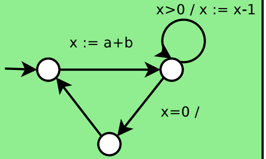
Refinement-Based CFG Reconstruction from Unstructured Programs

Sébastien Bardin, Philippe Herrmann, Franck Védrine

CEA LIST
(Paris, France)

Dagstuhl seminar
January 2012

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-level program analysis at CEA

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

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

CGFBuilder [VMCAI-11]

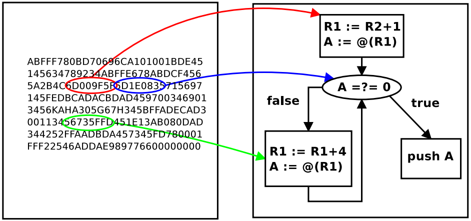
- safe CFG reconstruction (refinement-based static analysis)
- front-end : PPC

Dynamic Bitvector Automata (DBA) [CAV-11]
with Uni. Bordeaux & Paris 7

- concise formal model for binary code analysis
- small set of simple instructions, endianess and flags addressed in a simple way

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` ⟩ → 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.

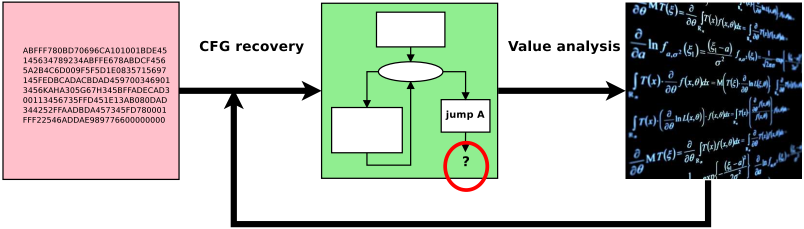
Sets of jump targets lack regularity

- arbitrary values chosen by compiler
- standard domains do not fit

False jump targets cannot be easily detected

- many addresses in an exec. file correspond to legal instructions

VA and CFG reconstruction must be interleaved



Difficulty 1 : small errors on jumps may have dramatic effects

imprecision on jumps in VA → imprecision on CFG → more propagation in VA → more imprecision on VA → . . .

Difficulty 2 : standard domains do not fit

Existing domains do not fit

`jump R`, with $R \in \{500, 530, 1000, 1500\}$

Stride intervals

- $x \in [a..b] \wedge x \equiv c[d]$
- imprecise here : $R \in [500..1500] \wedge x \equiv 500[10]$

Sets of bounded cardinality (k-sets)

- $x \in \{c_1, \ldots, c_q\}$ with $q \leq k$, or $\top$
- very imprecise if k is not sufficient : $R \in \top$
- precise if k is large enough : $R \in \{500, 530, 1000, 1500\}$
- precise but slow if k is too large

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

Our work [VMCAI 2011]

- A refinement-based approach dedicated to CFG reconstruction
- The technique is safe, moreover precise and efficient on our examples

Sketch of the procedure (1)

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 (2)

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

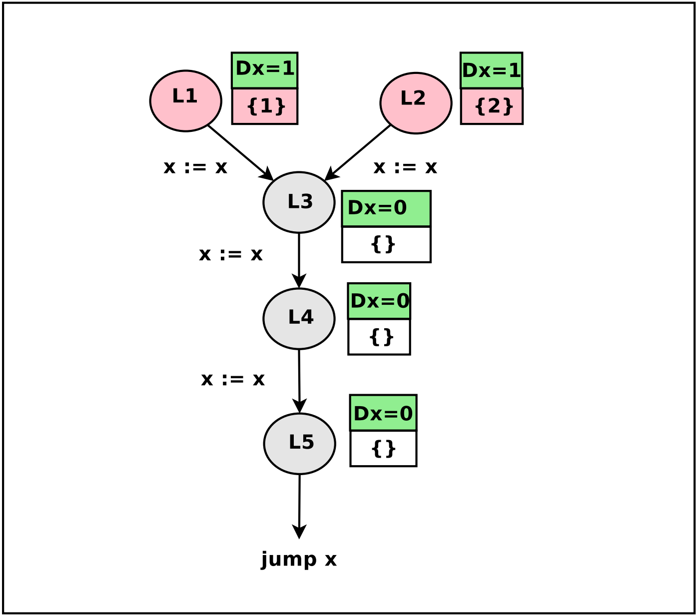
Finding the initial causes of precision loss

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

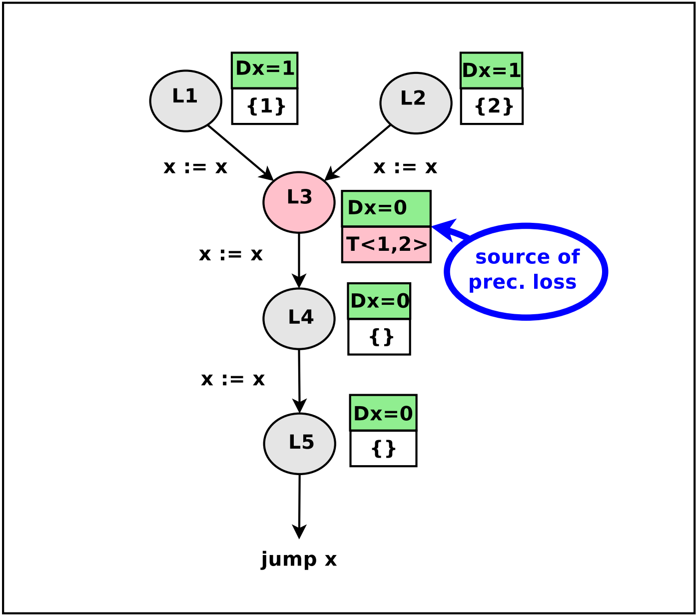
How to correct

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

Example

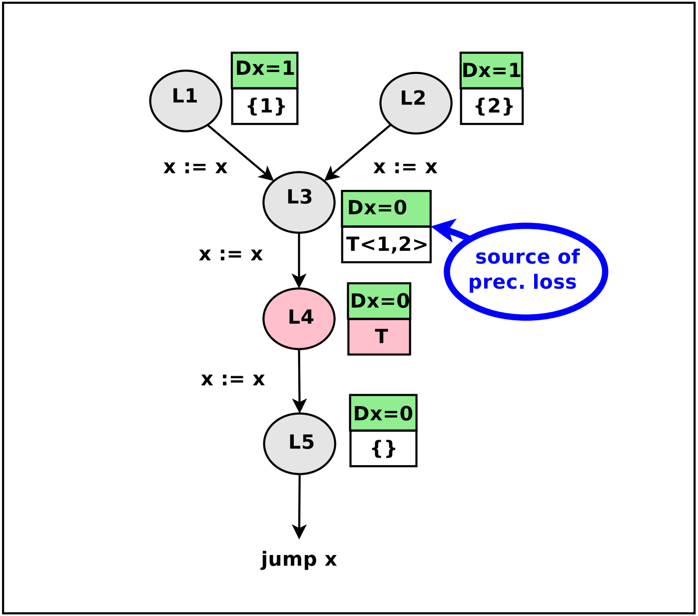


Example

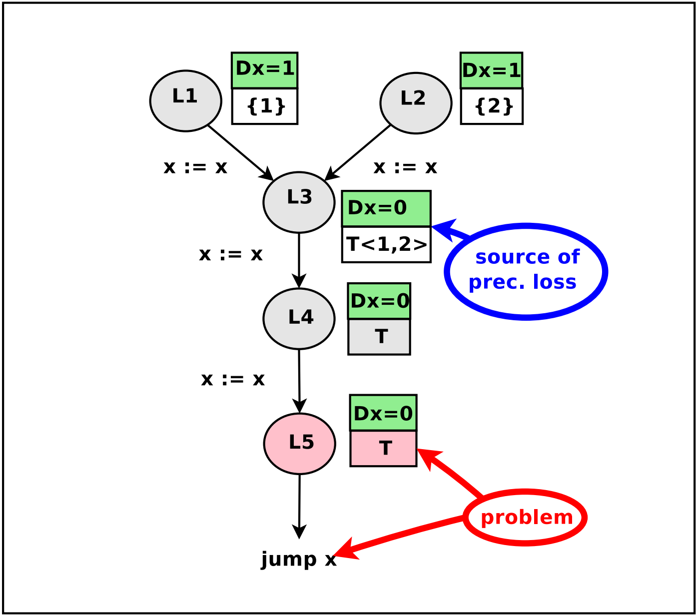


L1
Dx=1
{1}
L2
Dx=1
{2}
x := x
x := x
L3
Dx=0
T<1,2>
source of
prec. loss
x := x
L4
Dx=0
{}
x := x
L5
Dx=0
{}
jump x

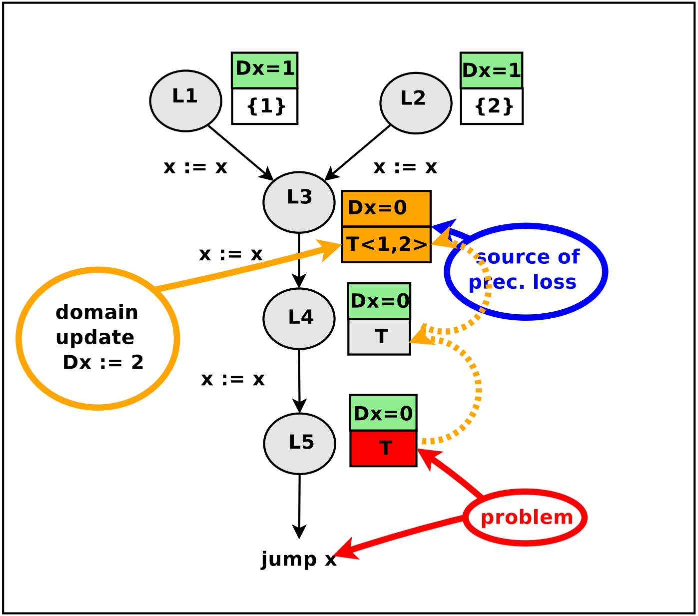
Example


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

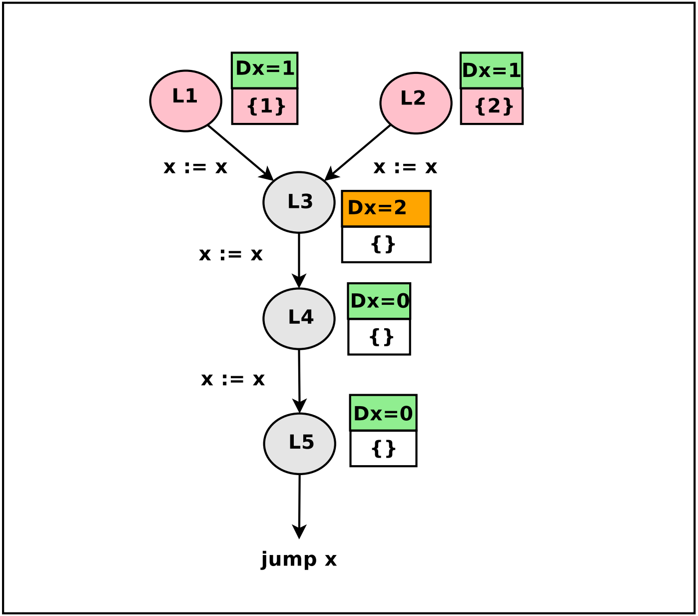
Example

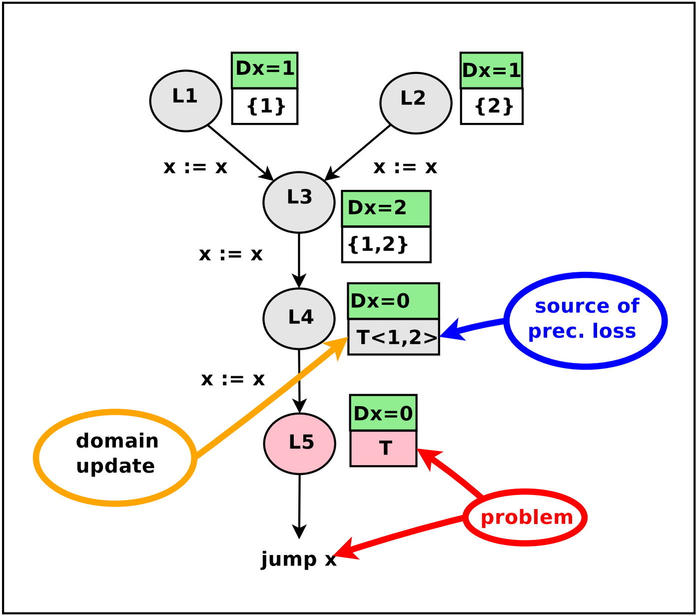

L1
Dx=1
{1}
L2
Dx=1
{2}
x := x
x := x
L3
Dx=0
T<1,2>
source of prec. loss
x := x
L4
Dx=0
T
x := x
L5
Dx=0
T
problem
jump x

Example

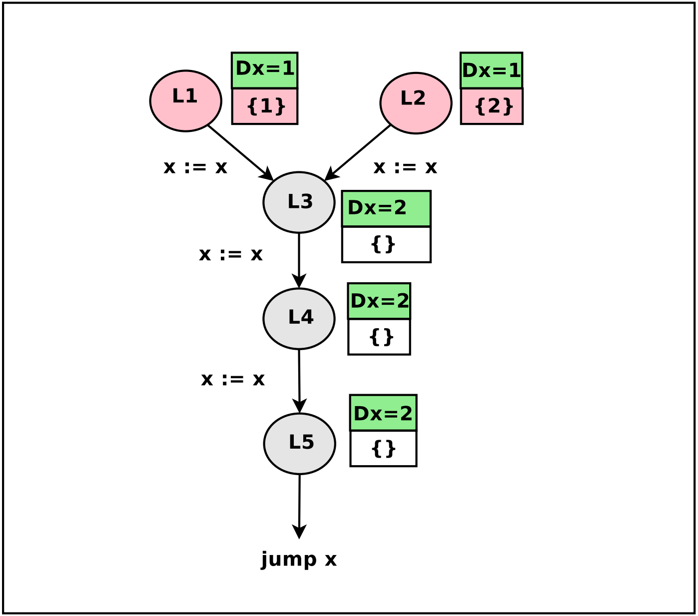

L1
Dx=1
{1}
L2
Dx=1
{2}
x := x
x := x
L3
Dx=0
T<1,2>
source of
prec. loss
x := x
domain
update
Dx := 2
L4
Dx=0
T
x := x
L5
Dx=0
T
problem
jump x

Example

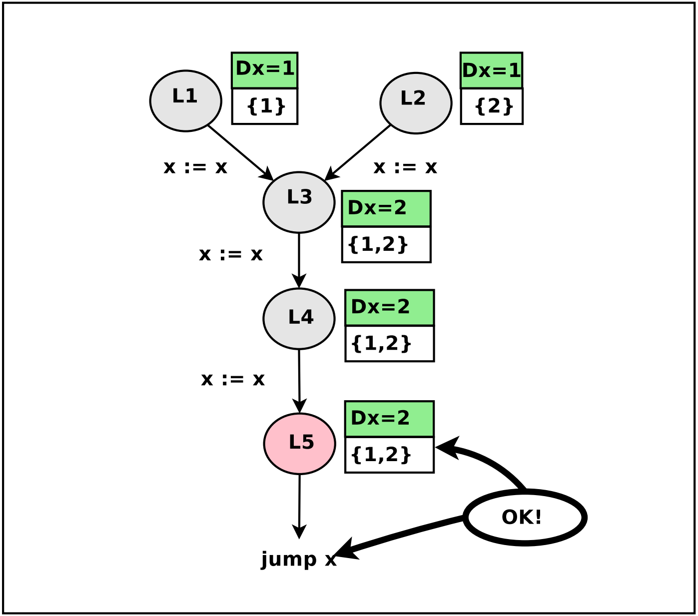



L1
Dx=1
{1}
L2
Dx=1
{2}
x := x
x := x
L3
Dx=2
{1,2}
x := x
L4
Dx=0
T<1,2>
source of
prec. loss
x := x
L5
Dx=0
T
domain
update
problem
jump x

Example

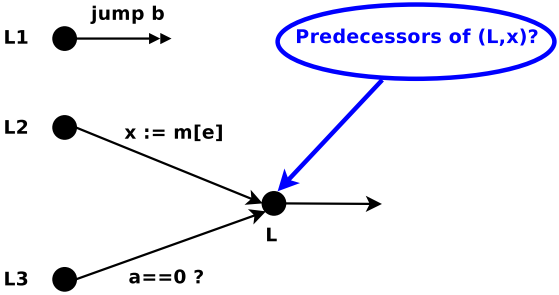


Example



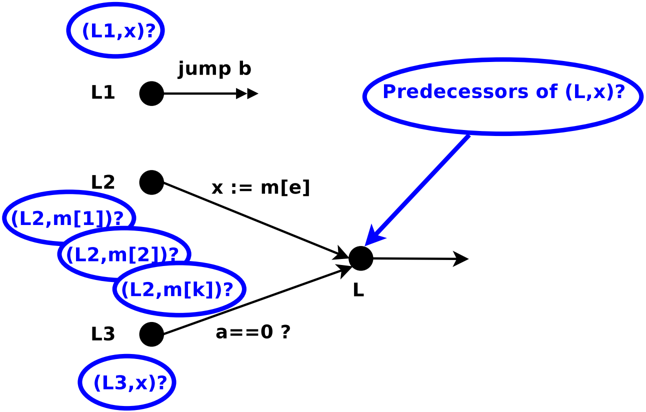
Problem during ispl search

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



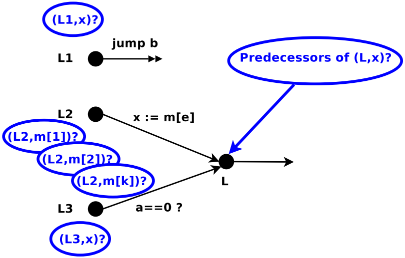
Problem during ispl search

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



Problem during ispl search

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



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

Prototype

- input : PPC executable + entrypoint + initial memory
- output :
  - map from jumps to targets
  - cfg, callgraph, assembly code
- main limitation : no dynamic memory allocation

Prototype (2)

Internal formal model (DBA)

- small set of instructions, no side effects
- concise and natural modelling of common ISAs
- pruning techniques to get rid of useless computations

Procedure inlining

- ⟨ formal stack , `addr` ⟩
- add precision, but no recursion

Memory model

- no difference yet between global memory region and stack (need some initial stack value)
- no dynamic memory allocation

Procedure enhancements

Improved algorithm [efficiency, robustness]

- $\#$ refinements indep. of $Kmax$
- chaining of domain updates

Domain combination [precision]

- equalities : $e = e$, where $e \ ::= \ R|k|@e$
- flags : $b \Leftrightarrow e\{<, \leq, =, \geq, >\}e$
- intervals : $x \in [a..b]$

Procedure enhancements (2)

Case 1 : compile `assume(X == Y)` into :
`R1:=X ; R2 := Y; B := (R1==R2), assume(B)`

- only k-sets : $B \in \{1\}$
- k-sets + equalities : $B \in \{1\} \wedge R_1 = X \wedge R_2 = Y$
- k-sets + equalities + flags : $B \in \{1\} \wedge R_1 = R_2 = X = Y$

Case 2 : prove that `@X := Y` does not affect `jump @100`

- if $X \in [101, +\infty[$, intervals ok, k-sets not ok
- requiring k-sets on write addresses might be overkill

Experiments

program	#I	#DJ	#T	max #T	#SDJ	FT	Time (sec)
aircraft	32405	51	461	16	51/51	10%	20s
SwitchCase	204	1	19	19	1/1	0%	<1s
SingleRowInput	158	1	6	6	1/1	0%	<1s
Keypad	224	1	8	8	1/1	0%	<1s
EmergencyStop	475	1	10	10	1/1	0%	17s
TaskScheduler'	171	1	5	5	1/1	0%	<1s
TaskScheduler	127	1	3	3	0/1	KO	<1s

I : instructions - DJ : dynamic jumps - T : feasible targets

SDJ : # dynamic jumps whose target $\neq \top$

FT : % of recovered false targets

Experiments (2)

- precision : resolve every jump but one, $\leq 10\%$ of false targets
- robustness to initial parameter : efficiency independent of $Kmax$ (if large enough)
- locality : tight value of max-k, low value of mean-k
- efficiency : ok here

Beware : aeronautic software are easier to verify than other software

Place in the design space

Main design choices

- stripped executable : ✓
- return address modification : ✓
- instructions overlapping : ✓
- self-modifying code : ✗
- recursion : ✗
- asynchronous interrupts : ✗

Other points

- float : ✓
- dynamic memory allocation : ●
- OS modelling : ●

Conclusion

Result : an original refinement-based procedure

- truly dedicated to CFG reconstruction [domains, refinement]
- safe
- precise and efficient on a few examples

On going work

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

Main design ideas

- small set of instructions
- concise and natural modelling of common ISAs
- low-level enough to allow bit-precise modelling
- standalone model : do not need any info on archi
- try to be “analysis”-agnostic
- mostly an executable reference semantics

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

- $\mathrm{array}[expr; k^{\#}]$, 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

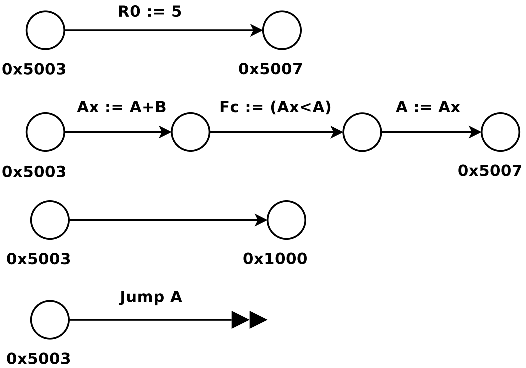
Modelling with DBA

0x5003 : move R0 5

0x5003 : add A B

0x5003 : goto 0x1000

0x5003 : goto A



What about the quality of the refinement ?

Relative completeness (RC) : PaR is relatively complete if PaR($P, Kmax$) returns successfully when $\rightarrow^*_{Kmax}$ does

Relative precision (RP) : PaR is relatively precise if when PaR($P, Kmax$) returns successfully, it returns the same set of targets than $\rightarrow^*_{Kmax}$ does

Bad news : no RC / RP in the general case

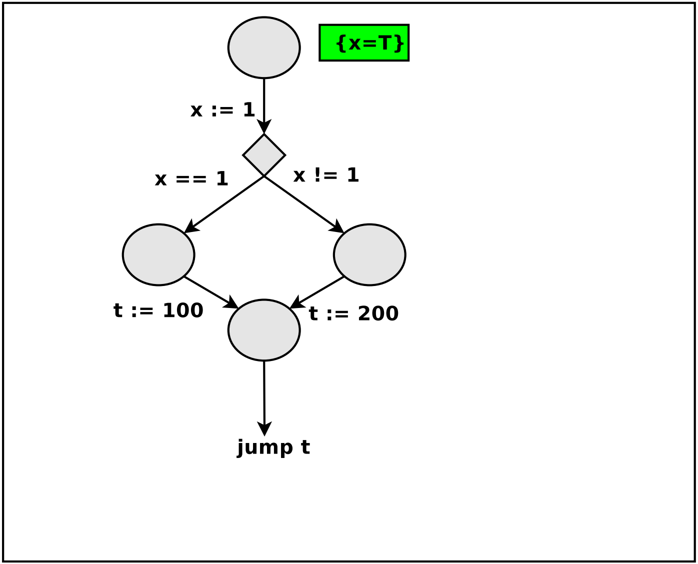
- mainly because of control dependencies

Good news : RC and RP for a non trivial subclass of programs

- non-deterministic branching [new : all branches feasible]
- restricted operators : $+, -, \times k$ ok, but not $\times$

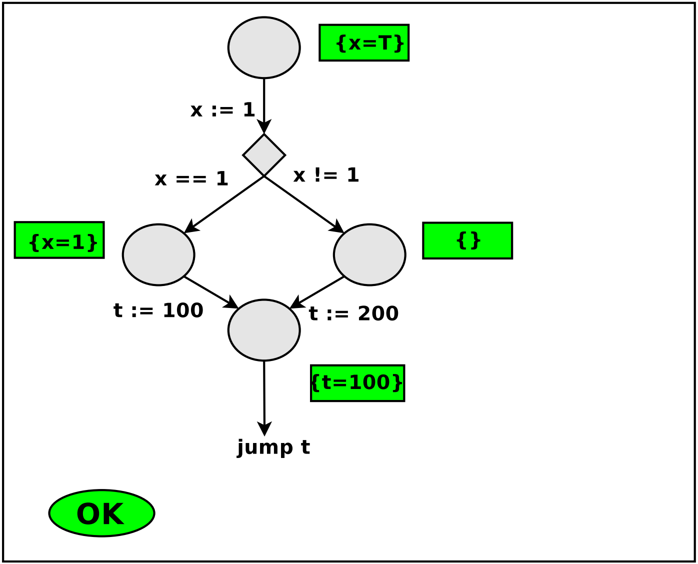
RC : why it does not work

let us suppose $Kmax = 1$

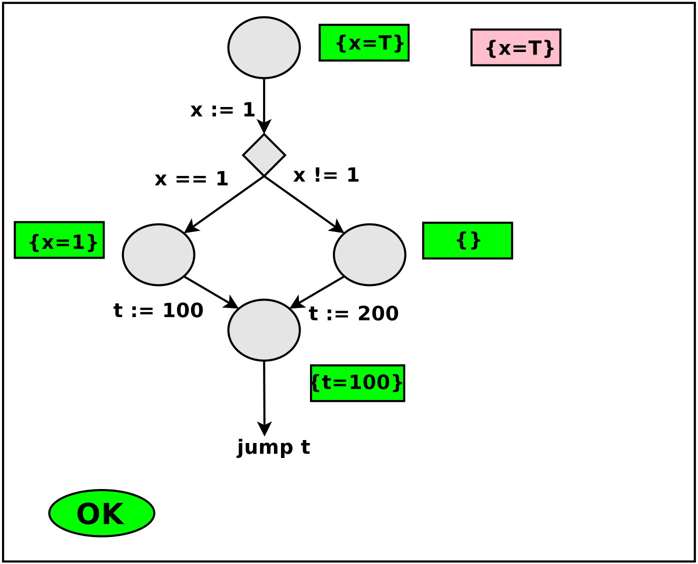


RC : why it does not work

let us suppose $Kmax = 1$



let us suppose $Kmax = 1$



RC : why it does not work

let us suppose $Kmax = 1$