DE LA RECHERCHE À L'INDUSTRIE

Recovering high-level conditions from binary programs

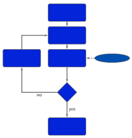
09/11/2016
FM 2016

Adel Djoudi
Sébastien Bardin
Éric Goubault

www.cea.fr

Outline

Binary code analysis : Why ?



source code

source analysis

- ×Proprietary software
- ×Analysis of malware
- ×Compiler independent
- ×Multi-languages progs

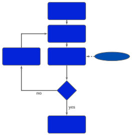
001011010101
100010101101
010110101010
001011010101
100010101101
100010101101
010110101010

binary code

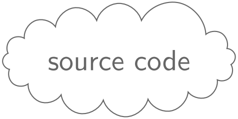
binary analysis

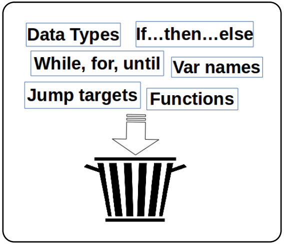
- ✓Proprietary software
- ✓Analysis of malware
- ✓Compiler independent
- ✓Multi-languages progs

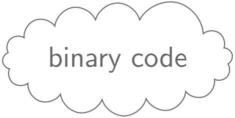
Binary code analysis : Why ?



001011010101
100010101101
010110101010
001011010101
100010101101
100010101101
010110101010







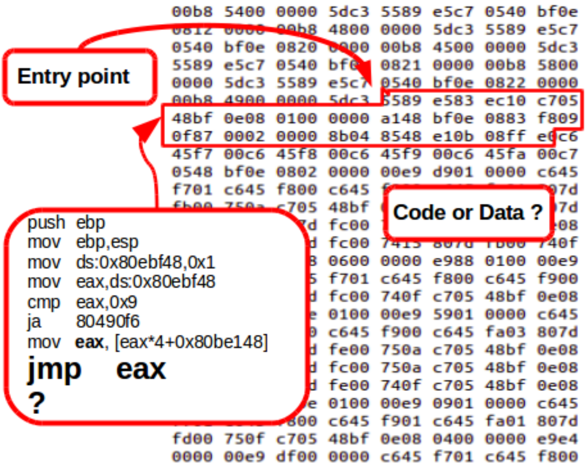
source analysis

- ×Proprietary software
- ×Analysis of malware
- ×Compiler independent
- ×Multi-languages progs

binary analysis

- ✓Proprietary software
- ✓Analysis of malware
- ✓Compiler independent
- ✓Multi-languages progs

Challenges of binary code analysis (1)



Challenges of binary code analysis (2)

- Low-level semantics of data
  - Machine arithmetic, bit-level operations
  - Systematic usage of untyped memory [big array]

  Difficult for current formal techniques

- Low-level semantics of control
  - No clear distinction data/instructions
  - Dynamic jumps (`jump eax`)

  No easy syntactic recovery of CFG

- Diversity of architectures and instruction sets
  - Too many instructions (ex. X86, $\geq$ 900 instructions)
  - Modeling issues : side effect, addressing mode, ...

  No platform independent concise formalism

Challenges of binary code analysis (2)

Nice progress since 2004

- Low-level semantics of data
  - Machine arithmetic, bit-level operations
  - Systematic usage of untyped memory [big array]

  Difficult for current formal techniques

- Low-level semantics of control
  - No clear distinction data/instructions
  - Dynamic jumps (`jump eax`)

  No easy syntactic recovery of CFG

- Diversity of architectures and instruction sets
  - Too many instructions (ex. X86, ≥ 900 instructions)
  - Modeling issues : side effect, addressing mode, ...

  No platform independent concise formalism

Intermediate languages

REIL [Zynamics]
BIL [CMU]
DBA [CEA, LaBRI]
RREIL [TUM] ...

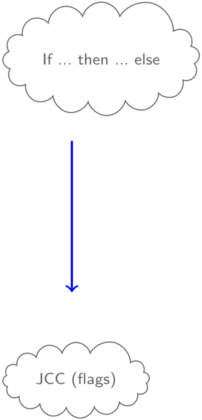
CFG recovery

CodeSurfer/x86 [GrammaTech]
Jakstab [TU München]
CFGBuilder [CEA]
...

Tests generation

SAGE [Microsoft]
OSMOSE [CEA]
Mayhem [ForAllSecure]
...

Challenge : High-level condition recovery



- High-level conditions translated into low-level flag predicates
- Conditional jumps depend on flags and not directly on registers
- Serious problem for formal analysis
  - abstract interpretation : precision
  - symb exec : solving cost

PowerPC translation example

```
if (ax > bx) X = -1;
else X = 1;
```

compilation

```
        cmpd    ax, bx
        bg      l1
        li      X, 1
        b       l2
l1:     li      X, -1
l2:
```

disassembly

```
        CR_L := (ax < bx)
        CR_G := (ax > bx)
        CR_E := (ax = bx)
        if (CR_G) goto l1
        X := 1
        goto l2
l1:     X := -1
l2:
```

Easy with relation propagation [folklore]

X86 translation example

```
if (ax > bx) X = -1;
else X = 1;
```

compilation

```
        cmp   ax, bx
        jg    l1
        mov   X, 1
        jmp   l2
l1:     mov   X, -1
l2:
```

disassembly

```
        OF := ((ax{31,31}≠bx{31,31}) &
                (ax{31,31}≠(ax-bx){31,31}));
        SF := (ax-bx) < 0;
        ZF := (ax-bx) = 0;
        if (¬ ZF ∧ (OF = SF)) goto l1
        X := 1
        goto l2
l1:     X := -1
l2:
```

The real difficulty

Problem with formal approaches

`4: cmp x 100;`	`ZF := (x=100)`	$x \mapsto \top$
`5: je a;`	`if (ZF) then goto a`	$x, ZF \mapsto \top, [0,1]$
`....`		
`a:...`		$x, ZF \mapsto \top, [1,1]$

Condition evaluation does not allow operand refinement

Problem with symbolic execution also

Goal & achievements

Goal : Source-level like reasoning

- through high-level condition recovery
- what we want : sound, generic, precise in practice

Achievements

- template-based condition recovery
- implementation in BINSEC/VA
- other binary-level tricks [see the paper]

Applications

- formal methods : abstract interpretation, symbolic execution
- help the reverse engineering

Outline

Logic-based recovery (1)

`4: cmp x 100;`	`ZF := (x=100)`	$x \mapsto \top$
`5: je a;`	`if (x=100) then goto a`	$x \mapsto \top$
`...`		
`a:`		$x \mapsto [100, 100]$

- Idea : flag predicate $\Longrightarrow$ operand predicate
- Problem : flag predicate $\overset{?}{\Longleftrightarrow}$ operand predicate
- Solution :
  - use abstract interpretation to propagate flag expressions
  - substitute flags by corresponding expressions at conditions
  - simplify conditions to recover operand predicates

[folklore] solution

Generic & sound in simple cases **only**

Logic-based recovery (2)

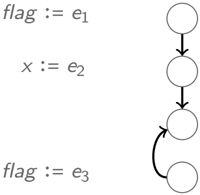
`4: cmp x 100;`	`ZF := (x=100)`	$x \mapsto \top$	$ZF \mapsto \top$
`5: je a;`	`if (x=100) then goto a`	$x \mapsto \top$	$ZF \mapsto (x = 100)$
`...`			
`a:`		$x \mapsto [100, 100]$	$ZF \mapsto (x = 100)$

- Abstract domain $\boxed{D^{\#} \triangleq \mathit{Flag} \rightarrow \mathit{Expr}}$

Logic-based recovery (2)

`4: cmp x 100;`	`ZF := (x=100)`	$x \mapsto \top$	$ZF \mapsto \top$
`5: je a;`	`if (x=100) then goto a`	$x \mapsto \top$	$ZF \mapsto (x = 100)$
`...`			
`a:`		$x \mapsto [100, 100]$	$ZF \mapsto (x = 100)$

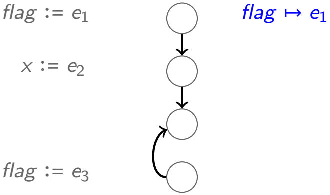
- Abstract domain $\boxed{D^{\#} \triangleq Flag \to Expr}$
- Propagation



Logic-based recovery (2)

`4: cmp x 100;`	`ZF := (x=100)`	$x \mapsto \top$	$ZF \mapsto \top$
`5: je a;`	`if (x=100) then goto a`	$x \mapsto \top$	$ZF \mapsto (x = 100)$
...			
`a:`		$x \mapsto [100, 100]$	$ZF \mapsto (x = 100)$

- Abstract domain $\boxed{D^{\#} \triangleq Flag \rightarrow Expr}$
- Propagation



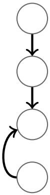
Logic-based recovery (2)

`4: cmp x 100;` `5: je a;` `...` `a:`	`ZF := (x=100)` `if (x=100) then goto a`	$x \mapsto \top$ $x \mapsto \top$ $x \mapsto [100, 100]$	$ZF \mapsto \top$ $ZF \mapsto (x = 100)$ $ZF \mapsto (x = 100)$

- Abstract domain $\boxed{D^{\#} \triangleq Flag \rightarrow Expr}$
- Propagation

$flag := e_1$ — $flag \mapsto e_1$

$x := e_2$ — $flag \mapsto \top$ if $(x \in^* e_1)$

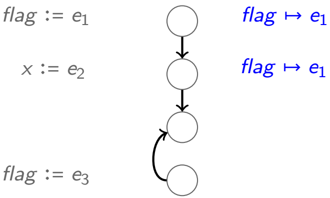


$flag := e_3$

Logic-based recovery (2)

`4: cmp x 100;`	`ZF := (x=100)`	$x \mapsto \top$	$ZF \mapsto \top$
`5: je a;`	`if (x=100) then goto a`	$x \mapsto \top$	$ZF \mapsto (x = 100)$
…			
`a:`		$x \mapsto [100, 100]$	$ZF \mapsto (x = 100)$

- Abstract domain $\boxed{D^\# \triangleq Flag \to Expr}$
- Propagation



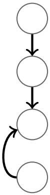
Logic-based recovery (2)

`4: cmp x 100;`	`ZF := (x=100)`	$x \mapsto \top$	$ZF \mapsto \top$
`5: je a;`	`if (x=100) then goto a`	$x \mapsto \top$	$ZF \mapsto (x = 100)$
`...`			
`a:`		$x \mapsto [100, 100]$	$ZF \mapsto (x = 100)$

- Abstract domain $\boxed{D^{\#} \triangleq Flag \rightarrow Expr}$
- Propagation

$flag := e_1$ $\quad$ $flag \mapsto e_1$

$x := e_2$ $\quad$ $flag \mapsto e_1$



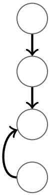
$flag \mapsto \top$ if $(e_1 \neq e_3)$

$flag := e_3$ $\quad$ $flag \mapsto e_3$

Logic-based recovery (2)

`4: cmp x 100;`	`ZF := (x=100)`	$x \mapsto \top$	$ZF \mapsto \top$
`5: je a;`	`if (x=100) then goto a`	$x \mapsto \top$	$ZF \mapsto (x = 100)$
…			
`a:`		$x \mapsto [100, 100]$	$ZF \mapsto (x = 100)$

- Abstract domain $\boxed{D^{\#} \triangleq Flag \to Expr}$
- Propagation



$flag := e_1$ — $flag \mapsto e_1$

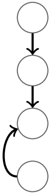
$x := e_2$ — $flag \mapsto e_1$

$flag \mapsto e_1$

$flag := e_3$ — $flag \mapsto e_3$

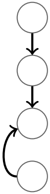
Logic-based recovery (3)

- Usage (at conditional instruction)

$flag := e_1$		$flag \mapsto e_1$
$x := e_2$		$flag \mapsto e_1$
$if\,(flag)\ldots$		$flag \mapsto \top$
$flag := e_3$		$flag \mapsto e_3$

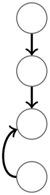
Logic-based recovery (3)

Usage (at conditional instruction)

Instruction		State	
flag $:= e_1$		*flag* $\mapsto e_1$	
$x := e_2$		*flag* $\mapsto e_1$	
if (*flag*)...		*flag* $\mapsto \top$	do nothing!
flag $:= e_3$		*flag* $\mapsto e_3$	

Logic-based recovery (3)

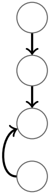
- Usage (at conditional instruction)



$flag := e_1$	$flag \mapsto e_1$
$x := e_2$	$flag \mapsto e_1$
$if\,(rewrite(e_1))\ldots$	$flag \mapsto e_1$
$flag := e_3$	$flag \mapsto e_3$

Logic-based recovery (3)

- Usage (at conditional instruction)

$flag := e_1$		$flag \mapsto e_1$
$x := e_2$		$flag \mapsto e_1$
$if\,(rewrite(e_1))\ldots$		$flag \mapsto e_1$
$flag := e_3$		$flag \mapsto e_3$

Problem solved ! Not yet...

Example

4: cmp x y;	OF := ((x{31,31}≠y{31,31}) & (x{31,31}≠(x-y){31,31}));	$x, y \mapsto [0, 11], [10, 20]$
	SF := (x-y) < 0;	
	ZF := (x-y) = 0;	
5: jg a;	if (¬ ZF ∧ (OF = SF)) then goto a	$x, y \mapsto [0, 11], [10, 20]$
...		
a:...		$x, y \mapsto [0, 11], [10, 20]$

$$\neg(\texttt{x-y = 0}) \wedge \Big((\texttt{x}_{\{31,31\}}\texttt{=}\texttt{y}_{\{31,31\}}) \;\&\; (\texttt{x}_{\{31,31\}}\texttt{=(x-y)}_{\{31,31\}})\Big) = (\texttt{x-y<0}) \overset{?}{\Longleftrightarrow}$$

$$x > y$$

Example

`4: cmp x y;`	`OF := ((x{31,31}≠y{31,31}) & (x{31,31}≠(x-y){31,31}));` `SF := (x-y) < 0;` `ZF := (x-y) = 0;`	$x, y \mapsto [0,11],[10,20]$
`5: jg a;`	`if (¬ ZF ∧ (OF = SF)) then goto a`	$x, y \mapsto [0,11],[10,20]$
`...`		
`a:...`		$x, y \mapsto [0,11],[10,20]$

Relation propagation does not help

$$\neg(x-y = 0) \wedge \big((x_{\{31,31\}}=y_{\{31,31\}}) \;\&\; (x_{\{31,31\}}=(x-y)_{\{31,31\}})\big) = (x-y<0) \overset{?}{\Longleftrightarrow} x > y$$

Natural high-level predicate

`4: cmp x y;`	`OF := ((x{31,31}≠y{31,31}) & (x{31,31}≠(x-y){31,31}));`	$x, y \mapsto [0, 11], [10, 20]$
	`SF := (x-y) < 0;`	
	`ZF := (x-y) = 0;`	
`5: jg a;`	`if (x > y) then goto a`	$x, y \mapsto [0, 11], [10, 20]$
...		
`a:...`		$x, y \mapsto [11, 11], [10, 10]$

- Idea : flag predicate $\Longrightarrow$ ***natural*** operands predicate
- Problem :
  - only simple high-level predicates can be handled by non-relational abstract domains
  - complex flag predicates can hide simple high-level predicates

Natural high-level predicate

4: cmp x y;	OF := ((x{31,31}≠y{31,31}) & (x{31,31}≠(x-y){31,31}));	$x, y \mapsto [0, 11], [10, 20]$
	SF := (x-y) < 0;	
	ZF := (x-y) = 0;	
5: jg a;	if (x > y) then goto a	$x, y \mapsto [0, 11], [10, 20]$
...		
a:...		$x, y \mapsto [11, 11], [10, 10]$

- Idea : flag predicate ⟹ *natural* operands predicate
- Problem :
  - only sim... e handled by non-rela...
  - complex ... e high-level predicates

Existing solutions

Flag patterns

Virtual flags

Pattern-based recovery (1)

- Depend on operations cmp / sub / test and their use
- Possible to ensure soundness
- Rely on decoding information

Sound, precise but architecture specific

Compilers may use their own patterns

Pattern-based recovery (2)

High level predicates for conditional jump instructions (x86) [1]

	flag predicate	cmp x y predicate	sub x y predicate [2]	test x y predicate
ja, jnbe	$\neg CF \wedge \neg ZF$	$x >_u y$	$x' \neq 0$	$x\&y \neq 0$
jae, jnb, jnc	$\neg CF$	$x \geq_u y$	true	true
jb, jnae, jc	CF	$x <_u y$	$x' \neq 0$	false
jbe, jna	$CF \vee ZF$	$x \leq_u y$	true	$x\&y = 0$
je, jz	ZF	$x = y$	$x' = 0$	$x\&y = 0$
jne, jnz	$\neg ZF$	$x \neq y$	$x' \neq 0$	$x\&y \neq 0$
jg, jnle	$\neg ZF \wedge (OF = SF)$	$x > y$	$x' > 0$	$(x\&y \neq 0) \wedge$ $(x \geq 0 \vee y \geq 0)$
jge, jnl	$(OF = SF)$	$x \geq y$	true	$(x \geq 0 \vee y \geq 0)$
jl, jnge	$(OF \neq SF)$	$x < y$	$x' < 0$	$(x < 0 \wedge y < 0)$
jle, jng	$ZF \vee (OF \neq SF)$	$x \leq y$	*true*	$(x\&y = 0) \vee$ $(x < 0 \wedge y < 0)$

1. G. Balakrishnan, T. Reps : WYSINWYX : What You See Is Not What You eXecute
2. x' = x - y
3. CF = OF = False

Non-standard examples

example	retrieved condition	patterns
`or eax, 0` `je ...`	`if (eax = 0) then goto ...`	✗
`cmp eax, 0` `jns ...`	`if (eax ⩾ 0) then goto ...`	✗
`sar ebp, 1` `je ...`	`if (ebp = 0) then goto ...`	✗
`dec ecx` `jg ...`	`if (ecx ⩾ 0) then goto ...`	✗

How many necessary patterns ?

Summary & proposal

Approach	archi. independent	Sound	Complete enough
Patterns	✗	✓/✗	✓/✗
Logic-based	✓	✓	✗

Summary & proposal

Approach	archi. independent	Sound	Complete enough
Patterns	✗	✓/✗	✓/✗
Logic-based	✓	✓	✗
Template-based	✓	✓	✓

Template-based approach

- direct extension of logic-based approach
- may combine with patterns (better recovery, speed)

Outline

Example

`4: cmp x y;`	`OF := ((x{31,31}≠y{31,31}) & (x{31,31}≠(x-y){31,31}));` `SF := (x-y) < 0;` `ZF := (x-y) = 0;`	$x, y \mapsto [0, 11], [10, 20]$
`5: jg a;`	`if (¬ ZF ∧ (OF = SF)) then goto a`	$x, y \mapsto [0, 11], [10, 20]$
`...`		
`a:...`		$x, y \mapsto [0, 11], [10, 20]$

$$\neg(\mathtt{x-y} = 0) \land \big((\mathtt{x}_{\{31,31\}} = \mathtt{y}_{\{31,31\}})\ \&\ (\mathtt{x}_{\{31,31\}} = (\mathtt{x-y})_{\{31,31\}})\big) = (\mathtt{x-y} < 0) \overset{?}{\Longleftrightarrow}$$

$$x > y$$

Template-based recovery (1)

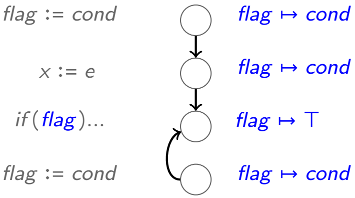
- Abstract domain $\boxed{D^{\#} \triangleq Flag \rightarrow Expr}$
- Propagation : same as in logic-based approach

Insights

- Complex predicates often hide simple predicates
- Only a few templates : $>_{u,s}, <_{u,s}, \geq_{u,s}, \leq_{u,s}, =, \neq$
- Try to find the appropriate one through equivalence checking
- Optimization :
  - Do it only once per loc (cache)
  - Cheap pruning through filtering

Template-based recovery (2)

- Usage at condition : `cond`
- Retrieve potential operands : `x` and `y` from `cond`



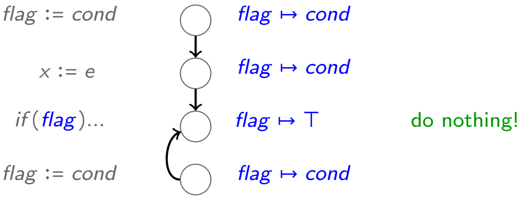
- Assert the equivalence of `cond` with :

$\texttt{cond} \iff \texttt{x} >_u \texttt{y}$	$\texttt{cond} \iff \texttt{x} <_u \texttt{y}$	$\texttt{cond} \iff \texttt{x} \geq_u \texttt{y}$	$\texttt{cond} \iff \texttt{x} \leq_u \texttt{y}$
$\texttt{cond} \iff \texttt{x} > \texttt{y}$	$\texttt{cond} \iff \texttt{x} < \texttt{y}$	$\texttt{cond} \iff \texttt{x} \geq \texttt{y}$	$\texttt{cond} \iff \texttt{x} \leq \texttt{y}$
$\texttt{cond} \iff \texttt{x} = \texttt{y}$	$\texttt{cond} \iff \texttt{x} \neq \texttt{y}$	s.t. $\texttt{x}, \texttt{y} \in_{syntax} \texttt{cond}$	

- If no assertion is satisfied then do nothing

Template-based recovery (2)

- Usage at condition : `cond`
- Retrieve potential operands : `x` and `y` from `cond`



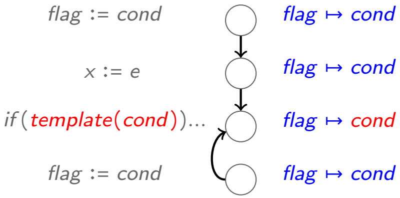
- Assert the equivalence of `cond` with :

$\texttt{cond} \iff \texttt{x} >_u \texttt{y}$	$\texttt{cond} \iff \texttt{x} <_u \texttt{y}$	$\texttt{cond} \iff \texttt{x} \geq_u \texttt{y}$	$\texttt{cond} \iff \texttt{x} \leq_u \texttt{y}$
$\texttt{cond} \iff \texttt{x} > \texttt{y}$	$\texttt{cond} \iff \texttt{x} < \texttt{y}$	$\texttt{cond} \iff \texttt{x} \geq \texttt{y}$	$\texttt{cond} \iff \texttt{x} \leq \texttt{y}$
$\texttt{cond} \iff \texttt{x} = \texttt{y}$	$\texttt{cond} \iff \texttt{x} \neq \texttt{y}$	s.t. $\texttt{x}, \texttt{y} \in_{syntax} \texttt{cond}$	

- If no assertion is satisfied then do nothing

Template-based recovery (2)

- Usage at condition : `cond`
- Retrieve potential operands : `x` and `y` from `cond`



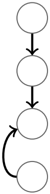
- Assert the equivalence of `cond` with :

$\texttt{cond} \iff \texttt{x} >_u \texttt{y}$	$\texttt{cond} \iff \texttt{x} <_u \texttt{y}$	$\texttt{cond} \iff \texttt{x} \geq_u \texttt{y}$	$\texttt{cond} \iff \texttt{x} \leq_u \texttt{y}$
$\texttt{cond} \iff \texttt{x} > \texttt{y}$	$\texttt{cond} \iff \texttt{x} < \texttt{y}$	$\texttt{cond} \iff \texttt{x} \geq \texttt{y}$	$\texttt{cond} \iff \texttt{x} \leq \texttt{y}$
$\texttt{cond} \iff \texttt{x} = \texttt{y}$	$\texttt{cond} \iff \texttt{x} \neq \texttt{y}$	s.t. $\texttt{x}, \texttt{y} \in_{syntax} \texttt{cond}$	

- If no assertion is satisfied then do nothing

Optimization 1 : Normalization cache

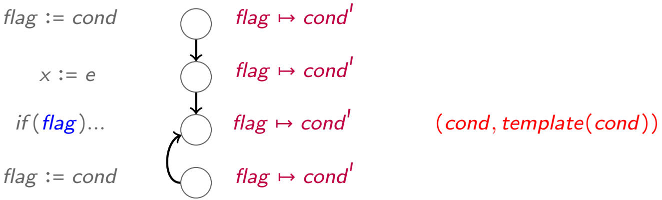
- Low-level condition saved in the cache at address a together with the retrieved high-level condition
- If the same condition at the same address a is met later in the analysis, the saved high-level condition can be safely reused



$flag := cond$		$flag \mapsto cond$	
$x := e$		$flag \mapsto cond$	
$if\,(template(cond))\ldots$		$flag \mapsto cond$	$(cond, template(cond))$
$flag := cond$		$flag \mapsto cond$	

Optimization 1 : Normalization cache

- Low-level condition saved in the cache at address a together with the retrieved high-level condition
- If the same condition at the same address a is met later in the analysis, the saved high-level condition can be safely reused



Optimization 1 : Normalization cache

- Low-level condition saved in the cache at address a together with the retrieved high-level condition
- If the same condition at the same address a is met later in the analysis, the saved high-level condition can be safely reused

$flag := cond$ — $flag \mapsto cond'$

$x := e$ — $flag \mapsto cond'$

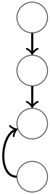
$if(template(cond))\ldots$ — $flag \mapsto cond'$ — $(cond, template(cond))$

$flag := cond$ — $flag \mapsto cond'$

If $cond = cond'$

Optimization 1 : Normalization cache

- Low-level condition saved in the cache at address a together with the retrieved high-level condition
- If the same condition at the same address a is met later in the analysis, the saved high-level condition can be safely reused



$flag := cond$	$flag \mapsto cond'$	
$x := e$	$flag \mapsto cond'$	
$if\,(template(cond'))\ldots$	$flag \mapsto cond'$	$(cond', template(cond'))$
$flag := cond$	$flag \mapsto cond'$	

Optimization 2 : Templates filtering

- Substitute operands in `cond` with special values
- Evaluate `cond` to eliminate obvious impossible predicates

$$\neg(\texttt{x-y = 0}) \land \left((\texttt{x}_{\{31,31\}}\texttt{=y}_{\{31,31\}}) \ \texttt{\&} \ (\texttt{x}_{\{31,31\}}\texttt{=(x-y)}_{\{31,31\}})\right) \texttt{ = (x-y<0)} \overset{?}{\Longleftrightarrow} x \diamond y$$

$$\text{Eval 1} : \texttt{cond}[0/\texttt{x},\ 0/\texttt{y}] = 0 \Rightarrow \diamond \notin \{=, \leq, \leq_u, \geq, \geq_u\}$$

$$\text{Eval 2} : \texttt{cond}[0/\texttt{x},\ 1/\texttt{y}] = 0 \Rightarrow \diamond \notin \{\neq, <, <_u\}$$

$$\text{Eval 3} : \texttt{cond}[0/\texttt{x}, -1/\texttt{y}] = 1 \Rightarrow \diamond \notin \{>_u\}$$

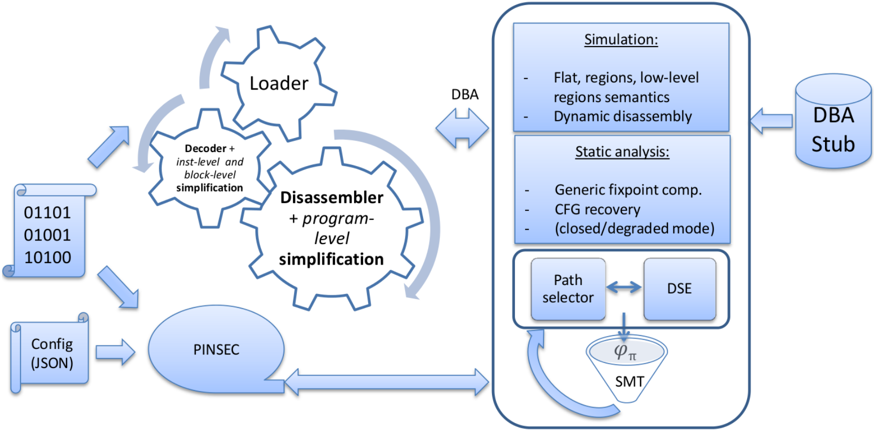
$$\boxed{\diamond \in \{>\}}$$

Tricky examples

example	retrieved condition	patterns	templates
`or eax, 0` `je ...`	`if (eax = 0) then goto ...`	✗	✓
`cmp eax, 0` `jns ...`	`if (eax ⩾ 0) then goto ...`	✗	✓
`sar ebp, 1` `je ...`	`if (ebp = 0) then goto ...`	✗	✓
`dec ecx` `jg ...`	`if (ecx ⩾ 0) then goto ...`	✗	✓
`add ecx, 0xfffefefe` `jae ...`	`if (ecx ⩾ 0xfffefefe) goto ...`	✗	✓
`test al, 0x8` `jne ...`	`if ((al & 0x8) ≠ 0) goto ...`	✓	✗
`stos [edi],eax`	`if(DF) goto ...`	✗	✗
`and edx, eax` `jp ...`	`if (PF) goto ...`	✗	✗
`shr ecx, 1` `jae ...`	`if (¬CF) goto ...`	✗	✗
`cmp [esp+0x64],eax` `mov eax, [esp+0x24]` `jg ...`	`if (¬ZF∧(OF=SF)) goto ...`	✗	✗

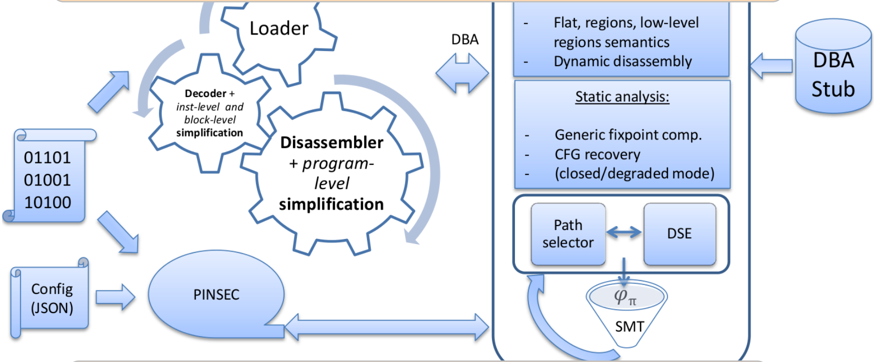
Outline

BINSEC Platform Overview



BINSEC Platform Overview

- Front-end [loader, decoder, disassembly, simplifications]
- Simulator
- Generic static analyzer
- Dynamic Symbolic Execution [ISSTA16, SANER16, BlackHatEU16]

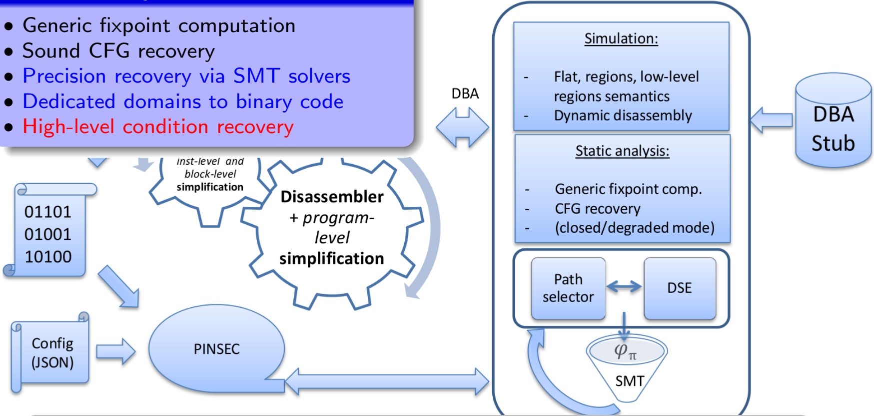


- Developed in OCaml [≈50 000 loc] [TACAS 2015]
- Within the BINSEC project [CEA, IRISA, LORIA, Univ-Genoble]

BINSEC Platform Overview

Static analysis

- Generic fixpoint computation
- Sound CFG recovery
- Precision recovery via SMT solvers
- Dedicated domains to binary code
- High-level condition recovery



Open source and available at : `http://binsec.gforge.inria.fr/tools.html`

Experiments (1)

progs	#loc†	#cond‡	#success⋆	#fail	time (s)	time$_{all}$ (s)
firefox	21488	150 **(137)**	**134** | 89% **(98%)**	16	1.40	55.91
cat	6490	132 **(125)**	**116** | 88% **(92%)**	16	1.08	259.24
chmod	8954	183 **(172)**	**159** | 87% **(92%)**	24	1.44	313.17
cp	67199	174 **(162)**	**152** | 87% **(94%)**	22	4.79	346.84
cut	7358	148 **(138)**	**132** | 89% **(96%)**	16	1.16	211.73
dir	9732	137 **(126)**	**118** | 86% **(94%)**	19	1.26	201.67
echo	8016	190 **(182)**	**168** | 88% **(92%)**	22	1.43	274.60
kill	6911	142 **(133)**	**125** | 88% **(94%)**	17	1.17	209.79
ln	88837	203 **(185)**	**177** | 87% **(96%)**	26	4.88	531.58
mkdir	6347	125 **(117)**	**109** | 87% **(93%)**	16	1.01	235.80
Verisec	11552	394 **(370)**	**370** | 87% **(100%)**	24	3.31	34.48
total	242884	1978 **(1847)**	**1760** | 89% **(95%)**	218	22.93	2674.81

† : number of analysed instructions only

‡ : total number of conditions (resp. high-level conditions). DF, PF and $x\&y = 0$ are not considered high-level.

⋆ : total number of successfully recovered conditions, ratio w.r.t. total number of conditions (resp. high-level conditions)

Experiments (1)

progs	#loc†	#cond‡	#success⋆	#fail	time (s)	$time_{all}$ (s)
firefox	21488	150 **(137)**	**134** | 89% **(98%)**	16	1.40	55.91
cat	6490	132 **(125)**	**116** | 88% **(92%)**	16	1.08	259.24
chmod	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	313.17
cp	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	46.84
cut	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	11.73
dir	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	01.67
echo	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	74.60
kill	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	09.79
ln	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	31.58
mkdir	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	35.80
Verisec	11552	394 **(370)**	**370** | 87% **(100%)**	24	3.31	34.48
total	242884	1978 **(1847)**	**1760** | 89% **(95%)**	218	22.93	2674.81

Conclusion

- Low overhead, 1% in average (column time *vs* $time_{all}$)
- Large part of high-level conditions recovered
- Templates are generic, sound and fully automatic

† : number of analysed instructions only

‡ : total number of conditions (resp. high-level conditions). DF, PF and $x\&y = 0$ are not considered high-level.

⋆ : total number of successfully recovered conditions, ratio w.r.t. total number of conditions (resp. high-level conditions)

Experiments (2)

method	#loc	#cond	#success	#fail	time	time$_{all}$
templates	242884	1978	**1760 (89%)**	218	**22.93**	2674.81
logic-based	247894	2260	**694 (31%)**	1566	0.003	2561.64
patterns	229255	1987	**1357 (68%)**	630	0.014	2373.33
templates+patterns	242884	1978	**1838 (92%)**	140	**9.17**	2659.95
templates **w/o cache**	242884	1978	**1760 (89%)**	218	29.76	2697.67
templates **w/o filtering**	242884	1978	**1760 (89%)**	218	51.13	2726.45
templates **w/o cache, filtering**	242884	1978	**1760 (89%)**	218	**66.52**	2752.73

Conclusion

- Templates achieve significantly better results
- Templates have affordable extra cost
- Optimizations allow to win a factor 3x on average
- Templates can be fruitfully combined with patterns

Fun application!

`cmp eax ebx`	`CF := (eax` $<_u$ `ebx)`
`cmc`	`CF := ¬CF`
`jae ...`	`if (¬CF) goto ...`

- Standard pattern :

  `cmp eax ebx` $\longrightarrow$ `if(eax` $\geq_u$ `ebx) goto ...`

  `jae ...`

- The true semantic : `if (eax` $<_u$ `ebx) goto ...`

patterns	templates
×	✓

Outline

Conclusion

- Template-based recovery : a sound and generic technique
- Performs significantly better than state-of-the-art approaches
- Helps to adapt analyses from source-level to binary-level
- Can be useful for reverse engineering
- Implemented in BinSec : [http ://binsec.gforge.inria.fr/tools.html]

$\boxed{\text{Questions ?}}$