Symbolic Execution:
a journey from safety to security

Sébastien Bardin
joint work with
Nikolai Kosmatov, Mickaël Delahaye, Robin David

CEA LIST
Software Safety & Security Lab
(Paris-Saclay, France)
Context = verification / testing of "non safety-critical" programs

**Dynamic Symbolic Execution (DSE)** is very promising

- robust, no false alarm, scale [in some ways]

DSE can be efficiently lifted to coverage-oriented testing

- unified view of coverage criteria [ICST 14]
- a dedicated variant DSE* [ICST 14]
- infeasibility detection is feasible [ICST 15]
- tool LTest (Frama-C plugin) [TAP 14]

DSE can be applied to binary-level security analysis

- binary-level formal methods
- applications to vulnerabilities and reverse
- tool BINSEC [TACAS 15, SANER 16]
Introduction

About formal verification

- Between Software Engineering and Theoretical Computer Science
- Goal = proves correctness in a mathematical way

Key concepts: $M \models \varphi$

- $M$: semantic of the program
- $\varphi$: property to be checked
- $\models$: algorithmic check

Kind of properties

- absence of runtime error
- pre/post-conditions
- temporal properties
Industrial reality in some area, especially safety-critical domains

- hardware, aeronautics [airbus], railroad [metro 14], smartcards, drivers [Windows], certified compilers [CompCert] and OS [Sel4], etc.
Industrial reality in some areas, especially safety-critical domains
- hardware, aeronautics [airbus], railroad [metro 14], smartcards, drivers [Windows], certified compilers [CompCert] and OS [Sel4], etc.

Ex: Airbus

Verification of
- runtime errors [Astrée]
- functional correctness [Frama-C]
- numerical precision [Fluctuat]
- source-binary conformance [CompCert]
- resource usage [Absint]
Industrial reality in some area, especially safety-critical domains

- hardware, aeronautics [airbus], railroad [metro 14], smartcards, drivers [Windows], certified compilers [CompCert] and OS [Sel4], etc.

Ex: Microsoft

Verification of drivers [SDV]

- conformance to MS driver policy
- home developers
- and third-party developers
Industrial reality in some area, especially safety-critical domains

- hardware, aeronautics [airbus], railroad [metro 14], smartcards, drivers [Windows], certified compilers [CompCert] and OS [Sel4], etc.

Things like even software verification, this has been the Holy Grail of computer science for many decades but now in some very key areas, for example, driver verification we’re building tools that can do actual proof about the software and how it works in order to guarantee the reliability.

- Bill Gates (2002)
- Apply formal methods to less-critical software
- Very different context: no formal spec, less developer involvement, etc.

Some difficulties
- robustness \([w.r.t. \text{software constructs}]\)
- no place for false alarms
- scale
- no model, sometimes no source

Guidelines & trends
- find sweetspots \([\text{drivers}]\)
- manage abstractions
- reduction to logic
Dynamic Symbolic Execution [since 2004-2005: dart, cute, pathcrawler]

- a very powerful formal approach to verification and testing
- many tools and successful case-studies since mid 2000's
- arguably one of the most wide-spread use of formal methods

Still a formal approach $M \models \varphi$

- semantically-founded, well-understood guarantees [under-approximation]
- follows the “reduction to logic” principle

But very good properties

✓ no false alarm
✓ robustness
✓ scales [in some ways]
Dynamic Symbolic Execution [since 2004–2005: dart, cute, pathcrawler]

- a very powerful formal approach to verification and testing
- many tools and successful case-studies since mid 2000's
- arguably one of the most wide-spread use of formal methods

Still a formal approach  \( M \models \varphi \)

- semantically-founded, well-understood guarantees
- follows the “reduction to logic”

But very good properties

✓ no false alarm
✓ robustness
✓ scales [in some ways]

A few tools

- PathCrawler (2004)
- Cute, DART (2005)
- Exe (2006)
- Osmose [icst 08-09, stvr 11, qsic 13]
- SAGE (2008)
- Pex, Klee, S2E, ...
Outline

- Introduction
- DSE in a nutshell
- Coverage-oriented DSE
- Binary-level DSE
- Conclusion
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! [no false alarm]
- old idea, recent renew interest [requires powerful solvers]
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! [no false alarm]
- 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]
DSE in a nutshell

Path predicate

**Definition**

- consider a program $P$ on input $v$, and a given path $\sigma$
- a path predicate $\phi_\sigma$ for $\sigma$ is a formula s.t.
  $$v \models \phi_\sigma \Rightarrow P(v) \text{ follows } \sigma$$

- $\phi$ is intuitively the **logical conjunction of all branching conditions** encountered on that path
- $\phi$ is **correct** if any solution cover that path [**mandatory**]
- $\phi$ is **complete** if any input covering the path is a solution [**less important**]

$\rightarrow$ The path predicate is the key concept of symbolic execution
Usually easy in a forward manner [introduce new logical variables at each step]

<table>
<thead>
<tr>
<th>Loc</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>input(y,z)</td>
</tr>
<tr>
<td>1</td>
<td>w := y+1</td>
</tr>
<tr>
<td>2</td>
<td>x := w + 3</td>
</tr>
<tr>
<td>3</td>
<td>if (x &lt; 2 * z) [True branch]</td>
</tr>
<tr>
<td>4</td>
<td>if (x &lt; z) [False branch]</td>
</tr>
</tbody>
</table>

Path predicate \( (input \, Y_0 \, et \, Z_0) \)
Usually easy in a forward manner

<table>
<thead>
<tr>
<th>Loc</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>input(y,z)</td>
</tr>
<tr>
<td>1</td>
<td>w := y+1</td>
</tr>
<tr>
<td>2</td>
<td>x := w + 3</td>
</tr>
<tr>
<td>3</td>
<td>if (x &lt; 2 * z) [True branch]</td>
</tr>
<tr>
<td>4</td>
<td>if (x &lt; z) [False branch]</td>
</tr>
</tbody>
</table>

Path predicate (input $Y_0$ et $Z_0$)
$\top \land W_1 = Y_0 + 1$
**Usually easy** in a forward manner [introduce new logical variables at each step]

<table>
<thead>
<tr>
<th>Loc</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>input(y,z)</td>
</tr>
<tr>
<td>1</td>
<td>w := y+1</td>
</tr>
<tr>
<td>2</td>
<td>x := w + 3</td>
</tr>
<tr>
<td>3</td>
<td>if (x &lt; 2 * z) [True branch]</td>
</tr>
<tr>
<td>4</td>
<td>if (x &lt; z) [False branch]</td>
</tr>
</tbody>
</table>

Path predicate (input Y₀ et Z₀)

\[ \top \land W_1 = Y_0 + 1 \land X_2 = W_1 + 3 \]
**Usually easy** in a forward manner

[introduce new logical variables at each step]

<table>
<thead>
<tr>
<th>Loc</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>input(y,z)</td>
</tr>
<tr>
<td>1</td>
<td>w := y+1</td>
</tr>
<tr>
<td>2</td>
<td>x := w + 3</td>
</tr>
<tr>
<td>3</td>
<td>if (x &lt; 2 * z) [True branch]</td>
</tr>
<tr>
<td>4</td>
<td>if (x &lt; z) [False branch]</td>
</tr>
</tbody>
</table>

Path predicate (input $Y_0$ et $Z_0$)

$$\top \land W_1 = Y_0 + 1 \land X_2 = W_1 + 3 \land X_2 < 2 \times Z_0$$
Path predicate: how to compute it?

Usually easy in a forward manner

<table>
<thead>
<tr>
<th>Loc</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>input(y,z)</td>
</tr>
<tr>
<td>1</td>
<td>w := y+1</td>
</tr>
<tr>
<td>2</td>
<td>x := w + 3</td>
</tr>
<tr>
<td>3</td>
<td>if (x &lt; 2 * z) [True branch]</td>
</tr>
<tr>
<td>4</td>
<td>if (x &lt; z) [False branch]</td>
</tr>
</tbody>
</table>

Path predicate (input $Y_0$ et $Z_0$)

$\top \land W_1 = Y_0 + 1 \land X_2 = W_1 + 3 \land X_2 < 2 \times Z_0 \land X_2 \geq Z_0$
**DSE in a nutshell**

**Path predicate: how to compute it?**

Usually easy in a forward manner

[introduce new logical variables at each step]

<table>
<thead>
<tr>
<th>Loc</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>input(y,z)</td>
</tr>
<tr>
<td>1</td>
<td>w := y+1</td>
</tr>
<tr>
<td>2</td>
<td>x := w + 3</td>
</tr>
<tr>
<td>3</td>
<td>if (x &lt; 2 * z) [True branch]</td>
</tr>
<tr>
<td>4</td>
<td>if (x &lt; z) [False branch]</td>
</tr>
</tbody>
</table>

Path predicate (input Y₀ et Z₀)

\[ \top \land W₁ = Y₀ + 1 \land X₂ = W₁ + 3 \land X₂ < 2 \times Z₀ \land X₂ \geq Z₀ \]

Alternative:

let \( W₁ ≝ Y₀ + 1 \) in

let \( X₂ ≝ W₁ + 3 \) in

\( X₂ < 2 \times Z₀ \land X₂ \geq Z₀ \)
Let us consider: \( x := a + b \)

Try 1: only logical variables

\[ X_{n+1} = A_n + B_n \]
Let us consider: $x := a + b$

**Try 1: only logical variables**

$X_{n+1} = A_n + B_n$

memory model: disjoint variables \{A, B, X, \ldots\}
Let us consider: 

\[ x := a + b \]

**Try 1: only logical variables**

\[ X_{n+1} = A_n + B_n \]

memory model: disjoint variables \{A, B, X, \ldots\}

*does not handle pointers*
Let us consider: $x := a + b$

Try 2: add a memory state $M$ [$\approx$ logical array with store/load functions]

$$M' = \text{store} (M, \text{addr}(X), \text{load}(M, \text{addr}(A)) + \text{load}(M, \text{addr}(B)))$$
Let us consider: \( x := a + b \)

**Try 2: add a memory state** \( M \) \( \approx \) logical array with store/load functions

\[
M' = \text{store}(M, \text{addr}(X), \text{load}(M, \text{addr}(A)) + \text{load}(M, \text{addr}(B)))
\]

**memory model**: map \( \{ \text{Addr}_1 \mapsto A, \text{Addr}_2 \mapsto B, \ldots \} \)
Let us consider: \( x := a + b \)

**Try 2**: add a memory state \( M \) \([\approx \text{logical array with store/load functions}]\)

\[
M' = \text{store}(M, \text{addr}(X), \text{load}(M, \text{addr}(A)) + \text{load}(M, \text{addr}(B)))
\]

memory model: map \( \{\text{Addr}_1 \mapsto A, \text{Addr}_2 \mapsto B, \ldots\} \)

ok for pointers, but type-safe access only (Java vs C)
Let us consider: \( x := a + b \)

**Try 3: byte-level** \( M \) [here: 3 bytes]

\[
\begin{align*}
\text{let } \text{tmpA} &= \text{load}(M, \text{addr}(A)) \odot \text{load}(M, \text{addr}(A)+1) \odot \text{load}(M, \text{addr}(A)+2) \\
\text{and } \text{tmpB} &= \text{load}(M, \text{addr}(B)) \odot \text{load}(M, \text{addr}(B)+1) \odot \text{load}(M, \text{addr}(B)+2) \\
\text{in}
\text{let } nX = \text{tmpA} + \text{tmpB} \\
\text{in}
M' &= \text{store}
\text{store}
\text{store}(M, \text{addr}(X), nX[0]),
\text{addr}(X) + 1, nX[1]),
\text{addr}(X) + 2, nX[2])
\end{align*}
\]
Let us consider: \( x \ := \ a + b \)

Try 3: byte-level \( M \) [here: 3 bytes]

\[
\begin{align*}
let \ tmpA &= \text{load}(M,\text{addr}(A)) \circ \text{load}(M,\text{addr}(A)+1) \circ \text{load}(M,\text{addr}(A)+2) \\
and \ tmpB &= \text{load}(M,\text{addr}(B)) \circ \text{load}(M,\text{addr}(B)+1) \circ \text{load}(M,\text{addr}(B)+2) \\
in \\
let \ nX = tmpA + tmpB \\
in \\
M' = \text{store}( \\
\text{store}( \\
\text{store}(M, \text{addr}(X), nX[0]), \\
\text{addr}(X) + 1, nX[1]), \\
\text{addr}(X) + 2, nX[2])
\end{align*}
\]

ok for C, but complex formula
Path predicate in some theory $T$

Trade off on $T$

- highly expressive: easy path predicate computation, hard solving
- poorly expressive: hard path predicate computation, easier solving

Remarks

- conjunctive quantifier-free formula are sufficient
- requires solution synthesis

Current consensus: anything that fits into SMT solvers [Z3, CVC4, etc.]

- typical choices: array + LIA, array + bitvectors
- solvers are usually good enough [with proper preprocessing]
- yet, still some challenges: string, float, heavy memory manipulations
**DSE in a nutshell**

**The Symbolic Execution Loop**

**input**: a program $P$

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

- pick a path $\sigma \in \text{Paths}^{\leq k}(P)$ [key 1]
- compute a *path predicate* $\varphi_{\sigma}$ of $\sigma$ [key 2]
- solve $\varphi_{\sigma}$ for satisfiability [key 3 - use smt solvers]
- SAT(s)? get a new pair $< s, \sigma >$
- loop until no more path to cover
**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
DSE in a nutshell

The Symbolic Execution Loop

**input**: a program $P$

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

- pick a path $\sigma \in \text{Paths}^{\leq k}(P)$ [key 1]
- compute a *path predicate* $\varphi_{\sigma}$ of $\sigma$ [key 2]
- solve $\varphi_{\sigma}$ for satisfiability [key 3 - use smt solvers]
- SAT(s) ? get a new pair $< s, \sigma >$
- loop until no more path to cover

Bardin et al. Séminaire Confiance Numérique
DSE in a nutshell

The Symbolic Execution Loop

**input**: a program $P$

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

- pick a path $\sigma \in \text{Paths}^{\leq k}(P)$  
  [key 1]
- compute a path predicate $\varphi_\sigma$ of $\sigma$  
  [key 2]
- solve $\varphi_\sigma$ for satisfiability  
  [key 3 - use smt solvers]
- SAT(s) ? get a new pair $< s, \sigma >$
- loop until no more path to cover
DSE in a nutshell

The Symbolic Execution Loop

**input**: a program $P$

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

- pick a path $\sigma \in Paths_{\leq k}(P)$ [key 1]
- compute a path predicate $\varphi_\sigma$ of $\sigma$ [key 2]
- solve $\varphi_\sigma$ for satisfiability
- SAT(s)? get a new pair $< s, \sigma >$
- loop until no more path to cover [key 3 - use smt solvers]
DSE in a nutshell

The Symbolic Execution Loop

**input**: a program $P$

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

- pick a path $\sigma \in Paths^{\leq k}(P)$ [key 1]
- compute a *path predicate* $\varphi_\sigma$ of $\sigma$ [key 2]
- solve $\varphi_\sigma$ for satisfiability [key 3 - use smt solvers]
- SAT(s)? get a new pair $< s, \sigma >$
- loop until no more path to cover
input: a program $P$
output: a test suite $TS$ covering all feasible paths of $Paths^{\leq k}(P)$

- pick a path $\sigma \in Paths^{\leq k}(P)$ [key 1]
- compute a path predicate $\varphi_\sigma$ of $\sigma$ [key 2]
- solve $\varphi_\sigma$ for satisfiability [key 3 - use smt solvers]
- SAT(s)? get a new pair $< s, \sigma >$
- loop until no more path to cover

![Diagram of symbolic execution loop]
DSE in a nutshell

**The Symbolic Execution Loop**

**input**: a program $P$

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

- pick a path $\sigma \in Paths^{\leq k}(P)$ [key 1]
- compute a *path predicate* $\varphi_\sigma$ of $\sigma$ [key 2]
- solve $\varphi_\sigma$ for satisfiability [key 3 - use smt solvers]
- SAT(s)? get a new pair $<s, \sigma>$
- loop until no more path to cover

Under-approximation
- correct
- relatively complete

✓ No false alarm
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]

Concretization: force a symbolic variable to take its runtime value

- application 1: follow only feasible path for free
- application 2: correct approximation of "difficult" constructs [out of scope or too expensive to handle]
**DSE in a nutshell**

**About robustness (2)**

**Goal** = find input leading to ERROR
(assume we have only a solver for linear integer arith.)

\[ g(\text{int } x) \{ \text{return } x \times x; \} \]
\[ f(\text{int } x, \text{int } y) \{ \text{z=g(x); if (y == z) ERROR; else OK} \} \]

**Symbolic Execution**

- create a subformula \( z = x \times x \), out of theory **[FAIL]**

**Dynamic Symbolic Execution**

- first concrete execution with \( x=3, \ y=5 \) **[goto OK]**
- during path predicate computation, \( x \times x \) not supported
  
  \( x \) is concretized to 3 and \( z \) is forced to 9
- resulting path predicate: \( x = 3 \land z = 9 \land y = z \)
- a solution is found: \( x=3, \ y=9 \) **[goto ERROR]** **[SUCCESS]**
Dynamic Symbolic Execution [since 2004-2005: dart, cute, pathcrawler]

✓ no false alarm
✓ robustness
✓ scales [in some ways]

Many applications

- smart fuzzing and bug finding [Microsoft SAGE, Klee, Mayhem, etc.]
- completion of existing test suites
- exploit generation
- reverse engineering

Still many challenges

- scale and high coverage (loop, function calls)
- coverage-oriented testing [cf. after], target-oriented testing
- path selection, right trade-off between concrete and symbolic
Coverage-oriented DSE

Outline

- Introduction
- DSE in a nutshell
- Coverage-oriented DSE
- Binary-level DSE
- Conclusion
Coverage-oriented DSE – The problem

Context: white-box software testing

```c
int myfun(int ...) {
    ....
    }
else ...
return ...;
```

SPEC

```
blablabla
blablabla
blablabla
blablabla
```
Testing process

- Generate a test input
- Run it and check for errors
- Estimate coverage: if enough stop, else loop
Coverage-oriented DSE – The problem

Context: white-box software testing

Testing process
- Generate a test input
- Run it and check for errors
- Estimate coverage:
  if enough stop, else loop

Diagram:
- Input
  - Code: int myfun(int ...)
    - if (x <= b) {
      - ....
      - }
    - else ...
    - return ...;
  - Run & observe
    - Enough?
      - Yes -> Good!
      - No -> Problem!
    - No
  - Spec: blablabla blablabla blablabla blablabla
  - ???
Coverage-oriented DSE – The problem

Context: white-box software testing

Testing process
- Generate a test input
- Run it and check for errors
- Estimate coverage: if enough stop, else loop

Coverage criteria [decision, mcdc, etc.]
- systematic way of deriving test objectives
- major role: guide testing, decide when to stop, assess quality
- beware: lots of different coverage criteria
- beware: infeasible test requirements
Coverage-oriented DSE – The problem

Context: white-box software testing

Testing process

- Generate a test input
- Run it and check for errors
- Estimate coverage: if enough stop, else loop

Coverage criteria [decision, mcdc, etc.]

- systematic way of deriving test objectives
- major role: guide testing, decide when to stop, assess quality
- beware: lots of different coverage criteria
- beware: infeasible test requirements
DSE is GREAT for automating structural testing

✓ very powerful approach to (white box) test generation
✓ many tools and many successful case-studies since mid 2000’s
Coverage-oriented DSE – The problem

The problem: coverage-oriented DSE

DSE is GREAT for automating structural testing
✓ very powerful approach to (white box) test generation
✓ many tools and many successful case-studies since mid 2000’s

Yet, no real support for structural coverage criteria
[except path coverage and branch coverage]

Would be useful:
■ when required to produce tests achieving some criterion
■ for producing “good” tests for an external oracle
  [functional correctness, security, performance, etc.]
Coverage-oriented DSE – The problem

The problem: coverage-oriented DSE

DSE is GREAT for automating structural testing
✓ very powerful approach to (white box) test generation
✓ many tools and many successful case-studies since mid 2000’s

Yet, no real support for structural coverage criteria
[except path coverage and branch coverage]

Recent efforts [Active Testing, Augmented DSE, Mutation DSE]
- limited or unclear expressiveness
- explosion of the search space [APex: 272x avg, up to 2,000x]
Our goals and results

Goals: extend DSE to a large set of structural coverage criteria

- support these criteria in a unified way
- support these criteria in an efficient way
- detect (some) infeasible test requirements
Goals: extend DSE to a large set of structural coverage criteria
- support these criteria in a unified way
- support these criteria in an efficient way
- detect (some) infeasible test requirements

Results
✓ generic low-level encoding of coverage criteria [ICST 14]
✓ efficient variant of DSE for coverage criteria [ICST 14]
✓ sound and quasi-complete detection of infeasibility [ICST 15]
Coverage-oriented DSE – Labels

Outline

■ Introduction
■ DSE in a nutshell
■ Coverage-oriented DSE
  ▶ The problem
  ▶ Labels: a unified view of coverage criteria
  ▶ Efficient DSE for Labels
  ▶ The LTest testing framework
  ▶ Infeasible label detection
■ Binary-level DSE
■ Conclusion
Annotate programs with **labels**
- predicate attached to a specific program instruction

Label \((loc, \varphi)\) is covered if a test execution
- reaches the instruction at \(loc\)
- satisfies the predicate \(\varphi\)

**Good for us**
- can easily encode a large class of coverage criteria [see after]
- in the scope of standard program analysis techniques
Coverage-oriented DSE – Labels

Simulation of standard coverage criteria

**Decision Coverage (DC)**

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

```
statement_1;
// l1: x==y && a<b
// l2: !(x==y && a<b)
if (x==y && a<b)
    {...};
statement_3;
```
Coverage-oriented DSE – Labels

Simulation of standard coverage criteria

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)
Coverage-oriented DSE – Labels

Simulation of standard coverage criteria

Coverage-oriented DSE – Labels

Simulation of standard coverage criteria

Multiple-Condition Coverage (MCC)
OBJ: generic specification mechanism for coverage criteria

- **IC, DC, FC, CC, MCC**
- **GACC** (a variant of MCDC)
- large part of Weak Mutations
- Input Domain Partition
- Run-Time Error
Coverage-oriented DSE – Labels

Simulation of standard coverage criteria

OBJ : generic specification mechanism for coverage criteria

- IC, DC, FC, CC, MCC
- GACC (a variant of MCDC)
- large part of Weak Mutations
- Input Domain Partition
- Run-Time Error

Out of scope:
- strong mutations, MCDC
- (side-effect weak mutations)
Outline

- Introduction
- DSE in a nutshell
- Coverage-oriented DSE
  - The problem
  - Labels: a unified view of coverage criteria
  - Efficient DSE for Labels
  - The LTest testing framework
  - Infeasible label detection
- Binary-level DSE
- Conclusion
Coverage-oriented DSE – DSE*

Direct instrumentation

Coverage label $l \iff$ Covering branch $\text{True}$

- ✓ sound & complete instrumentation
- ✗ complexification of the search space [#paths, shape of paths]
- ✗ dramatic overhead [theory & practice] [Apex : avg 272x, max 2000x]
Direct instrumentation is not good enough

Non-tightness 1

\[ \times \quad \text{P'} \text{ has exponentially more paths than P} \]
Coverage-oriented DSE – DSE*

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
Coverage-oriented DSE – DSE*

Our approach

The DSE* algorithm [ICST 14]

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

DSE*: Tight Instrumentation

Covering label \( l \) ⇔ Covering exit(0)

✓ sound & complete instrumentation
✓ no complexification of the search space
Coverage-oriented DSE – DSE*

DSE* : Tight Instrumentation (2)

Direct instrumentation

1

\[ p_1 \]

True False

2

\[ p_N \]

True False

N

Tight Instrumentation

1

\[ \text{non}_{-}\text{det} \]

\[ \text{assert}(p_1) \]

2

\[ \text{non}_{-}\text{det} \]

\[ \text{assert}(p_N) \]

N
Coverage-oriented DSE – DSE*  
DSE* : Tight Instrumentation (2)

**Direct instrumentation**

1

$p_1$

2

$p_N$

N

$2^N$ paths

**Tight Instrumentation**

1

non_det

assert($p_1$)

2

non_det

assert($p_N$)

N

N+1 paths
Coverage-oriented DSE – DSE*

DSE*: Tight Instrumentation (2)

Direct instrumentation

Tight Instrumentation

2^N paths

N+1 paths

no combination

no additional constraint

asser(p1)

assert(pN)
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. label coverage

Solution: Iterative Label Deletion

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

Iterative Label Deletion is relatively complete w.r.t. label coverage
Coverage-oriented DSE – DSE*  

DSE* : Iterative Label Deletion (2)
Coverage-oriented DSE – DSE*

Experiments

Benchmark: Standard (test generation) benchmarks [Siemens, Verisec, Mediabench]
- 12 programs (50-300 loc), 3 criteria (CC, MCC, WM)
- 26 pairs (program, coverage criterion)
- 1,270 test requirements

Performance overhead

<table>
<thead>
<tr>
<th></th>
<th>DSE</th>
<th>DSE’</th>
<th>DSE*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min</td>
<td>×1</td>
<td>×1.02</td>
<td>×0.49</td>
</tr>
<tr>
<td>Median</td>
<td>×1</td>
<td>×1.79</td>
<td>×1.37</td>
</tr>
<tr>
<td>Max</td>
<td>×1</td>
<td>×122.50</td>
<td>×7.15</td>
</tr>
<tr>
<td>Mean</td>
<td>×1</td>
<td>×20.29</td>
<td>×2.15</td>
</tr>
<tr>
<td>Timeouts</td>
<td>0</td>
<td>5 *</td>
<td>0</td>
</tr>
</tbody>
</table>

*: TO are discarded for overhead computation
cherry picking: 94s vs TO [1h30]
Coverage-oriented DSE – DSE*

Experiments

Benchmark: Standard (test generation) benchmarks [Siemens, Verisec, Mediabench]

- 12 programs (50-300 loc), 3 criteria (CC, MCC, WM)
- 26 pairs (program, coverage criterion)
- 1,270 test requirements

Coverage

<table>
<thead>
<tr>
<th></th>
<th>Random</th>
<th>DSE</th>
<th>DSE*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min</td>
<td>37%</td>
<td>61%</td>
<td>62%</td>
</tr>
<tr>
<td>Median</td>
<td>63%</td>
<td>90%</td>
<td>95%</td>
</tr>
<tr>
<td>Max</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Mean</td>
<td>70%</td>
<td>87%</td>
<td>90%</td>
</tr>
</tbody>
</table>

vs DSE: +39% coverage on some examples
Conclusion

- DSE* performs significantly better than DSE’
- The overhead of handling labels is kept reasonable
- high coverage, better than DSE
Introduction

DSE in a nutshell

Coverage-oriented DSE
  ▶ The problem
  ▶ Labels: a unified view of coverage criteria
  ▶ Efficient DSE for Labels
  ▶ The LTest testing framework
  ▶ Infeasible label detection

Binary-level DSE

Conclusion
LTest: All-in-one automated testing toolkit for C

- plugin of the FRAMA-C verification platform (open-source)
- based on PATHCRAWLER for test generation
- the plugin itself is open-source except test generation
Coverage-oriented DSE – LTest

**LTest overview**

- **Supported criteria**
  - DC, CC, MCC, GACC
  - FC, IDC, WM

- **Encoded with labels** [ICST 2014]
  - managed in a unified way
  - rather easy to add new ones
DSE* procedure [ICST 2014]

- DSE with native support for labels
- extension of PATHCRAWLER
Reuse static analyzers from FRAMA-C

- sound detection!
- several modes: VA, WP, VA ⊕ WP
Coverage-oriented DSE – LTest

**LTest overview**

**Reuse static analyzers from Frama-C**
- sound detection!
- several modes: VA, WP, VA ⊕ WP

**Service cooperation**
- share label statuses
- Covered, Infeasible, ?
Coverage-oriented DSE – Infeasibility detection

Outline

- Introduction
- DSE in a nutshell
- Coverage-oriented DSE
  - The problem
  - Labels: a unified view of coverage criteria
  - Efficient DSE for Labels
  - The LTest testing framework
  - Infeasible label detection
- Binary-level DSE
- Conclusion
Infeasible test objectives

- waste generation effort, imprecise coverage ratios
- cause: structural coverage criteria are ... structural
- can be a nightmare [mcdc, mutations]
- detecting infeasible test requirements is undecidable

Basic ideas

- rely on existing sound verification methods
  - label \((loc, \varphi)\) infeasible \(\iff\) assertion \((loc, \neg\varphi)\) invariant
- grey-box combination of existing approaches [VA ⊕ WP]
Coverage-oriented DSE – Infeasibility detection

Overview of the approach

- labels as a unifying criteria
- label infeasibility ⇔ assertion validity
- s-o-t-a verification for assertion checking

- only soundness is required (verif)
  - label encoding not required to be perfect
  - mcddc and strong mutation ok
Two broad categories of sound assertion checkers

- **Forward abstract interpretation (VA)** [state approximation]
  - compute an invariant of the program
  - then, analyze all assertions (labels) in one run

- **Weakest precondition calculus (WP)** [goal-oriented]
  - perform a dedicated check for each assertion
  - a single check usually easier, but many of them
Two broad categories of sound assertion checkers

- **Forward abstract interpretation (VA)** [state approximation]
  - compute an invariant of the program
  - then, analyze all assertions (labels) in one run

- **Weakest precondition calculus (WP)** [goal-oriented]
  - perform a dedicated check for each assertion
  - a single check usually easier, but many of them

The paper is more generic
### Focus: checking assertion validity (2)

<table>
<thead>
<tr>
<th>Feature</th>
<th>VA</th>
<th>WP</th>
</tr>
</thead>
<tbody>
<tr>
<td>sound for assert validity</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>blackbox reuse</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>local precision</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>calling context</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>calls / loop effects</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>global precision</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>scalability wrt. #labels</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>scalability wrt. code size</td>
<td>×</td>
<td>✓</td>
</tr>
</tbody>
</table>

**hypothesis:** VA is interprocedural
int main() {
    int a = nondet(0 .. 20);
    int x = nondet(0 .. 1000);
    return g(x, a);
}

int g(int x, int a) {

    int res;
    if(x+a >= x)
        res = 1;
    else
        res = 0;
    //l1: res == 0
}
VA and WP may fail

```c
int main () {
    int a = nondet (0 .. 20);
    int x = nondet (0 .. 1000);
    return g(x, a);
}

int g(int x, int a) {
    int res;
    if (x + a >= x)
        res = 1;
    else
        res = 0;
    // @assert res != 0
}
```
Coverage-oriented DSE – Infeasibility detection

VA and WP may fail

```c
int main () {
    int a = nondet(0 .. 20);
    int x = nondet(0 .. 1000);
    return g(x,a);
}

int g(int x, int a) {
    int res;
    if(x+a >= x)
        res = 1;
    else
        res = 0;
    //@assert res != 0 // both VA and WP fail
}
```
Goal = get the best of the two worlds

- idea: VA passes to WP the global info. it lacks

Which information, and how to transfer it?

- VA computes (internally) some form of invariants
- WP naturally takes into account assumptions //@ assume

Solution: VA exports its invariants on the form of WP-assumptions
Coverage-oriented DSE – Infeasibility detection

Proposal: VA ⊕ WP (1)

Goal = get the best of the two worlds
- idea: VA passes to WP the global info. it lacks

Which information, and how to transfer it?
- VA computes (internally) some form of invariants
- WP naturally takes into account assumptions //@ assume

solution VA exports its invariants on the form of WP-assumptions

- Should work for any reasonable VA and WP engine
- No manually-inserted WP assumptions
```c
int main () {
    int a = nondet (0 .. 20);
    int x = nondet (0 .. 1000);
    return g(x,a);
}

int g(int x, int a) {

    int res;
    if(x+a >= x)
        res = 1;
    else
        res = 0;
    //l1: res == 0
}
```
int main() {
    int a = nondet(0 .. 20);
    int x = nondet(0 .. 1000);
    return g(x, a);
}

int g(int x, int a) {
    //@assume 0 \leq a \leq 20
    //@assume 0 \leq x \leq 1000
    int res;
    if(x+a >= x)
        res = 1;
    else
        res = 0;
    //@assert res != 0
}
int main() {
    int a = nondet(0 .. 20);
    int x = nondet(0 .. 1000);
    return g(x,a);
}

int g(int x, int a) {
    //-@assume 0 <= a <= 20
    //-@assume 0 <= x <= 1000
    int res;
    if(x+a >= x)
        res = 1;
    else
        res = 0;
    //-@assert res != 0       // VA ⊕ WP succeeds
}
Exported invariants

- only names appearing in the program (params, lhs, vars)
  - independent from memory size

- non-relational information numerical constraints (sets, intervals, congruence)
  - linear in the number of names

- only numerical information
  - sets, intervals, congruence
Proposal: VA ⊕ WP (3)

**Soundness** ok as long as VA is sound

**Exhaustivity** of “export” only affect deductive power [full export has only a little overhead]
## Summary

<table>
<thead>
<tr>
<th></th>
<th>VA</th>
<th>WP</th>
<th>VA ⊕ WP</th>
</tr>
</thead>
<tbody>
<tr>
<td>sound for assert validity</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>blackbox reuse</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>local precision</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>calling context</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>calls / loop effects</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>global precision</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>scalability wrt. #labels</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>scalability wrt. code size</td>
<td>×</td>
<td>✓</td>
<td>?</td>
</tr>
</tbody>
</table>
Detection power

Reuse the same benchmarks [Siemens, Verisec, Mediabench]

- 12 programs (50-300 loc), 3 criteria (CC, MCC, WM)
- 26 pairs (program, coverage criterion)
- 1,270 test requirements, 121 infeasible ones

<table>
<thead>
<tr>
<th></th>
<th>#Lab</th>
<th>#Inf</th>
<th>VA</th>
<th>WP</th>
<th>VA ⊕ WP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#d</td>
<td>%d</td>
<td>#d</td>
<td>%d</td>
<td>#d</td>
</tr>
<tr>
<td>Total</td>
<td>1,270</td>
<td>121</td>
<td>84</td>
<td>69%</td>
<td>73</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>60%</td>
<td></td>
<td>118</td>
</tr>
<tr>
<td>Min</td>
<td>0</td>
<td>0</td>
<td>0%</td>
<td>0%</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>67%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max</td>
<td>29</td>
<td>29</td>
<td>100%</td>
<td>15</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>100%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mean</td>
<td>4.7</td>
<td>3.2</td>
<td>63%</td>
<td>2.8</td>
<td>4.5</td>
</tr>
<tr>
<td></td>
<td>95%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#d : number of detected infeasible labels
%d : ratio of detected infeasible labels
Coverage-oriented DSE – Infeasibility detection

Detection power

Reuse the same benchmarks [Siemens, Verisec, Mediabench]

- 12 programs (50-300 loc), 3 criteria (CC, MCC, WM)
- 26 pairs (program, coverage criterion)
- 1,270 test requirements, 121 infeasible ones

<table>
<thead>
<tr>
<th></th>
<th>#Lab</th>
<th>#Inf</th>
<th>VA</th>
<th>WP</th>
<th>VA ⊕ WP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#d</td>
<td>%d</td>
<td>#d</td>
<td>%d</td>
<td>#d</td>
</tr>
<tr>
<td>Total</td>
<td>1,270</td>
<td>121</td>
<td>84</td>
<td>69%</td>
<td>73</td>
</tr>
<tr>
<td></td>
<td>121</td>
<td>84</td>
<td>69%</td>
<td>73</td>
<td>118</td>
</tr>
<tr>
<td>Min</td>
<td>0</td>
<td>0</td>
<td>0%</td>
<td>0%</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0%</td>
<td>0%</td>
<td>2</td>
</tr>
<tr>
<td>Max</td>
<td>29</td>
<td>29</td>
<td>100%</td>
<td>15</td>
<td>29</td>
</tr>
<tr>
<td>Mean</td>
<td>4.7</td>
<td>3.2</td>
<td>63%</td>
<td>2.8</td>
<td>4.5</td>
</tr>
</tbody>
</table>

#d : number of detected infeasible labels
%d : ratio of detected infeasible labels

- VA ⊕ WP achieves almost perfect detection
- detection speed is reasonable [≤ 1s/obj.]

Bardin et al. Séminaire Confiance Numérique
Coverage-oriented DSE – Infeasibility detection

Impact on test generation

report more accurate coverage ratio

<table>
<thead>
<tr>
<th>Detection method</th>
<th>Coverage ratio reported by DSE*</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>None</td>
</tr>
<tr>
<td>Total</td>
<td>90.5%</td>
</tr>
<tr>
<td>Min</td>
<td>61.54%</td>
</tr>
<tr>
<td>Max</td>
<td>100.00%</td>
</tr>
<tr>
<td>Mean</td>
<td>91.10%</td>
</tr>
</tbody>
</table>

* preliminary, manual detection of infeasible labels
optimisation : speedup test generation : take care!

<table>
<thead>
<tr>
<th>DSE*-OPT vs DSE*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min.</td>
</tr>
<tr>
<td>Max.</td>
</tr>
<tr>
<td>Mean</td>
</tr>
</tbody>
</table>

Ideal speedup

<table>
<thead>
<tr>
<th>RT(1s) + LUnCOV + DSE*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min</td>
</tr>
<tr>
<td>Max</td>
</tr>
<tr>
<td>Mean</td>
</tr>
</tbody>
</table>

RT : random testing
Speedup wrt. DSE* alone

Bardin et al.
DSE for binary code analysis

Outline

- Introduction
- DSE in a nutshell
- Coverage-oriented DSE
- Binary-level DSE
- Conclusion
Advantages over source-level analysis

-Executable always available
-No "compiler gap"
-Open new fields of applications [cots, mobile code]
-Especially security [vulnerabilities, reverse, malware]

But more challenging

- Low-level data [including raw memory]
- Low-level control [very specific]
- Which semantics?
DSE for binary code analysis

Basic challenge: modelling

Example: x86
- more than 1,000 instructions
  - ≈ 400 basic
  - + float, interrupts, mmx
- many side-effects
- error-prone decoding
  - addressing mode, prefixes, ...
DSE for binary code analysis

Basic challenge: safe CFG recovery [reverse]

Input

- an executable code (array of bytes)
- an initial address
- a basic decoder: file × address $\mapsto$ instruction × size

Output: (surapprox of) the program CFG [basic artifact for verif]

- question: successor of $\langle$addr: goto a $\rangle$? [chicken-and-egg issue]
A platform for binary-level formal analysis
. open-source, developed mostly in OCaml
. developed within the BINSEC project [CEA, IRISA, LORIA, VERIMAG]
. intermediate representation [cav 11, tacas 15] + formal analysis
DSE for binary code analysis
The BINSEC platform

- lhs := rhs
- goto addr, goto expr
- ite(cond)? goto addr : goto addr'
- goto expr
- assume, assert, nondet, malloc, free

Simulation:
- Flat, regions, low-level regions semantics
- Dynamic disassembly

Static analysis:
- Generic fixpoint computation
- Interleaved CFG recovery (closed/degraded mode)
DSE for binary code analysis

The BINSEC platform

- ELF loader, x86 decoder
- 460/500 instructions: 380/380 basic, 80/120 SIMD, no float/system
- prefixes: op size, addr size, repetition, segments

Tested on Unix Coreutils, Windows malware, Verisec/Juliet, opcode32.asm, etc.
DSE for binary code analysis

The BINSEC platform

Disassembly
- Linear / Recursive
- Linear and Recursive
- Dynamic

Simplifications
- Instruction-level
- Intra-bloc
- Inter-blocs

Simulation:
- Flat, regions, low-level regions semantics
- Dynamic disassembly

Decoding + instruction-level and block-level simplification

stubs @replace, @insert

DBA Stub
DSE for binary code analysis

The BINSEC platform

Static analysis

- generic fixpoint computation
- safe CFG recovery [vmcai 11]
- basic features [non-relational domains, unrolling, etc.]

DSE

- whole dse engine [path pred, solving, path selection]
- generic path selection [in progress]
- optimized path predicate construction [in progress]
- start to scale [coreutils, malwares]
Example: solving FlareOn #1

**Context**: FlareOn, set of challenges published by FireEye every year

**Goal**: Finding a 24-character key *(simple decryption loop)*

input string $\mapsto \{\text{success, failure}\}$
Context: FlareOn, set of challenges published by FireEye every year
Goal: Finding a 24-character key (simple decryption loop)
Context: FlareOn, set of challenges published by FireEye every year

Goal: Finding a 24-character key (simple decryption loop)

BINSE/SE can help:
- iterate over all paths until success
- yield the given key: bunny_sl0pe@flare-on.com
Example: solving FlareOn #1

Context: FlareOn, set of challenges published by FireEye every year
Goal: Finding a 24-character key (simple decryption loop)

BINSE/SE can help:
- Iterate over all paths until success
- Yield the given key: bunny_sl0pe@flare-on.com

Other reverse tasks: dynamic jumps, call/ret
Outline

- Introduction
- DSE in a nutshell
- Coverage-oriented DSE
- Binary-level DSE
- Conclusion
Dynamic Symbolic Execution is very promising

- robust, no false alarm, scale [in some ways]

Can be efficiently lifted to coverage-oriented testing

- unified view of coverage criteria [ICST 14, TAP 14]
- a dedicated variant DSE* [ICST 14]
- infeasibility detection is feasible [ICST 15]

Can help for binary-level security analysis

- applications to vulnerabilities and reverse [SANER 16]
Focus: Simulation of Weak Mutations

- 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 (1)

**Mutant M1**

```plaintext
statement i-1;
x := f(d);
y := e;
statement i+2;
```

**Program P**

```plaintext
statement i-1;
x := d;
y := e;
statement i+2;
```

**Mutant M2**

```plaintext
statement i-1;
x := d;
y := g(e);
statement i+2;
```

**Program with label**

```plaintext
statement i-1:
x := d;
//d != f(d)
y := e;
//e != g(e)
statement i+2;
```

**labels:**
- predicates
- no side-effects
- just monitoring
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
One label per mutant

Mutation inside a statement

**Theorem**

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

Mutation inside a decision

- if (cond) → if (cond’)
  - add label: $\text{cond} \oplus \text{cond’}$

**Beware**: no side-effect inside labels
int fun(int a, int b, int c) {
//@assume a [...]  
//@assume b [...]  
//@assume c [...]  
int x=c;

//@assert a < b
if(a < b)
  {...}
else
  {...}
}
int fun(int a, int b, int c) {

    int x=c;
    //@assume a [...]
    //@assume b [...]
    //@assert a < b
    if(a < b)
    {
    }
    else
    {
    }
}
int fun(int a, int b, int c) {
//@assume a [...]
//@assume b [...]
//@assume c [...]
int x = c;
//@assume x [...]
//@assume a [...]
//@assume b [...]
//@assert a < b
if (a < b)
    {...}
else
    {...}
}
Conclusion – Bonus

Invariant export strategies

```c
int fun(int a, int b, int c) {
    //@assume a [...]
    //@assume b [...]
    //@assume c [...]
    int x=c;
    //@assume x [...]
    //@assume a [...]
    //@assume b [...]
    //@assert a < b
    if(a < b)
        {...}
    else
        {...}
}
```

Conclusion: Complete annotation very slight overhead (but label annotation experimentaly the best trade-off).