Structural Testing of Executables

Sébastien Bardin    Philippe Herrmann

CEA-LIST, Software Reliability Lab.
Overview

Structural testing at the machine code level

- automatic test data generation
- goal: structural coverage or bug finding
- do not address the problem of the oracle

Conceptual framework: symbolic/concolic execution

Three main contributions

- show how to adapt existing techniques to machine code
- combination of concolic execution and static analysis
- implementation of the tool OSMOSE

Limitations

- no floating-point numbers, no interruptions
Why binary-level analysis?

No source code available

- Components Off the Shelf (COTS)
- legacy code
- mobile code, malware
- certification of third-party software

Low confidence in the compiling process

- compilers may contain bugs
- optimisations preserve (?) correctness, what about security?
- What You See Is Not What You eXecute

High precision of the analysis

- quality of service (QoS): wcet, maximal stack height, etc.
- security
Outline

Motivations
- About machine code
- The Osmose tool
- Test data generation
- Bit-level constraint solving
- IR recovery
- Experiments
- Related work
- Conclusion
Motivations

About machine code

The OSMOSE tool

Test data generation

Bit-level constraint solving

IR recovery

Experiments

Related work

Conclusion
About machine code

The machine code is interpreted:

1. **PC** is the entry-point
2. decode **instr** at address **PC**
3. execute **instr**, update **PC**
4. goto 2

Instructions

- data: +, -, ×, /, >>, <<, xor, and, not, ...
- control: if, goto 10, cgoto A

Memory / variables

- registers and RAM (very large array)
- **PC** contains next instruction
- **SP** is the stack pointer
# Machine code vs Structured language

## Structured language

**Variables**
- unbounded # of variables
- types

**Functions**
- binding of arguments
- local context
- return to the caller
- generic function

**Control-flow**
- structured
- given *a priori*

## Machine code

**Variables**
- few registers + RAM
- a single type: bit-vectors

**Functions**
- goto callee_addr
- no context, no binding
- goto caller_addr
- goto x, where x may vary

**Control-flow**
- unstructured
- discovered at run-time
Motivations
Machine code
The OSMOSE Tool
Test data generation
Bit-vector solver
IR recovery
Experiments
Related work
Conclusion

Difficulties of binary-level analysis

No control-flow given a priori

- the CFG has to be discovered on-the-fly (IR recovery)
- cgoto \( x \): which values of \( x \) are legal?

Unstructured control-flow

- arbitrary goto
- low-level mechanism for next instruction and function call
  - interruptions

Bit-level instructions

- machine arithmetic, signed and unsigned operations
- bit-vector operations: rotate, extract, concat, etc.
  - floating-point numbers
Motivations

About machine code

The **OSMOSE** tool

Test data generation

Bit-level constraint solving

IR recovery

Experiments

Related work

Conclusion
Motivations

About machine code

The OSMOSE tool

Test data generation

Bit-level constraint solving

IR recovery

Experiments

Related work

Conclusion
Osmose = tool for automatic analysis of machine code

- Reverse-engineering
- Automatic test data generation

**Motivations**

- Machine code

**The Osmose Tool**

- Test data generation
- Bit-vector solver
- IR recovery

**Experiments**

**Related work**

**Conclusion**
Current state

Architecture support

- processors Motorola 6800, Intel 8051, Power PC 550
- all instructions for 8051, all instructions but one for 6800, most *user-level* instructions for PowerPC

Test objectives

- structural coverage: paths / branches / instructions
- quantitative objective

Environment

- entry point
- volatile memory
- initialised memory

Output

- test suite
- control-flow graph, call graph
- statistics (# branches, coverage, etc.)
Key technologies

Test data generation
- Concolic execution
- Bit-precise constraint solving

IR recovery
- Combination of static and dynamic recovery

Multiple architecture support
- Internal normalised instruction set, parametrised by an architecture template
- Template: size of a memory word, memory regions and registers
Motivations
About machine code
The OSMOSE tool
Test data generation
Bit-level constraint solving
IR recovery
Experiments
Related work
Conclusion
Outline

- Motivations
- About machine code
- The OSMOSE tool
- Test data generation
- Bit-level constraint solving
- IR recovery
- Experiments
- Related work
- Conclusion
**Framework: symbolic execution**

**“Path-based” test data generation**

1. Select a path $\pi$ in the CFG
2. Compute the path predicate $\varphi_\pi$
3. A solution to $\varphi_\pi = a$ test datum exercising the path
4. If still something to cover then goto 1

**Recent approach for programs**: PathCrawler, Dart, Cute, Exe

**Parameters**

- How to explore the set of paths?
- Which theory for $\varphi_\pi$?
- Memory model and alias management
- How to handle function calls?
Concolic execution: combination of concrete and symbolic executions

[GKS 05, SMA 05, WMM 04]

Symbolic execution: path predicate generation

Concrete execution: help the symbolic execution

- follow feasible paths only
- approximate non-linear constraints
- approximate library function calls
- approximate multiple-level pointers
- other?
Symbolic/Concolic execution in Osmose

Path enumeration: Bounded depth-first

Theory for path predicate: Bit-vector theory
- modulo arithmetic, signed and unsigned view
- extraction, concatenation, shift, and, or, xor, etc.

Functions: inlining

Concolic
- follow feasible paths only
  - detect legal alias relationship along a path
  - detect legal targets of cgoto A
  - semi-concrete execution to detect easy cases of unsat

Memory model and alias management: usually, for structured languages, possible aliasing are found according to variable types
- no notion of memory in our path predicate
- aliasing enforced a priori w.r.t. concrete execution
- aliasing depending on memory layout rather than types
Outline

- Motivations
- About machine code
- The OSMOSE tool
  - Test data generation
- Bit-level constraint solving
- IR recovery
- Experiments
- Related work
- Conclusion
About constraint programming (CP)

Constraint Programming: smart exploration of the space of valuations to find a solution

Constraint Programming = search + propagation

- Search: standard search algorithm (labelling, backtrack) in the tree of possible valuations
- Propagation: between two labelling steps, variable domains are narrowed according to propagation rules.

Pros & cons

- very general framework: any constraint over finite domains
- trade-off: propagation rules +/- complex
- quite efficient to find solutions of “easy” formulas
- theory over finite domains
- not very good at proving UNSAT
Based on a CP solver for bounded arithmetic [Bruno Marre]
- already used in the MBT tool Gatel for Lustre/Scade
- efficient propagators for linear/non-linear constraints
- mechanisms to detect UNSAT as soon as possible

We add a layer dedicated to the bit-vector theory
- modulo arithmetic with overflow and carry flags
- logical bit-wise operations
- other exotic constraints, e.g. countLeadingZero

Our approach
- rely on bounded arithmetic as much as possible
- add optimisations according to experiments
  - specific propagation rules: bit-wise operations and masks
  - specific constraints: compare(A,B,Res)
Motivations

About machine code

The Osmose tool

Test data generation

- Bit-level constraint solving

IR recovery

Experiments

Related work

Conclusion
Outline

- Motivations
- About machine code
- The Osmose tool
- Test data generation
- Bit-level constraint solving
  - IR recovery
- Experiments
- Related work
- Conclusion
Basic static analysis [IDA Pro]

- sound (find only legal instructions)
- cheap and easy to implement
- really incomplete: stop at each cgoto A

Advanced static analysis [T. Reps]

- complete: all legal instructions/targets are covered
- unsound: may find (too many) unfeasible targets
- very difficult to implement and get precise: cgoto \top
Concolic execution for IR recovery

- concrete execution may find new (legal) targets
- at each cgoto A: predicate to find new (legal) targets
  - sound: only legal targets are discovered
  - incomplete
  - CP solvers are not very good for ≠-constraints

Static analysis is used to cheaply provide the symbolic execution with possible targets

- constant propag. but ⊤ on alias and cgoto not propagated
  - easy to implement, efficient
  - neither sound nor complete

In Osmose: both techniques are interleaved

- still sound, more efficient on small tricky examples
- still incomplete (but test is incomplete in essence)
Motivations

About machine code

The OSMOSE tool

Test data generation

Bit-level constraint solving

IR recovery

Experiments

Related work

Conclusion
Motivations

About machine code

The OSMOSE tool

Test data generation

Bit-level constraint solving

IR recovery

Experiments

Related work

Conclusion
6 small C programs cross-compiled to PowerPC 550 (gcc) and Intel 8051 (sdcc)

Programs

- **msquare** (40 loc, 1 fun): # constraints is exponential
- **hysteresis** (30 loc, 2 fun): need long sequences of inputs
- **merge** (60 loc, 3 fun)
- **triangle** (20 loc, 3 fun)
- **cell** (20 loc, 3 fun): small tricky program given in [GKS 05]
- **list** (20 loc, 1 fun)

Remarks

- more machine code instructions than C instructions
- compiler optimisations are turned off (more difficult here)
- executables may vary greatly from one architecture to another (merge and sort)
Experiments II

Intel Pentium M 2Ghz, 1.2 GBytes RAM, Linux

Time out for the solver: 1 minute

Processor 8051 (8 bits)

<table>
<thead>
<tr>
<th>program</th>
<th>I</th>
<th>C</th>
<th>Branch cover</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>msquare 3×3</td>
<td>272</td>
<td>23</td>
<td>82%-100%</td>
<td>5.5</td>
</tr>
<tr>
<td>msquare 4×4</td>
<td>274</td>
<td>23</td>
<td>86%-100%</td>
<td>129</td>
</tr>
<tr>
<td>hysteresis</td>
<td>91</td>
<td>8</td>
<td>100%</td>
<td>45</td>
</tr>
<tr>
<td>merge</td>
<td>56</td>
<td>12</td>
<td>100%</td>
<td>13</td>
</tr>
<tr>
<td>triangle</td>
<td>102</td>
<td>19</td>
<td>52%-100%</td>
<td>0.8</td>
</tr>
<tr>
<td>cell</td>
<td>23</td>
<td>4</td>
<td>100%</td>
<td>0.4</td>
</tr>
<tr>
<td>list</td>
<td>13</td>
<td>3</td>
<td>100%</td>
<td>0.5</td>
</tr>
</tbody>
</table>

I: #instructions, C: #conditional branches, Time in seconds
## Experiments II

Intel Pentium M 2Ghz, 1.2 GBytes RAM, Linux

Time out for the solver: 1 minute

### Processor PowerPC 550 (32 bits)

<table>
<thead>
<tr>
<th>program</th>
<th>I</th>
<th>C</th>
<th>Branch cover</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>msquare 3×3</td>
<td>226</td>
<td>15</td>
<td>93% - 100%</td>
<td>7</td>
</tr>
<tr>
<td>msquare 4×4</td>
<td>226</td>
<td>15</td>
<td>82%</td>
<td>40</td>
</tr>
<tr>
<td>hysteresis</td>
<td>76</td>
<td>8</td>
<td>100%</td>
<td>66</td>
</tr>
<tr>
<td>merge</td>
<td>188</td>
<td>8</td>
<td>100%</td>
<td>0.5</td>
</tr>
<tr>
<td>triangle</td>
<td>40</td>
<td>9</td>
<td>100%</td>
<td>0.7</td>
</tr>
<tr>
<td>cell</td>
<td>18</td>
<td>4</td>
<td>100%</td>
<td>0.5</td>
</tr>
<tr>
<td>list</td>
<td>15</td>
<td>3</td>
<td>100%</td>
<td>0.5</td>
</tr>
</tbody>
</table>

I: #instructions, C: #conditional branches, Time in seconds
Motivations

About machine code

The OSMOSE tool

Test data generation

Bit-level constraint solving

IR recovery

Experiments

Related work

Conclusion
Motivations
About machine code
The OSMOSE tool
Test data generation
Bit-level constraint solving
IR recovery
Experiments
Related work
Conclusion
Some related work

Commercial tools from the Absint company
- static analysis for QoS properties
- critical systems, annotated C program, table of symbols

[Esparza-Schwoon et al. 01,07]
- structural testing of Java byte-code via model checking
- Java byte-code is very high-level compare to machine code

[Reps-Balakrishnan 04,05]
- static analysis for IR recovery and verification
- complementary to our technique

Structural testing via concolic execution
- many tools and teams: Cute, Dart, Exe, PathCrawler
- we share the same conceptual framework
- all these works consider C programs
Motivations
About machine code
The OSMOSE tool
Test data generation
Bit-level constraint solving
IR recovery
Experiments
Related work
Conclusion
## Conclusion

### About Osmose

- The tool performs well on small experiments
- First time structural testing is applied on machine code

### Lessons learned

- Automatic testing of machine code seems feasible
- Concolic execution and CP are our key concepts
- Concolic execution dramatically simplifies IR recovery
- CP can handle all operations on bit-vectors. Quick prototyping of all constraints, then optimise the bottlenecks
Future work

Experiments on real-size problems
- currently: case-studies from aeronautics and energy

Technical improvements
- better interface
- use infos from the table of symbols

Scientific challenges
- alias and memory management
- floating-point arithmetic
- interruptions
- scalability