The Insight binary code analysis platform

Aymeric Vincent

LaBRI

Torrents – 11 March 2013

Administrative Context

- Started during the ANR project BINCOA (2009-2012)
- Currently involved in the FUI project Marshal (2012-2014)
- A project of the “Formal Methods” team at LaBRI

Outline of the talk

1. Motivation
   - Object under study
   - Analysis goals
2. The Insight Microcode
3. The Insight platform
4. CFG Recovery

Object under study (1/5)

We want to analyze binary code. It can come as:

- an executable file,
- an object file,
- a dynamic library,
- a memory dump,
- ...

We don't have the corresponding high-level source code.

Object under study (2/5)

Let's see how it's different from C code.

An addition function in C

```
int
addition(int x, int y) {
    return x + y;
}
```

- We can compile this to assembly
- We can compile this to a binary object
- Let's do it and compare those versions.

Object under study (3/5)

```
$ gcc -S -m32 examples/addition-function.c
```

Addition function compiled into assembly

```
        .file   "addition-function.c"
        .text
        .globl  addition
        .type   addition, @function
addition:
.LFB0:
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        movl    12(%ebp), %eax
        movl    8(%ebp), %edx
        addl    %edx, %eax
```

Object under study (4/5)

```
$ objdump -d examples/addition-function.o
```

Addition function disassembled from object code

```
addition-function.o:     file format elf32-i386


Disassembly of section .text:

00000000 <addition>:
   0: 55                      push   %ebp
   1: 89 e5                   mov    %esp,%ebp
   3: 8b 45 0c                mov    0xc(%ebp),%eax
   6: 8b 55 08                mov    0x8(%ebp),%edx
   9: 01 d0                   add    %edx,%eax
   b: 5d                      pop    %ebp
   c: c3                      ret
```

Object under study (5/5)

We can notice the following losses between versions:

- From C to assembly
  - typing information of variables
  - variables have become “a piece of memory” or a register
  - the structure (and associated intent) of the code
- From assembly to binary
  - almost nothing
  - identification of functions
  - ease of reading

We want to use binary code as our input.

Why binary code

The lack of source code, for example:

- Low-level assembly code
- Off-the-shelf components (COTS)
- Legacy code
- Application stores
- Malware

Mistrust in the compilation chain:

- C compiler possibly buggy
- Optimization very probably buggy, yet optimized code could reduce hardware cost
- Link-time symbol resolution to unexpected symbol

Binary code is the real thing which will be executed

Analysis goals

The kind of analyses we would like to do:

- Software verification
  - check violation of memory boundaries
  - check arithmetic overflows
  - recover types
  - check reachability properties
  - check invariants
- Compute memory footprint
- Reverse engineering
- But, recovering the code itself is already non trivial...

Dynamic jumps

Recovering the code is non trivial, for two reasons:

- dynamic jumps occur frequently in assembly code
- code and data can live in the same memory space

Compared to a C like language:

- target candidates of a dynamic jump are every address instead of the functions of a compatible type
- the compiler can introduce dynamic jumps on its own.
  Example follows

Dynamic jump introduced by the compiler (1/6)

Function using a switch statement

```
enum { DIGIT, AT, BANG, MINUS }
f(char c) {
    switch (c) {
    case '0': case '1': case '2': case '3': case '4':
    case '5': case '6': case '7': case '8': case '9':
        return DIGIT;
    case '@':
        return AT;
    case '!':
        return BANG;
    case '-':
        return MINUS;
    }
}
```

Dynamic jump introduced by the compiler (2/6)

Compiled version of the function

```
f:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $4, %esp
    movl    8(%ebp), %eax
    movb    %al, -4(%ebp)
    movsbl -4(%ebp), %eax
    subl    $33, %eax
    cmpl    $31, %eax
    ja      .L2
    movl    .L7(,%eax,4), %eax
    jmp     *%eax
```

Dynamic jump introduced by the compiler (3/6)

Compiled version of the function

```
f:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $4, %esp
    movl    8(%ebp), %eax
    movb    %al, -4(%ebp)
    movsbl -4(%ebp), %eax
    subl    $33, %eax                ; ASCII for '!'
    cmpl    $31, %eax                ; 64 is ASCII for '@'
    ja      .L2                      ; Out of bounds - quit
    movl    .L7(,%eax,4), %eax ; Character becomes
    jmp     *%eax                    ;  offset in jump table
```

Dynamic jump introduced by the compiler (4/6)

The jump table

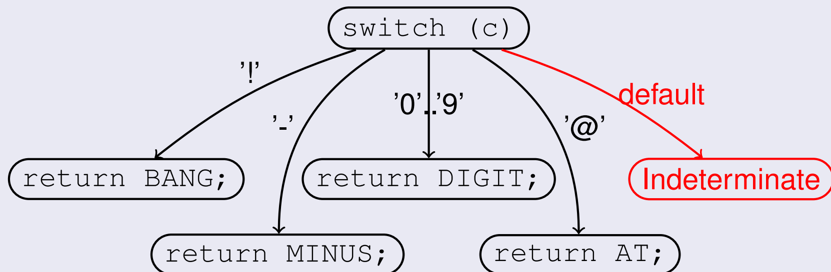
```
.L7:
  .long  .L3 ; '!'
  .long  .L2
  ...
  .long  .L2
  .long  .L4 ; '-'
  .long  .L2
  .long  .L2
  .long  .L5 ; '0'
  .long  .L5 ; '1'
  .long  .L5 ; '2'
  ...
  .long  .L2
  .long  .L6 ; '@'
```

Code pointed to by jump table

```
.L5:  movl  $0, %eax
      jmp   .L8
.L6:  movl  $1, %eax
      jmp   .L8
.L3:  movl  $2, %eax
      jmp   .L8
.L4:  movl  $3, %eax
      jmp   .L8
.L2:  jmp   .L1
.L8:
.L1:
      leave
      ret
```

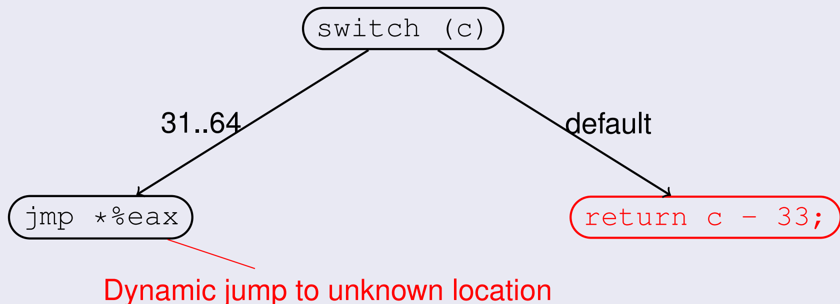
Dynamic jump introduced by the compiler (5/6)

Control Flow Graph from C version



- The CFG is completely known.
- The absence of default case appears immediately in the CFG, and is a potential bug.

Dynamic jump introduced by the compiler (6/6)



- The CFG ends in a dynamic jump which can lead to any place in memory
- The missing default case becomes a deterministic computation

The Insight Microcode

Our intermediate representation is a directed graph

- Nodes are labelled by memory locations
- Edges contain a guard and a statement

Nodes and edges can be annotated by arbitrary objects, for example

- assembly instructions which produced this microcode
- procedure calls/returns known or found
- procedure start/end
- higher-level constructs discovered
- ...

Microcode instructions

Microcode instructions are very limited:

- **Skip**
  does nothing
- **Assign**
  assigns the value of an expression to a l-value
- **Jump**
  jumps to the address computed by an expression (dynamic jump)
- **External**
  specifies a relation between current variable values and next variable values. This allows to model in a very abstract way a piece of code

Note that static jumps are not instructions, edges model them.

Microcode expressions

Microcode expressions

- operate on bitvectors
- are used in instructions and guards
- are very expressive
  - arithmetic operators
  - concatenation, sign extensions, bit reversal, ...
  - every expression can extract a sub-bitvector
- boolean expressions are expressions of bit-size 1

Example: “stackpointer minus four”

```
%esp{0:32} -{0:32} 4{0:32}
```

Microcode example

Assembly

```
0x8049284: push %eax
0x8049285: test %eax, %eax
0x8049287: ...
```

becomes microcode:

```
x8049284,0: %esp{0:32} := %esp{0:32} SUB 4{0:32}     -> x8049284,1
x8049284,1: [%esp{0:32},4,le] := %eax{0:32}          -> x8049285,0
x8049285,0: %tmp{0:32} := %eax{0:32} AND %eax{0:32} -> x8049285,1
x8049285,1: %pf{0:1} := %tmp{0:32} LT 0{0:32}       -> x8049285,2
x8049285,2: %zf{0:1} := %tmp{0:32} EQ 0{0:32}       -> x8049285,3
x8049285,3: %pf{0:1} := %tmp{0:1} XOR %tmp{1:1} XOR -> x8049285,4
x8049285,4: %cf{0:1} := 0{0:1}                      -> x8049285,5
x8049285,5: %of{0:1} := 0{0:1}                      -> x8049287,0
x8049287,0: ...
```

Memory model (1/2)

- Memory is a linear array of bytes.
- Regions of memory are defined by:
  - start address
  - size
  - access rights (read/write)

Our model takes into account the endianness. Remember:
The number $(cafebabe)_{16}$ can be stored in memory at address x in several ways, like:

Big endian

x:	CA	FE	BA	BE

Little endian

x:	BE	BA	FE	CA

Memory model (2/2)

L-values can be:

- a register
  - known, fixed size
- a contiguous memory area defined by its
  - start address (expression)
  - size (constant)
  - endianness (constant)

We chose to specify the endianness in each expression instead of fixing it for a given microcode model.

Microcode extracts are thus architecture agnostic

Memory trick (1/2)

The problem of byte by byte memory copies

Example with intervals: we know that memory at x contains a 4-byte integer in the interval $[2500, 2562] = [9.256 + 196, 10.256 + 2]$, stored as little endian.
Copying byte by byte means considering $x[0]$, $x[1]$, $x[2]$, $x[3]$.
We can at best compute:

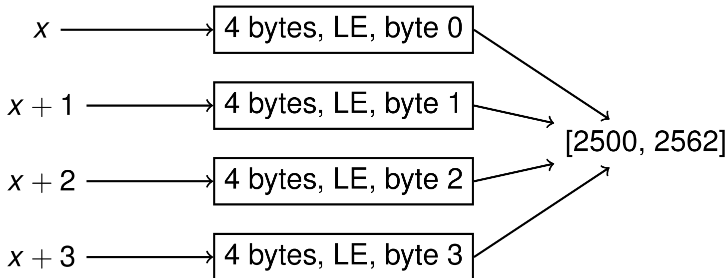
Intervals of sub bytes

$$\begin{aligned} x[0] \in & \quad [0, 0] \\ x[1] \in & \quad [0, 0] \\ x[2] \in & \quad [9, 10] \\ x[3] \in & \quad [196, 2] \end{aligned}$$

If we then read again this value at its destination address, we can at best obtain the interval $[9.256 + 0, 10.256 + 255] = [2304, 2815]$.

Memory trick (2/2)

To avoid this, we do not split abstract values unless required.



If x is read again with the same size and endianness, we can give back the exact correct interval.
If one byte is altered, we cancel this structure.

The Insight platform

The Insight platform is

- available at `http://insight.labri.fr/`
- under a 2-clause BSD-license
- written in C++, using templates
- reasonably portable across POSIX systems (uses GNU autotools)

It's split in three main components:

- The libinsight library which provides all features
- Tools to exercise the library (currently `cfgrecovery`)
- A test suite

Architecture

The architecture of Insight is split in five main components:

- Kernel
  - manipulates expressions, microcode, memory, annotations
- Loader
  - can read binary files (based on GNU libbfd)
  - can trace a process (uses `ptrace(2)`)
- Decoder
  - responsible for generating microcode from binary code
  - uses GNU libopcode and the custom C++ code
  - supports (32 bit) x86 and ARM
- Interpreter
  - can simulate code over an abstract domain
  - current domains: concrete, *k*-sets, intervals
- Support classes and functions

CFG Recovery

Usually, data flow analysis requires a known control flow graph.

But here, we need to perform data flow analysis to recover the CFG...

We need to combine both analyses.

CFG Recovery

Here are common ways to recover assembly code from binary:

Linear sweep

- Decode instruction
- Move to next instruction
- Loop

Recursive traversal

- Decode instruction
- If a call, enqueue target address
- Else if a ret, dequeue address
- Else move to next instruction

Recursive traversal gives more interesting results, but:

- How to know that an instruction is a call?
- How to know that an instruction is a ret?
- How to know the target address of a call?

Both fail on non trivial examples.

CFG Recovery

Our `cfgrecovery` tool offers:

- Linear sweep
- Recursive traversal
  - Calls and Rets are given by annotations from the decoder
  - Only static calls are honored
- Symbolic simulation
  - Uses an SMT solver (currently Mathsat or z3)
  - Keeps track of executions leading to a location by a formula

Demo on the toy callbacks example.

Pointers to others' works

- Jakstab, by Johannes Kinder, Munich, now at EPFL
- McVeto, Tom Reps et al., University of Wisconsin
- BitBlaze, UC Berkeley
- Binary Analysis Platform, CMU
- GDSL (Generic Decoder Specification Language), Axel Simon et al., Munich

Future directions

Here are things we plan to explore:

- Support and use the GDSL project
- Discovery of a possible execution trace from a core dump
- Recovery of variables and their types
- A symbolic data structure to keep track of relations between variables
- ...