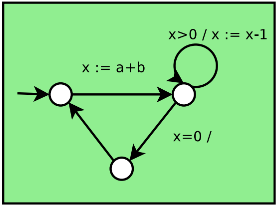
Refinement-Based CFG Reconstruction from Unstructured Programs

Sébastien Bardin, Philippe Herrmann, Franck Védrine

CEA LIST
(Paris, France)

Binary code analysis

Model



Source code

```
int foo(int x, int y) {
 int k= x;
 int c=y;
 while (c>0) do {
   k++;
   c--;}
 return k;
}
```

Assembly

```
_start:
  load  A 100
  add B A
  cmp B 0
  jle label

label:
  move @100 B
```

Executable

```
ABFFF780BD70696CA101001BDE45
145634789234ABFFE678ABDCF456
5A2B4C6D009F5F5D1E0835715697
145FEDBCADACBDAD459700346901
3456KAHA305G67H345BFFADECAD3
00113456735FFD451E13AB080DAD
344252FFAADBDA457345FD780001
FFF22546ADDAE989776600000000
```

Binary code analysis at a glimpse

Recent renew interest

[Codesurfer/x86, SAGE, Jakstab, Osmose, TraceAnalyzer, McVeto, Vine, BAP]

Many promising applications

- off-the-shelf components (including libraries)
- mobile code (including malware)
- third-party certification

Advantages over source-code analysis

- always available
- no "compilation gap"
- allows precise quantitative analysis (ex : wcet)

Very challenging

- conceptual challenges
- practical issues

Outline

- A gentle introduction to binary-level program analysis
- Focus : refinement-based CFG reconstruction
- Conclusion and perspectives

Main challenges of binary code analysis

Low-level semantic of data

Low-level semantic of control [see technical focus]

Practical issues

PB1 : Low-level semantic of data

machine (integer) arithmetic

- overflows, flags

bit-vector operations

- bitwise logical operations, shifts, rotate, etc.

systematic usage of memory (stack)

- only very few variables and one single very large array

up-to-date formal techniques do not adress well these issues

PB1 : Low-level semantic of data (2)

Example 1 : value analysis with machine arithmetic (8 bit)

- $[250..255] + 5 = [0..4] \cup [255]$
- with any convex-domain : $[250..255] +^{\#} 5 = [0..255]$

Example 2 : decision procedures with machine arithmetic

- a popular theory on integers is difference logic $\bigwedge_i x_i - y_i \leq k_i$
- reasonably expressive and in **P**
- but difference logic over modular arithmetic is **NP-hard**

Example 3 : reified comparisons + move from memory to registers

- `R := @100; Flag := cmp(R,0); assert(Flag == 1);`
- perfect deduction after `assert` :
  $Flag = 1 \wedge R = 0 \wedge @100 = 0$
- standard forward deduction after `assert` :
  $Flag = 1$

PB2 : Low-level semantic of control

No clear distinction between data and control

No clean encapsulation of procedure calls

Dynamic jumps (`goto R0`) [the enemy !]

And others : instruction overlapping, self-modifying code

Recovering the Control Flow Graph (CFG) is already non-trivial

PB3 : Practical issues

Engineering issue : many different (large) ISAs

- supporting a new ISA : time-consuming, error-prone, tedious
- consequence : each tool support only a few ISAs (often one !)

Semantic issue : each tool comes with its own formal(?) model

- exact semantics seldom available
- modelling hypothesises often unclear

Consequences

- lots of redundant engineering work between analysers
- difficult to achieve empiric comparisons
- difficult to combine / reuse tools

A renew of interest since 2000's

- CFG reconstruction [Reps et al.] [Kinder et al.] [Brauer et al.] [BHV]
- variables and types recovery [Reps et al.]
- test data generation [Godefroid et al.] [BH]
- malware analysis and other security analyses [Song et al.]
- semantics [Reps et al.] [Bardin et al.] [Brumley et al.]
- dedicated Dagstuhl seminar in 2012

More or less related topics

Analysis of low-level C programs

- many low-level constructs : `*f`, `longjump`, stack overflow, etc.
- BUT
  - ANSI-C forbids most of the nasty behaviours
  - most analyzers consider a very nice subset of C

Analysis of Java bytecode

- Java byte-code is very high level
  - strong static typing for primitive types
  - clean functional abstraction
  - very restricted dynamic jumps

Analysis of assembly languages

- should be the same than binary code
- but often rely on very optimistic assumptions
  - no hidden instruction, sets of dynamic jumps known in advance, call/return policy

Binary-level program analysis at CEA

Osmose [ICST-08,ICST-09,STVR-11]

- automatic test data generation (dynamic symbolic execution)
- bitvector reasoning [TACAS-10]
- front-ends : PPC, M6800, Intel c509

TraceAnalyzer [VMCAI-11] [see technical focus]

- safe CFG reconstruction (refinement-based static analysis)
- front-end : PPC

Dynamic Bitvector Automata [CAV-11]

- concise formal model for binary code analysis
- basic tool support : OCaml type, XML DTD
- safe DBA reduction

Outline

- A gentle introduction to binary-level program analysis
- Focus : Refinement-based CFG reconstruction
- Conclusion and perspectives

CFG recovery

A key issue for binary-level program analysis

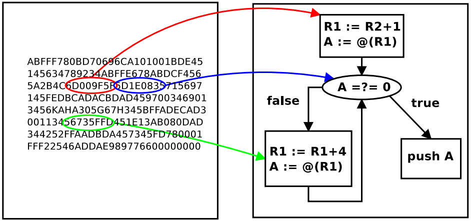
- prior to any other static analysis (SA)
- must be safe : otherwise, other SA unsafe
- must be precise : otherwise, other SA imprecise

Our approach [VMCAI-11]

- safe, precise, efficient and robust technique
- based on abstraction-refinement

Input

- an executable file, i.e. an array of bytes
- the address of the initial instruction
- a basic decoder : exec f. $\times$ address $\mapsto$ instruction $\times$ size



Output : CFG of the program

CFG reconstruction (2)

Successor addresses are often syntactically known

- ⟨ `addr: move a b` ⟩ $\rightarrow$
- ⟨ `addr: goto 100` ⟩ $\rightarrow$
- ⟨ `addr: ble 100` ⟩ $\rightarrow$

CFG reconstruction (2)

Successor addresses are often syntactically known

- ⟨ `addr: move a b` ⟩ → successor at `addr+size`
- ⟨ `addr: goto 100` ⟩ → successor at `100`
- ⟨ `addr: ble 100` ⟩ → successors at `100` and `addr+size`

CFG reconstruction (2)

Successor addresses are often syntactically known

- ⟨ `addr: move a b` ⟩ → successor at `addr+size`
- ⟨ `addr: goto 100` ⟩ → successor at `100`
- ⟨ `addr: ble 100` ⟩ → successors at `100` and `addr+size`

But not always : successors of ⟨`addr: goto a` ⟩ ?

CFG reconstruction (2)

Successor addresses are often syntactically known

- ⟨ `addr: move a b` ⟩ → successor at `addr+size`
- ⟨ `addr: goto 100` ⟩ → successor at `100`
- ⟨ `addr: ble 100` ⟩ → successors at `100` and `addr+size`

But not always : successors of ⟨`addr: goto a` ⟩ ?

Dynamic jump is the enemy !

Know your enemy

Dynamic jumps are pervasive [introduced by compilers]

- `switch`, function pointers, virtual methods, etc.

Sets of jump targets lack regularity [arbitrary values from compiler]

- convex sets plus congruence information are not well-suited

False jump targets cannot be easily detected

- almost any address in an exec. file correspond to a legal instruction
- no pragmatic trick like "detect pb - warn user - correct value"

Unsafe approaches to CFG recovery

... current industrial practise ...

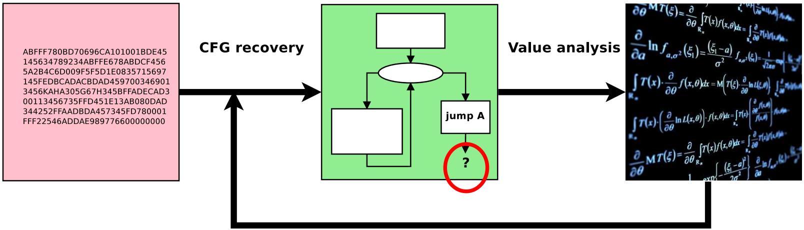
Linear sweep decoding [brute force]

- decode instructions at each code address
- miss every "dynamic" edge of the CFG
- may still miss instructions [too optimistic hypothesises]

Recursive traversal

- decode recursively from entry point, stop on dynamic jump
- miss large parts of CFG

VA and CFG reconstruction must be interleaved



Very difficult to get precise : imprecision on jumps in VA $\rightarrow$ imprecision on CFG $\rightarrow$ more propagation / imprecision on VA $\rightarrow$ ...

Existing safe approaches

CodeSurfer/x86 [Balakrishnan-Reps 04,05,07,...]

- abstract domain : strided intervals (+ affine relationships)

- imprecise : abstract domain not suited to sets of jump targets (arbitrary values from compiler)
- in practise many false targets

Jakstab [Kinder-Veith 08,09,10]

- abstract domain : sets of bounded cardinality (k-sets)
- precise when the bound k is well-tuned

- not robust to the parameter k : possibly inefficient if k too large, very imprecise if k not large enough

Our work

Key observations

- k-sets are the only domain well-suited to precise CFG reconstruction
- for most programs, only a few facts need to be tracked precisely to resolve dynamic jumps
- good candidate for abstraction-refinement

Contribution [VMCAI 2011]

- A refinement-based approach dedicated to CFG reconstruction
- An implementation and a few experiments
- The technique is safe, precise, robust and efficient

Formalisation

Unstructured Programs : $P = (L, V, A, T, l_0)$

- $L \subseteq \mathbb{N}$ finite set of code addresses
- V finite set of program variables
- A finite set of arrays
- T maps code addresses to instructions
- l_0 initial code address

Instructions

- assignments v`:=`e and $a[e_1]$`:=`e_2
- static jumps `goto` l
- branching instructions `ite`$(cond, l_1, l_2)$
- dynamic jumps `cgoto`(v)

Formalisation (2)

Our problem

- input : an unstructured program P
- output : compute an invariant of P such that no dynamic target expression evaluates to $\top$, or fail

Informal requirements

- do not fail “too often”
- do not add “too many” false targets

Sketch of the procedure

Abstract domain : k-sets with local cardinality bounds

- gain efficiency through loss of precision
- still a global bound *Kmax* over local bounds
- domain refinement = increase some k-set cardinality bounds

Ingredient 1 : (slightly) modified forward propagation

- propagation takes local bounds into account
- add tags to $\top$-values to record origin : $\top, \top_{init}, \top_{\langle c_1,\ldots,c_n \rangle}$
  - dedicated propagation rules : $\top_{init}$ and $\top_{\langle \ldots \rangle}$ stay in place
  - pinpoint "initial sources of precision loss" (ispl)
  - give clues for refinement (where and how much)

Ingredient 2 : refinement mechanism

- decide which local bound must be updated, to which value
- helped by $\top$-tags

The procedure

Procedure PaR : $(P, Kmax) \mapsto ?Invariant(P)$

1. Dom $:= \{(loc, v) \mapsto 0\}$
2. forward propagate until a dynamic target exp. evaluates to $\top$
3. if no target exp. evaluates to $\top$, return the fixpoint (OK !)

   else, try to refine the domain to avoid fault

   - if no refinement then fail (KO !)
   - else restart with refined domain (goto 2)

Refinement

For each target evaluating to $\top$

- follows backward data dependencies
- only interested in $\top$-values (other locations are safe until now)
- only interested in correcting initial causes of precision loss

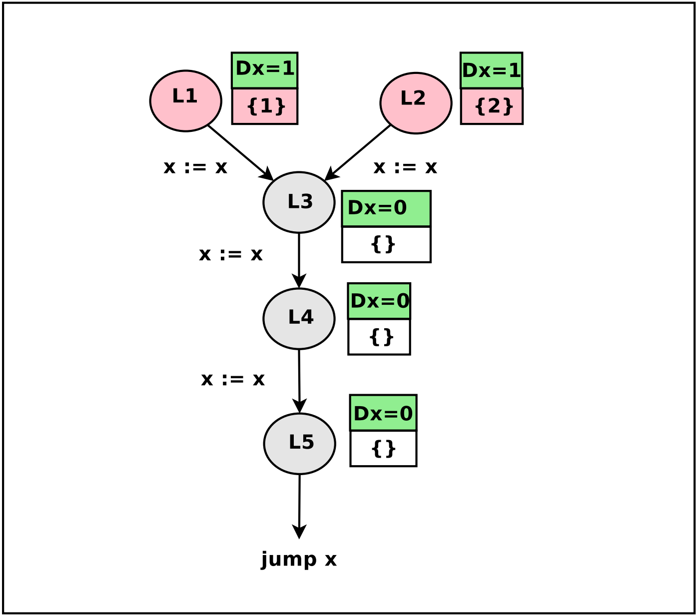
Finding the initial causes of precision loss

- initial causes of precision loss are of the form $\top_{init}, \top_{\langle c_1,\ldots,c_n \rangle}$

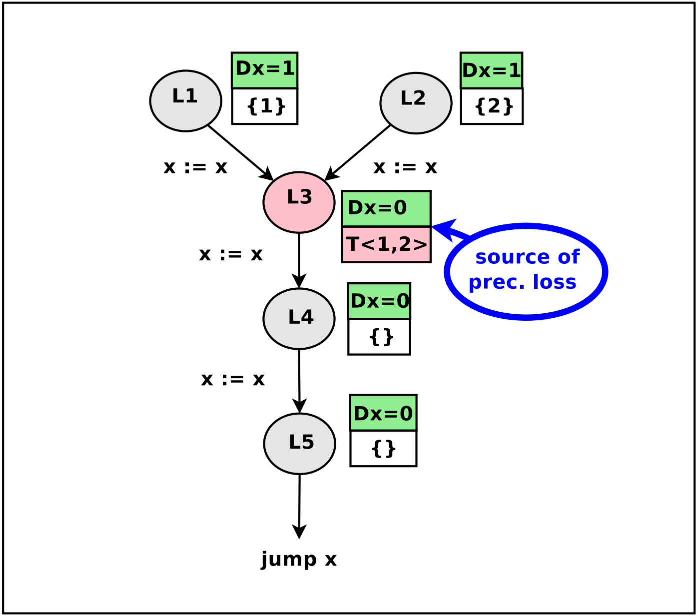
How to correct

- $\top_{init}$ cannot be avoided
- $\top_{\langle c_1,\ldots,c_n \rangle}$ may be avoided if $n \leq Kmax$ (set local bound to n)

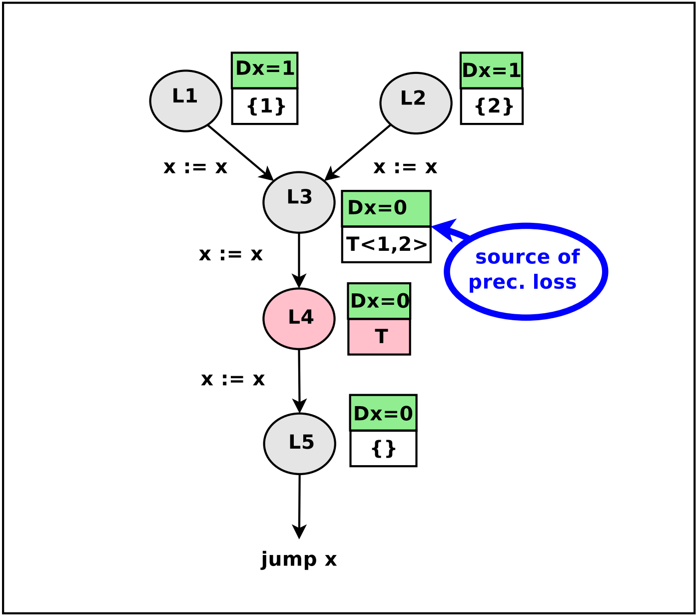
Example



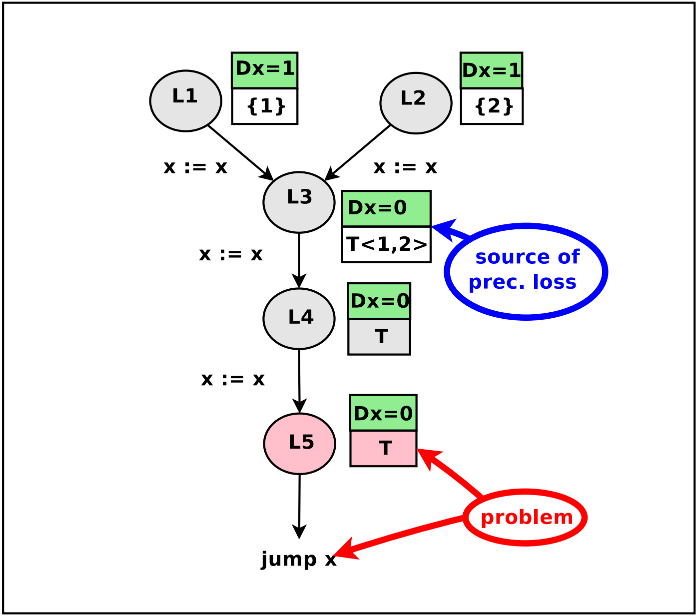
Example


L1
Dx=1
{1}
L2
Dx=1
{2}
x := x
x := x
L3
Dx=0
T<1,2>
source of
prec. loss
x := x
L4
Dx=0
{}
x := x
L5
Dx=0
{}
jump x

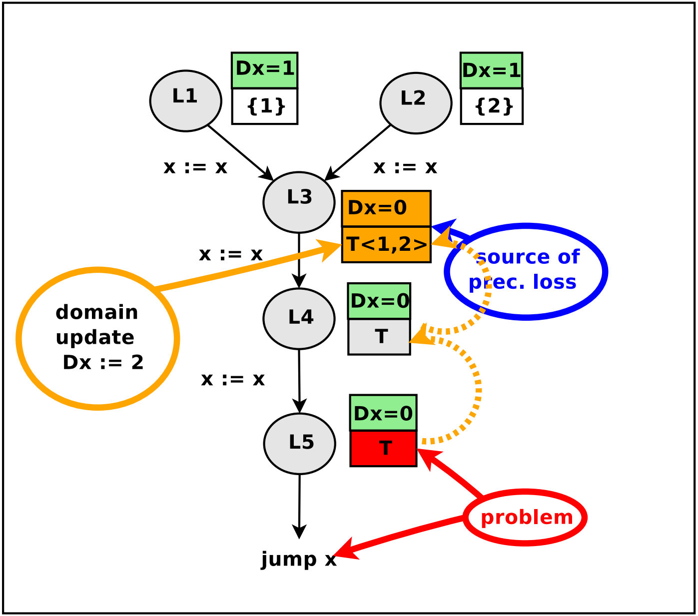
Example



L1
Dx=1
{1}
L2
Dx=1
{2}
x := x
x := x
L3
Dx=0
T<1,2>
source of
prec. loss
x := x
L4
Dx=0
T
x := x
L5
Dx=0
{}
jump x

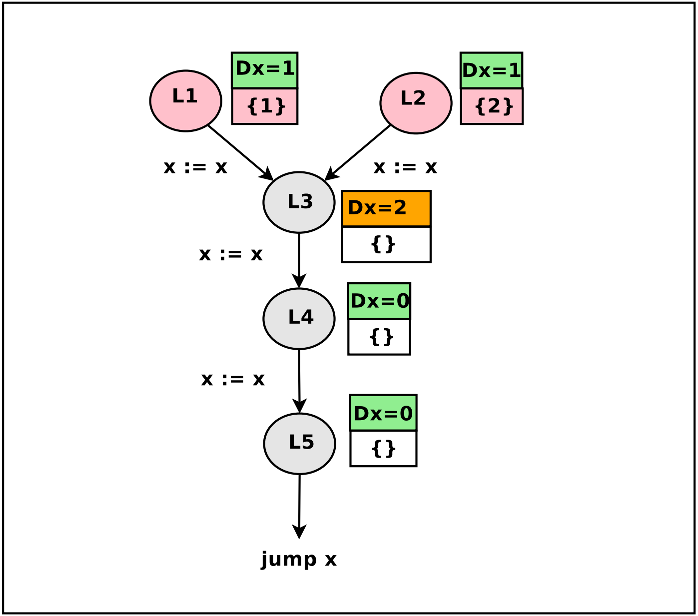

L1
Dx=1
{1}
L2
Dx=1
{2}
x := x
x := x
L3
Dx=0
T<1,2>
source of
prec. loss
x := x
L4
Dx=0
T
x := x
L5
Dx=0
T
problem
jump x

Example

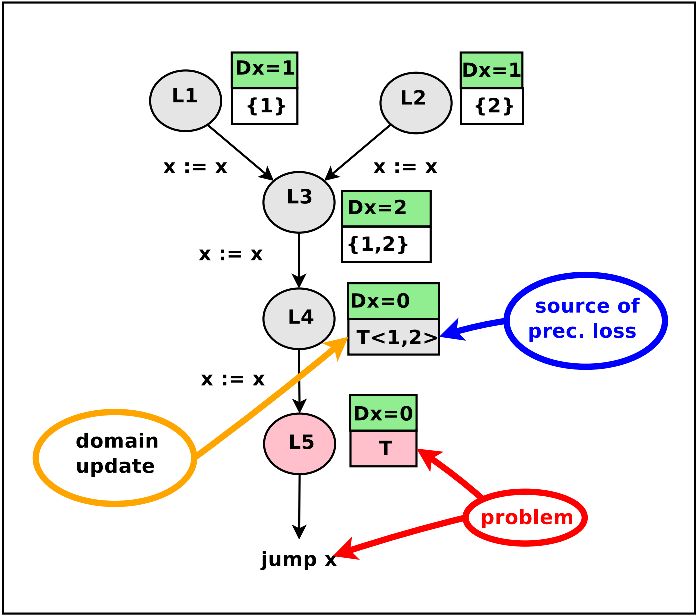


L1
Dx=1
{1}
L2
Dx=1
{2}
x := x
x := x
L3
Dx=0
T<1,2>
source of
prec. loss
x := x
domain
update
Dx := 2
L4
Dx=0
T
x := x
L5
Dx=0
T
problem
jump x

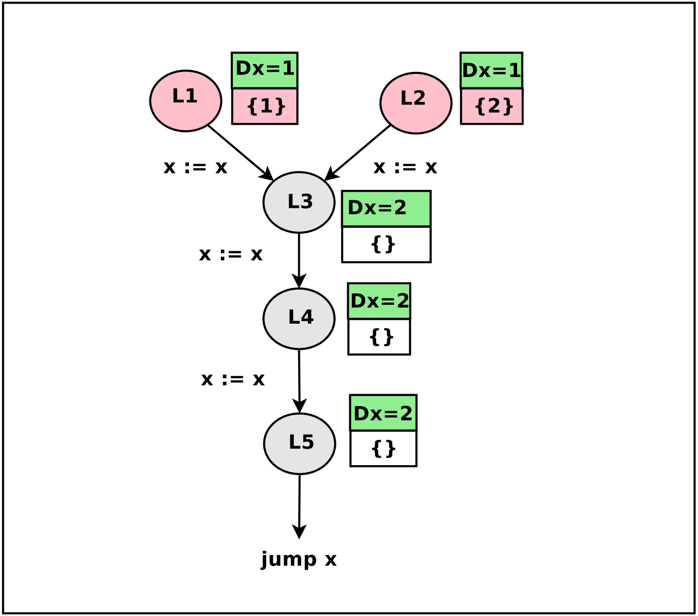
Example



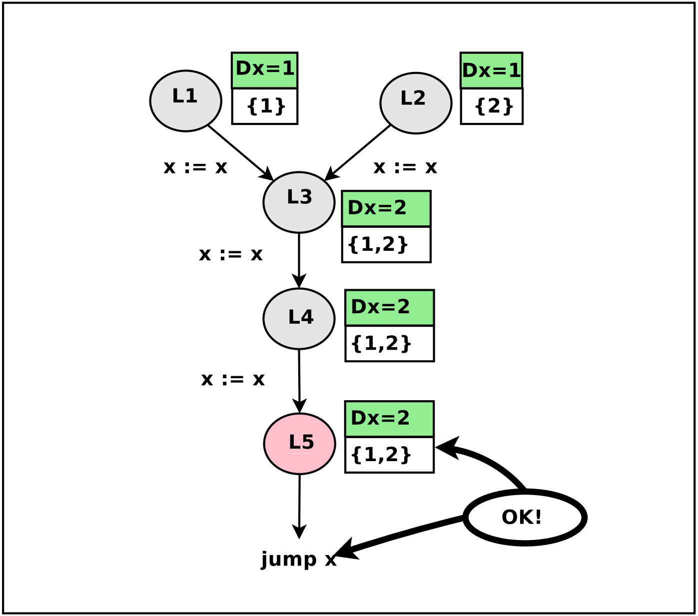
Example


L1
Dx=1
{1}
L2
Dx=1
{2}
x := x
x := x
L3
Dx=2
{1,2}
x := x
L4
Dx=0
T<1,2>
source of prec. loss
x := x
domain update
L5
Dx=0
T
problem
jump x

Example



Example



Technical detail I : failure policy

Two possible failure policies during refinement

- optimistic : fails only when no ispl is corrected
- pessimistic : fails as soon as one ispl cannot be corrected

Optimistic policy succeeds more, but more refinements

Pessimistic policy fails earlier, but may unduly fail

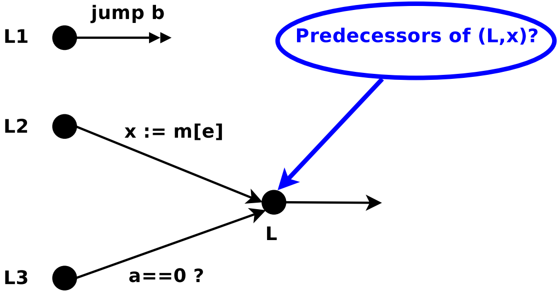
ispl computation	under-approx	exact	over-approx
pessimistic	x	RC	x
optimistic	x	RC	RC (perf --)

RC : relative completeness [see after]

Technical detail II : journal

Problem during ispl search

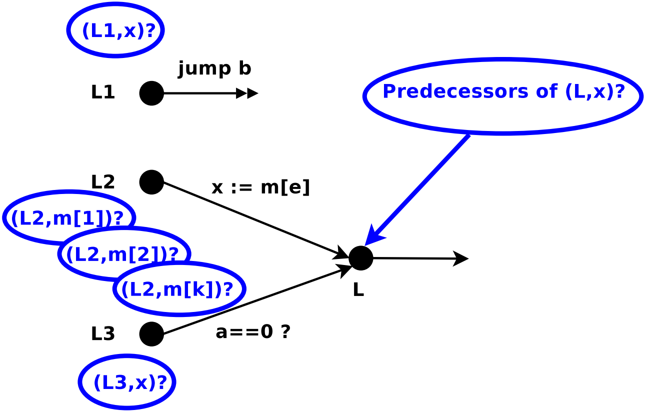
- syntactic computation of (data) predecessors (for assignments with alias and dynamic jumps) is either unsafe or imprecise
  [cf failure policy]



Technical detail II : journal

Problem during ispl search

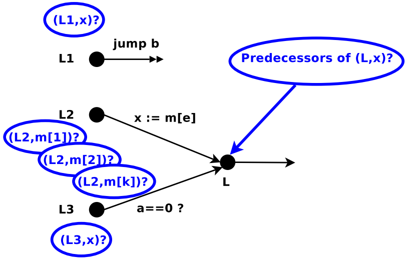
- syntactic computation of (data) predecessors (for assignments with alias and dynamic jumps) is either unsafe or imprecise [cf failure policy]



Technical detail II : journal

Problem during ispl search

- syntactic computation of (data) predecessors (for assignments with alias and dynamic jumps) is either unsafe or imprecise
  [cf failure policy]



Solution : a journal of the propagation

- record observed feasible branches / alias / dynamic targets
- prune backward data dependencies accordingly
- updated during propagation, used during ispl search

Basic theoretical properties

Soundness : PaR(P,Kmax) returns either an invariant such that no jump target evaluates to $\top$, or FAIL

Complexity : polynomial number of refinements

Completeness : perfect relative completeness for a non trivial subclass of programs (see next)

What about the quality of the refinement ?

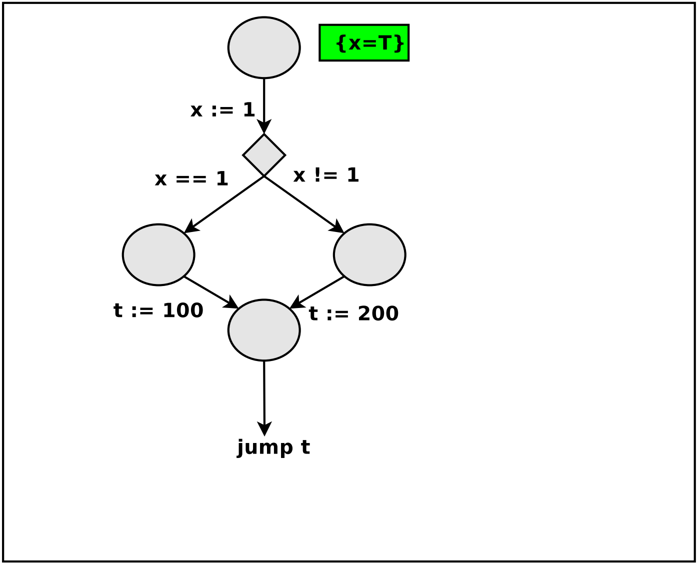
Relative completeness (RC) : PaR is relatively complete if PaR($P, Kmax$) returns successfully when the forward k-set propagation with parameter $Kmax$ does

Bad news : no RC in the general case

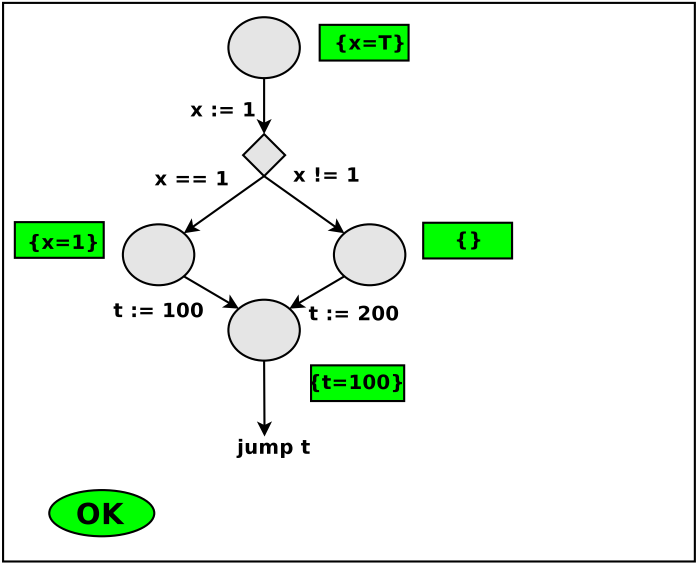
- mainly because of control dependencies

Good news : RC for a non trivial subclass of programs

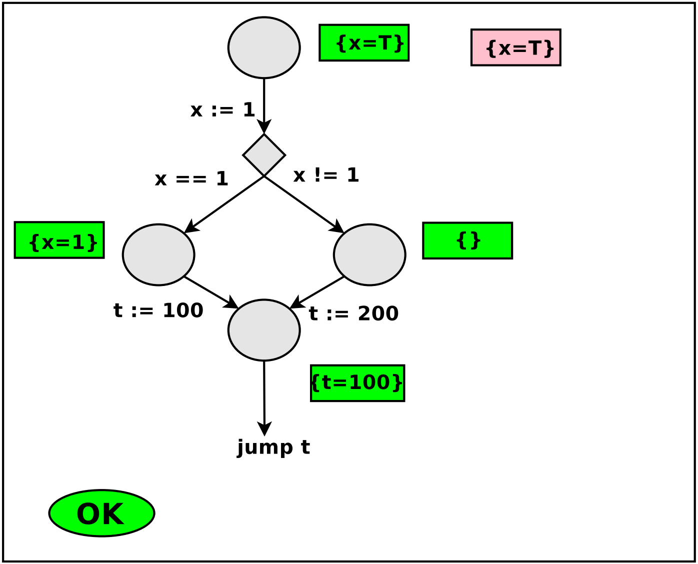
let us suppose $Kmax = 1$



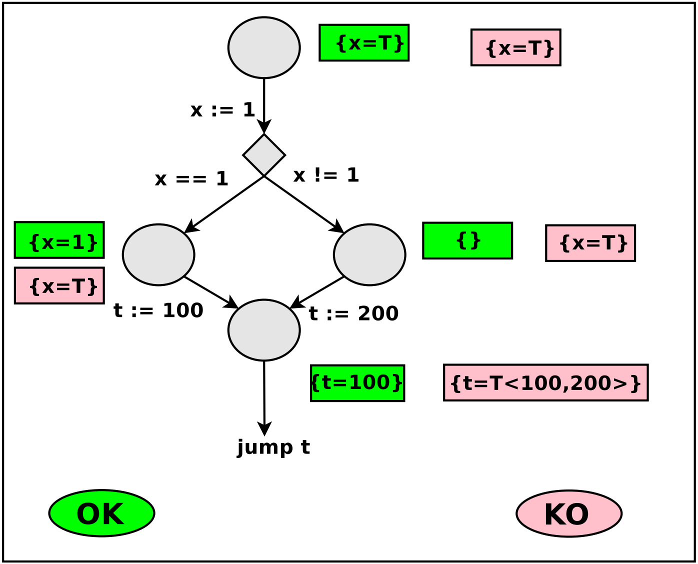
let us suppose $Kmax = 1$



let us suppose $Kmax = 1$



let us suppose $Kmax = 1$



RC on a subclass

RC holds for a subclass of unstructured programs
[even with "pessimistic failure"]

- Non-deterministic branching [new : all branches feasible]
- only $\top$-propagating operators ($+, -, \times k$ ok, but not $\times$)
- guarded aliases

skip proof

RC on a subclass : sketch of proof

Reason over traces of the forward propagation procedure

From faulty trace in PaR, build faulty trace in $\rightarrow^*_{D^{max}}$

⋆ Assume

- $M_0 \xrightarrow{\pi}_D M_n$, $M_n(l_n, v_n) = \top$ *// failure*
- refinement fails on M_n and (l_n, v_n)

⋆ Prove that $M_0 \xrightarrow{\pi}_{D^{max}} M'_n$, $M'_n(l_n, v_n) = \top$

Proof steps

- prove for restricted[2] subclass : no jump / alias
- generalisation : guarded jumps and guarded aliases

sketch of proof (2)

Fragment with < NDBranching - no alias - no dynamic jump >

Find a non correctable ispl of (l_n, v_n) such that

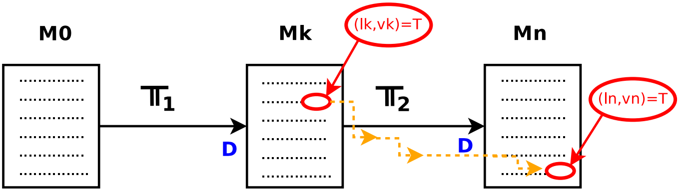
- $\pi = \pi_1 \cdot \pi_2$
- $M_0 \xrightarrow{\pi_1}_D M_k \xrightarrow{\pi_2}_D M_n$
  and (l_k, v_k) ispl of (l_n, v_n)
  and
  $$k = 0, M_k(l_k, v_k) = \top_{init}$$
  or
  $$M_k(l_k, v_k) = \top_{\langle c_1 \ldots c_q \rangle}, q > K_{max} \text{ and } M_{k-1}(l_k, v_k) \neq \top$$

We want to prove that

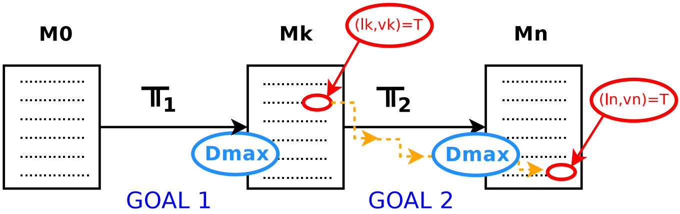
Goal1 ispl (l_k, v_k) still evaluates to $\top$ in D^{max} after π_1
$M_0 \xrightarrow{\pi_1}_{D^{max}} M'_k$ and $M'_k(l_k, v_k) = \top$

Goal2 value of (l_k, v_k) still propagate to (l_n, v_n) in D^{max} after π_2
$M'_k \xrightarrow{\pi_2}_{D^{max}} M'_n$ and $M'_n(l_n, v_n) = \top$

ASSUME



PROVE



sketch of proof (3)

Two fundamental lemmas

Lemma 1 : $\xrightarrow{\sigma}_D$ and $\xrightarrow{\sigma}_{D^{max}}$ computes the same proper k-sets

- hint : the only cause of precision loss is early $\top$-cast
  . does not create bigger proper k-sets, but $\top$
  . we can know if a set is (relatively) approximated or not
- note : very specific to k-sets, false when unfeasible branches

Lemma 2 : $\xrightarrow{\sigma}_D$ and $\xrightarrow{\sigma}_{D^{max}}$ define the same data dependencies

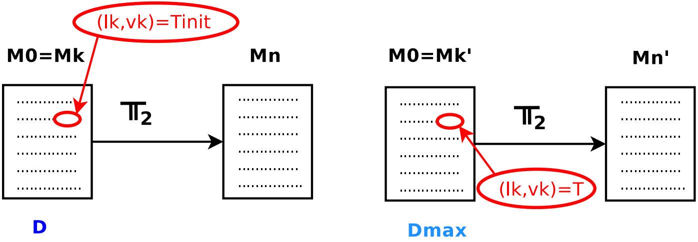
- easy here, all data dep. are static

[the two proofs are interleaved]

Goal1 : ispl (l_k, v_k) still evaluates to $\top$ in D^{max} after π_1
$M_0 \xrightarrow{\pi_1}_{D^{max}} M'_k$ and $M'_k(l_k, v_k) = \top$

Case 1 : $M_k(l_k, v_k) = \top_{init}$

- $\top_{init}$ created in initial state
- (l_k, v_k) will also take value $\top$ in M'_k

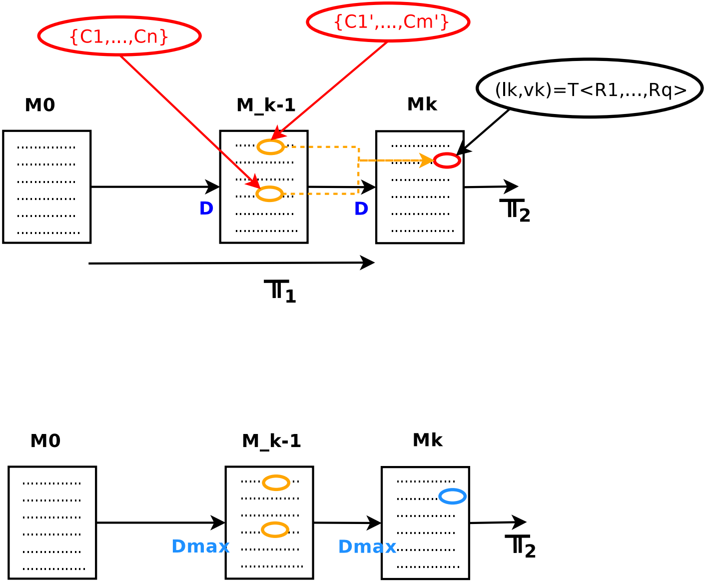


Goal1 : ispl (l_k, v_k) still evaluates to $\top$ in D^{max} after π_1
$M_0 \xrightarrow{\pi_1}_{D^{max}} M'_k$ and $M'_k(l_k, v_k) = \top$

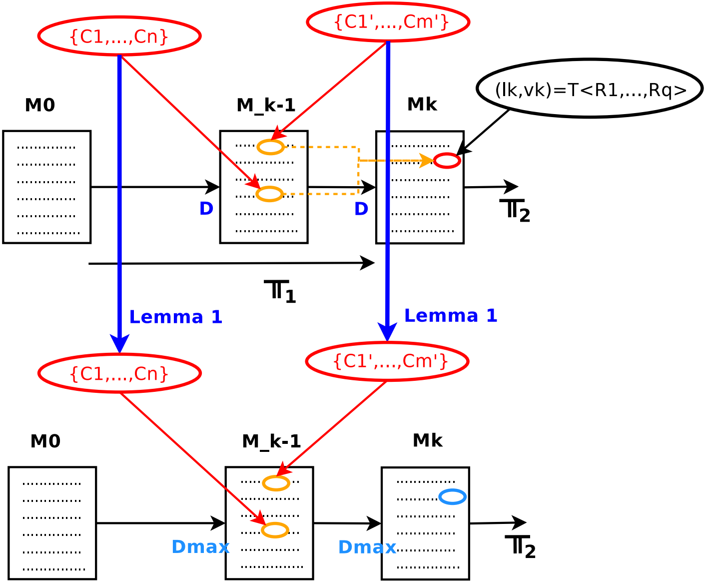
Case 2 : $M_k(l_k, v_k) = \top_{\langle c_1 \dots c_q \rangle}$ and $q > K_{max}$

- $(\star)$ predecessors of (k, l_k, v_k) for $\xrightarrow{\pi_1}_D$ are all proper k-sets
  // rest. op : otherwise $M_k(l_k, v_k) = \top$
- lemma 2 + $(\star)$ + lemma 3 : predecessors of (k, l_k, v_k) for $\xrightarrow{\pi_1}_{D^{max}}$ are the same locations than for $\xrightarrow{\pi_1}_D$, and evaluate to the same proper k-sets
- hence, $M'_k(l_k, v_k) = \alpha_{K_{max}}(\{c_1 \dots c_q\}) = \top$ // $q > Kmax$

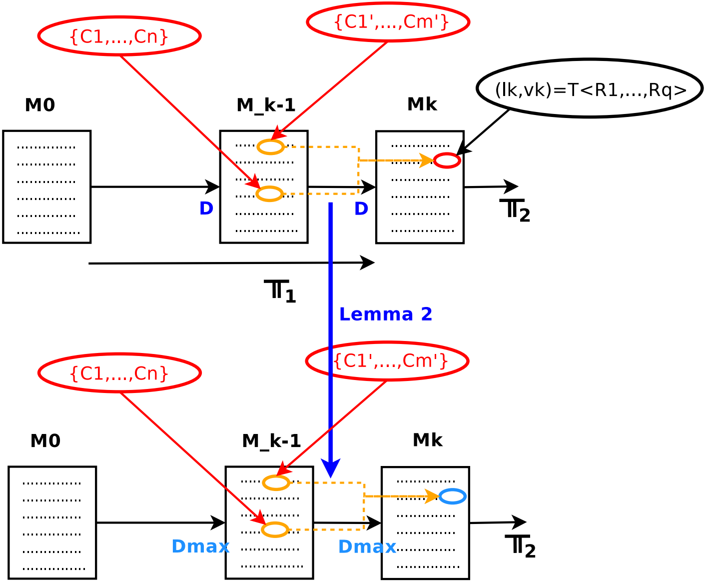
sketch of proof (5') : goal 1, case 2



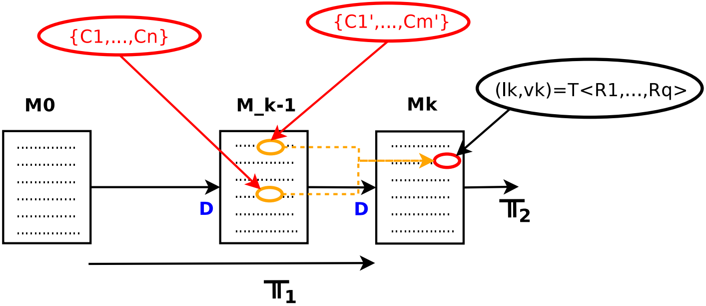
sketch of proof (5') : goal 1, case 2

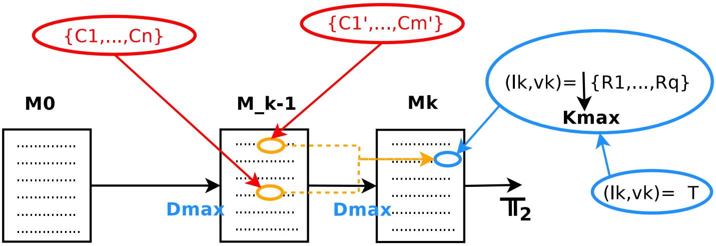


sketch of proof (5') : goal 1, case 2



sketch of proof (5') : goal 1, case 2





sketch of proof (6)

Goal2 : value of (l_k, v_k) still propagate to (l_n, v_n) in D^{max} after π_2
$M'_k \xrightarrow{\pi_2}_{D^{max}} M'_n$ and $M'_n(l_n, v_n) = \top$

- ok because of lemma 2 and restricted operators ($\top$-must propagate)

Full Proof of RC : goal1 + goal2

sketch of proof (7) : generalisation

More general case : guarded alias and guarded dynamic jumps

Basically same technique, handle alias and jumps with care

Key : guarded jumps enforce proper ksets on jump exp, or fail

- lemma 1 still holds (until failure state)
- lemma 2 still holds (until failure state)

  note for both lemma : need the journal to track back only "feasible" ispl

Same trick for guarded aliases

Relative Precision (not in VMCAI 2011)

Relative precision (RP) : PaR is relatively precise if when PaR($P, Kmax$) returns successfully, it returns the same set of targets than the forward k-set propagation with parameter $Kmax$ does

RP holds for the subclass of unstructured programs

Summary : RC+RP (on the restricted subclass)

- PaR($P, Kmax$) terminates iff forward k-set propagation with parameter $Kmax$ does
- in case of success, they compute the same set of targets

Experiments

Implementation

- input : PPC executable + entrypoint
- output : cfg, callgraph, sets of targets, assembly code
- details : procedure inlining, efficient data-structures
- limitation : no dynamic memory allocation
- 29 kloc of C++

Test bench 1 : 12 small hand-written C programs compiled with gcc. From 60 to 1000 PPC instructions

Test bench 2 : Safety-critical program from Sagem

- 32 kloc, 51 dynamic jumps, up to 16 targets a jump

Experiments (2)

Experimental results for the aeronautic program

- precision : resolve every jump, only 7% of false targets

  (standard program analysis cannot recover better than between 400% and 4000% of false targets)

- robustness : efficiency independent of *Kmax* (if large enough)

- locality : tight value of max-k, low value of mean-k

- efficiency : terminates in 5 min
  - already sufficient for some (safety-critical) applications
  - however procedure inlining may be an issue
  - rooms for improvement

Outline

- A gentle introduction to binary-level program analysis
- Focus : Refinement-based CFG reconstruction
- Conclusion and perspectives

News from the front

Improved algorithm [efficiency, robustness]

- # refinements indep. of *Kmax*
- chaining of updates

Compositionality : product of domains KSET $\times$ D

- more precise than just KSET

Implementation

- domain = KSET $\times$ I $\times$ Formulas $x\{<, \leq, =, \geq, >\}y$
- Sagem : $\approx$ 10 sec

Conclusion on CFG reconstruction

Result : an original refinement-based procedure

- truly dedicated to CFG reconstruction [domains, refinement]
- safe, precise, robust and efficient
- both theoretical and empirical evidence

Future work

- experiments on non-critical programs [dynamic alloc]
- ultimate goal : executables coming from large C++ programs

A few last words

Binary code analysis shows both great promises and challenges

Many open problems

- which semantic for binary code ? common formal model ?
- which properties are worth to investigate ?
- is binary-code analysis so different than program analysis ?

A few years ago, only a few scattered teams and works

Things are changing [CAV 11, VMCAI 11, EMSOFT 11, SSV 11]

- time for more collaboration ?
- benchmarks, meetings, workshops / conference, projects ?

Backup slides

Dynamic bitvector automata (DBA)

Osmose

Main design ideas

- small set of instructions
- concise and natural modelling of common ISAs
- low-level enough to allow bit-precise modelling

Can model : instruction overlapping, return address smashing, endianness, overlapping memory read/write

Limitations : (strong) no self-modifying code, (weak) no dynamic memory allocation, no FPA

Dynamic Bitvector Automata (2)

Extended automata-like formalism

- bitvector variables and arrays of bytes
- all bv sizes statically known, no side-effects
- standard operations from BVA

Feature 1 : Dynamic transitions

- for dynamic jumps

Feature 2 : Directed multiple-bytes read and write operations

- for endianness and word load/store

Feature 3 : Memory zone properties

- for (simple) environment

Dynamic Bitvector Automata (2)

Feature 1 : Dynamic transitions

- some nodes are labelled by an address
- dynamic transitions have no predefined destination
- destination computed dynamically via a target expression

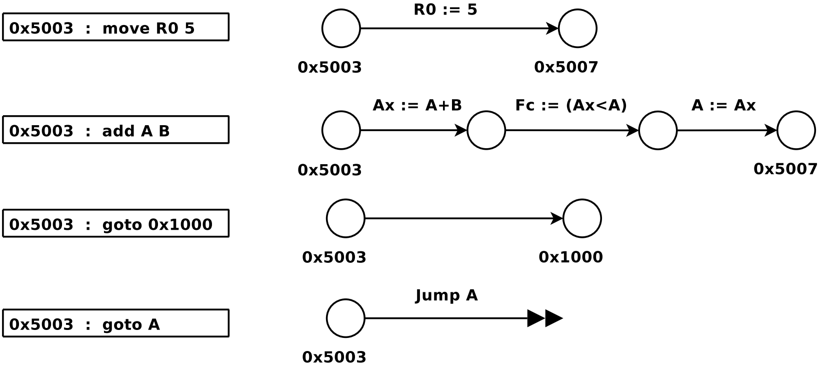
Feature 2 : Directed multiple-bytes read and write operations

- $\mathrm{array}[\mathit{expr}; k^{\#}]$, where $k \in \mathbb{N}$ and $\# \in \{\leftarrow, \rightarrow\}$

Feature 3 : Memory zone properties

- specify special behaviour for some segments of memory
- volatile, write-aborts, write-ignored, read-aborts

Modelling with DBA



Procedure calls / returns : encoded as static / dynamic jumps

Memory zone properties, a few examples : ROM *(write-ignored)*, memory controlled by env *(volatile)*, code section *(write-aborts)*

DBA toolbox

Open-source Ocaml code for basic DBA manipulation

Features

- a datatype for DBAs
- basic "typing" (size checking) over DBAs
- import (export) from (to) a XML format
- DBA simplification (see next)

GPL license, based on xml-light, $\approx$ *3 kloc*

DBA toolbox - simplifications

Goal : simplify unduly complex DBAs typically obtained from instruction-wise translation

- useless flag computations / auxiliary variables / etc.

Inspired by standard compilation techniques [peephole, dead code, etc.]

- beware of partial DBAs and dynamic jumps !
- rethink these standard techniques in a partial CFG setting

Results : size reduction of -50% (all instrs), and between -30% and -50% (non-goto instrs)

Osmose : test data generation

Osmose (CEA) [ICST-08, STVR-11]

- automatic test data generation (dynamic symbolic execution)
- 75 kloc of OCaml, front-ends : PPC, M6800, Intel c509
- case-studies : programs from aeronautics and energy

Supported architectures : Motorola 6800, Intel 8051, Power PC 550

Key technologies

Multiple-architecture support [BH-11]

- Generic assembly language (GAL) [current move to DBAs]

Test data generation through Concolic Execution [BH-08,BH-11]

- exploration of all (bounded) paths of the program
- symbolic reasoning to discover new dynamic targets
- path pruning optimisations [BH-09]

Bit-precise constraint solving [BHP-10]