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

Outline

- 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

:100D43000A0A4320636F6D70696C65722064656D78
:100D53006F6E7374726174696F6E2070726F6772F5
:100D6300616D0A0A00496E707574206F7065726157
:100D730074696F6E3A20272B272028414444292089
:100D83006F7220272D27202853554229203F20000A
:100CEA00759852758920758869758DF37BFF7A0D21
:100CFA007943120862120DDB8E648F65120DDB8E4A
:100D0A00668F677BFF7A0D7968120862120E0BEF05
:100D1A00B42B03D38001C3921130110CE56525670A
:100D2A00FFE5643566FE800BC3E5659567FFE564FC
:090D3A009566FE120DF780BD2242
:100E1C00496E707574204E756D626572203F2000AE
:100DBA008B688A69896A120E0BAB68056AE56AAAAA
:100DCA0069700205691 4F9EF120C63FFEFB40AE6C1
:010DDA0022F6

Instructions

- data: +, -, $\times$, /, `>>`, `<<`, `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

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

Outline

- 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

OSMOSE

OSMOSE = tool for automatic analysis of machine code

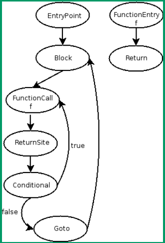
- Reverse-engineering
- Automatic test data generation

Input

:100D43000A0A4320636F6D70696C65722064656D78
:100D53006F6E7374726174696F6E2070726F6772F5
:100D6300616D0A0A00496E707574206F7065726157
:100D730074696F6E3A20272B272028414444292089
:100D83006F7220272D272028535554229203F20000A
:100CEA0075985275892075886975 8DF37BFF7A0D21
:100CFA00794312086212 0DDB8E648F65120DDB8E4A
:100D0A00668F677BFF7A0D7968120862120E0BEF05
:100D1A00B42B03D38001C3921130110CE56525670A
:100D2A00FFE5643566FE800BC3E5659567FFE564FC
:090D3A009566FE120DF780BD2242
:100E1C00496E707574204E756D626572203F2000AE
:100DBA008B688A69896A120E0BAB68056AE56AAAAA
:100DCA0069700205 6914F9EF120C63FFEFB40AE6C1
:010DDA0022F6

+ environment
+ objective

Outputs



+ test suite
+ report

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

Outline

- 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

"Path-based" test data generation

1. Select a path π in the CFG
2. Compute the path predicate φ_π
3. a solution to φ_π = 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 φ_π?
- 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

Outline

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. `count_leading_zero`

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)`

Outline

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 $\neq$-constraints

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

- constant propag. but $\top$ 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)*

Outline

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

Outline

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)

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

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)

program	I	C	Branch cover	Time
msquare 3×3	226	15	93% - 100%	7
msquare 4×4	226	15	82%	40
hysteresis	76	8	100%	66
merge	188	8	100%	0.5
triangle	40	9	100%	0.7
cell	18	4	100%	0.5
list	15	3	100%	0.5

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

Outline

- 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

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

Outline

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

Outline

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

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