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)

Take away

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

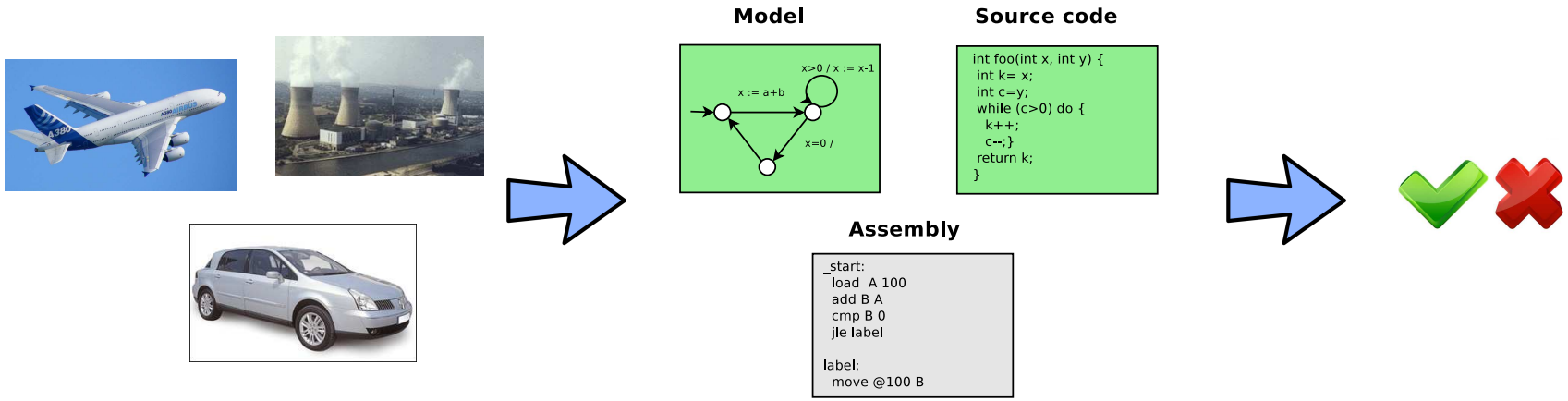
- unified view of coverage criteria [ICST 14]
- a dedicated variant DSE* [ICST 14]
- infeasibility detection is feasible [ICST 15]
- tool LTest (Frama-C plugin) [TAP 14]

DSE can be applied to binary-level security analysis

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

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
- φ : property to be checked
- $\models$: algorithmic check

Kind of properties

- absence of runtime error
- pre/post-conditions
- temporal properties

Success stories

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.

Success stories

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 : Airbus

Verification of

- runtime errors [Astrée]
- functional correctness [Frama-C]
- numerical precision [Fluctuat]
- source-binary conformance [CompCert]
- ressource usage [Absint]

Success stories

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

Success stories

Industrial reality in some area, especially safety-critical domains

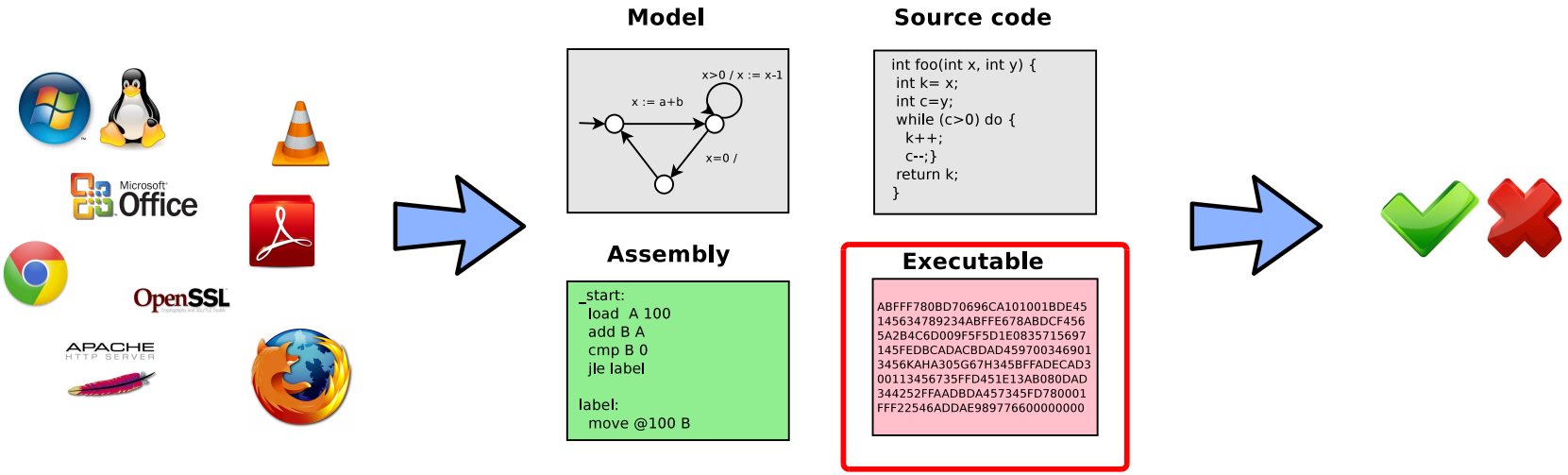
- hardware, aeronautics [airbus], railroad [metro 14], smartcards, drivers [Windows], certified compilers [CompCert] and OS [Sel4], etc.

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)

Next big challenge

- Apply formal methods to less-critical software
- Very different context : no formal spec, less developer involvement, etc.



Some difficulties

- robustness [w.r.t. software constructs]
- no place for false alarms
- scale
- no model, sometimes no source

Guidelines & trends

- find sweetspots [drivers]
- manage abstractions
- reduction to logic

Introducing DSE

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]

Introducing DSE

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-unde
- 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

(Dynamic) Symbolic Execution

Symbolic Execution [King 70's]

- consider a program P on input v, and a given path σ
- a path predicate φ_σ for σ is a formula s.t.

$$\mathtt{v} \models \varphi_\sigma \Rightarrow \mathtt{P}(\mathtt{v}) \text{ follows } \sigma$$

- can be used for bounded-path testing ! [no false alarm]
- old idea, recent renew interest [requires powerful solvers]

(Dynamic) Symbolic Execution

Symbolic Execution [King 70's]

- consider a program P on input v, and a given path σ
- a path predicate φ_σ for σ is a formula s.t.

$$\mathtt{v} \models \varphi_\sigma \Rightarrow \mathtt{P}(\mathtt{v}) \text{ follows } \sigma$$

- 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]

Path predicate

Definition

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

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

$\rightarrow$ The path predicate is the key concept of symbolic execution

Path predicate : how to compute it ?

Usually easy in a forward manner [introduce new logical variables at each step]

Loc	Instruction
0	input(y,z)
1	w := y+1
2	x := w + 3
3	if (x < 2 * z) [True branch]
4	if (x < z) [False branch]

Path predicate (input Y_0 et Z_0)

$\top$

Path predicate : how to compute it ?

Usually easy in a forward manner [introduce new logical variables at each step]

Loc	Instruction
0	input(y,z)
1	w := y+1
2	x := w + 3
3	if (x < 2 * z) [True branch]
4	if (x < z) [False branch]

Path predicate (input Y_0 et Z_0)

$\top \wedge W_1 = Y_0 + 1$

Path predicate : how to compute it ?

Usually easy in a forward manner [introduce new logical variables at each step]

Loc	Instruction
0	input(y,z)
1	w := y+1
2	x := w + 3
3	if (x < 2 * z) [True branch]
4	if (x < z) [False branch]

Path predicate (input Y_0 et Z_0)

$\top \wedge W_1 = Y_0 + 1 \wedge X_2 = W_1 + 3$

Path predicate : how to compute it ?

Usually easy in a forward manner [introduce new logical variables at each step]

Loc	Instruction
0	input(y,z)
1	w := y+1
2	x := w + 3
3	if (x < 2 * z) [True branch]
4	if (x < z) [False branch]

Path predicate (input Y_0 et Z_0)

$$\top \wedge W_1 = Y_0 + 1 \wedge X_2 = W_1 + 3 \wedge X_2 < 2 \times Z_0$$

Path predicate : how to compute it ?

Usually easy in a forward manner [introduce new logical variables at each step]

Loc	Instruction
0	input(y,z)
1	w := y+1
2	x := w + 3
3	if (x < 2 * z) [True branch]
4	if (x < z) [False branch]

Path predicate (input Y_0 et Z_0)

$$\top \wedge W_1 = Y_0 + 1 \wedge X_2 = W_1 + 3 \wedge X_2 < 2 \times Z_0 \wedge X_2 \geq Z_0$$

Path predicate : how to compute it ?

Usually easy in a forward manner [introduce new logical variables at each step]

Loc	Instruction
0	input(y,z)
1	w := y+1
2	x := w + 3
3	if (x < 2 * z) [True branch]
4	if (x < z) [False branch]

Path predicate (input Y_0 et Z_0)

$$\top \wedge W_1 = Y_0 + 1 \wedge X_2 = W_1 + 3 \wedge X_2 < 2 \times Z_0 \wedge X_2 \geq Z_0$$

Alternative :

let $W_1 \triangleq Y_0 + 1$ in
  let $X_2 \triangleq W_1 + 3$ in
    $X_2 < 2 \times Z_0 \wedge X_2 \geq Z_0$

Path predicate : how to compute it ? (2)

Let us consider : `x := a + b`

Try 1 : only logical variables

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

Path predicate : how to compute it ? (2)

Let us consider : `x := a + b`

Try 1 : only logical variables

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

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

Path predicate : how to compute it ? (2)

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

Path predicate : how to compute it ? (2)

Let us consider : `x := a + b`

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

$$M' = store(M, addr(X), load(M, addr(A)) + load(M, addr(B)))$$

Path predicate : how to compute it ? (2)

Let us consider : `x := a + b`

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

$M' = store(M, addr(X), load(M, addr(A)) + load(M, addr(B)))$

memory model : map $\{Addr_1 \mapsto A, Addr_2 \mapsto B, \ldots\}$

Path predicate : how to compute it ? (2)

Let us consider : `x := a + b`

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

$M' = store(M, addr(X), load(M, addr(A)) + load(M, addr(B)))$

memory model : map $\{Addr_1 \mapsto A, Addr_2 \mapsto B, \ldots\}$

ok for pointers, but type-safe access only (Java vs C)

Path predicate : how to compute it ? (2)

Let us consider : `x := a + b`

Try 3 : byte-level M [here : 3 bytes]

$$
\begin{array}{l}
\textit{let } tmpA = load(M,addr(A)) \ @ \ load(M,addr(A)+1) \ @ \ load(M,addr(A)+2) \\
\textit{and } tmpB = load(M,addr(B)) \ @ \ load(M,addr(B)+1) \ @ \ load(M,addr(B)+2) \\
\textit{in} \\
\textit{let } nX = tmpA+tmpB \\
\textit{in} \\
\quad M' = store(\\
\qquad store(\\
\qquad\quad store(M, addr(X), nX[0]), \\
\qquad\quad addr(X)+1, nX[1]), \\
\qquad addr(X)+2, nX[2])
\end{array}
$$

Path predicate : how to compute it ? (2)

Let us consider : `x := a + b`

Try 3 : byte-level M [here : 3 bytes]

$$
\begin{array}{l}
\textit{let } tmpA = load(M,addr(A)) \ @ \ load(M,addr(A)+1) \ @ \ load(M,addr(A)+2) \\
\textit{and } tmpB = load(M,addr(B)) \ @ \ load(M,addr(B)+1) \ @ \ load(M,addr(B)+2) \\
\textit{in} \\
\textit{let } nX = tmpA+tmpB \\
\textit{in} \\
\quad M' = store(\\
\qquad store(\\
\qquad\quad store(M, addr(X), nX[0]), \\
\qquad\quad addr(X)+1, nX[1]), \\
\qquad addr(X)+2, nX[2])
\end{array}
$$

ok for C, but complex formula

Path predicate : how to solve it ?

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

The Symbolic Execution Loop

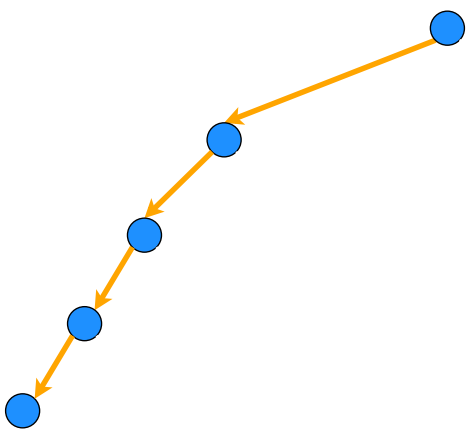
input : a program P
output : a test suite *TS* covering all feasible paths of $Paths^{\leq k}(\mathtt{P})$

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

The Symbolic Execution Loop

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

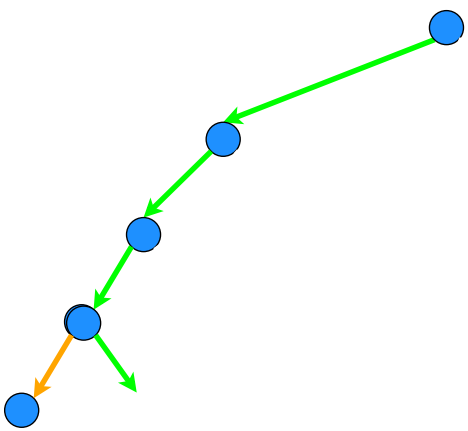
- pick a path $\sigma \in Paths^{\leq k}(\mathtt{P})$ [key 1]
- compute a *path predicate* φ_σ of σ [key 2]
- solve φ_σ for satisfiability [key 3 - use smt sovlers]
- SAT(s) ? get a new pair $< \mathrm{s}, \sigma >$
- loop until no more path to cover



The Symbolic Execution Loop

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

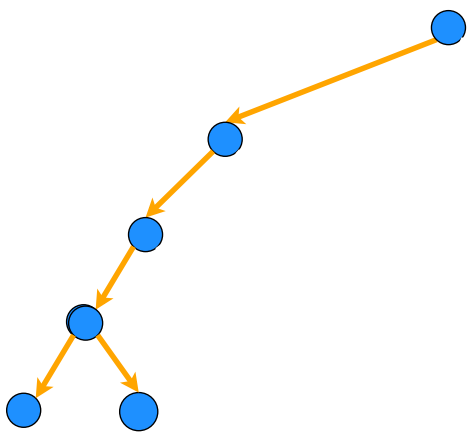
- pick a path $\sigma \in Paths^{\leq k}(\mathtt{P})$ [key 1]
- compute a *path predicate* φ_σ of σ [key 2]
- solve φ_σ for satisfiability [key 3 - use smt sovlers]
- SAT(s) ? get a new pair $< \mathrm{s}, \sigma >$
- loop until no more path to cover



The Symbolic Execution Loop

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

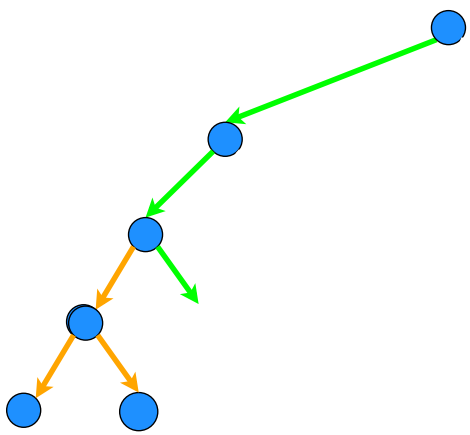
- pick a path $\sigma \in Paths^{\leq k}(\mathtt{P})$ [key 1]
- compute a *path predicate* φ_σ of σ [key 2]
- solve φ_σ for satisfiability [key 3 - use smt sovlers]
- SAT(s) ? get a new pair $< \mathrm{s}, \sigma >$
- loop until no more path to cover



The Symbolic Execution Loop

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

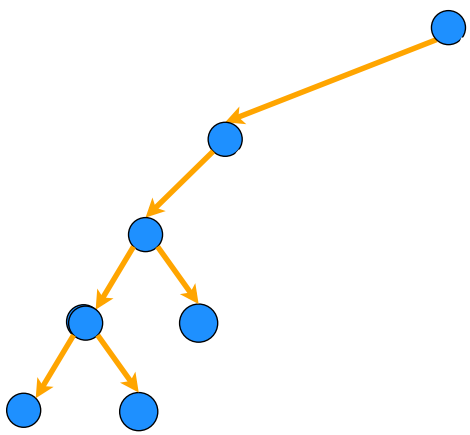
- pick a path $\sigma \in Paths^{\leq k}(\mathtt{P})$ [key 1]
- compute a *path predicate* φ_σ of σ [key 2]
- solve φ_σ for satisfiability [key 3 - use smt sovlers]
- SAT(s) ? get a new pair $< \mathrm{s}, \sigma >$
- loop until no more path to cover



The Symbolic Execution Loop

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

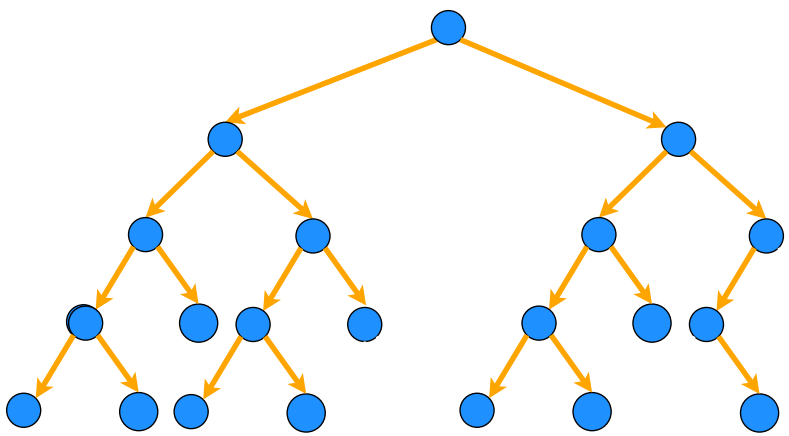
- pick a path $\sigma \in Paths^{\leq k}(P)$ [key 1]
- compute a *path predicate* φ_σ of σ [key 2]
- solve φ_σ for satisfiability [key 3 - use smt sovlers]
- SAT(s) ? get a new pair $< s, \sigma >$
- loop until no more path to cover



The Symbolic Execution Loop

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

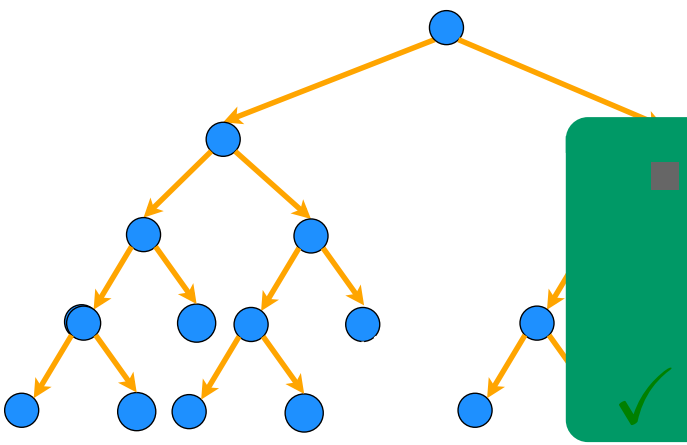
- pick a path $\sigma \in Paths^{\leq k}(\mathtt{P})$ [key 1]
- compute a *path predicate* φ_σ of σ [key 2]
- solve φ_σ for satisfiability [key 3 - use smt sovlers]
- SAT(s) ? get a new pair $< \mathrm{s}, \sigma >$
- loop until no more path to cover



The Symbolic Execution Loop

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

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



- Under-approximation
  - correct
  - relatively complete
- ✓ No false alarm

About robustness

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]

About robustness (2)

Goal = find input leading to `ERROR`

(assume we have only a solver for linear integer arith.)

```
g(int x) {return x*x; }
f(int x, int y) {z=g(x); if (y == z) ERROR; else OK }
```

Symbolic Execution

- create a subformula $z = x * x$, out of theory [FAIL]

Dynamic Symbolic Execution

- first concrete execution with `x=3, y=5` [goto OK]
- during path predicate computation, $x * x$ not supported
  . x is concretized to 3 and z is forced to 9
- resulting path predicate : $x = 3 \wedge z = 9 \wedge y = z$
- a solution is found : `x=3, y=9` [goto ERROR] [SUCCESS]

Summary

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

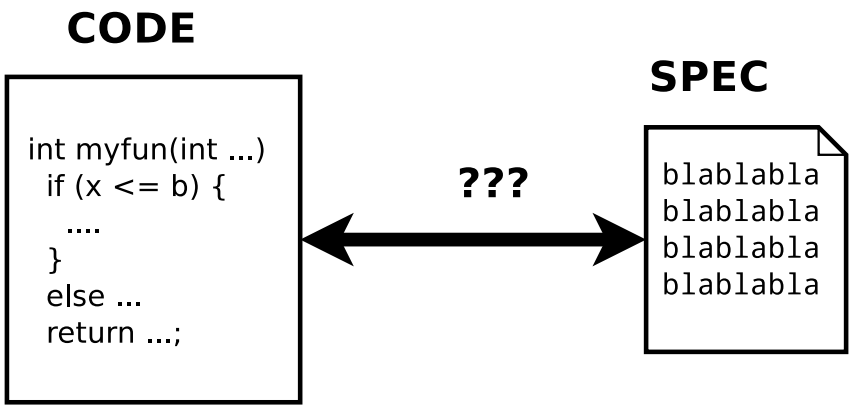
Still many challenges

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

Outline

- Introduction
- DSE in a nutshell
- Coverage-oriented DSE
- Binary-level DSE
- Conclusion

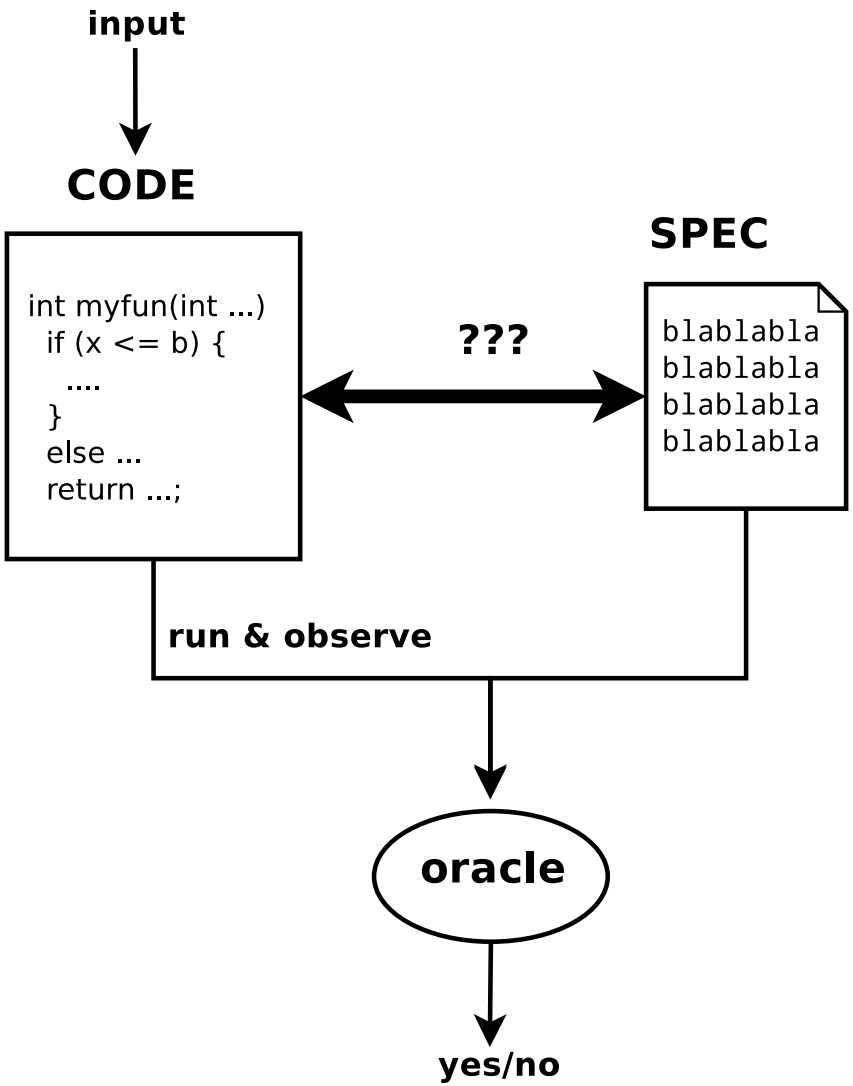
Context : white-box software testing



Context : white-box software testing

Testing process

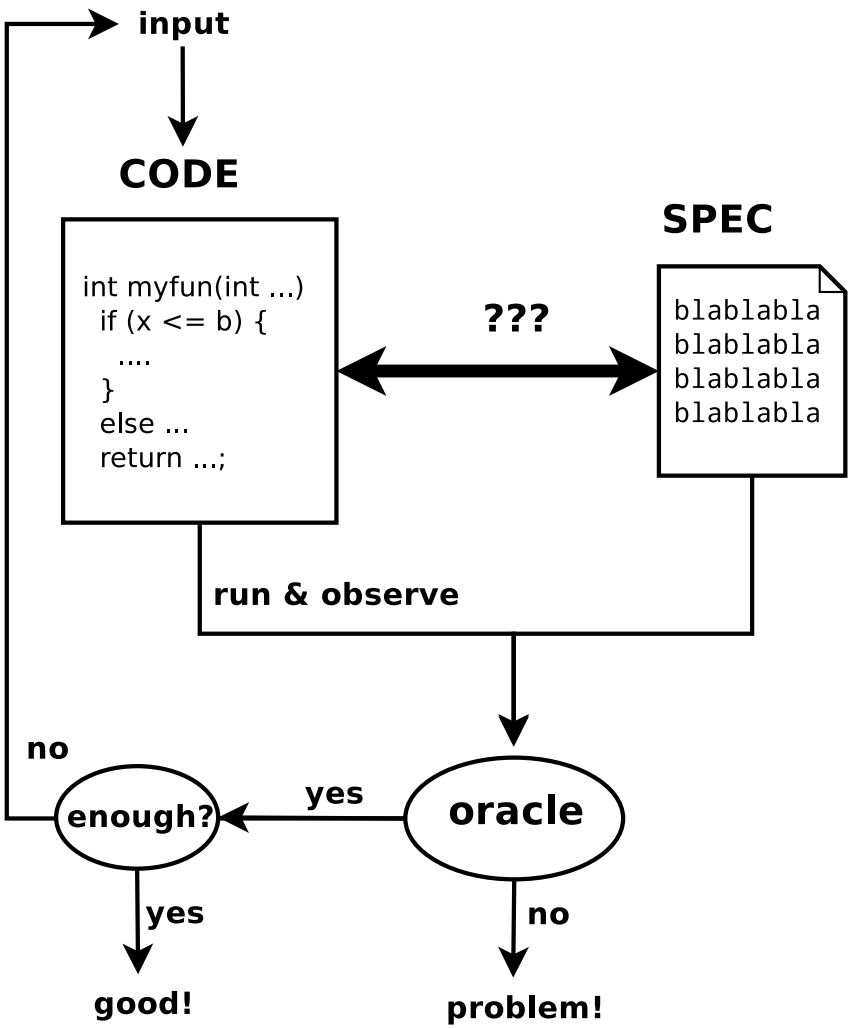
- Generate a test input
- Run it and check for errors
- Estimate coverage :
  if enough stop, else loop



Context : white-box software testing

Testing process

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



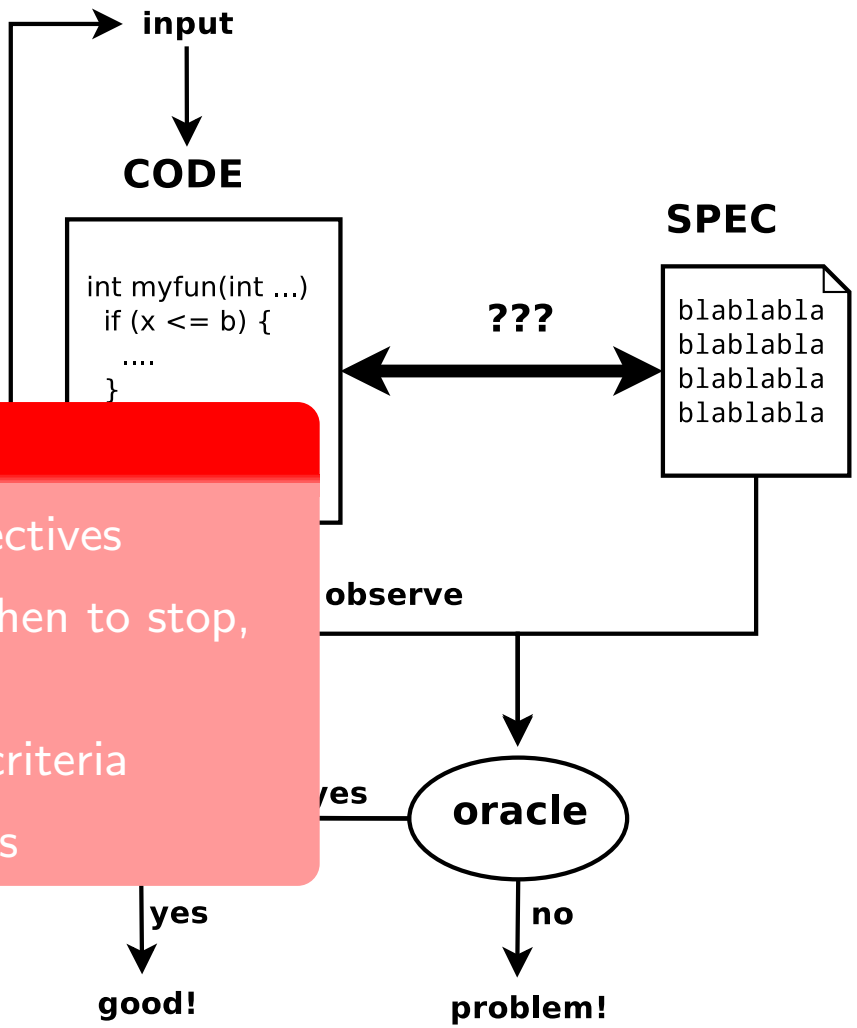
Context : white-box software testing

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



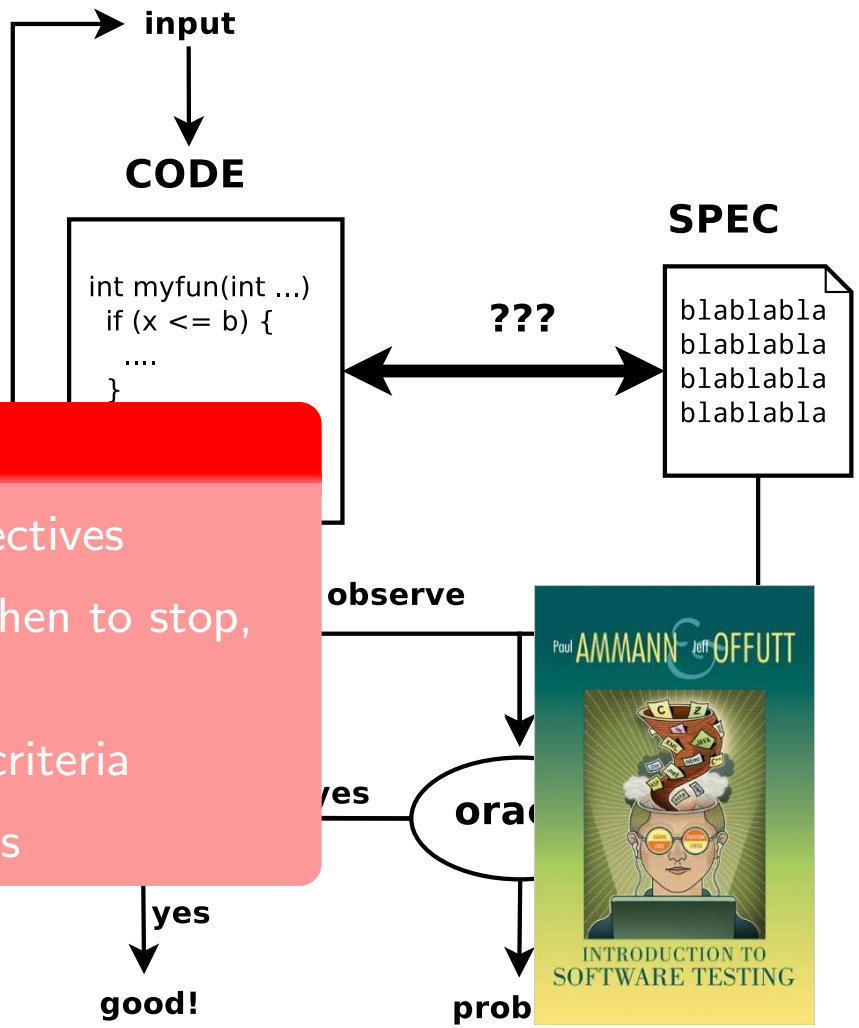
Context : white-box software testing

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



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

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.]

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

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

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]

Outline

Focus : Labels

- Annotate programs with **labels**
  - predicate attached to a specific program instruction
- Label (loc, φ) is covered if a test execution
  - reaches the instruction at loc
  - satisfies the predicate φ
- **Good for us**
  - can easily encode a large class of coverage criteria [see after]
  - in the scope of standard program analysis techniques

Simulation of standard coverage criteria

```
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;
```

Decision Coverage (**DC**)

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**)

Simulation of standard coverage criteria

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

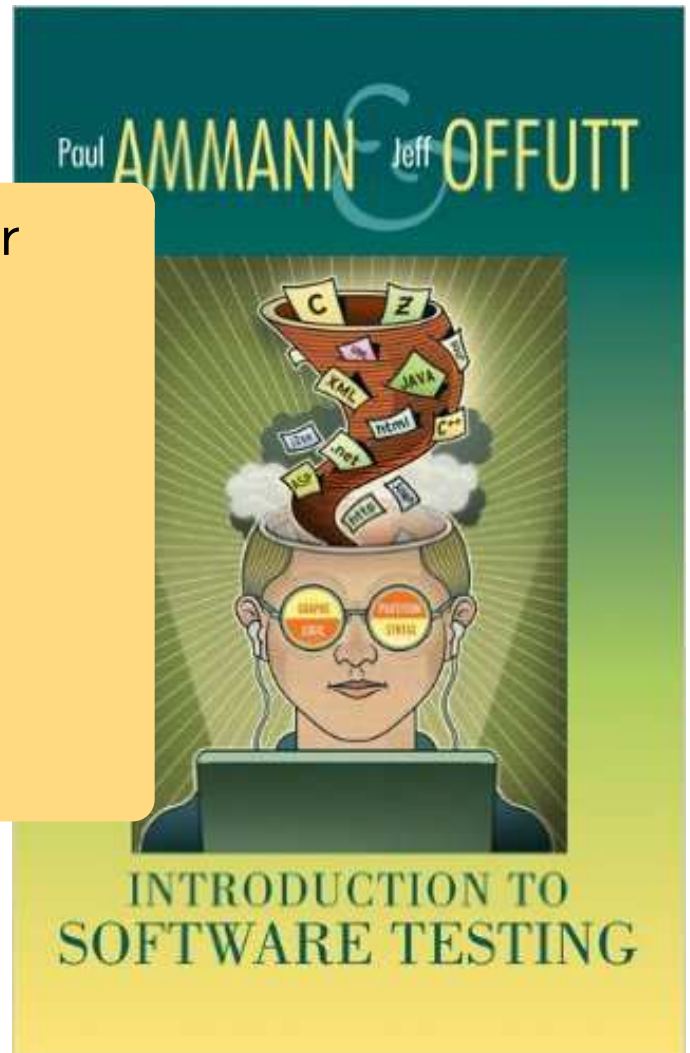
```
statement_1;
// l1: x==y && a<b
// l2: x==y && a>=b
// l3: x!=y && a<b
// l4: x!=y && a>=b
if (x==y && a<b)
   {...};
statement_3;
```

Multiple-Condition Coverage (**MCC**)

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

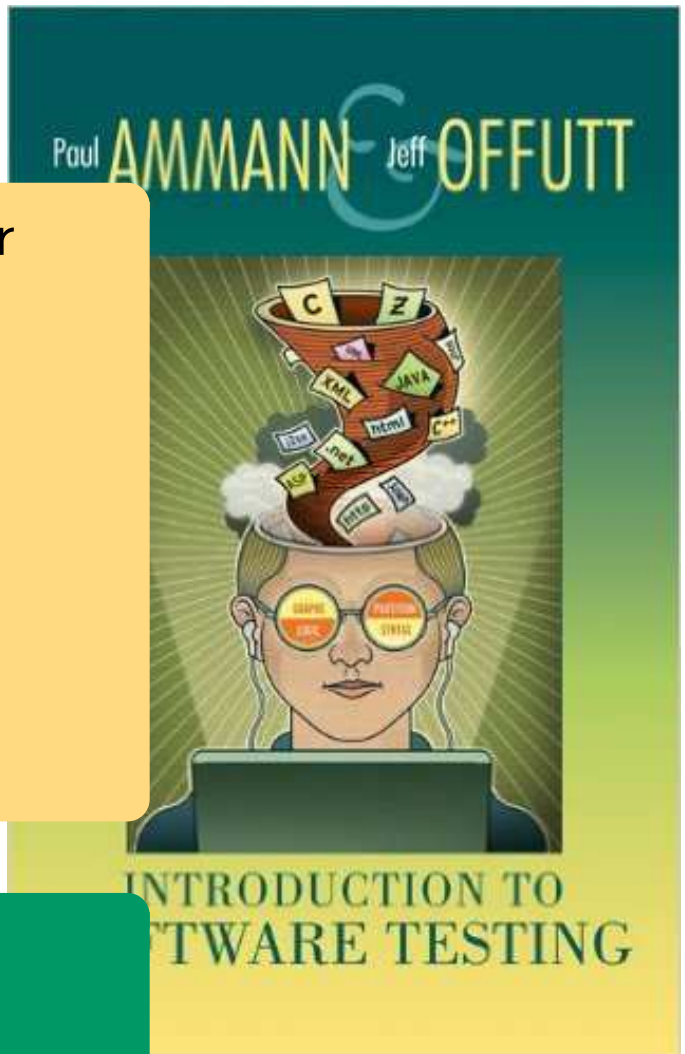


Simulation of standard coverage criteria

OBJ : generic specification mechanism for coverage criteria ✓

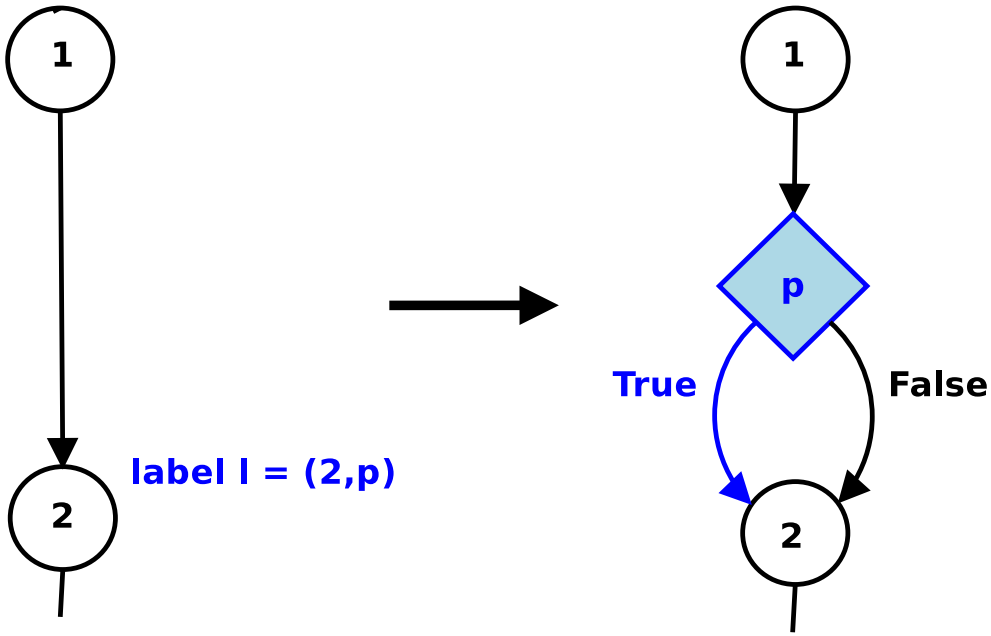
- **IC**, **DC**, **FC**, **CC**, **MCC**
- **GACC** (a variant of MCDC)
- **large part of Weak Mutations**
- Input Domain Partition
- Run-Time Error

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



Outline

Direct instrumentation



Covering label l $\Leftrightarrow$ Covering branch `True`

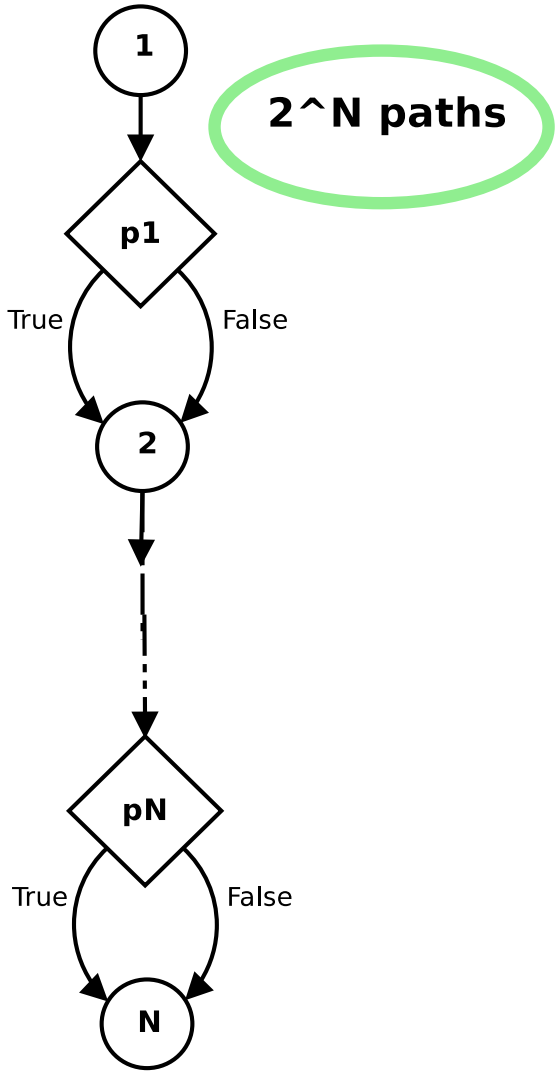
- ✓ sound & complete instrumentation
- ✗ complexification of the search space [#paths, shape of paths]
- ✗ dramatic overhead [theory & practice] [Apex : avg 272x, max 2000x]

Direct instrumentation is not good enough

Non-tightness 1

× P' has exponentially more paths than P

Direct instrumentation



Direct instrumentation is not good enough

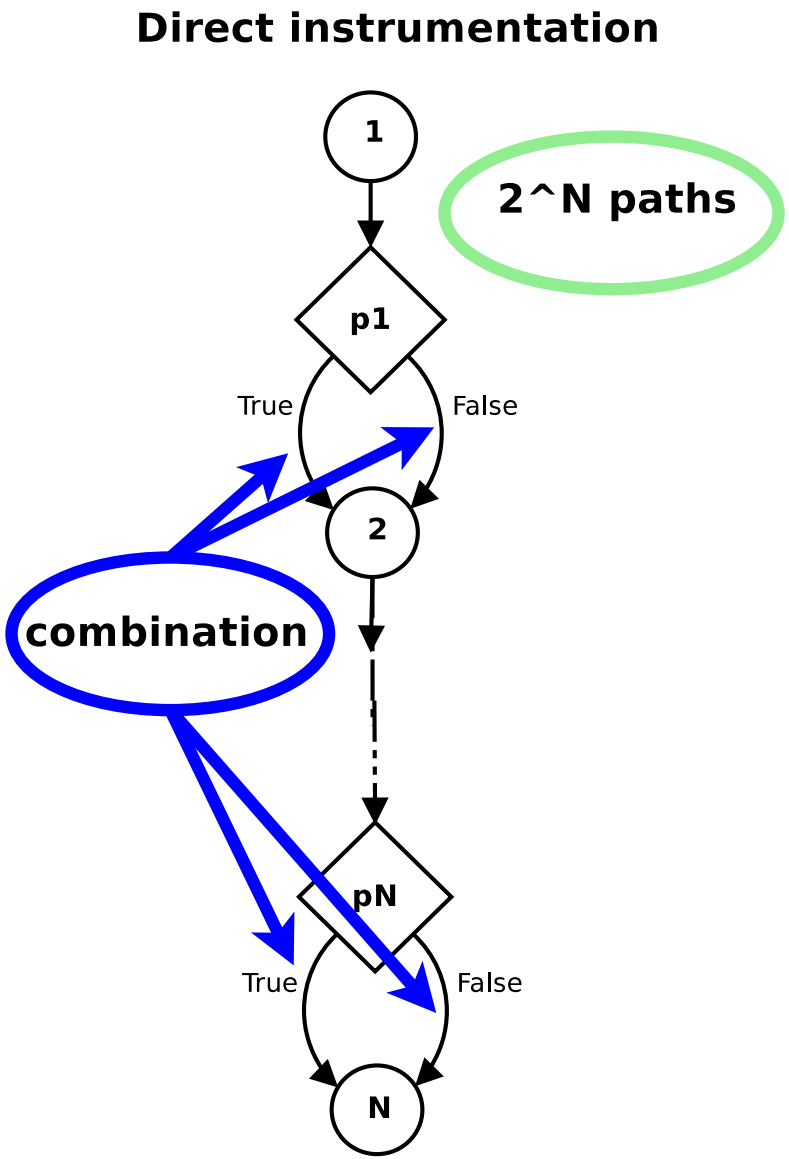
Non-tightness 1

× 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 $\neg p$
- π' covers up to N labels

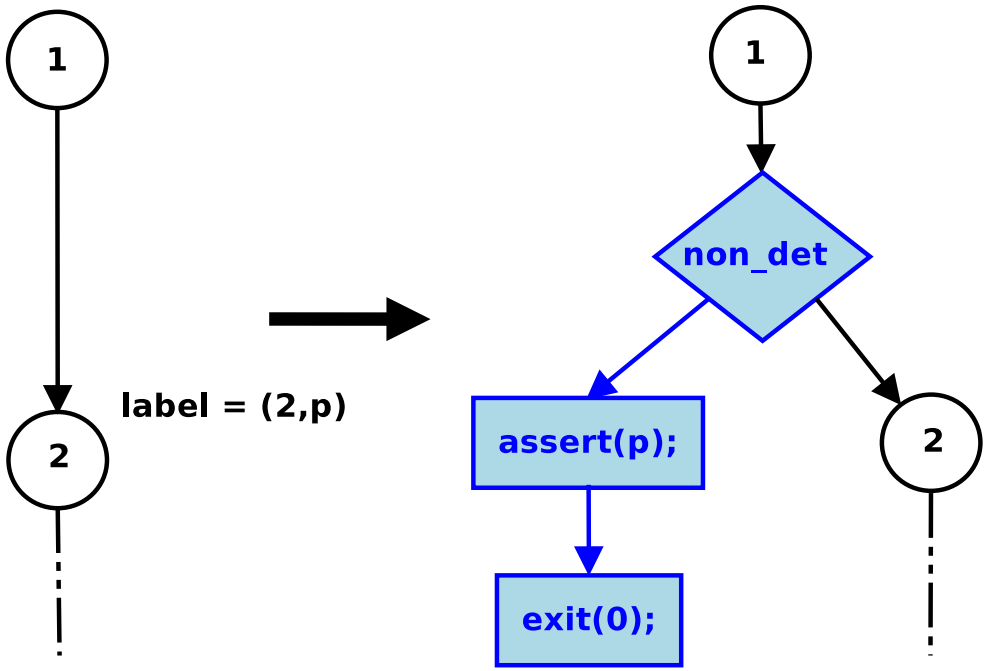


Our approach

The DSE* algorithm [ICST 14]

- Tight instrumentation $P^\star$: totally prevents "complexification"
- Iterative Label Deletion : discards some redundant paths
- Both techniques can be implemented in black-box

DSE⋆ : Tight Instrumentation

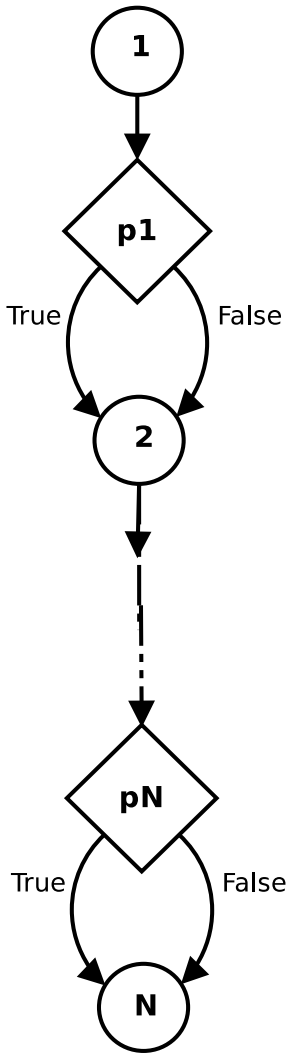


Covering label l $\Leftrightarrow$ Covering `exit(0)`

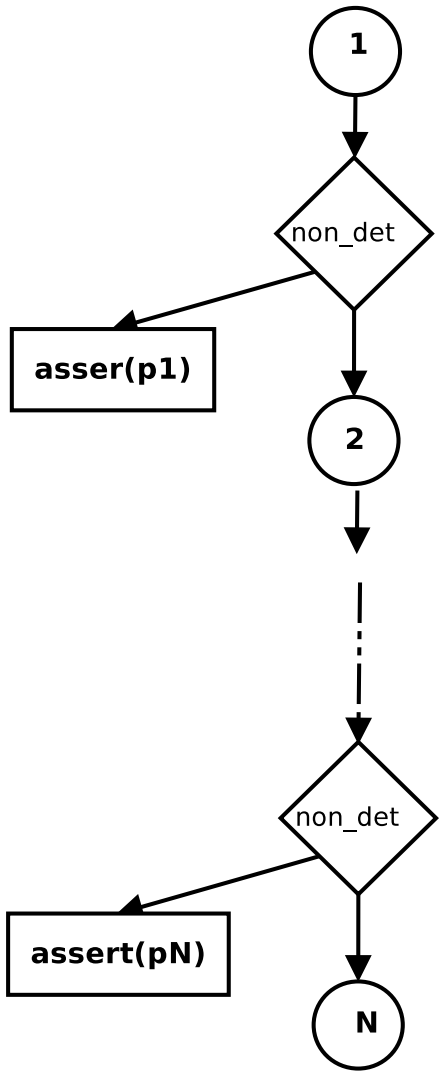
- ✓ sound & complete instrumentation
- ✓ no complexification of the search space

DSE⋆ : Tight Instrumentation (2)

Direct instrumentation

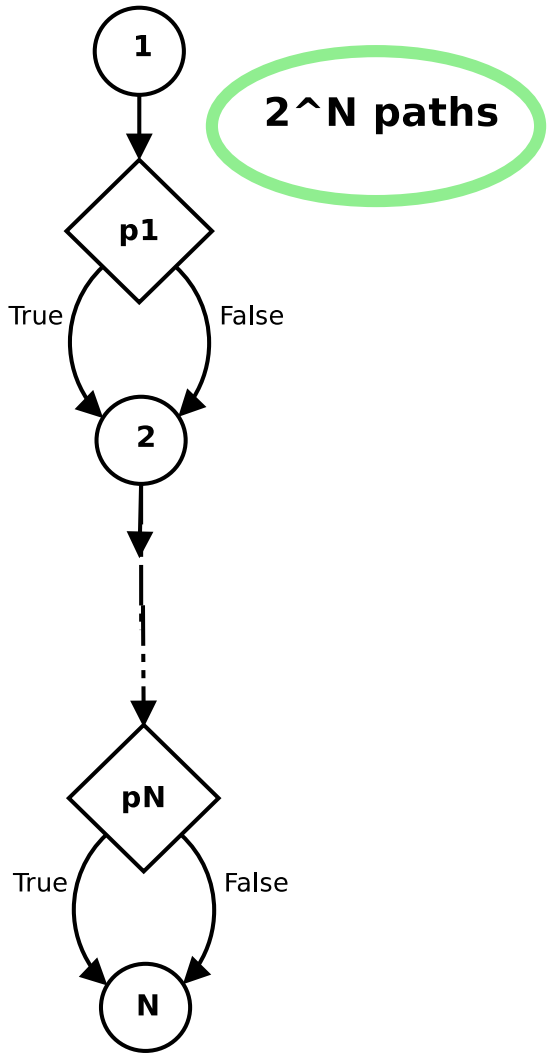


Tight Instrumentation

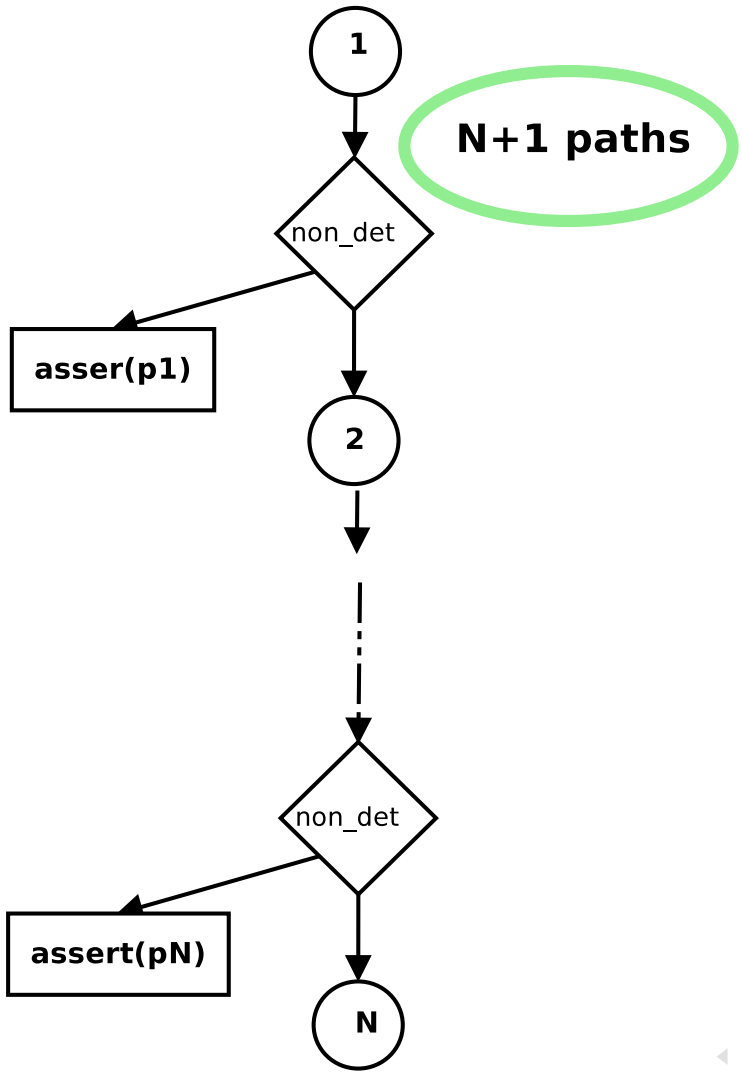


DSE⋆ : Tight Instrumentation (2)

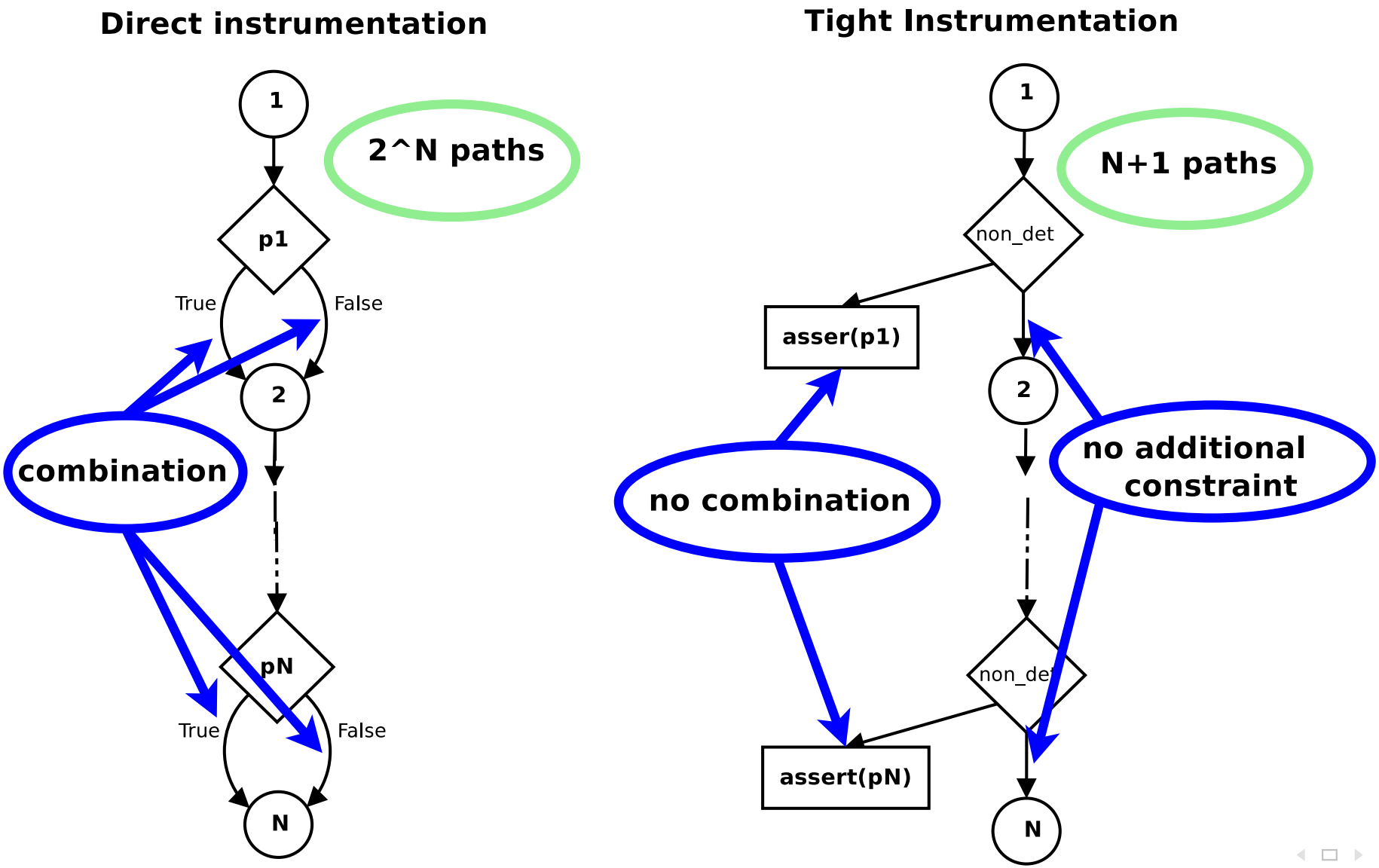
Direct instrumentation



Tight Instrumentation



DSE⋆ : Tight Instrumentation (2)



DSE⋆ : Iterative Label Deletion

Observations

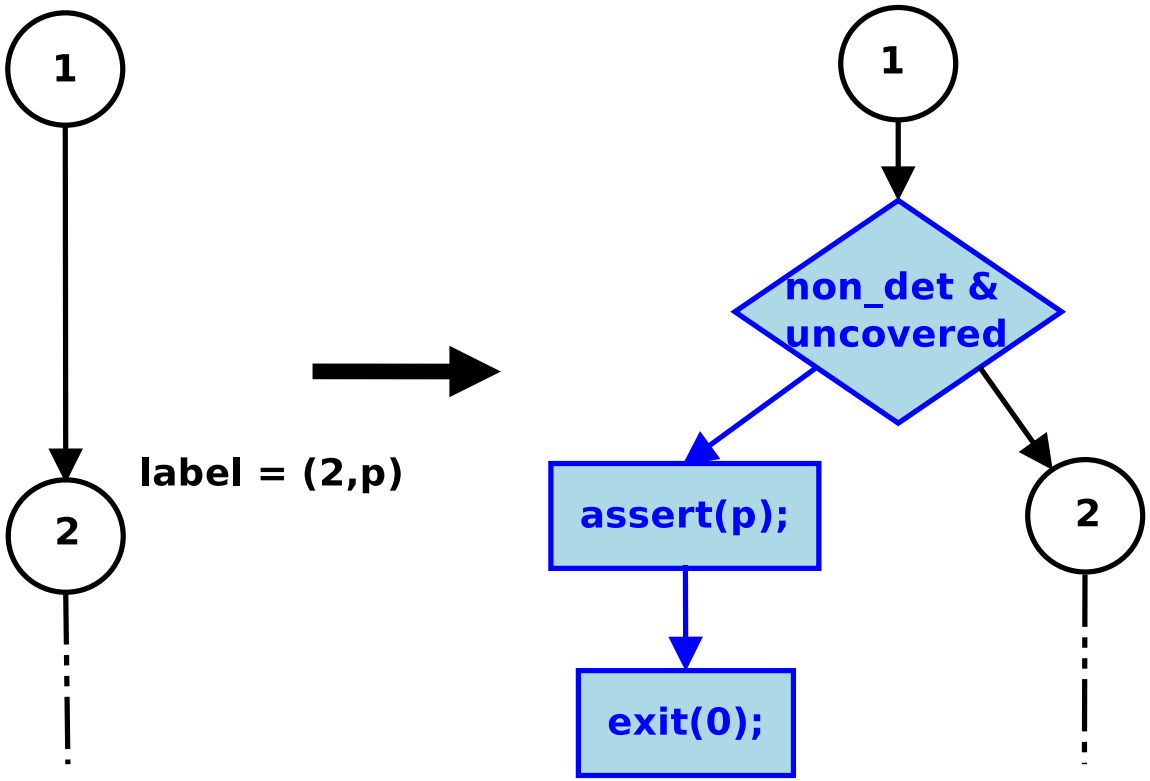
- we need to cover each label only once
- yet, DSE explores paths of P⋆ ending in already-covered labels
- burden DSE with "useless" paths w.r.t. label coverage

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

DSE⋆ : Iterative Label Deletion (2)



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

	DSE	DSE'	DSE*
Min	×1	×1.02	×0.49
Median	×1	×1.79	×1.37
Max	×1	**×122.50**	**×7.15**
Mean	×1	**×20.29**	**×2.15**
Timeouts	0	**5** *	**0**

* : TO are discarded for overhead computation
cherry picking : 94s vs TO [1h30]

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

	Random	DSE	DSE*
Min	37%	61%	62%
Median	63%	90%	**95%**
Max	100%	100%	100%
Mean	70%	87%	**90%**

vs DSE : +39% coverage on some examples

Experiments

Conclusion

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

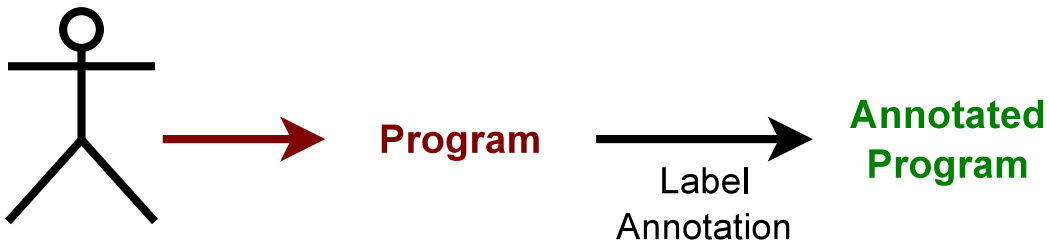
Outline

LTest overview

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

LTest overview



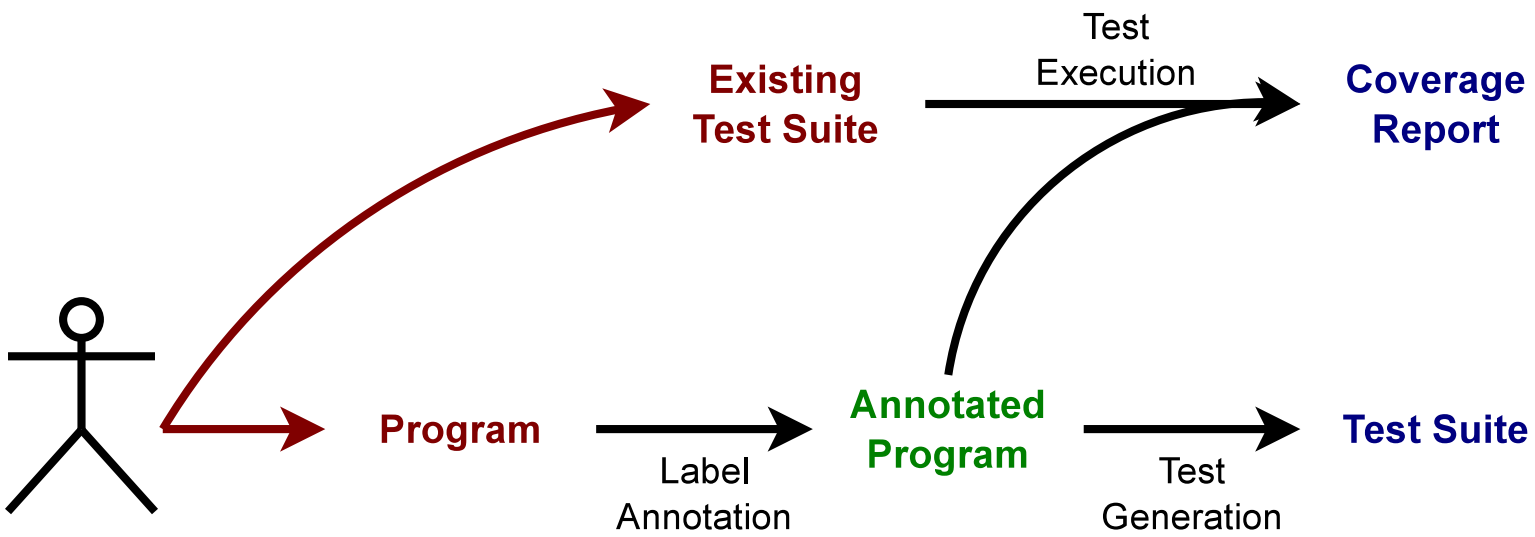
Supported criteria

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

Encoded with labels [ICST 2014]

- managed in a unified way
- rather easy to add new ones

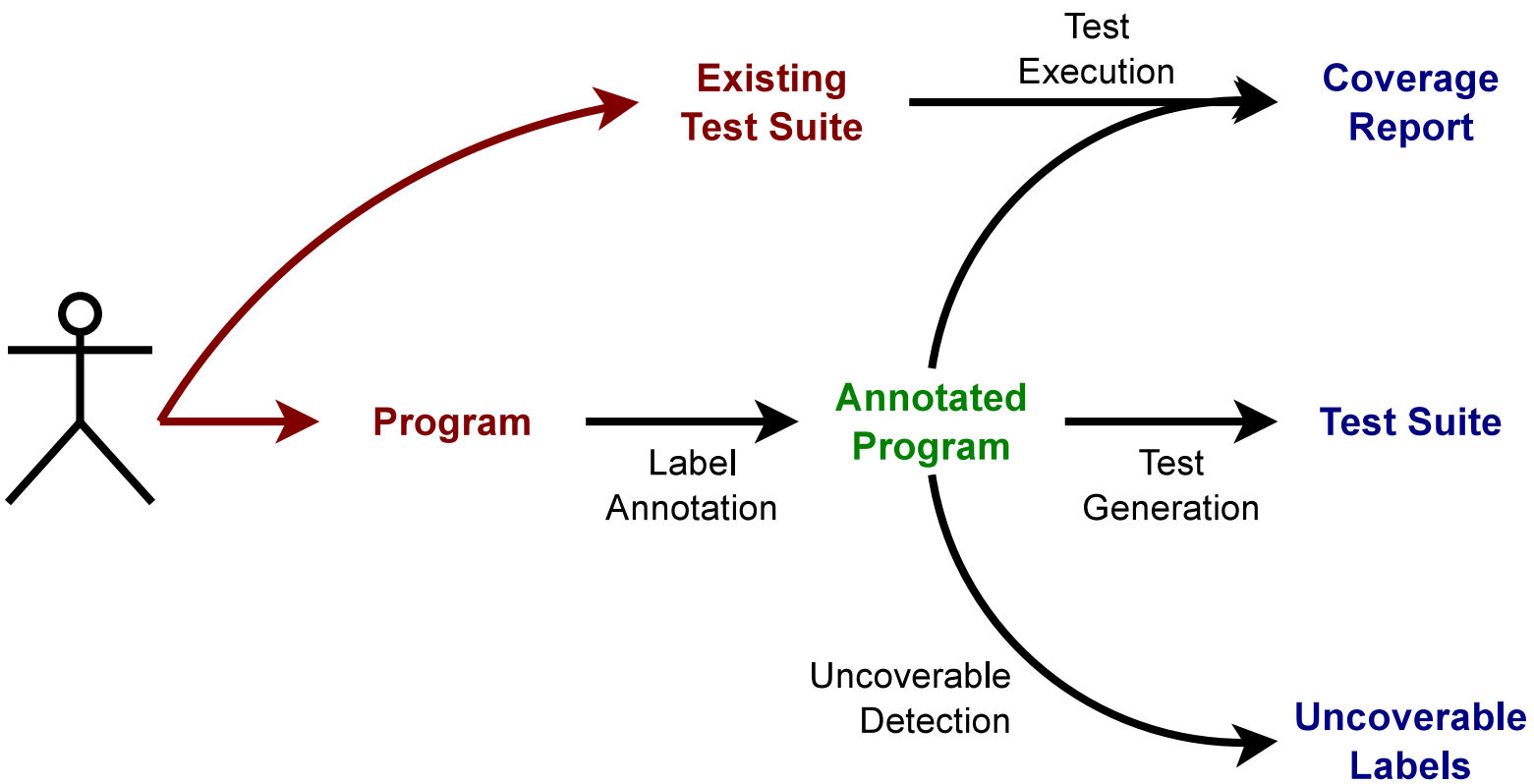
LTest overview



DSE* procedure [ICST 2014]

- DSE with native support for labels
- extension of PathCrawler

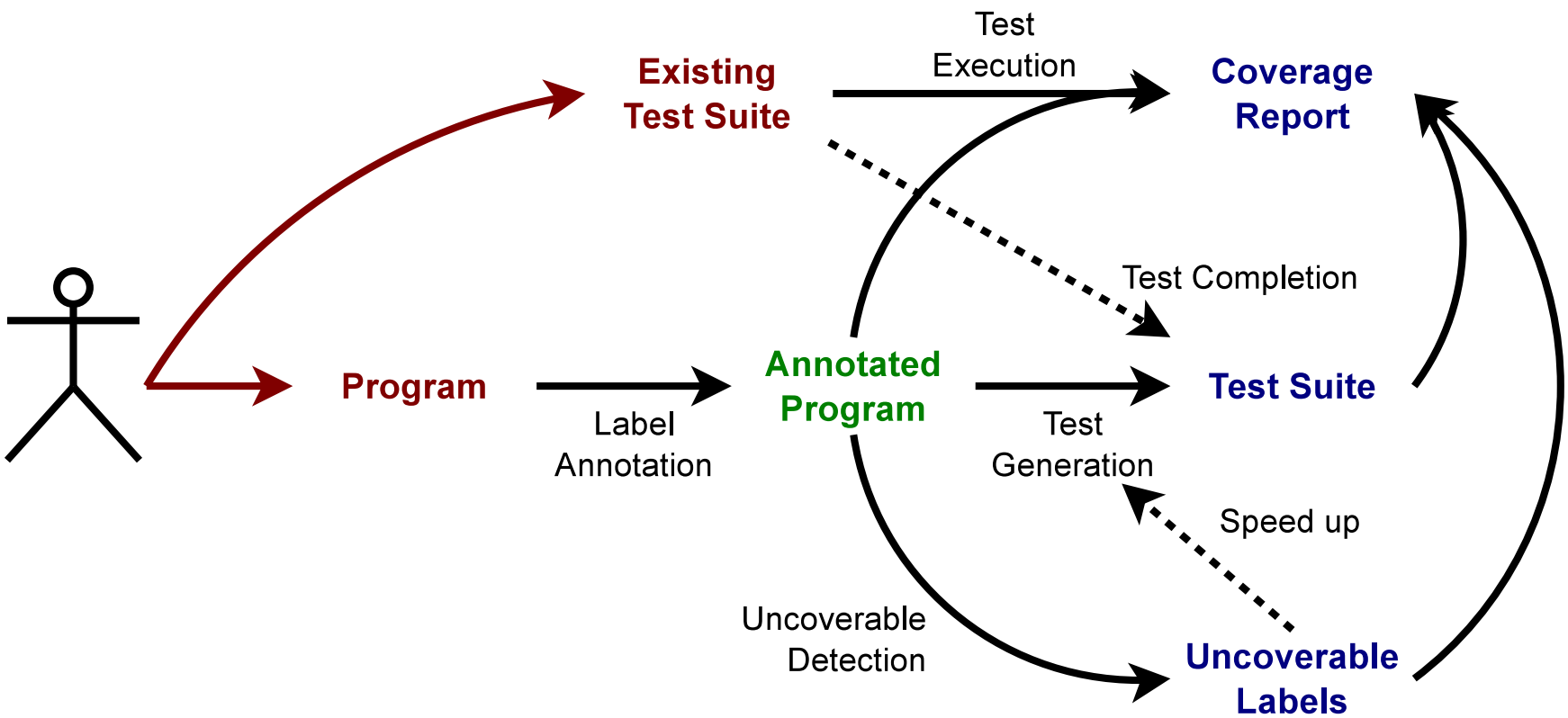
LTest overview



Reuse static analyzers from Frama-C

- sound detection !
- several modes : VA, WP, VA $\oplus$ WP

LTest overview



Reuse static analyzers from Frama-C

- sound detection !
- several modes : VA, WP, VA $\oplus$ WP

Service cooperation

- share label statuses
- `Covered`, `Infeasible`, `?`

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

Infeasibility detection

Infeasible test objectives

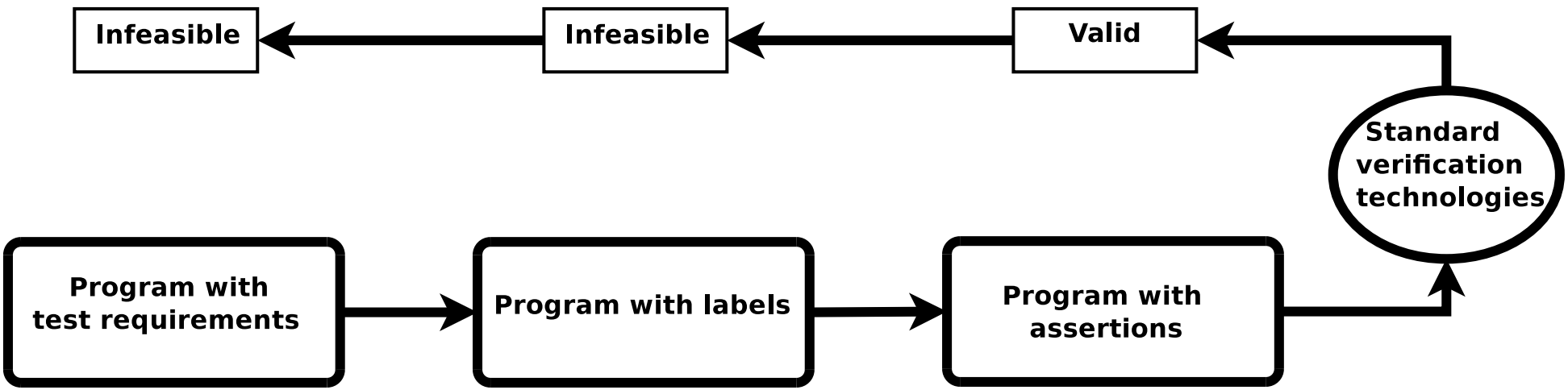
- waste generation effort, imprecise coverage ratios
- cause : structural coverage criteria are ... structural
- can be a nightmare [mcdc, mutations]
- detecting infeasible test requirements is undecidable

Basic ideas

- rely on existing sound verification methods
  . label (loc, φ) infeasible $\Leftrightarrow$ assertion $(loc, \neg\varphi)$ invariant
- grey-box combination of existing approaches [VA $\oplus$ WP]

Overview of the approach

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



- only soundness is required (verif)
- ▶ label encoding not required to be perfect
  . mcdc and strong mutation ok

Focus : checking assertion validity

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

Focus : checking assertion validity

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)

	VA	WP
sound for assert validity	✓	✓
blackbox reuse	✓	✓
local precision	✗	✓
calling context	✓	✗
calls / loop effects	✓	✗
global precision	✗	✗
scalability wrt. #labels	✓	✓
scalability wrt. code size	✗	✓

hypothesis : VA is interprocedural

VA and WP may fail

```
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

```
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
}
```

VA and WP may fail

```
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
}
```

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**

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

VA⊕WP succeeds!

```
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⊕WP succeeds!

```
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!

```
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
}
```

Proposal : VA ⊕ WP (2)

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

	VA	WP	VA ⊕ WP
sound for assert validity	✓	✓	✓
blackbox reuse	✓	✓	✓
local precision	✗	✓	✓
calling context	✓	✗	✓
calls / loop effects	✓	✗	✓
global precision	✗	✗	✗
scalability wrt. #labels	✓	✓	✓
scalability wrt. code size	✗	✓	?

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

	#Lab	#Inf	VA		WP		VA ⊕ WP	
			#d	%d	#d	%d	#d	%d
Total	1,270	121	84	69%	73	**60%**	118	**98%**
Min		0	0	0%	0	0%	2	**67%**
Max		29	29	100%	15	100%	29	100%
Mean		4.7	3.2	63%	2.8	**82%**	4.5	**95%**

#d : number of detected infeasible labels

%d : ratio of detected infeasible labels

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

	#Lab	#Inf	VA		WP		VA ⊕ WP	
			#d	%d	#d	%d	#d	%d
Total	1,270	121	84	69%	73	**60%**	118	**98%**
Min		0	0	0%	0	0%	2	**67%**
Max		29	29	100%	15	100%	29	100%
Mean		4.7	3.2	63%	2.8	**82%**	4.5	**95%**

#d : number of detected infeasible labels

%d : ratio of detected infeasible labels

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

Impact on test generation

report more accurate coverage ratio

	Coverage ratio reported by DSE*		
Detection method	**None**	**VA ⊕WP**	**Perfect***
Total	**90.5%**	**99.2%**	100.0%
Min	**61.54%**	**91.7%**	100.0%
Max	100.00%	100.0%	100.0%
Mean	**91.10%**	**99.2%**	100.0%

* preliminary, manual detection of infeasible labels

Impact on test generation

optimisation : speedup test generation : take care !

Ideal speedup

	DSE⋆-OPT vs DSE⋆
Min.	0.96×
Max.	592.54×
Mean	49.04×

in practice

		DSE⋆-OPT vs DSE⋆
RT(1s) +LUncov +DSE⋆	**Min**	0.1x
	Max	**55.4x**
	Mean	**3.8x**

RT : random testing
Speedup wrt. DSE⋆ alone

Outline

- Introduction
- DSE in a nutshell
- Coverage-oriented DSE
- Binary-level DSE
- Conclusion

Benefits & challenges of binary code analysis

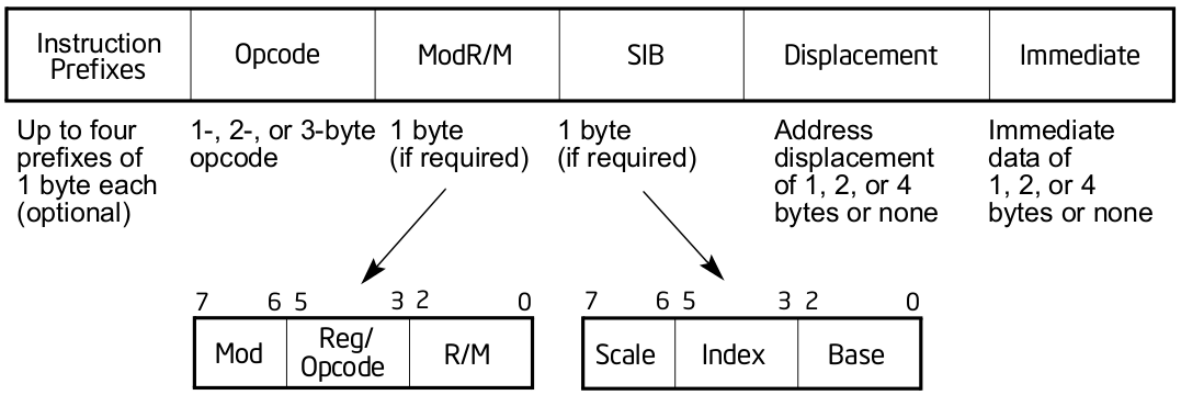
Advantages over source-level analysis

- 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 ?

Basic challenge : modelling



r8(/r)		AL	CL	DL	BL	AH	CH	DH	BH
r16(/r)		AX	CX	DX	BX	SP	BP	SI	DI
r32(/r)		EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
mm(/r)		MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
xmm(/r)		XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
sreg		ES	CS	SS	DS	FS	GS	res.	res.
eee		CR0	invd	CR2	CR3	CR4	invd	invd	invd
eee		DR0	DR1	DR2	DR3	DR4[1]	DR5[1]	DR6	DR7
(In decimal) /digit (Opcode)		0	1	2	3	4	5	6	7
(In binary) REG =		000	001	010	011	100	101	110	111
Effective Address	Mod	R/M	Value of ModR/M Byte (in Hex)							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[sib]		100	04	0C	14	1C	24	2C	34	3C
disp32		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX]+disp8	01	000	40	48	50	58	60	68	70	78
[ECX]+disp8		001	41	49	51	59	61	69	71	79
[EDX]+disp8		010	42	4A	52	5A	62	6A	72	7A
[EBX]+disp8		011	43	4B	53	5B	63	6B	73	7B
[sib]+disp8		100	44	4C	54	5C	64	6C	74	7C
[EBP]+disp8		101	45	4D	55	5D	65	6D	75	7D
[ESI]+disp8		110	46	4E	56	5E	66	6E	76	7E
[EDI]+disp8		111	47	4F	57	5F	67	6F	77	7F
[EAX]+disp32	10	000	80	88	90	98	A0	A8	B0	B8
[ECX]+disp32		001	81	89	91	99	A1	A9	B1	B9
[EDX]+disp32		010	82	8A	92	9A	A2	AA	B2	BA
[EBX]+disp32		011	83	8B	93	9B	A3	AB	B3	BB
[sib]+disp32		100	84	8C	94	9C	A4	AC	B4	BC
[EBP]+disp32		101	85	8D	95	9D	A5	AD	B5	BD
[ESI]+disp32		110	86	8E	96	9E	A6	AE	B6	BE
[EDI]+disp32		111	87	8F	97	9F	A7	AF	B7	BF
AL/AX/EAX/ST0/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
CL/CX/ECX/ST1/MM1/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
DL/DX/EDX/ST2/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
BL/BX/EBX/ST3/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
AH/SP/ESP/ST4/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
CH/BP/EBP/ST5/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
DH/SI/ESI/ST6/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
BH/DI/EDI/ST7/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

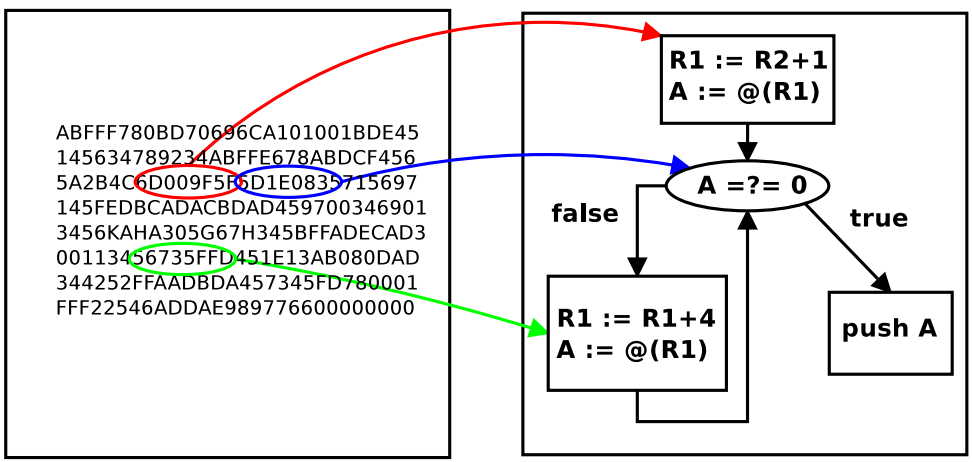
Example : x86

- more than 1,000 instructions
  - . ≈ 400 basic
  - . + float, interrupts, mmx
- many side-effects
- error-prone decoding
  - . addressing mode, prefixes, ...

Basic challenge : safe CFG recovery [reverse]

Input

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

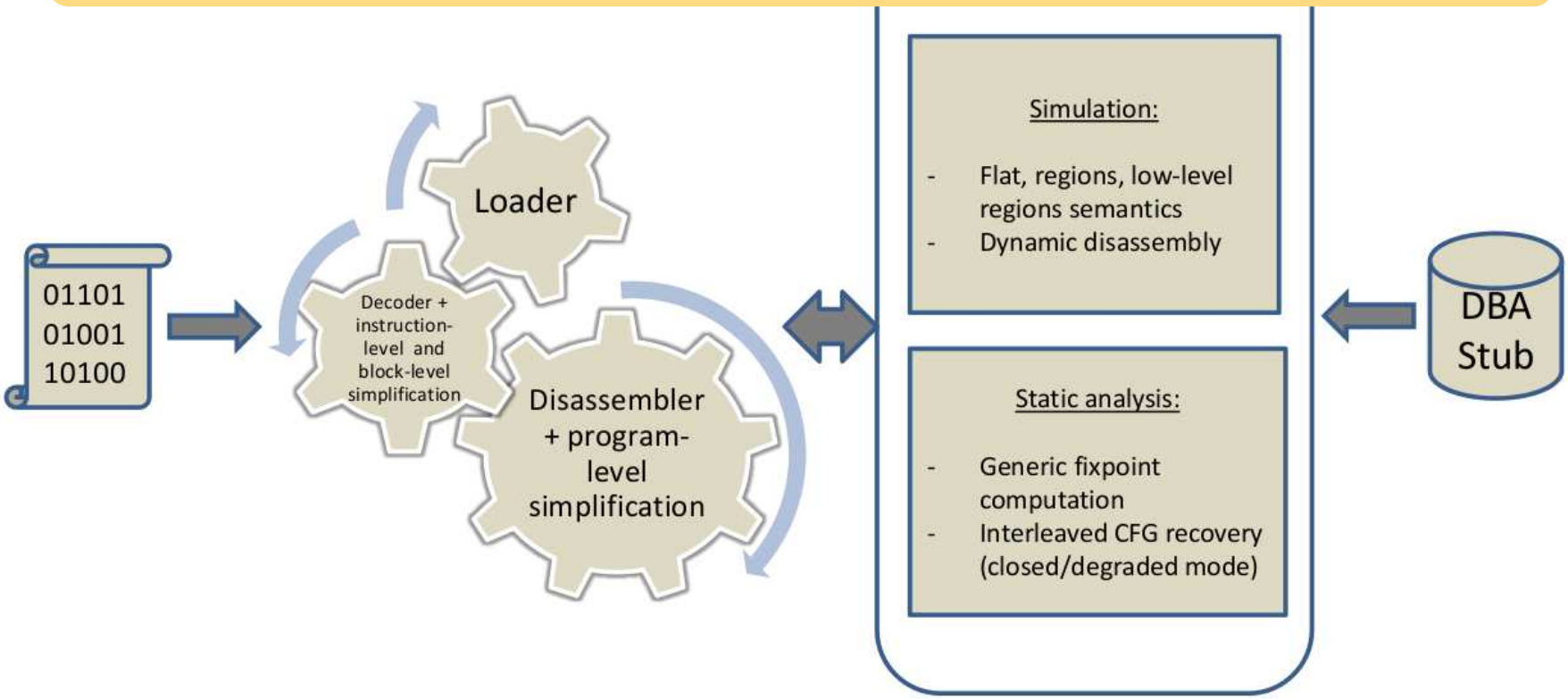


Output : (surapprox of) the program CFG [basic artifact for verif]

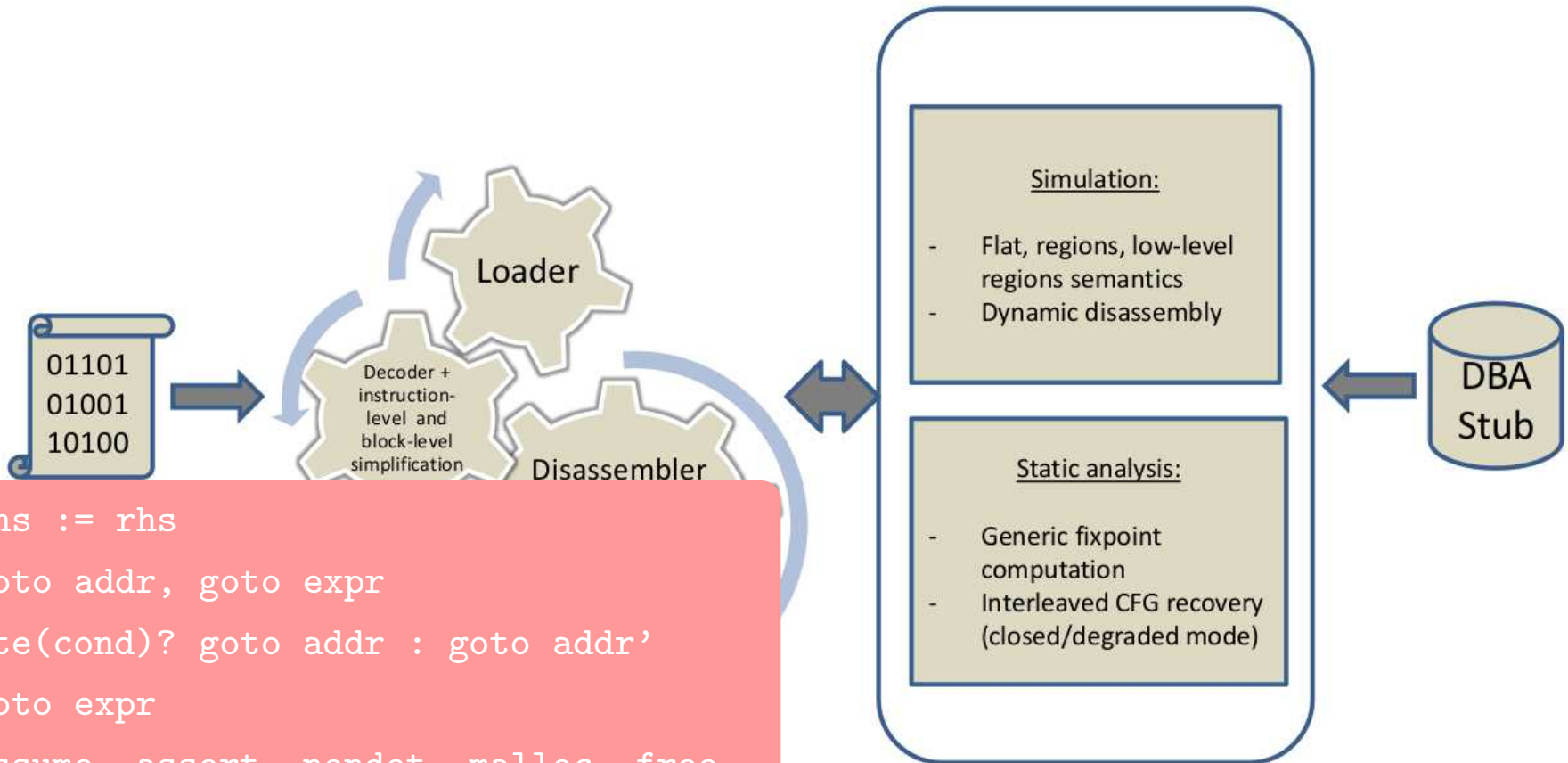
- question : successor of $\langle$`addr: goto a`$\rangle$? [chicken-and-egg issue]

The BINSEC platform

A platform for binary-level formal analysis
. open-source, developped mostly in OCaml
. developped within the BINSEC project [CEA, IRISA, LORIA, VERIMAG]
. intermediate representation [cav 11, tacas 15] + formal analysis



The BINSEC platform

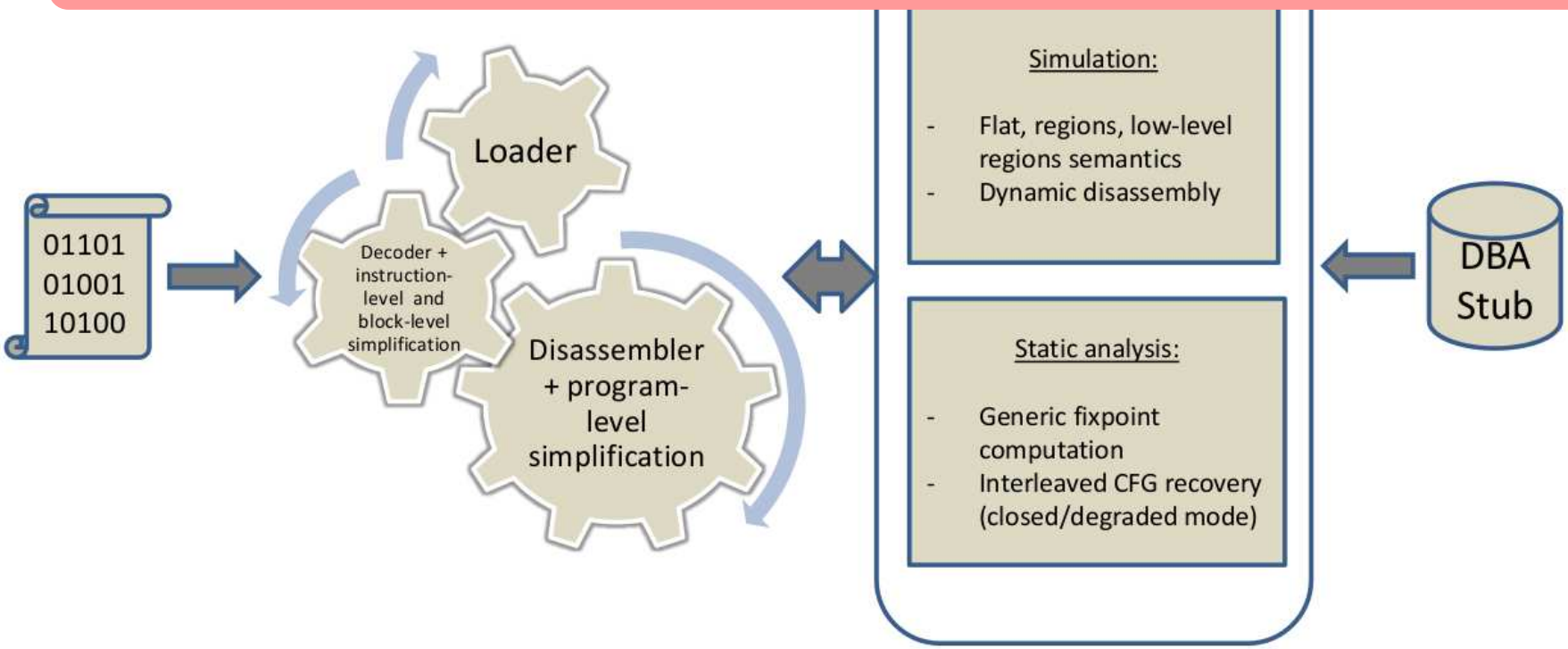


- `lhs := rhs`
- `goto addr, goto expr`
- `ite(cond)? goto addr : goto addr'`
- `goto expr`
- `assume, assert, nondet, malloc, free`

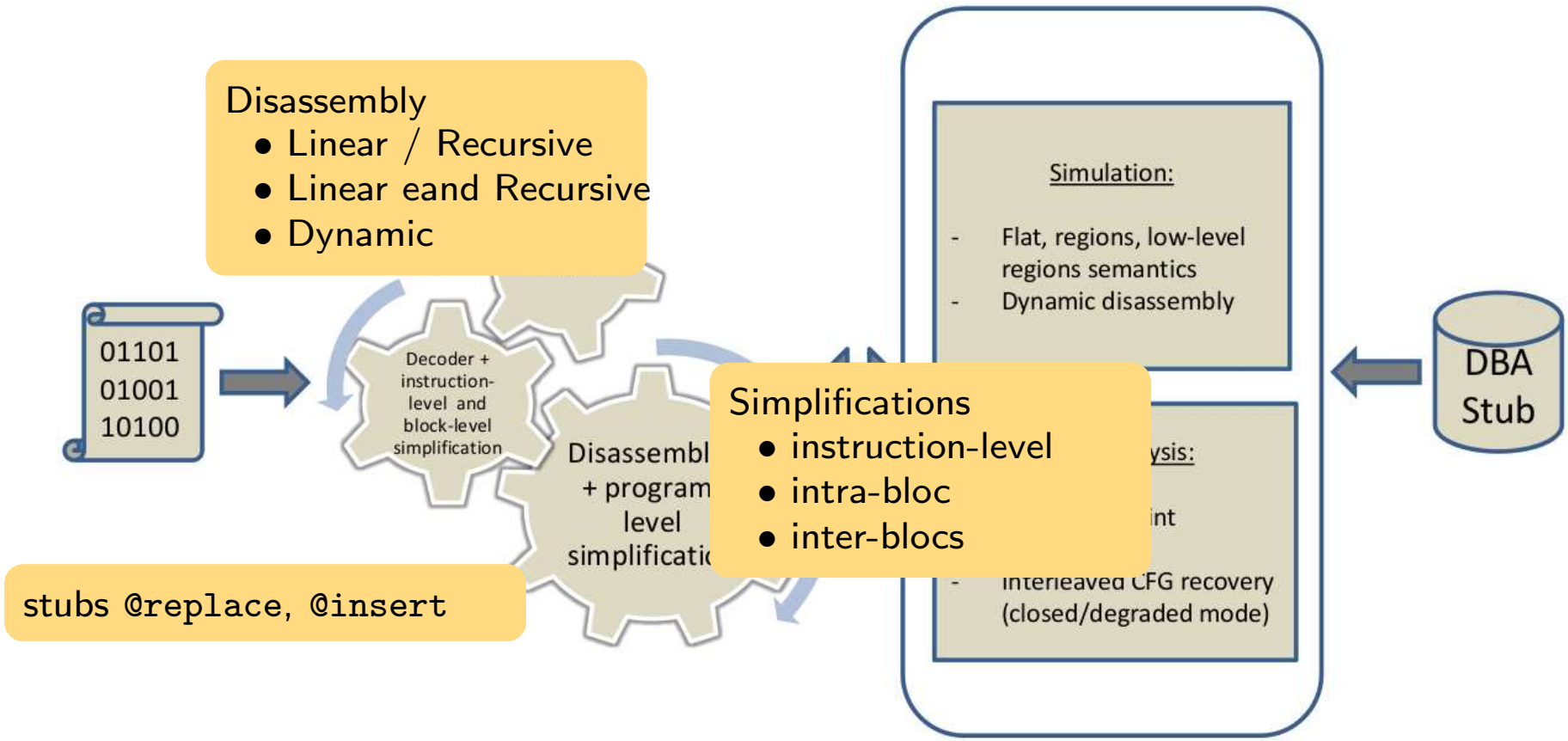
The BINSEC platform

- 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.



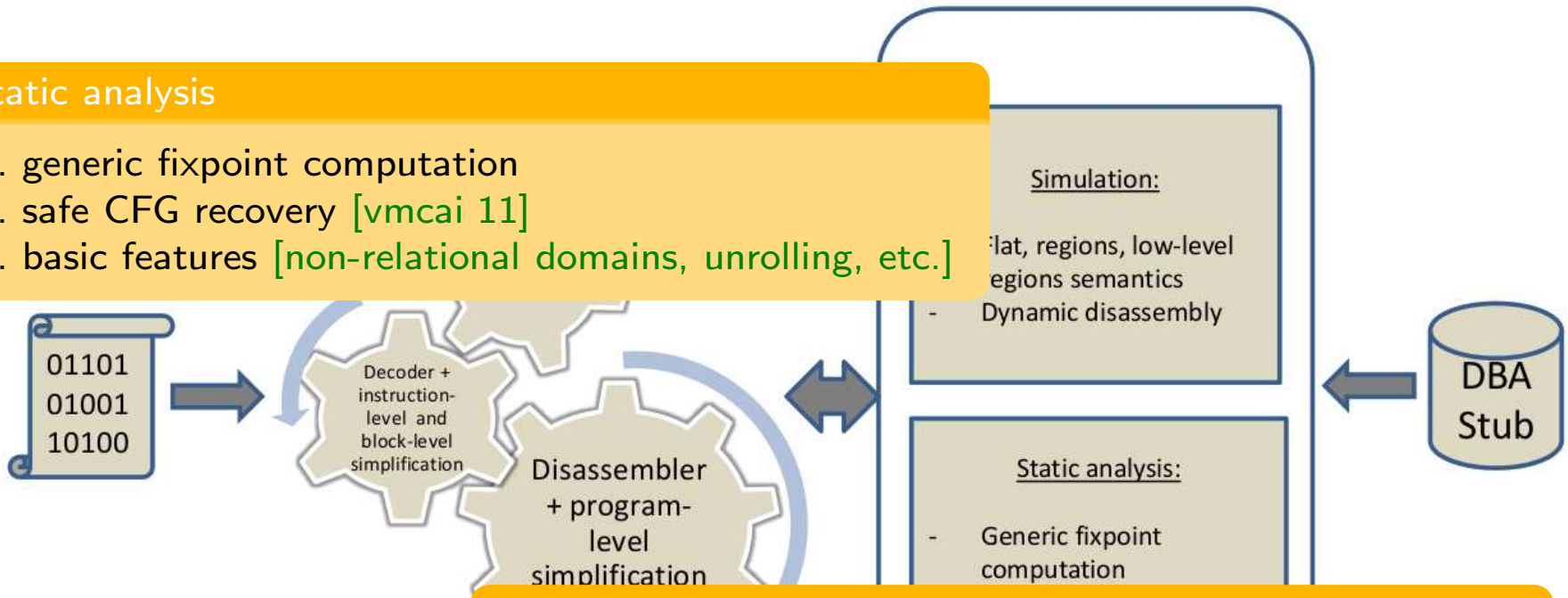
The BINSEC platform



The BINSEC platform

Static analysis

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

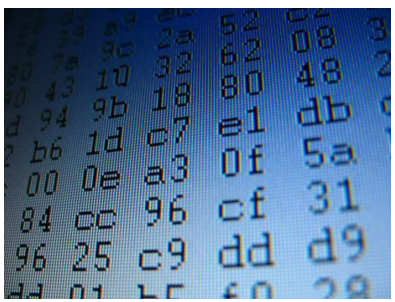


DSE

. 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

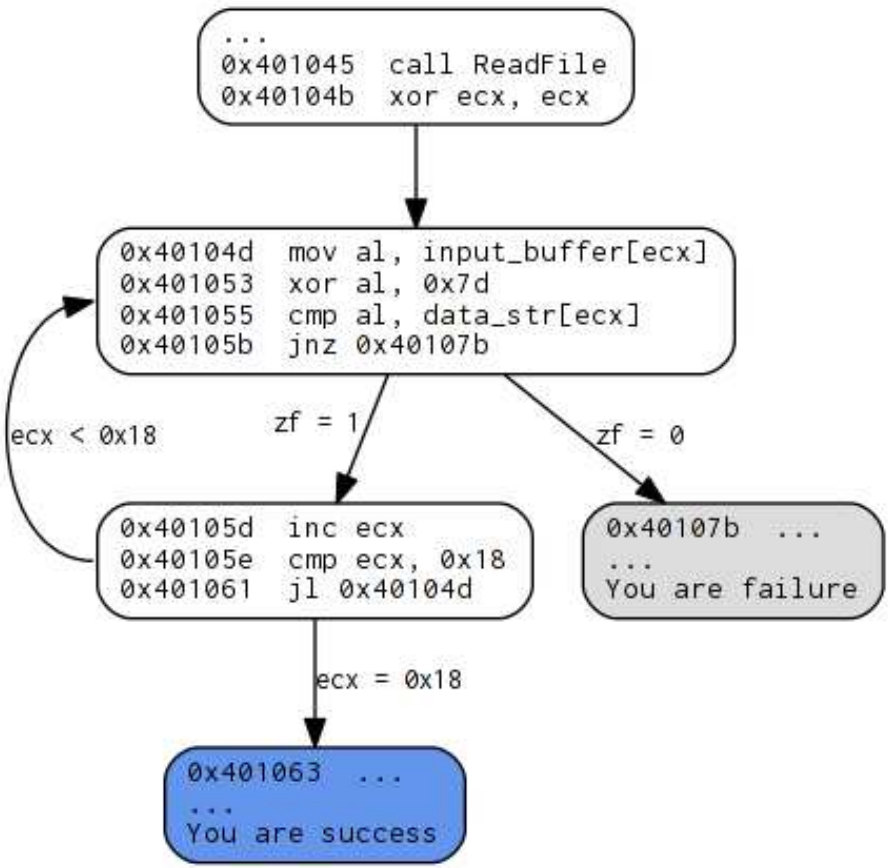
Context : FlareOn, set of challenges published by FireEye every year
Goal : Finding a 24-character key *(simple decryption loop)*



input string $\mapsto$ { success, failure }

Example : solving FlareOn #1

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

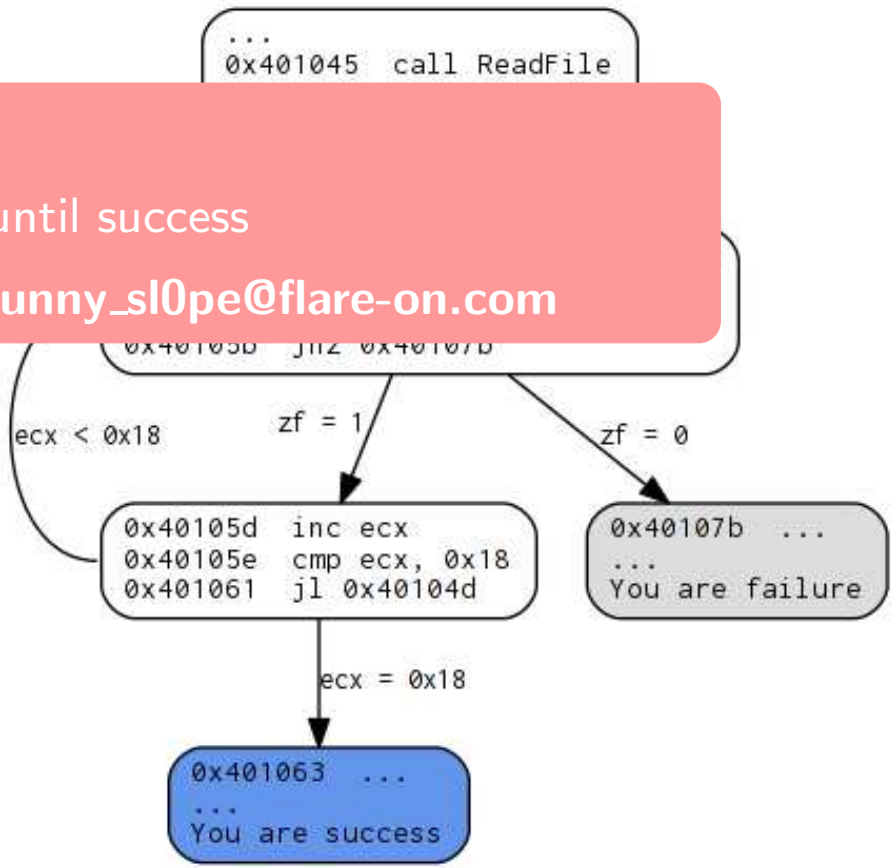


Example : solving FlareOn #1

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

BINSE/SE can helps

- iterate over all paths until success
- yield the given key : **bunny_sl0pe@flare-on.com**

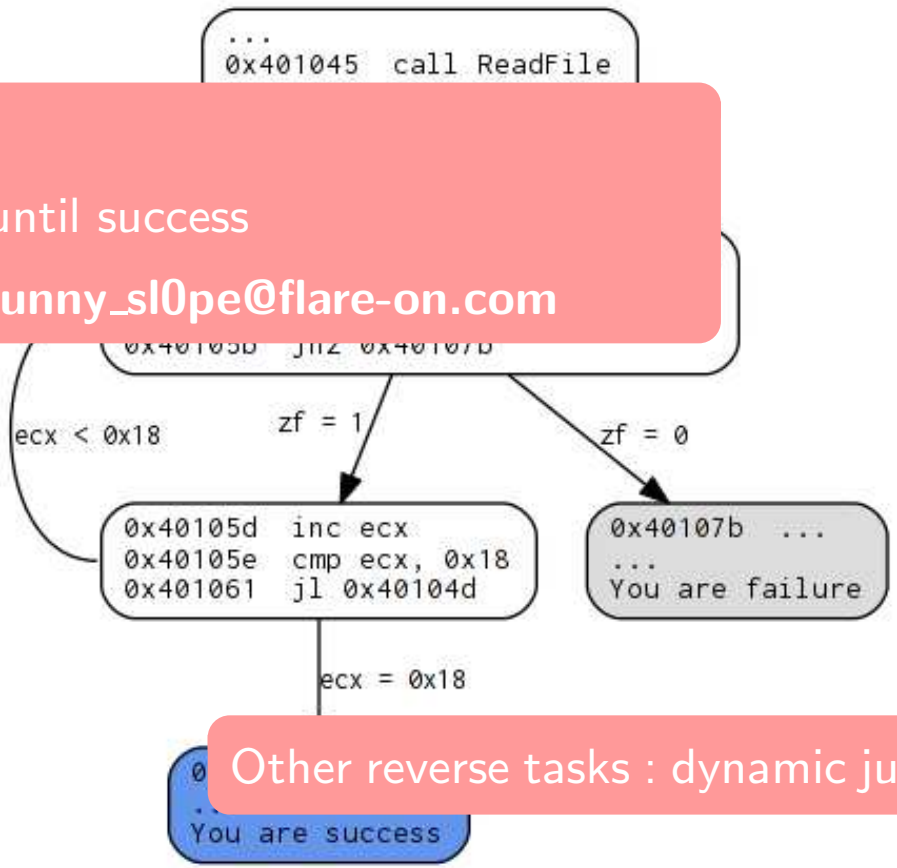


Example : solving FlareOn #1

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

BINSE/SE can helps

- iterate over all paths until success
- yield the given key : **bunny_sl0pe@flare-on.com**



Other reverse tasks : dynamic jumps, call/ret

Outline

Conclusion

Dynamic Symbolic Execution is very promising

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

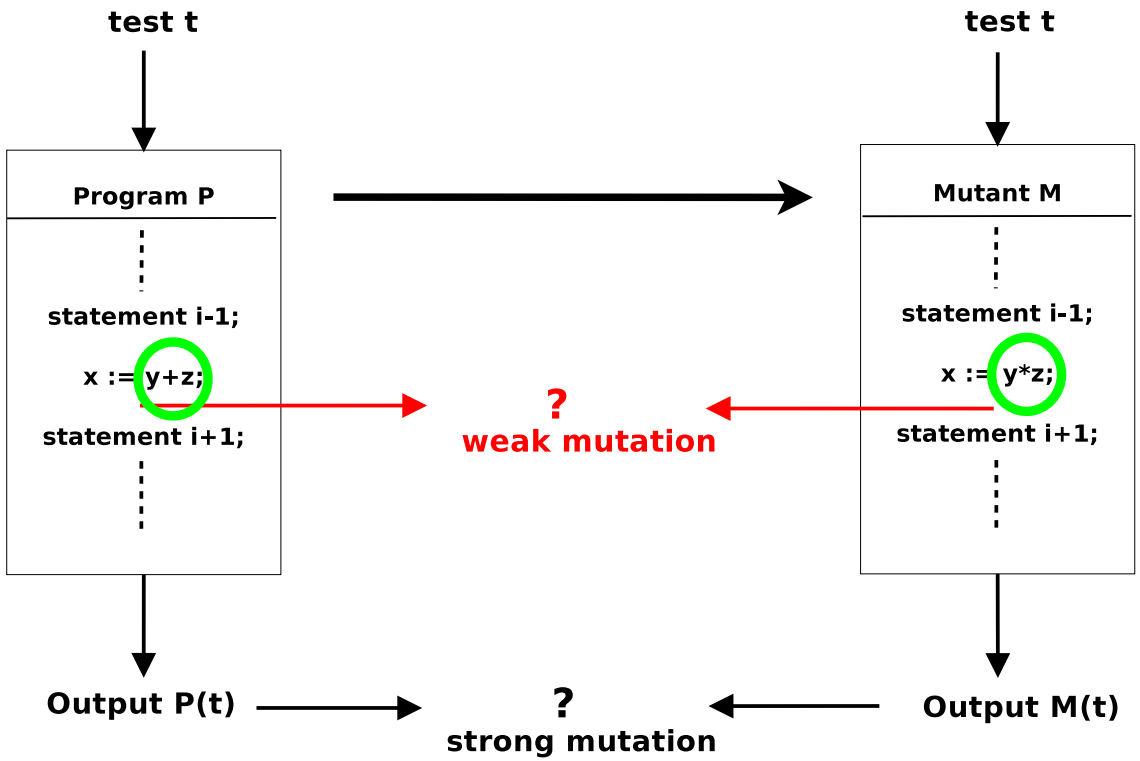
Can be efficiently lifted to coverage-oriented testing

- ✓ 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

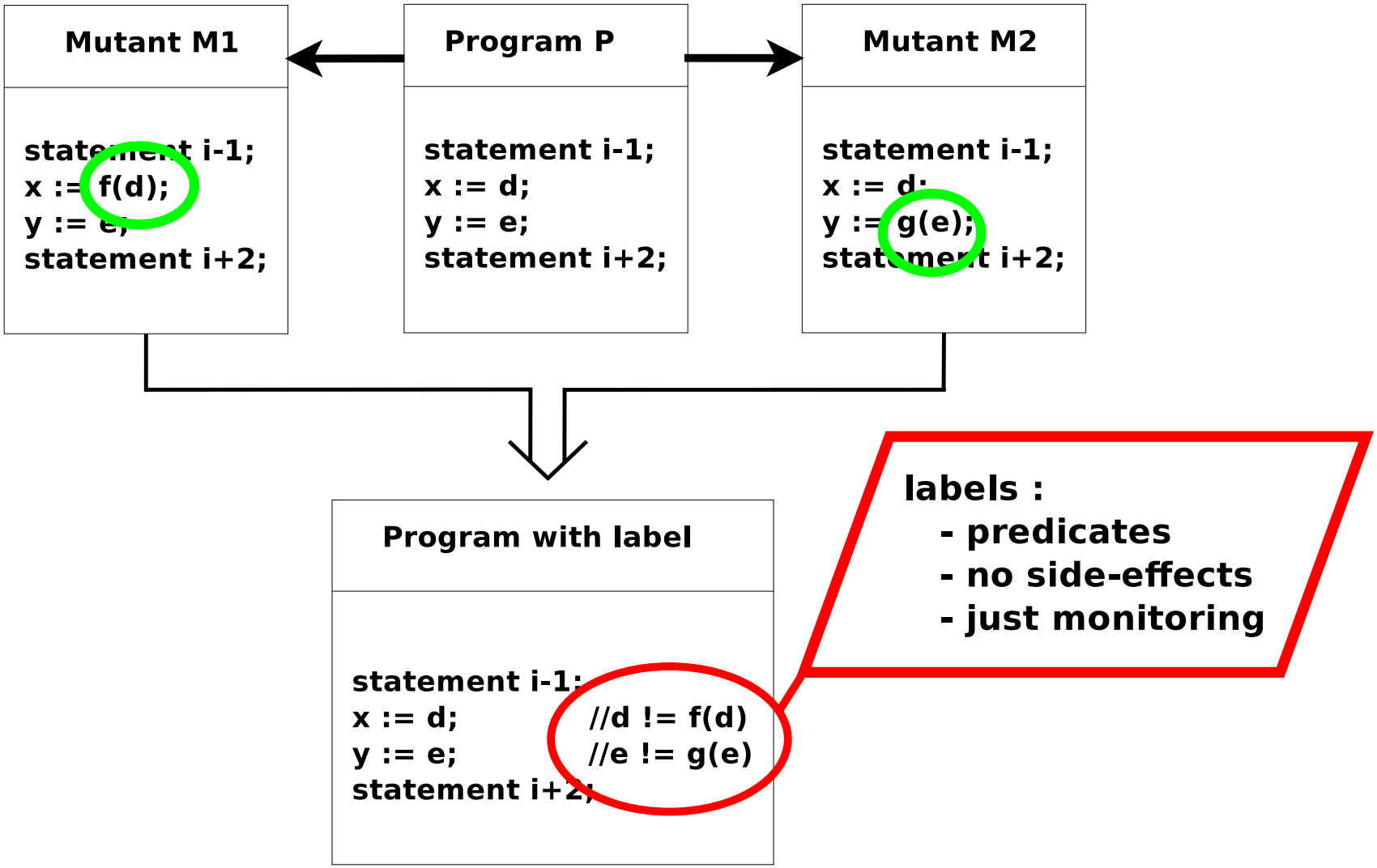
- applications to vulnerabilities and reverse [SANER 16]

Focus : Simulation of Weak Mutations



- mutant M = syntactic modification of program P
- weakly covering M = finding t such that $P(t) \neq M(t)$ just after the mutation

From weak mutants to labels (1)



From weak mutants to labels (2)

One label per mutant

Mutation inside a statement

- `lhs := e` $\mapsto$ `lhs := e'`
  - add label : $e \neq e'$
- `lhs := e` $\mapsto$ `lhs' := e`
  - add label : $\&lhs \neq \&lhs' \wedge (lhs \neq e \vee lhs' \neq e)$

Mutation inside a decision

- `if (cond)` $\mapsto$ `if (cond')`
  - add label : $cond \oplus cond'$

Beware : no side-effect inside labels

From weak mutants to labels (2)

One label per mutant

Mutation inside a statement

Theorem

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

Mutation inside a decision

- `if (cond)` $\mapsto$ `if (cond')`
  - add label : $cond \oplus cond'$

Beware : no side-effect inside labels

Invariant export strategies

```
int fun(int a, int b, int c) {
  //@assume a [...]
  //@assume b [...]
  //@assume c [...]
  int x=c;



  //@assert a < b
  if(a < b)
    {...}
  else
    {...}
}
```

Parameters annotations

Invariant export strategies

```
int fun(int a, int b, int c) {



  int x=c;

  //@assume a [...]
  //@assume b [...]
  //@assert a < b
  if(a < b)
    {...}
  else
    {...}
}
```

Label annotations

Invariant export strategies

```
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
    {...}
}
```

Complete annotations

Invariant export strategies

```
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).