Binary-level Software Analysis

Sébastien Bardin
joint work with Franck Védrine and Philippe Herrmann

CEA-LIST, Laboratoire de Sûreté Logicielle
Binary-level software analysis

Model

Source code

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

Assembly

_exec:
    load A 100
    add B A
    cmp B 0
    jle label

label:
    move @100 B

Executable

ABFFF780BD70696CA101001BDE45
145634789234ABFFE678ABDCF456
5A2B4C6D009F5F5D1E0835715697
145FEDBCADACBDAD459700346901
3456KAHA305G67H345BFFADECAD3
00113456735FFD451E13AB080DAD
344252FFAADBDA457345FD780001
FFF22546ADDAE989776600000000
Benefits of binary code analysis

Advantages over source-level analysis
- executable always available
- no “compiler gap” (security, safety)

New fields of application
- COTS (including libraries)
- mobile code (including malware)
- third-party certification

BUT : (much ?) more challenging than source code analysis
Challenges of binary code analysis

D1 : Low-level semantics of data
- machine arithmetic, bit-level operations
- systematic usage of untyped memory \([\approx \text{big array}]\)
  - difficult for current formal techniques

D2 : Low-level semantics of control
- no clear distinction data / instructions
- dynamic jumps (\texttt{goto A})
  - no easy syntactic recovery of CFG
  - while it is an implicit prerequisite for most formal techniques

D3 : Diversity of architectures and instruction sets
- support for many instructions, modelling issues
Challenges of binary code analysis

**D1 : Low-level semantics of data**
- machine arithmetic, bit-level operations
- systematic usage of untyped memory \( \approx \) big array
- difficult for current formal techniques

**D2 : Low-level semantics of control**
- no clear distinction data / instructions
- dynamic jumps (\texttt{goto A})
- no easy syntactic recovery of CFG
- while it is an implicit prerequisite for most formal techniques

**D3 : Diversity of architectures and instruction sets**
- support for many instructions, modelling issues
Challenges of binary code analysis

D1 : Low-level semantics of data
- machine arithmetic, bit-level operations
- systematic usage of untyped memory \( \approx \) big array
  - difficult for current formal techniques

D2 : Low-level semantics of control
- no clear distinction data / instructions
- dynamic jumps (\texttt{goto A})
  - no easy syntactic recovery of CFG
  - while it is an implicit prerequisite for most formal techniques

D3 : Diversity of architectures and instruction sets
- support for many instructions, modelling issues
Challenges of binary code analysis

D1: Low-level semantics of data
- machine arithmetic, bit-level operations
- systematic usage of untyped memory \([\approx\) big array]\)
  - difficult for current formal techniques

D2: Low-level semantics of control
- no clear distinction data / instructions
- dynamic jumps (\texttt{goto A})
  - no easy syntactic recovery of CFG
  - while it is an implicit prerequisite for most formal techniques

D3: Diversity of architectures and instruction sets
- support for many instructions, modelling issues
Focus: safe CFG recovery problem

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

Output: (surapproximation) of the program CFG
The successor instructions can often be identified syntactically

- \( \langle \text{addr: move a b} \rangle \rightarrow \text{successor at addr+size+1} \)
- \( \langle \text{addr: goto 100} \rangle \rightarrow \text{successor at 100} \)
- \( \langle \text{addr: ble 100} \rangle \rightarrow \text{successors at 100 and addr+size+1} \)
The successor instructions can often be identified syntactically

- `<addr: move a b>` → successor at `addr+size+1`
- `<addr: goto 100>` → successor at `100`
- `<addr: ble 100>` → successors at `100` and `addr+size+1`

But not always: successor of `<addr: goto a>`?

Challenge = compute a superset of all targets of each jump
Need to combine syntactic CFG recovery with value analysis (VA)
Focus: safe CFG recovery problem

Need to combine syntactic CFG recovery with value analysis (VA)

“Chicken and egg” problem

VA imprecise on goto A
⇒ too many instructions / branches added to CFG
⇒ more propagation / imprecision in VA
⇒ VA more imprecise on goto A ⇒ and so on
Binary-level program analysis at CEA LSL
Binary-level program analysis at CEA LSL

**DBA**
- small set of instructions
- no side effects
- bit-precise modelling
- easy modelling

- Done:
  - PPC
  - (part of) x86
  - M6800, C509
OSMOSE

- test data generation

- input:
  . executable
  . entry, env.

- criteria:
  . paths / branches / instr.

- output:
  . test suite
  . partial CFG, coverage
Binary-level program analysis at CEA LSL

**CFGBuilder**
- safe CFG recovery
- input:
  - executable
  - entry, env
- output:
  - jump targets
  - cfg, cg, etc.

**OSMOSE**

**safe control-flow graph**

**test input partial cfg coverage**
Key Technologies

OSMOSE: Dynamic Symbolic Execution [ICST-08, STVR-11]
- exploration of all (bounded) paths of the program
- bit-precise constraint solving [TACAS-10]
- symbolic reasoning to discover new dynamic targets [STVR-11]
- path pruning optimisations [ICST-09]

DBA formal model [CAV-11]

CFGBuilder: Refinement-based analysis [VMCAI-11]
- static analysis through abstract interpretation
- abstract domain = k-sets (finite sets of at most k constants)
- the size is controlled by an iterative refinement mechanism
Outline

Motivations and challenges

Modelling

Test data generation

Safe CFG recovery

Conclusion & Future work
Main design ideas [CAV 11, with LABRI]

- small set of instructions
- no side-effect
- concise and natural modelling of common ISAs
- low-level enough to allow bit-precise modelling
- standalone model: do not need any info on architecture
- try to be “analysis”-agnostic

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)

Basis

- bitvector variables and arrays of bytes
- all bv sizes statically known
- standard operations from bitvector arithmetic
- some instructions are labelled by addresses
Dynamic Bitvector Automata (2)

Basis

- bitvector variables and arrays of bytes
- all bv sizes statically known
- standard operations from bitvector arithmetic
- some instructions are labelled by addresses

Instructions

- \( \text{lhs} := \text{rhs}, \text{goto addr} \)
- goto addr
- ite(\text{cond})? goto addr : goto addr'
- goto expr
Dynamic Bitvector Automata (2)

Basis
- bitvector variables and arrays of bytes
- all bv sizes statically known
- standard operations from bitvector arithmetic
- some instructions are labelled by addresses

Conditions
- any expr evaluating to a bv of size 1
- including: \( \{ <u,s, \leq u,s, =, \neq, \geq u,s, > u,s \} \) expr
Dynamic Bitvector Automata (2)

Basis

- bitvector variables and arrays of bytes
- all bv sizes statically known
- standard operations from bitvector arithmetic
- some instructions are labelled by addresses

Expressions

- \(0xFF10 \times 16\), \(X\times size\)
- \(@ (expr, \rightarrow k), @ (expr, \leftarrow k)\)
- \(expr\{i .. j\}, ext_{u,s}(expr,n)\)
- \(expr\{<_{u,s}, \leq_{u,s}, =, \neq, \geq_{u,s}, >_{u,s}\}\) \(expr\)
- \(expr\{+,-,\times,/_{u,s}, \%_{u,s}\}\) \(expr\)
- \(expr\{\wedge, \vee, \oplus\}\) \(expr, !expr\)
- \(expr\{\ll_{u,s}, \gg_{u,s}, ::\}\) \(expr\)
no procedure calls, only jumps

- return becomes jump @SP
Memory layout: actually only one memory space (= one array)

- very precise
- but difficult to handle for symbolic reasoning

Would need extension

- several memory spaces: Stack, Heap, Constant, Malloc(id)
- values are pairs: \((\text{memspace}, \text{bv})\)
- operations with memspaces are restricted:
  - \((\text{Constant}, v) \text{ op } (\text{Constant}, v') = (\text{Constant}, v \text{ op } v')\)
  - \((\text{R}, v) - (\text{R}, v') = (\text{Constant}, v - v')\)
  - \((\text{R}, v) + (\text{Constant}, v') = (\text{R}, v + v')\)
OSMOSE: test data generation

Goal = automatic test data generation [assume an external oracle]

- under-approximation analysis
- computes witnesses of reachability
- cannot prove invariance

Input: executable, entry point, initial state [including volatile]

Output

- a set of pairs < input, intended execution path >
- (under-approximated) CFG
- (under-approximated) coverage measure

[First dse tool over exec. code, with SAGE - Godefroid-08]
Core concept: Dynamic Symbolic Execution

Symbolic Execution (assume a program $P$)

- choose a path $\pi$ of $P$
- compute a path predicate $\varphi_{\pi}$:
  $\forall v \models \varphi_{\pi} \Rightarrow P(v)$ follows $\pi$  
  \[ \text{[wpre, spost]} \]
- solve $\varphi_{\pi}$ for satisfiability
- $\text{SAT}(s)$? get a new pair $<s, \pi>$, update coverage
- loop until nothing more to cover

Old idea [King-70], but requires powerful solvers
Dynamic Symbolic Execution [Williams+ 04, Godefroid+ 05]

- Interleave concrete and symbolic executions
- Drive the search towards feasible paths for free
- Give hints for relevant under-approximations [concretization]
Dynamic Symbolic Execution [Williams+ 04, Godefroid+ 05]

- Interleave concrete and symbolic executions
- Drive the search towards feasible paths for free
- Give hints for relevant under-approximations [concretization]

Example of concretization

- Suppose instruction \( X := A \times B \), but only LIA available
- Need a proper underapproximation
- Suppose another exec. pass through same instr., with \( A = 5 \)
- Get the following underapprox: \( X = 5 \times B \)
Follows feasible paths for free

X < 0
X >= 0
X <> 12
X = 12
X >= -3
X < -3
Follows feasible paths for free

dynamic run with (arbitrary) $X=12$
Follows feasible paths for free

dynamic run with (arbitrary) $X=12$
backtrack + solve, $X = 5$
Follows feasible paths for free

dynamic run with (arbitrary) $X=12$
backtrack + solve, $X = 5$
dynamic run with $X=5$
Follows feasible paths for free

dynamic run with (arbitrary) $X=12$
backtrack + solve, $X=5$
dynamic run with $X=5$
backtrack + solve, unsat
Follows feasible paths for free

GOOD : never search inside the unfeasible path space
Concrete execution allows partial CFG recovery

Symbolic reasoning can be used too [STVR 11]

- assume a prefix $\pi$ finishing on jump $e$
- assume we already know targets $t_1, \ldots, t_k$
- solve $\varphi_\pi \land \bigwedge_{i=1}^{k} e \neq t_i$
- a solution will lead to a new jump target !!

All targets recovered by OSMOSE are truly reachable!
More efficient solving (even with blackbox solver)

- preprocessing [constant propag, equality propag, variable removal]
- partial reuse of solution of $\varphi_\pi$ for solving $\varphi_\pi \cdot \sigma$
  
  transform $\varphi_\pi \cdot \sigma = \varphi_\pi \land \varphi_\sigma$
  
  into $\varphi_\pi \cdot \sigma = (\varphi_1 \land \varphi_2) \land \varphi_\sigma$
  
  s.t. $\varphi_1$ does not share any var with $\varphi_2$ and $\varphi_\sigma$
- split $\varphi$ into independent subformulas $\varphi_1 \land \varphi_2$

Lower path explosion [ICST 09]

- prune paths which cannot reach uncovered items
- smarter searches than DFS (faster coverage)
A few other features

Test completion
- allow to chain several search heuristics

Export symbolic constraints
- useful for initialization phases and modular reasoning

Directives restricting the search space
- exit, no-branch
- repeat addr1 at most N (with reset on addr2)
Coverage of a medium-size aircraft program (Hispano-Suiza)

- 30,000 instructions, 250 functions, max calldepth = 10
- good coverage results for procedures with low height in the call graph (even with 2,000 instructions)
- robustness issue with higher-level procedures
- tested on 40 functions with call-depth \( \leq 4 \):
  - full cover for 31 functions
  - bad cover (\(< 50\%\)) for 5 functions
<table>
<thead>
<tr>
<th>name</th>
<th>I</th>
<th>Br</th>
<th>Osmose cover</th>
<th>Osmose time</th>
<th>Osmose #tests</th>
<th>random cover</th>
<th>random time</th>
</tr>
</thead>
<tbody>
<tr>
<td>aircraft0</td>
<td>237</td>
<td>36</td>
<td>100%</td>
<td>10</td>
<td>19</td>
<td>40%</td>
<td>20</td>
</tr>
<tr>
<td>aircraft1</td>
<td>290</td>
<td>140</td>
<td>98%</td>
<td>60</td>
<td>43</td>
<td>64%</td>
<td>100</td>
</tr>
<tr>
<td>aircraft2</td>
<td>201</td>
<td>72</td>
<td>100%</td>
<td>10</td>
<td>37</td>
<td>35%</td>
<td>20</td>
</tr>
<tr>
<td>aircraft3</td>
<td>977</td>
<td>190</td>
<td>50%</td>
<td>60</td>
<td>3</td>
<td>96%</td>
<td>60</td>
</tr>
<tr>
<td>aircraft4</td>
<td>2347</td>
<td>500</td>
<td>87%</td>
<td>600</td>
<td>15</td>
<td>68%</td>
<td>600</td>
</tr>
<tr>
<td>aircraft5</td>
<td>121</td>
<td>2</td>
<td>100%</td>
<td>1</td>
<td>2</td>
<td>100%</td>
<td>10</td>
</tr>
<tr>
<td>aircraft6</td>
<td>250</td>
<td>18</td>
<td>94%</td>
<td>100</td>
<td>9</td>
<td>83%</td>
<td>120</td>
</tr>
<tr>
<td>aircraft7</td>
<td>506</td>
<td>20</td>
<td>80%</td>
<td>20</td>
<td>4</td>
<td>75%</td>
<td>500</td>
</tr>
<tr>
<td>aircraft8</td>
<td>957</td>
<td>14</td>
<td>14%</td>
<td>10</td>
<td>3</td>
<td>50%</td>
<td>500</td>
</tr>
<tr>
<td>aircraft9</td>
<td>627</td>
<td>74</td>
<td>77%</td>
<td>600</td>
<td>12</td>
<td>63%</td>
<td>600</td>
</tr>
</tbody>
</table>

Time in sec.  Random tests : 1000 tests - unit testing
Experiments

Control-command program written in assembly language (EdF)

- Third-party software, sparse documentation
- Small program, but required very long sequences
- Complete functional tests
  + help understand the code (unfeasible branches, entries, etc.)
  + help to pinpoint problems in doc (ack. by vendor)

Comparison of binary coverage vs source coverage [S. Labbé, EdF]

- OSMOSE achieves better binary-coverage than test suites covering source-level MCDC
- on the examples, test suites generated by OSMOSE often achieve source-level MCDC
<table>
<thead>
<tr>
<th>name</th>
<th>I</th>
<th>Br</th>
<th>Osmose cover</th>
<th>Osmose time</th>
<th>Osmose #tests</th>
<th>random cover</th>
<th>random time</th>
</tr>
</thead>
<tbody>
<tr>
<td>msquare $3 \times 3$</td>
<td>226</td>
<td>30</td>
<td>100%</td>
<td>10</td>
<td>34</td>
<td>56%</td>
<td>110</td>
</tr>
<tr>
<td>msquare $4 \times 4$</td>
<td>226</td>
<td>30</td>
<td>82%</td>
<td>60</td>
<td>125</td>
<td>50%</td>
<td>120</td>
</tr>
<tr>
<td>hysteresis</td>
<td>76</td>
<td>16</td>
<td>100%</td>
<td>60</td>
<td>251</td>
<td>20%</td>
<td>60</td>
</tr>
<tr>
<td>merge</td>
<td>188</td>
<td>16</td>
<td>100%</td>
<td>1</td>
<td>2</td>
<td>100%</td>
<td>1</td>
</tr>
<tr>
<td>check-pressure</td>
<td>59</td>
<td>10</td>
<td>100%</td>
<td>10</td>
<td>4</td>
<td>90%</td>
<td>160</td>
</tr>
<tr>
<td>buf-get</td>
<td>262</td>
<td>18</td>
<td>100%</td>
<td>10</td>
<td>14</td>
<td>66%</td>
<td>600</td>
</tr>
<tr>
<td>strtok</td>
<td>316</td>
<td>40</td>
<td>100%</td>
<td>450</td>
<td>183</td>
<td>90%</td>
<td>180</td>
</tr>
<tr>
<td>strlen</td>
<td>134</td>
<td>18</td>
<td>100%</td>
<td>10</td>
<td>22</td>
<td>94%</td>
<td>120</td>
</tr>
<tr>
<td>findcolor</td>
<td>283</td>
<td>36</td>
<td>97%</td>
<td>800</td>
<td>328</td>
<td>61%</td>
<td>800</td>
</tr>
<tr>
<td>countlib</td>
<td>328</td>
<td>44</td>
<td>100%</td>
<td>120</td>
<td>48</td>
<td>54%</td>
<td>300</td>
</tr>
</tbody>
</table>

Time in sec. Random tests: 1000 tests
Outline

Motivations and challenges

Modelling

Test data generation

Safe CFG recovery

Conclusion & Future work
Reminder: safe CFG recovery problem

Input
- an executable code (array of bytes)
- an initial address
- a basic decoder: file $\times$ address $\mapsto$ instruction $\times$ size
Reminder: safe CFG recovery problem

"Chicken and egg" problem
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
Standard domains do not fit

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

Stride intervals [Balakrishnan-Reps 04,05,07]

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

Sets of bounded cardinality (k-sets) [Kinder-Veith 08,09,10]

- $x \in \{c_1, \ldots, c_q\}$ 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

Contribution [VMCAI 2011]

- A refinement-based approach dedicated to CFG reconstruction
- The technique is safe, moreover precise and efficient on our examples
Abstract domain: k-sets with local cardinality bounds

- gain efficiency through loss of precision
- still a global bound $K_{max}$ 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

Ingredient 2: refinement mechanism

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

For each target evaluating to $\top$

- follows backward data dependencies
- follows only $\top$-values (other locations are safe until now)
- stop on initial sources of precision loss: $\top_{\text{init}}$, $\top_{\langle c_1, \ldots, c_n \rangle}$

How to correct

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

Procedure PaR : $(P, K_{max}) \mapsto \text{Invariant}(P)$

pre : an unstructured program $P$ and a global bound $K_{max}$
post : compute an invariant of $P$ such that no dynamic target expression evaluates to $\top$, or fail

1. $\text{Dom} := \{(loc, v) \mapsto 0\}$
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)
Example

L1 \(\text{Dx}=1\) {1}

L2 \(\text{Dx}=1\) {2}

L3
\(x := x\)

L4 \(\text{Dx}=0\)
\(T<1,2>\)

L5 \(\text{Dx}=0\)

\(\text{Dx}=0\) {}

\(\text{Dx}=0\) {}

\(\text{jump } x\)

\text{source of prec. loss}
Example

```
L1
Dx=1
{1}

L2
Dx=1
{2}

L3
x := x

L4
Dx=0
T<1,2>

L5
Dx=0
{}

jump x
```

source of prec. loss
Example

L1: \( D_x = 1 \)  
\{1\}

L2: \( D_x = 1 \)  
\{2\}

L3:

\( x := x \)

L4:

\( x := x \)

\( T < 1, 2 > \)

\( D_x = 0 \)

L5:

\( x := x \)

\( T \)

\( D_x = 0 \)

Jump to L5

L3:

\( x := x \)

\( T \)

\( D_x = 0 \)

Source of prec. loss

Domain update \( D_x := 2 \)

Problem
Example

L1 \[ D_x=1 \]
{1}

x := x

L3

x := x

L4 \[ D_x=0 \]
{}

x := x

L5 \[ D_x=0 \]
{}

jump x
Technical details

Failure policy

- optimistic: fails only when no ispl is corrected
  [succeeds more, but more refinements]
- pessimistic: fails as soon as one ispl cannot be corrected
  [fails earlier, but may unduly fail]

Journal of the forward propagation phase

- record observed feasible branches, alias, dynamic targets
- prune backward data dependencies when searching ispl

Procedure inlining

- ⟨ formal stack , addr ⟩
- add precision, but forbid recursion
**Guarantees**

**Relative completeness**: PaR is relatively complete if PaR$(P, C)$ with parameter $K_{max}$ returns successfully when the forward k-set propagation with parameter $K_{max}$ does.

No relative completeness in the general case mainly because of control dependencies

Relative completeness for a non trivial subclass

- non-deterministic branching
- “simple” operators ($+, -, abs, \times k$ are ok, but not $\times$)
- array indexes are required to be $\neq \top$
Fresh news!

**Improved algorithm** [efficiency, robustness]

- # refinements indep. of $K_{max}$
- chaining of domain updates

**Combination of abstract domains** [precision]

- equalities: $e = e$, where $e ::= R|k|\circ e$
- flags: $b \Leftrightarrow e\{<, \leq, =, \geq, >\}e$
- intervals: $x \in [a..b]$
Case 1: compile \( \text{assume}(X == Y) \) into:

\[
\text{R1} := X; \text{R2} := Y; B := (\text{R1}==\text{R2}), \text{assume}(B)
\]

- only k-sets: \( B \in \{1\} \)
- k-sets + equalities: \( B \in \{1\} \land R_1 = X \land R_2 = Y \)
- k-sets + equalities + flags: \( B \in \{1\} \land R_1 = R_2 = X = Y \)

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

<table>
<thead>
<tr>
<th>program</th>
<th>#I</th>
<th>#DJ</th>
<th>#T</th>
<th>max #T</th>
<th>#SDJ</th>
<th>FT</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>aircraft</td>
<td>32405</td>
<td>51</td>
<td>461</td>
<td>16</td>
<td>51/51</td>
<td>10%</td>
<td>20s</td>
</tr>
<tr>
<td>SwitchCase</td>
<td>204</td>
<td>1</td>
<td>19</td>
<td>19</td>
<td>1/1</td>
<td>0%</td>
<td>&lt;1s</td>
</tr>
<tr>
<td>SingleRowInput</td>
<td>158</td>
<td>1</td>
<td>6</td>
<td>6</td>
<td>1/1</td>
<td>0%</td>
<td>&lt;1s</td>
</tr>
<tr>
<td>Keypad</td>
<td>224</td>
<td>1</td>
<td>8</td>
<td>8</td>
<td>1/1</td>
<td>0%</td>
<td>&lt;1s</td>
</tr>
<tr>
<td>EmergencyStop</td>
<td>475</td>
<td>1</td>
<td>10</td>
<td>10</td>
<td>1/1</td>
<td>0%</td>
<td>17s</td>
</tr>
<tr>
<td>TaskScheduler'</td>
<td>171</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>1/1</td>
<td>0%</td>
<td>&lt;1s</td>
</tr>
<tr>
<td>TaskScheduler</td>
<td>127</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>0/1</td>
<td>KO</td>
<td>&lt;1s</td>
</tr>
<tr>
<td>8 puzzles</td>
<td>66 - 1000</td>
<td>21</td>
<td>123</td>
<td>9</td>
<td>21/21</td>
<td>7%</td>
<td>7s</td>
</tr>
</tbody>
</table>

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

# SDJ : # dynamic jumps whose target ≠ ⊤
FT : % of recovered false targets
<table>
<thead>
<tr>
<th>program</th>
<th>#I</th>
<th>#DJ</th>
<th>#T</th>
<th>max #T</th>
<th>#SDJ</th>
<th>FT</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>aircraft</td>
<td>32405</td>
<td>51</td>
<td>461</td>
<td>16</td>
<td>51/51</td>
<td>10%</td>
<td>20s</td>
</tr>
<tr>
<td>SwitchCase</td>
<td>204</td>
<td>1</td>
<td>19</td>
<td>19</td>
<td>1/1</td>
<td>0%</td>
<td>&lt;1s</td>
</tr>
<tr>
<td>SingleRowInput</td>
<td>158</td>
<td>1</td>
<td>6</td>
<td>6</td>
<td>1/1</td>
<td>0%</td>
<td>&lt;1s</td>
</tr>
<tr>
<td>Keypad</td>
<td>224</td>
<td>1</td>
<td>8</td>
<td>8</td>
<td>1/1</td>
<td>0%</td>
<td>&lt;1s</td>
</tr>
<tr>
<td>EmergencyStop</td>
<td>475</td>
<td>1</td>
<td>10</td>
<td>10</td>
<td>1/1</td>
<td>0%</td>
<td>17s</td>
</tr>
<tr>
<td>TaskScheduler'</td>
<td>171</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>1/1</td>
<td>0%</td>
<td>&lt;1s</td>
</tr>
<tr>
<td>TaskScheduler</td>
<td>127</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>0/1</td>
<td>KO</td>
<td>&lt;1s</td>
</tr>
<tr>
<td>8 puzzles</td>
<td>66 - 1000</td>
<td>21</td>
<td>123</td>
<td>9</td>
<td>21/21</td>
<td>7%</td>
<td>7s</td>
</tr>
</tbody>
</table>

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

# SDJ : # dynamic jumps whose target $\neq \top$
FT : % of recovered false targets

Here : precise, efficient, tight refinement
<table>
<thead>
<tr>
<th>program</th>
<th>#I</th>
<th>#DJ</th>
<th>#T</th>
<th>max #T</th>
<th>#SDJ</th>
<th>FT</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>aircraft</td>
<td>32405</td>
<td>51</td>
<td>461</td>
<td>16</td>
<td>51/51</td>
<td>10%</td>
<td>20s</td>
</tr>
<tr>
<td>SwitchCase</td>
<td>204</td>
<td>1</td>
<td>19</td>
<td>19</td>
<td>1/1</td>
<td>0%</td>
<td>&lt;1s</td>
</tr>
<tr>
<td>SingleRowInput</td>
<td>158</td>
<td>1</td>
<td>6</td>
<td>6</td>
<td>1/1</td>
<td>0%</td>
<td>&lt;1s</td>
</tr>
<tr>
<td>Keypad</td>
<td>224</td>
<td>1</td>
<td>8</td>
<td>8</td>
<td>1/1</td>
<td>0%</td>
<td>&lt;1s</td>
</tr>
<tr>
<td>EmergencyStop</td>
<td>475</td>
<td>1</td>
<td>10</td>
<td>10</td>
<td>1/1</td>
<td>0%</td>
<td>17s</td>
</tr>
<tr>
<td>TaskScheduler'</td>
<td>171</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>1/1</td>
<td>0%</td>
<td>&lt;1s</td>
</tr>
<tr>
<td>TaskScheduler</td>
<td>127</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>0/1</td>
<td>KO</td>
<td>&lt;1s</td>
</tr>
<tr>
<td>8 puzzles</td>
<td>66 - 1000</td>
<td>21</td>
<td>123</td>
<td>9</td>
<td>21/21</td>
<td>7%</td>
<td>7s</td>
</tr>
</tbody>
</table>

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

# SDJ : # dynamic jumps whose target \( \neq T \)

FT : % of recovered false targets

Here : precise, efficient, tight refinement

[Beware : aeronautic software easier to verify than other software]
Motivations and challenges

Modelling

Test data generation

Safe CFG recovery

Conclusion & Future work
What we have seen so far
Place in the design space

Main design choices

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

Other points

- float: ✓ CFGBuilder, × CFGBuilder
- dynamic memory allocation: ●
- OS modelling: ●
On the road to security


- INRIA Rennes, LORIA, VERIMAG, EADS, VUPEN
- scale up to non-critical systems (alloc/free, libc, etc.)
- explore applications to malware & crash analysis
- also: self-modifying code, obfuscation, C++

Crash analysis [Internship 2013, with ML. Pottet & L. Mounier]

- start from a buggy but non-exploitable trace
- try to generalize the trace into an exploit
- idea: symbolic reasoning, trace folding

Extension of CFG recovery to non-critical executables

- dynamic mem. allocation, size++, libraries
- [PhD student starting soon (?), with Éric Goubault]