroid+]
- interleave dynamic and symbolic executions
- drive the search towards feasible paths for free
- give hints for relevant under-approximations [robustness]
Dynamic Symbolic Execution (2)

**input**: a program $P$

**output**: a test suite $TS$ covering all feasible paths of $Paths_{\leq k}(P)$

- pick a path $\sigma \in Paths_{\leq k}(P)$
- compute a *path predicate* $\varphi_\sigma$ of $\sigma$
- solve $\varphi_\sigma$ for satisfiability
- $\text{SAT}(s)$? get a new pair $< s, \sigma >$
- loop until no more path to cover

[sm", "t solver]
input: a program $P$
output: a test suite $TS$ covering all feasible paths of $\text{Paths}^{\leq k}(P)$

- pick a path $\sigma \in \text{Paths}^{\leq k}(P)$
- compute a path predicate $\varphi_\sigma$ of $\sigma$  
- solve $\varphi_\sigma$ for satisfiability
- $\text{SAT}(s)$? get a new pair $< s, \sigma >$
- loop until no more path to cover

[wpre, spost]  
[smt solver]
**Dynamic Symbolic Execution (2)**

**input:** a program $P$

**output:** a test suite $TS$ covering all feasible paths of $\text{Paths}^{\leq k}(P)$

- pick a path $\sigma \in \text{Paths}^{\leq k}(P)$
- compute a path predicate $\varphi_\sigma$ of $\sigma$
- solve $\varphi_\sigma$ for satisfiability
- SAT(s) ? get a new pair $< s, \sigma >$
- loop until no more path to cover

Bardin et al. ICST 2014 3/29
**Dynamic Symbolic Execution (2)**

**input**: a program \( P \)

**output**: a test suite \( TS \) covering all feasible paths of \( Paths^{≤k}(P) \)

- pick a path \( σ ∈ Paths^{≤k}(P) \)
- compute a *path predicate* \( ϕ_σ \) of \( σ \) \([wpre, spost]\)
- solve \( ϕ_σ \) for satisfiability \([smt solver]\)
- SAT(s)? get a new pair \( < s, σ > \)
- loop until no more path to cover
**Dynamic Symbolic Execution (2)**

**input**: a program $P$

**output**: a test suite $TS$ covering all feasible paths of $\text{Paths}^{\leq k}(P)$

- pick a path $\sigma \in \text{Paths}^{\leq k}(P)$
- compute a *path predicate* $\varphi_\sigma$ of $\sigma$
- solve $\varphi_\sigma$ for satisfiability
- SAT(s)? get a new pair $<s, \sigma>$
- loop until no more path to cover

[wpre, spost]
[smt solver]
Dynamic Symbolic Execution (2)

**input**: a program \( P \)

**output**: a test suite \( TS \) covering all feasible paths of \( Paths^{\leq k}(P) \)

- pick a path \( \sigma \in Paths^{\leq k}(P) \)
- compute a path predicate \( \varphi_{\sigma} \) of \( \sigma \)
- solve \( \varphi_{\sigma} \) for satisfiability
- \( SAT(s) \)? get a new pair \( < s, \sigma > \)
- loop until no more path to cover

[wpre, spost]
[smt solver]
Dynamic Symbolic Execution (2)

**Input:** a program \( P \)

**Output:** a test suite \( TS \) covering all feasible paths of \( Paths^{\leq k}(P) \)

- pick a path \( \sigma \in Paths^{\leq k}(P) \)
- compute a *path predicate* \( \varphi_\sigma \) of \( \sigma \)
- solve \( \varphi_\sigma \) for satisfiability
- \( \text{SAT}(s) \) ? get a new pair \( < s, \sigma > \)
- loop until no more path to cover
The problem

Dynamic Symbolic Execution

✓ very powerful approach to (white box) test generation
✓ arguably one of the most wide-spread use of formal methods in “common software”
Dynamic Symbolic Execution

✓ very powerful approach to (white box) test generation
✓ arguably one of the most wide-spread use of formal methods in “common software”
× support only basic coverage criteria (IC, DC)

DSE is limited in the following cases:

■ generate TS achieving a given coverage criterion
■ generate a “good” TS for an external oracle
  [functional correctness, security, performance, etc.]
The problem

**Dynamic Symbolic Execution**

✓ very powerful approach to (white box) test generation
✓ arguably one of the most wide-spread use of formal methods in “common software”
✗ support only basic coverage criteria (IC, DC)

---

**Challenge**: extend DSE to a large class of coverage criteria

- well-known problem
- recent efforts in this direction through instrumentation
  
  [Active Testing, Mutation DSE, Augmented DSE]

- limitations:
  - exponential explosion of the search space [APEX : 272x avg]
  - very implementation-centric mechanisms
  - unclear expressiveness
Our results

**Labels**: a well-defined specification mechanism for coverage criteria
- based on predicates, can easily encode a large class of criteria
- w.r.t related work: semantic view, more formal treatment

**DSE**: an efficient integration of labels into DSE
- no exponential blowup of the search space
- can be added to DSE in blackbox

**Implem.** in **PathCrawler**
- huge savings compared to existing approaches
- handle labels for only a low overhead (2x average, up to 7x)
Given a program $P$, a label $l$ is a pair $(loc, \varphi)$, where:

- $\varphi$ is well-defined in $P$ at location $loc$
- $\varphi$ contains no side-effect expression

**Basic definitions**

- an annotated program is a pair $\langle P, L \rangle$, with $L$ set of labels
- a test datum $t$ covers $l$ if $P(t)$ reaches $loc$ and satisfies $\varphi$
- new criterion $\text{LC}$ (label coverage) for annotated programs

**Notations**

- $t \rightsquigarrow_{\langle P, L \rangle} l$ for “$t$ covers $l$”
- $TS \rightsquigarrow_{\langle P, L \rangle} \text{LC}$ for “$TS$ covers all labels of $\langle P, L \rangle$”
Goal = reasoning about the relative expressiveness of \( \text{LC} \)

A labelling function \( \psi \) maps a program \( P \) into an annotated program \( \psi(P) \triangleq \langle P, L \rangle \)

**Definition (Simulation)**

A coverage criterion \( C \) can be simulated by \( \text{LC} \) if there exists a labelling function \( \psi \) such that for any program \( P \) and any test suite \( TS \), we have: \( TS \sim_P C \) iff \( TS \sim_{\psi(P)} \text{LC} \).
Simulation of standard coverage criteria (1)

Decision Coverage (DC)
Simulation of standard coverage criteria (2)

```
statement_1;
if (x==y && a<b)
    {...};
statement_3;
```

```
statement_1;
// l1: x==y
// l2: !(x==y)
// l3: a<b
// l4: !(a<b)
if (x==y && a<b)
    {...};
statement_3;
```

Condition Coverage (CC)
Multiple-Condition Coverage (MCC)
Weak Mutation testing in a nutshell

- mutant $M = \text{syntactic modification of program } P$
- weakly covering $M = \text{finding } t \text{ such that } P(t) \neq M(t) \text{ just after the mutation}$
From weak mutants to labels

One label per mutant

Mutation inside a statement

- \( \text{lhs} := e \mapsto \text{lhs} := e' \)
  - add label: \( e \neq e' \)
- \( \text{lhs} := e \mapsto \text{lhs}' := e \)
  - add label: \&\text{lhs} \neq \&\text{lhs}' \land (\text{lhs} \neq e \lor \text{lhs}' \neq e) \)

Mutation inside a decision

- \( \text{if (cond)} \mapsto \text{if (cond')} \)
  - add label: \( \text{cond} \oplus \text{cond}' \)

Beware: no side-effect inside labels
From weak mutants to labels

One label per mutant

Mutation inside a statement

\[ \text{lhs := e} \implies \text{lhs := e}' \]

Theorem

*For any finite set \( O \) of side-effect free mutation operators, \( \text{WM}_O \) can be simulated by \( \text{LC} \).*

Mutation inside a decision

\[ \text{if (cond)} \implies \text{if (cond')} \]

- add label: \( \text{cond} \oplus \text{cond}' \)

Beware: no side-effect inside labels
From weak mutants to labels

One label per mutant

Mutation inside a statement

\[
\text{lhs := } e \rightarrow \text{lhs := e'}
\]

Theorem

For any finite set \( O \) of side-effect free mutation operators, \( \text{WM}_O \) can be simulated by \( \text{LC} \).

Mutation inside a decision

Theorem

More simulations:
\( \text{IC, DC, FC, CC, MCC, } \)
Input Domain Partition,
Run-Time Error

Beware: no side-effect inside labels

Theorem

For any finite set \( O \) of side-effect free mutation operators, \( \text{WM}_O \) can be simulated by \( \text{LC} \).
From weak mutants to labels

One label per mutant

Mutation inside a statement

- \( \text{lhs} := e \rightarrow \text{lhs} := e' \)

Theorem

For any finite set \( \mathcal{O} \) of side-effect free mutation operators, \( \text{WM}_\mathcal{O} \) can be simulated by \( \text{LC} \).

Theorem

More simulations:
- IC, DC, FC, CC, MCC,
- Input Domain Partition,
- Run-Time Error

Goals

✓ OBJ1: generic specification mechanism for coverage criteria

OBJ2: efficient integration into DSE

Beware: no side-effect inside labels
Outline

- Introduction
- Labels
- Efficient DSE for labels
- Experiments
- Conclusion
Covering label $l \Leftrightarrow$ Covering branch True

label $l = (2, p)$
Covering label $l \iff$ Covering branch $\text{True}$

✓ sound & complete instrumentation w.r.t. LC
Direct instrumentation is not good enough

\[ 2^N \text{ paths} \]
Direct instrumentation is not good enough

Non-tightness 1

\[ P' \text{ has exponentially more paths than } P \]
Direct instrumentation is not good enough

Non-tightness 1

× P’ has exponentially more paths than P

Non-tightness 2

× Paths in P’ too complex
  ▶ at each label, require to cover p or to cover ¬p
  ▶ π’ covers up to N labels

Direct instrumentation

2^N paths

combination

N

pN

p1

1

2

True

False

True

False
Direct instrumentation is not good enough

✓ sound & complete instrumentation w.r.t. LC
× dramatic overhead [theory & practice]
Our approach

The DSE$^*$ algorithm

- Tight instrumentation $P^*$: totally prevents “complexification”
- Iterative Label Deletion: discards some redundant paths
- Both techniques can be implemented in black-box
DSE*: Tight Instrumentation

Covering label l ⇔ Covering exit(0)
DSE* : Tight Instrumentation

Covering label l ⇔ Covering exit(0)

✓ sound & complete instrumentation w.r.t. LC
Direct instrumentation

1 -> p1 (Diamond) -> True -> 2 -> pN (Diamond) -> True -> N
   False

Tight Instrumentation

1 -> non_det (Diamond) -> assert(p1) -> 2 -> non_det
   assert(pN) -> N
Direct instrumentation:

1

\(2^N\) paths

\(p_1\)

True False

2

\(p_N\)

True False

N

Tight Instrumentation:

1

\(N+1\) paths

non_det

\text{assert}(p_1)

2

non_det

\text{assert}(p_N)

N
DSE* : Tight Instrumentation (2)

Direct instrumentation

1

2

\(2^N\) paths

1

p1

True

False

2

pN

True

False

N

combination

Tight Instrumentation

1

N+1 paths

1

non_det

asser(p1)

2

no combination

no additional constraint

2

non_det

assert(pN)

N

Bardin et al ICST 2014 19/ 29
DSE*: Tight Instrumentation (2)

Direct instrumentation

\[ 2^N \text{ paths} \]

Tight Instrumentation

\[ N+1 \text{ paths} \]

- P* has (only) linearly more paths than P
- Paths in P* are simple: covers \( \leq 1 \) label
DSE*: Tight Instrumentation (2)

Direct instrumentation

- 2^N paths
- True
- False
- p1
- pN
- N

Tight Instrumentation

- N+1 paths
- non_det
- asser(p1)
- asser(pN)
- non_det

✔ sound & complete instrumentation w.r.t. LC
✔ no complexification of the search space
Observations

- we need to cover each label only once
- yet, DSE explores paths of \( P^* \) ending in already-covered labels
- burden DSE with "useless" paths w.r.t. \( L_C \)
Observations

- we need to cover each label only once
- yet, DSE explores paths of $P^*$ ending in already-covered labels
- burden DSE with “useless” paths w.r.t. LC

Solution : Iterative Label Deletion

- keep a cover status for each label
- symbolic execution ignores paths ending in a covered label
- dynamic execution updates cover status [truly requires DSE]

Implementation

- symbolic part : a slight modification of $P^*$
- dynamic part : a slight modification of $P'$
DSE*: Iterative Label Deletion

Observations

- we need to cover each label only once
- yet, DSE explores paths of $P^*$ ending in already-covered labels
- burden DSE with “useless” paths w.r.t. LC

Solution: Iterative Label Deletion

- keep a cover status for each label
- symbolic execution ignores paths ending in a covered label
- dynamic execution updates cover status [truly requires DSE]

Implementation

- symbolic part: a slight modification of $P^*$
- dynamic part: a slight modification of $P'$

Iterative Label Deletion is relatively complete w.r.t. LC
DSE* : Iterative Label Deletion (2)

1

label = (2, p)

2

1

non_det & uncovered

assert(p);

exit(0);

2
DSE*: Iterative Label Deletion (2)

- **P**
  - query next TD
  - DSE on P*
  - TD found for a new path
    - update
    - run TD on P'
  - no more TD
    - STOP

coverage info [initially empty]
The DSE\(^*\) algorithm

- **Tight instrumentation** \(P^*\): totally prevents “complexification”
- **Iterative Label Deletion**: discards some redundant paths
- **Both techniques** can be implemented in black-box
The DSE* algorithm

**Goals**

- OBJ1: generic specification mechanism for coverage criteria
- OBJ2: efficient integration into DSE
Outline

- Introduction
- Labels
- Efficient DSE for labels
- Experiments
- Conclusion
Experiments

Implementation

- inside PathCrawler
- follows DSE*
- search heuristics: “label-first DFS”
- run in deterministic mode

Goal of experiments

- evaluate DSE* versus DSE’
- evaluate overhead of handling labels

Benchmark programs

- 12 programs taken from standard DSE benchmarks (Siemens, Verisec, MediaBench) [beware: small programs]
- 3 coverage criteria: **CC, MCC, WM**
  [uncoverable labels not discarded]
Results

- DSE’ : 4 TO, max overhead 122x [excluding TO]
- DSE* : no TO, max overhead 7x (average : 2.4x)
- on gd_5-wm : 94s vs TO [1h30]
- also : DSE* achieves very high LC-coverage [> 90% on 28/36]
Results

- DSE’: 4 TO, max overhead 122x [excluding TO]
- DSE*: no TO, max overhead 7x (average: 2.4x)
- on gd_5-wm: 94s vs TO [1h30]
- also: DSE* achieves very high LC-coverage [> 90% on 28/36]
Results

- DSE’ : 4 TO, max overhead 122x [excluding TO]
- DSE*: no TO, max overhead 7x (average : 2.4x)
- on gd_5-wm : 94s vs TO [1h30]
- also: DSE* achieves very high LC-coverage [> 90% on 28/36]
Results

- DSE’ : 4 TO, max overhead 122x \textit{[excluding TO]}
- DSE* : no TO, max overhead 7x (average : 2.4x)
- on gd\_5\_wm : 94s vs TO [1h30]
- also : DSE* achieves very high \textbf{LC}-coverage [> 90% on 28/36]

Conclusion

- DSE* performs significantly better than DSE’
- The overhead of handling labels is kept reasonable
- still room for improvement
### A few results

<table>
<thead>
<tr>
<th></th>
<th>DSE</th>
<th>DSE'</th>
<th>DSE*</th>
</tr>
</thead>
<tbody>
<tr>
<td>utf8-5</td>
<td>108 loc</td>
<td>84 /</td>
<td>11,111</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>40s / 82 / 84</td>
</tr>
<tr>
<td>utf8-7</td>
<td>108 loc</td>
<td>84 /</td>
<td>81,133</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>576s / 82 / 84</td>
</tr>
<tr>
<td>tcas</td>
<td>124 loc</td>
<td>111 /</td>
<td>300,213</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>662s / 101 / 111</td>
</tr>
<tr>
<td>replace</td>
<td>100 loc</td>
<td>79 /</td>
<td>87,498</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>245s / 70 / 79</td>
</tr>
<tr>
<td>get_tag-6</td>
<td>240 loc</td>
<td>20 /</td>
<td>76,456</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3,011s / TO</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>76,456</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3,011s / TO</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>TO</td>
</tr>
</tbody>
</table>
Conclusion

Goal = extend DSE to a large class of coverage criteria

Results

- **Labels**: a well-defined and expressive specification mechanism for coverage criteria
- **DSE**: an efficient integration of labels into DSE
  - no exponential blowup of the search space
  - only a low overhead [huge savings w.r.t. related work]
Conclusion

Goal = extend DSE to a large class of coverage criteria

Results

- Labels: a well-defined and expressive specification mechanism for coverage criteria
- DSE*: an efficient integration of labels into DSE
  - no exponential blowup of the search space
  - only a low overhead [huge savings w.r.t. related work]

Dynamic Symbolic Execution [dart, cute, exe, sage, pex, klee, ...]

✓ very powerful approach to (white box) test generation
✓ arguably one of the most wide-spread use of formal methods in “common software”
Conclusion

Goal = extend DSE to a large class of coverage criteria

Results

□ Labels: a well-defined and expressive specification mechanism for coverage criteria

□ DSE*: an efficient integration of labels into DSE
  ▶ no exponential blowup of the search space
  ▶ only a low overhead [huge savings w.r.t. related work]

Dynamic Symbolic Execution [dart, cute, exe, sage, pex, klee, ...]

✓ very powerful approach to (white box) test generation

✓ arguably one of the most wide-spread use of formal methods in “common software”

✗ support only basic coverage criteria
Conclusion

Goal = extend DSE to a large class of coverage criteria

Results

- Labels: a well-defined and expressive specification mechanism for coverage criteria
- DSE*: an efficient integration of labels into DSE
  - no exponential blowup of the search space
  - only a low overhead [huge savings w.r.t. related work]

Dynamic Symbolic Execution [dart, cute, exe, sage, pex, klee, ...]

✓ very powerful approach to (white box) test generation
✓ arguably one of the most wide-spread use of formal methods in “common software”
✓ can be efficiently extended to a large class of coverage criteria
Program P

Instrumentation step

Annotated Program \(<P+L>\)

Programs \(P' \& P^*\)

Cover. crit.

Pathcrawler

Frama-C

Test generation (DSE)

Uncoverability check (static analysis)

coverage measurement