Binary-Level Testing of Embedded Programs

Sébastien Bardin

joint work with
P. Baufreton, N. Cornuet, P. Herrmann and S. Labbé

CEA LIST, Software Safety Lab (Paris area, France)

QSIC 2013
Focus on binary-level testing of safety-critical programs

We have been developing the OSMOSE tool since 2006 [ICST-08, ICST-09, TACAS-10, STVR-11]

- rely on Dynamic Symbolic Execution (DSE)
- first DSE tool over executable code, with SAGE [Godefroid-08]

Collaborations with industrial partners from energy and aeronautics
Focus on binary-level testing of safety-critical programs

We have been developing the OSMOSE tool since 2006 [ICST-08, ICST-09, TACAS-10, STVR-11]

- rely on Dynamic Symbolic Execution (DSE)
- first DSE tool over executable code, with SAGE [Godefroid-08]

Collaborations with industrial partners from energy and aeronautics

Contribution: between research and experience report

- original and practically-relevant features for DSE over safety-critical programs
- experience report on several case-studies
- (up-to-date description of OSMOSE)
choose a path $\pi$ of $P$

compute a *path predicate* $\varphi_\pi$:

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

solve $\varphi_\pi$ for satisfiability

SAT(s)? get a new pair $< s, \pi >$, update coverage

loop until nothing more to cover
choose a path $\pi$ of $P$

- compute a *path predicate* $\varphi_\pi$:
  $$v \models \varphi_\pi \Rightarrow P(v) \text{ follows } \pi$$

- solve $\varphi_\pi$ for satisfiability

- SAT(s) ? get a new pair $< s, \pi >$, update coverage

- loop until nothing more to cover
Dynamic Symbolic Execution

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

FIGURE
Dynamic Symbolic Execution

- choose a path $\pi$ of $P$
- compute a path predicate $\varphi_\pi$:
  $v \models \varphi_\pi \Rightarrow P(v)$ follows $\pi$
- solve $\varphi_\pi$ for satisfiability
- SAT($s$)? get a new pair $\langle s, \pi \rangle$, update coverage
- loop until nothing more to cover
Dynamic Symbolic Execution

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

[\text{wpre, spost}]
Dynamic Symbolic Execution

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

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

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

Model

\[ x > 0 \rightarrow x := x - 1 \]

\[ x := a + b \]

\[ x = 0 \rightarrow \]

Source code

```c
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

```
ABFF780BD70696CA101001BDE45
145634789234ABFF678ABDCF456
5A2B4C6D009F5F5D1E0835715697
145FEDBCADACBDAD459700346901
3456KAHA305G67H345BFFADECAD3
00113456735FFD451E13AB080DAD
344252FFAADBDA457345FD780001
FFF22546ADDAD9E989776600000000
```
Safety-Critical Programs

- Highly critical
- Reactive, embedded
- Very demanding certification processes
A nice class of programs

- no dynamic memory allocation, no dynamic thread creation
- smaller size, self-contained code (no huge libraries)

Typical program structure

- a (big) non-terminating main loop
  - read input, perform internal computations, update output
  - all other loops are statically bounded
- a few programming idioms, for example self-tests
  - \( A := 0; \text{assert}(A == 0); \)

Very strong validation requirements

- unit testing aims at very high coverage
- all uncovered objectives must be justified
- automated tools must come with some guarantees
**Motivation 1**: validation w/o any access to source code
- commercial off-the-shelf components
- legacy code

**Motivation 2**: “compiler-aware” validation
- ex: aeronautics and optimizing compilers
Motivation 1: validation w/o any access to source code
- commercial off-the-shelf components
- legacy code

Motivation 2: “compiler-aware” validation
- ex: aeronautics and optimizing compilers

Appealing, but more challenging than source code analysis
Challenges of binary code analysis

D1: Low-level semantics of data
- machine arithmetic, bit-level operations, untyped memory
  - difficult for any state-of-the-art formal technique

D2: Low-level semantics of control
- no distinction data / instructions, dynamic jumps (goto A)
- no (easy) syntactic recovery of Control-Flow Graph (CFG)
  - violate an implicit prerequisite for most formal techniques

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

D1: Low-level semantics of data
- machine arithmetic, bit-level operations, untyped memory
  - difficult for any state-of-the-art formal technique

D2: Low-level semantics of control
- no distinction data / instructions, dynamic jumps (goto A)
- no (easy) syntactic recovery of Control-Flow Graph (CFG)
  - violate an implicit prerequisite for most formal techniques

D3: Diversity of architectures and instruction sets
- support for many instructions, modelling issues
  - tedious, time consuming and error prone
Outline

- Introduction
- The OSMOSE tool
- New features for DSE over safety-critical programs
- Case-studies
- Conclusion
Outline

- Introduction
- The OSMOSE tool
- New features for DSE over safety-critical programs
- Case-studies
- Conclusion
The OSMOSE tool

- **PPC**
  - 0
  - 110111001
  - 001001000
  - 001111011
  - 110101101

- **DBA formal model**

- **CFGBuilder**
  - Safe control-flow graph

- **OSMOSE**
  - Test input partial cfg coverage
The OSMOSE tool

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

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

Instructions

- \texttt{lhs} := \texttt{rhs}, \texttt{goto addr}
- \texttt{goto addr}
- \texttt{goto expr}
- \texttt{ite(cond)?goto addr:goto addr’}

DBA

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

- Done:

- PPC
- (part of) x86
- M6800, C509
The OSMOSE tool

Instructions

- `expr{i .. j}`, `ext_{u,s}`, `::`
- `@(expr, k)`
- `+,-,*,/_{u,s},_%{u,s},=,\leq_{u,s},...`
- `!,\&,\|,\oplus,\ll,\gg_{u,s}`

DBA

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

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

coverage
The OSMOSE tool

- encode ISA, then simulation and analysis for free
- independent of computing power of targeted architecture

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

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

- **PPC**
- **DBA formal model**
- **CFGBuilder**
- **safe control-flow graph**
- **OSMOSE**
- **test input partial cfg coverage**
The OSMOSE tool

- **OSMOSE**
  - test data generation
  - input:
    - executable
    - entry, env.
  - criteria:
    - paths / branches / instr.
  - output:
    - test suite
    - partial CFG, coverage
- GBuilder
- safe control-flow graph
- test input partial cfg coverage
The OSMOSE tool

- **OSMOSE**
  - test data generation
  - input:
    - executable
    - entry, env.
  - criteria:
    - paths / branches / inst.
  - output:
    - test suite
    - partial CFG, coverage

- **GBuilder**

- **safe control-flow graph**

- **Dynamic symbolic execution**
  [ICST-08, STVR-11]

- **Bit-precise constraint solving**
  [TACAS-10]

- **Symbolic reasoning to discover new dynamic targets**
  [STVR-11]

- **Path pruning optimisations**
  [ICST-09]

- **Solver-independent optimizations**
  (preprocessing, solution reuse, etc.)
Limits of our approach

Constraints

- memory model or strings: nothing fancy, but sufficient for critical programs
- floats: only programs without tricky reasoning on floats
  [real issue] [orthogonal challenge]

Low-level synchronization mechanisms

- interrupts, multi-threading, time-based synchronization
- left to the validation expert (methodology)
- match current methodologies at SAGEM and EDF
Outline

- Introduction
- The OSMOSE tool
- New features for DSE over safety-critical programs
- Case-studies
- Conclusion
New features

- generic search engine
- search directives
- test suite replay and completion
- output of concrete and symbolic states
- specification of dynamic targets
- goal-oriented testing

Remember our goals

- very high coverage
- reliable results
- flexibility, allows guidance from user
DFS has a low “coverage speed”

Many heuristics have been defined in the literature, but no best one.

(a) DFS

(b) BFS
Idea = Generic search engine for easy integration of new searches

Our search engine requires

- an abstract data type \textsc{score}
- function \texttt{score : path \mapsto \textsc{score}}
- function \texttt{cmp : \textsc{score} \times \textsc{score} \mapsto \{<, =, >\}}

Algorithm

- rank all active paths \hspace{1em} (active \approx uncovered)
- choose one among the best
- solve its path predicate, add the resulting new paths
Generic search engine (3)
Generic search engine (3)
Generic search engine (3)
Easy to encode many existing heuristics

- **DFS**: score based on length, cmp on $\textit{max}$
- **BFS**: score based on length, cmp on $\textit{min}$
- random prefix: score is random, cmp is $\textit{min}$ (arbitrary)
- generational search [Godefroid et al, 2008]: score is $(\textit{generation}, \textit{gain})$, cmp is $\textit{max}_{\textit{generation}} \circ \textit{max}_{\textit{gain}}$

In OSMOSE

- generic search engine:
  - DFS, BFS, random path
  - minCall-DFS, minCall-BFS
- a dedicated DFS-based DSE engine [more memory efficient]
- random data generation
Directives restricting the search space

- unsat_br (addr, bool)
- repeat addr1 at most N (with reset on addr2)
- maxTryBranch (addr, bool) N

Test replay and completion

- validation: replay test suite in external simulator
- incremental testing: complete existing test suites, smooth integration with existing test process
- combination of search heuristics
Other features (2)

Export (and reuse) of symbolic states

- useful for modular reasoning (typically: initialization)
- beware: may lose completeness or correctness (no silver-bullet)

Specification of dynamic targets

- by a human or a static analyser
- the coverage measure reported by OSMOSE is sound w.r.t. the specification
- OSMOSE checks the specification along the DSE process, but no completeness
<table>
<thead>
<tr>
<th>features</th>
<th>High coverage</th>
<th>Trust</th>
<th>Flexibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>generic search engine († goal-oriented testing)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>search directives</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>test suite replay and completion</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>output of conc/symb. states</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>specification of dynamic targets</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
Outline

- Introduction
- The OSMOSE tool
- New features for DSE over safety-critical programs
- Case-studies
- Conclusion
Experiments

- automatic unit testing of medium-sized aircraft application
- full testing of a small (but tricky) aircraft application
- testing and comprehension of a third-party program
- experimental comparison of source vs binary coverage criteria
First case-study

Medium-size aircraft program (Sagem)
- 30,000 instructions, 250 functions
- max calldepth = 10

Goal: unit testing, no expert guidance

Results
- good coverage results for procedures with low height in the call graph (even with 2,000 instructions)
- tested on 40 functions with call-depth \( \leq 4 \):
  - full cover for 31 functions (in less than a few minutes)
  - bad cover (< 50%) for only 5 functions
- robustness issue with higher-level procedures
Second case-study

Small program (17 procedures and 2,600 instructions), SAGEM

Goal = full testing from the program entry point

Program recognized hard to cover by testing teams

- random testing or DFS-DSE stuck to 50% coverage
- many infeasible paths
- huge search space:
  - one loop must be unrolled $\geq 380$ times
  - artifical paths due to read-loops on volatile memory
Second case-study

Small program (17 procedures and 2,600 instructions), SAGEM

Goal = full testing from the program entry point

Program recognized hard to cover by testing teams

Approach

- search directives (main loop, read-loops)
- combination of MinCall-BFS and MinCall-DFS

Results

- 100% coverage of 15/17 procedures
- 50% coverage of 2 “library” procedures
- several uncovered branches have been shown to be uncoverable (in progress)
Third case-study

Toy control-command program written in assembly language (EdF)
- 3,000 instructions, 10 modules and 10 “library functions”
- Third-party software, sparse documentation

Complex to analyse: many unsat branches, long init

A modular approach
- Analyse library functions in isolation to detect likely-unsat branches or other issues (e.g., volatile memory)
- Insert `unsat_br` directives
- Modular analysis through export of the symbolic state obtained after initialization
Third case-study

Toy control-command program written in assembly language (EdF)
- 3,000 instructions, 10 modules and 10 “library functions”
- Third-party software, sparse documentation

Complex to analyse : many unsat branches, long init

Results
- achieve high coverage in 2 min only (otherwise : 35 min)
- help to understand the code (unfeasible branches, volatile memory, entries, etc.)
- help to pinpoint problems in doc (ack. by vendor)
Outline

- Introduction
- The OSMOSE tool
- New features for DSE over safety-critical programs
- Case-studies
- Conclusion
Conclusion

Binary-level testing of safety-critical programs

- important issue
- DSE is an interesting tool

Our contribution

- original and practically-relevant features for DSE over safety-critical programs
- experience report on several case-studies

Current challenges

- improve scaling w.r.t. call depth
- floats
- low-level synchronization (can handle through methodology)
- automatic sound CFG recovery
## Experiments (2)

<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