Binary Program Analysis: Theory and Practice

(what you code is not what you execute)

Emmanuel Fleury
<emmanuel.fleury@labri.fr>

Joint work with:
Gérald Point <gerald.point@labri.fr>,
Aymeric Vincent <aymeric.vincent@labri.fr>.

LaBRI, Université Bordeaux 1, France

June 13, 2013

Overview

Overview

Program Analysis

Definition

Program analysis is the process of **automatically** deriving **properties** about the behavior of computer programs.

Dynamic Program Analysis

Analysis is performed by executing the program on chosen inputs. Traces of the actual executions are collected and processed. Properties about program behavior is deduced based on the analysis of these concrete executions.

Techniques

- Software Testing
- Performance Analysis
- ...

Static Program Analysis

Analysis is performed without actually executing the program. An abstract model of the program is issued and symbolically executed. Properties about program behavior is deduced from the analysis of these symbolic executions.

Techniques

- Abstract Interpretation
- Data-flow Analysis
- Model-checking
- Theorem Proving
- ...

Input Program Formats for Analysis

- **Abstract Model**: All **unnecessary information** for the analysis have been removed. Only **necessary information** remains.
- **Source Code**: Keep track of high-level information about the program such as **variables**, **types**, **functions**. But also, variable and function **names**, and **pragmas** or **code decorations**.
- **Bytecode**: May vary depending on the bytecode considered, but keep track of few high-level information about the program such as **types** and **functions**. But, programs are **unstructured**.
- **Binary File**: Only keep track of the **instructions** in an **unstructured way** (no for-loop, no clear argument passing in procedures, . . .). **No type**, **no naming**. But, the binary file may enclose **meta-data** that might be helpful (symbols, debug, . . .).
- **Memory Dump**: Pure assembler **instructions** with a full memory state of the current execution. We do not have anymore the **meta-data** of the executable file.

Input Program Formats for Analysis

- **Abstract Model**: All **unnecessary information** for the analysis have been removed. Only **necessary information** remains.
- **Source Code**: Keep track of high-level information about the program such as **variables**, **types**, **functions**. But also, variable and function **names**, and **pragmas** or **code decorations**.
- **Bytecode**: May vary depending on the bytecode considered, but keep track of few high-level information about the program such as **types** and **functions**. But, programs are **unstructured**.
- **Binary File**: Only keep track of the **instructions** in an **unstructured way** (no for-loop, no clear argument passing in procedures, ...). **No type**, **no naming**. But, the binary file may enclose **meta-data** that might be helpful (symbols, debug, ...).
- **Memory Dump**: Pure assembler **instructions** with a full memory state of the current execution. We do not have anymore the **meta-data** of the executable file.

Input Program Formats for Analysis

- **Abstract Model**: All **unnecessary information** for the analysis have been removed. Only **necessary information** remains.

- **Source Code**: Keep track of high-level information about the program such as **variables**, **types**, **functions**. But also, variable and function **names**, and **pragmas** or **code decorations**.

- **Bytecode**: May vary depending on the bytecode considered, but keep track of few high-level information about the program such as **types** and **functions**. But, programs are **unstructured**.

- **Binary File**: Only keep track of the **instructions** in an **unstructured way** (no for-loop, no clear argument passing in procedures, ...). **No type**, **no naming**. But, the binary file may enclose **meta-data** that might be helpful (symbols, debug, ...).

- **Memory Dump**: Pure assembler **instructions** with a full memory state of the current execution. We do not have anymore the **meta-data** of the executable file.

Input Program Formats for Analysis

UNIVERSITÉ DE BORDEAUX

- **Abstract Model**: All **unnecessary information** for the analysis have been removed. Only **necessary information** remains.
- **Source Code**: Keep track of high-level information about the program such as **variables**, **types**, **functions**. But also, variable and function **names**, and **pragmas** or **code decorations**.
- **Bytecode**: May vary depending on the bytecode considered, but keep track of few high-level information about the program such as **types** and **functions**. But, programs are **unstructured**.
- **Binary File**: Only keep track of the **instructions** in an **unstructured way** (no for-loop, no clear argument passing in procedures, ...). **No type**, **no naming**. But, the binary file may enclose **meta-data** that might be helpful (symbols, debug, ...).
- **Memory Dump**: Pure assembler **instructions** with a full memory state of the current execution. We do not have anymore the **meta-data** of the executable file.

Binary code is the closest format of what will be executed !

Why Analyze Binary Program?

UNIVERSITÉ DE BORDEAUX

The Lack of High-Level Source Code

- Low-level assembly code built-in the source code
- Legacy code
- Commercial Off-the-shelf software (COTS)
- Application stores (for cell phones and tablets)
- Malware or any "*hostile*" programs
- Technology forecasting

Mistrust in the Compilation Chain

- C compiler possibly buggy
- Optimization probably buggy, yet optimized code reduce hardware cost
- Checking low-level bugs (exploitability of a stack buffer-overflow)
- Bugs with a strong interconnection with hardware
- **What you code is not what you execute**[1] (see further example)

[1] Inspired by G. Balakrishnan and T. Reps.

Binary Code vs. Source Code (1/3)

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

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

We don't rely on getting the corresponding **high-level source code**.

Binary Code vs. Source Code (1/3)

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

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

We don't rely on getting the corresponding **high-level source code**.

Until now, most of the analysis techniques have been designed for source code analysis. So, what do we loose exactly at looking at binary programs only ?

Binary Code vs. Source Code (2/3)

- Compile this to assembly
- Compile this to a binary object
- Let's compare those versions.

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

Binary Code vs. Source Code (2/3)

- Compile this to assembly
- Compile this to a binary object
- Let's compare those versions.

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

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

```
.file "addition-function.c"
.text
.globl  addition
.type addition, @function
addition:
.LFB0:
pushl  %ebp
movl   %esp, %ebp
movl   12(%ebp), %eax
movl   8(%ebp), %edx
addl   %edx, %eax
popl   %ebp
ret
.LFE0:
.size addition, .-addition
.ident  "GCC: (Debian 4.7.3-4) 4.7.3"
.section  .note.GNU-stack,"",@progbits
```

Binary Code vs. Source Code (2/3)

- Compile this to assembly
- Compile this to a binary object
- Let's compare those versions.

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

$ gcc -S -m32 addition-function.c

```
.file "addition-function.c"
.text
.globl addition
.type addition, @function
addition:
.LFB0:
pushl %ebp
movl  %esp, %ebp
movl  12(%ebp), %eax
movl  8(%ebp), %edx
addl  %edx, %eax
popl  %ebp
ret
.LFE0:
.size addition, .-addition
.ident "GCC:␣(Debian␣4.7.3-4)␣4.7.3"
.section .note.GNU-stack,"",@progbits
```

$ objdump -d addition-function.o

```
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
```

Binary Code vs. Source Code (3/3)

We can notice the following losses between versions:

- **From C to assembly**
  - Typing information of variables;
  - Variables are turned into “a piece of memory” or a register;
  - The structure (and associated intent) of the code.
- **From assembly to binary**
  - Almost nothing;
  - Function names;
  - Ease of reading.

So, we loose information but this is not all, because:

Binary Code vs. Source Code (3/3)

We can notice the following losses between versions:

- **From C to assembly**
  - Typing information of variables;
  - Variables are turned into “a piece of memory” or a register;
  - The structure (and associated intent) of the code.

- **From assembly to binary**
  - Almost nothing;
  - Function names;
  - Ease of reading.

So, we loose information but this is not all, because:

“What you code is not what you execute!”

What You Code Is Not What You Execute (1/5)

Let consider a function using a simple "`switch`" statement, and suppose we forget the "`default`" case in the source code:

```
/* Function with 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;
    }
}
```

What You Code Is Not What You Execute (1/5)

UNIVERSITÉ DE BORDEAUX

Let consider a function using a simple "`switch`" statement, and suppose we forget the "`default`" case in the source code:

```
/* Function with 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;
    }
}
```

What happen when we have "`f('a')`" ?

What You Code Is Not What You Execute (2/5)

UNIVERSITÉ DE BORDEAUX

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
```

What You Code Is Not What You Execute (2/5)

UNIVERSITÉ DE BORDEAUX

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      .quit                 ; Out of bounds - quit
    movl    .L7(,%eax,4), %eax    ; Character becomes an
    jmp     *%eax                 ;  offset in jump table
```

What You Code Is Not What You Execute (3/5)

UNIVERSITÉ DE BORDEAUX

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 ; '@'
```

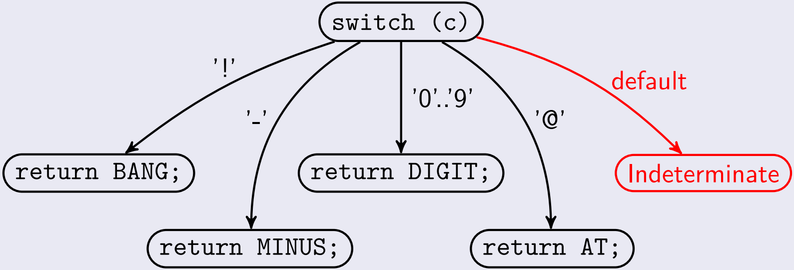
Code pointed to by jump table

```
.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
```

What You Code Is Not What You Execute (4/5)

UNIVERSITÉ DE BORDEAUX

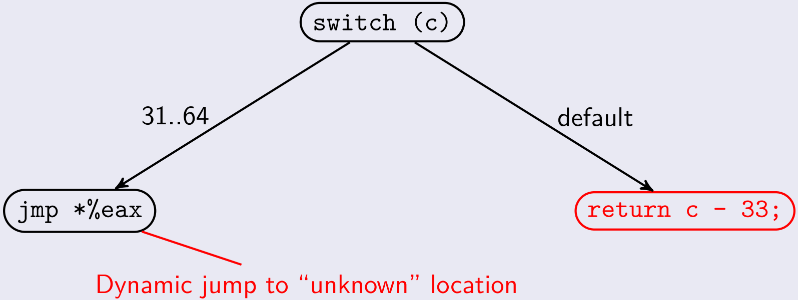
Control Flow Graph from C version



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

What You Code Is Not What You Execute (5/5)

Control Flow Graph from assembly version



- The CFG ends in a dynamic jump which can lead to any place in memory if we do not know what values can be stored in %eax.
- The missing default case becomes a deterministic computation.

Problems Specifically Linked to Binary Analysis

UNIVERSITÉ DE BORDEAUX

Compared to source code analysis the main difficulties are:

- We cannot start the analysis with a complete description of the program;
- We do not have types of the values that are manipulated (data/address/code);
- We lack of high-level structure on the recovered program (variables, functions, modules, . . .);
- The compiler can introduce dynamic jumps on its own;
- Code and data can live in the same memory space (*e.g.* self-modifying code).

And, the problems encountered in code analysis are still here:

- Undecidability of most of the interesting problems;
- Scalability problem (state-space explosion, line of code analyzed, . . .);

Analysis goals

Software Verification

- Check violation of memory boundaries;
- Check arithmetic overflows;
- Check reachability properties;
- Check invariants.

Reverse-Engineering

- Automatically rebuild as much as the control flow;
- Recover types and data structure in memory;
- Guess procedures or modules;
- Allow the user to check formally his hypothesis;
- Safe automated desobfuscation.

Analysis goals

Software Verification

- Check violation of memory boundaries;
- Check arithmetic overflows;
- Check reachability properties;
- Check invariants.

Reverse-Engineering

- Automatically rebuild as much as the control flow;
- Recover types and data structure in memory;
- Guess procedures or modules;
- Allow the user to check formally his hypothesis;
- Safe automated desobfuscation.

But, recovering the control-flow itself is already non trivial. . .

Overview

UNIVERSITÉ DE BORDEAUX

A Bit of Vocabulary

Processing Units

- **Loader**: Open the input file, parse the **meta-data** enclosed in the binary file and extract the **code** to be mapped in memory.
- **Decoder**: Given a **sequence of bytes**, translate it to an **assembler instruction** represented in a machine readable format.
- **Disassembler**: Combination of a **decoder** and a **strategy** to browse through the memory in order to recover all the program.
- **Decompiler**: Translate the assembly code into a high-level language with variables, functions and more (modules, objects, classes, ...).

Assembler Specific Terms

- **Opcode**: Hexadecimal code for assembler instructions;
- **Operands**: Hexadecimal code for arguments of an instruction;
- **Mnemonic**: Human-readable name of an assembler instruction;
- **Instruction**: Basic assembler operation;
- **Registers**: Basic unit of storage, usually of the size of a memory word.
- **Memory**: A (finite) table of bytes.

Program Representations and Collected Data

- **Context**: A valuation for the **memory** and the **registers**. The context is changed by the instructions.

- **Trace**: Given a memory state and an instruction, a **trace** is a valid sequence of instructions given their semantics.

- **Run**: A complete **trace** starting from the entrypoint of the program and from a correct initial memory.

- **Control-Flow (Graph)**: Oriented graph resulting of the union of all possible runs taken by the program. And,where:
  - node = (memory address $\times$ instruction)
  - edges = relation given by the union of all the possible runs

Disassembler Accuracy

The disassembler will output a CFG representing the union of all (potential) behaviors found by the disassembler. Namely, there are four types of disassemblers:

- **Exact**: The disassembler output the exact CFG that cover all the possible behaviors of the input program.
- **Under-approximation**: The disassembler output a subset of all the possible behaviors of the input program.
- **Over-approximation**: The disassembler output a set of behaviors that enclose the set of all possible ones.
- **Incorrect**: The disassembler output a set that may miss some behaviors and add some extra as well (we cannot say anything from this output).

Overview

UNIVERSITÉ DE BORDEAUX

Linear Sweep

Linear Sweep

1. Decode the first instruction at the entrypoint and store it;
2. Move (syntactically) the instruction pointer to the next instruction;
3. Decode the instruction and go to 2 if you are not out of the memory.

Linear Sweep

Linear Sweep

1. Decode the first instruction at the entrypoint and store it;
2. Move (syntactically) the instruction pointer to the next instruction;
3. Decode the instruction and go to 2 if you are not out of the memory.

Is It an Over-approximation ?

Linear Sweep

Linear Sweep

1. Decode the first instruction at the entrypoint and store it;
2. Move (syntactically) the instruction pointer to the next instruction;
3. Decode the instruction and go to 2 if you are not out of the memory.

Is It an Over-approximation ?

Lets disassemble this piece of binary code:

```
0804846c: eb04         jmp   0x804846e+4
0804846e: efbeadde     dd    0xdeadbeef    # Data hidden among instructions
08048472: a16e840408   mov   eax, [0x804846e]
08048477: 83c00a       add   eax, 0xa
```

Linear Sweep

Linear Sweep

1. Decode the first instruction at the entrypoint and store it;
2. Move (syntactically) the instruction pointer to the next instruction;
3. Decode the instruction and go to 2 if you are not out of the memory.

Is It an Over-approximation ?

Lets disassemble this piece of binary code:

```
0804846c: eb04        jmp   0x804846e+4
0804846e: efbeadde    dd    0xdeadbeef   # Data hidden among instructions
08048472: a16e840408  mov   eax, [0x804846e]
08048477: 83c00a      add   eax, 0xa
```

```
0804846c: eb04        jmp   0x804846e+4
```

Linear Sweep

Linear Sweep

1. Decode the first instruction at the entrypoint and store it;
2. Move (syntactically) the instruction pointer to the next instruction;
3. Decode the instruction and go to 2 if you are not out of the memory.

Is It an Over-approximation ?

Lets disassemble this piece of binary code:

```
0804846c: eb04         jmp   0x804846e+4
0804846e: efbeadde     dd    0xdeadbeef    # Data hidden among instructions
08048472: a16e840408   mov   eax, [0x804846e]
08048477: 83c00a       add   eax, 0xa
```

```
0804846c: eb04        jmp   0x804846e+4
0804846e: ef          out   dx, eax
```

Linear Sweep

Linear Sweep

1. Decode the first instruction at the entrypoint and store it;
2. Move (syntactically) the instruction pointer to the next instruction;
3. Decode the instruction and go to 2 if you are not out of the memory.

Is It an Over-approximation ?

Lets disassemble this piece of binary code:

```
0804846c: eb04        jmp  0x804846e+4
0804846e: efbeadde    dd   0xdeadbeef   # Data hidden among instructions
08048472: a16e840408  mov  eax, [0x804846e]
08048477: 83c00a      add  eax, 0xa
```

```
0804846c: eb04       jmp  0x804846e+4
0804846e: ef         out  dx, eax
0804846f: beaddea16e mov  esi, 0x6ea1dead
```

Linear Sweep

Linear Sweep

1. Decode the first instruction at the entrypoint and store it;
2. Move (syntactically) the instruction pointer to the next instruction;
3. Decode the instruction and go to 2 if you are not out of the memory.

Is It an Over-approximation ?

Lets disassemble this piece of binary code:

```
0804846c: eb04          jmp   0x804846e+4
0804846e: efbeadde      dd    0xdeadbeef   # Data hidden among instructions
08048472: a16e840408    mov   eax, [0x804846e]
08048477: 83c00a        add   eax, 0xa
```

```
0804846c: eb04         jmp    0x804846e+4
0804846e: ef           out    dx, eax
0804846f: beaddea16e   mov    esi, 0x6ea1dead
08048474: 840408       test   [eax+ecx], al
```

Linear Sweep

Linear Sweep

1. Decode the first instruction at the entrypoint and store it;
2. Move (syntactically) the instruction pointer to the next instruction;
3. Decode the instruction and go to 2 if you are not out of the memory.

Is It an Over-approximation ?

Lets disassemble this piece of binary code:

```
0804846c: eb04        jmp   0x804846e+4
0804846e: efbeadde    dd    0xdeadbeef    # Data hidden among instructions
08048472: a16e840408  mov   eax, [0x804846e]
08048477: 83c00a      add   eax, 0xa
```

```
0804846c: eb04        jmp   0x804846e+4
0804846e: ef          out   dx, eax
0804846f: beaddea16e  mov   esi, 0x6ea1dead
08048474: 840408      test  [eax+ecx], al
08048477: 83c00a      add   eax, 0xa
```

Linear Sweep

Linear Sweep

1. Decode the first instruction at the entrypoint and store it;
2. Move (syntactically) the instruction pointer to the next instruction;
3. Decode the instruction and go to 2 if you are not out of the memory.

Is It an Over-approximation ?

Lets disassemble this piece of binary code:

```
0804846c: eb04        jmp   0x804846e+4
0804846e: efbeadde    dd    0xdeadbeef    # Data hidden among instructions
08048472: a16e840408  mov   eax, [0x804846e]
08048477: 83c00a      add   eax, 0xa
```

```
0804846c: eb04        jmp   0x804846e+4
0804846e: ef          out   dx, eax
0804846f: beaddea16e  mov   esi, 0x6ea1dead
08048474: 840408      test  [eax+ecx], al
08048477: 83c00a      add   eax, 0xa
```

Yes!

Linear Sweep

Linear Sweep

1. Decode the first instruction at the entrypoint and store it;
2. Move (syntactically) the instruction pointer to the next instruction;
3. Decode the instruction and go to 2 if you are not out of the memory.

Linear Sweep

UNIVERSITÉ DE BORDEAUX

Linear Sweep

1. Decode the first instruction at the entrypoint and store it;
2. Move (syntactically) the instruction pointer to the next instruction;
3. Decode the instruction and go to 2 if you are not out of the memory.

Is It an Under-approximation ?

Linear Sweep

Linear Sweep

1. Decode the first instruction at the entrypoint and store it;
2. Move (syntactically) the instruction pointer to the next instruction;
3. Decode the instruction and go to 2 if you are not out of the memory.

Is It an Under-approximation ?

Lets disassemble this piece of binary code (entrypoint = 0x5):

```
00000000: b80003c1bb     inc   eax
00000005: b900000005     mov   $0, %eax   # Entrypoint here !
0000000a: 01c8           mov   $0, %ecx
0000000c: ff2502000000   jmp   *%eax
```

Linear Sweep

Linear Sweep

1. Decode the first instruction at the entrypoint and store it;
2. Move (syntactically) the instruction pointer to the next instruction;
3. Decode the instruction and go to 2 if you are not out of the memory.

Is It an Under-approximation ?

Lets disassemble this piece of binary code (entrypoint = 0x5):

```
00000000: b80003c1bb     inc   eax
00000005: b900000005     mov   $0, %eax   # Entrypoint here !
0000000a: 01c8           mov   $0, %ecx
0000000c: ff2502000000   jmp   *%eax
```

```
00000005: b900000005     mov   $0, %eax
```

Linear Sweep

Linear Sweep

1. Decode the first instruction at the entrypoint and store it;
2. Move (syntactically) the instruction pointer to the next instruction;
3. Decode the instruction and go to 2 if you are not out of the memory.

Is It an Under-approximation ?

Lets disassemble this piece of binary code (entrypoint = 0x5):

```
00000000: b80003c1bb     inc   eax
00000005: b900000005     mov   $0, %eax   # Entrypoint here !
0000000a: 01c8           mov   $0, %ecx
0000000c: ff2502000000   jmp   *%eax
```

```
00000005: b900000005     mov   $0, %eax
0000000a: 01c8           mov   $0, %ecx
```

Linear Sweep

Linear Sweep

1. Decode the first instruction at the entrypoint and store it;
2. Move (syntactically) the instruction pointer to the next instruction;
3. Decode the instruction and go to 2 if you are not out of the memory.

Is It an Under-approximation ?

Lets disassemble this piece of binary code (entrypoint = 0x5):

```
00000000: b80003c1bb     inc   eax
00000005: b900000005     mov   $0, %eax   # Entrypoint here !
0000000a: 01c8           mov   $0, %ecx
0000000c: ff2502000000   jmp   *%eax
```

```
00000005: b900000005     mov   $0, %eax
0000000a: 01c8           mov   $0, %ecx
0000000c: ff2502000000   jmp   *%eax
```

Linear Sweep

Linear Sweep

1. Decode the first instruction at the entrypoint and store it;
2. Move (syntactically) the instruction pointer to the next instruction;
3. Decode the instruction and go to 2 if you are not out of the memory.

Is It an Under-approximation ?

Lets disassemble this piece of binary code (entrypoint = 0x5):

```
00000000: b80003c1bb     inc   eax
00000005: b900000005     mov   $0, %eax   # Entrypoint here !
0000000a: 01c8           mov   $0, %ecx
0000000c: ff2502000000   jmp   *%eax
```

```
00000005: b900000005     mov   $0, %eax
0000000a: 01c8           mov   $0, %ecx
0000000c: ff2502000000   jmp   *%eax
```

Yes!

Linear Sweep

Linear Sweep

1. Decode the first instruction at the entrypoint and store it;
2. Move (syntactically) the instruction pointer to the next instruction;
3. Decode the instruction and go to 2 if you are not out of the memory.

Incorrect

Is It an Under-approximation ?

Lets disassemble this piece of binary code (entrypoint = 0x5):

```
00000000: b80003c1bb     inc   eax
00000005: b900000005     mov   $0, %eax   # Entrypoint here !
0000000a: 01c8           mov   $0, %ecx
0000000c: ff2502000000   jmp   *%eax
```

```
00000005: b900000005     mov   $0, %eax
0000000a: 01c8           mov   $0, %ecx
0000000c: ff2502000000   jmp   *%eax
```

Yes!

Recursive Traversal

Introduce a partial support of one type of dynamic jump (`call`/`ret`) with almost no semantics support.

Recursive Traversal

1. Do linear sweep until encountering a '`call`' or a '`ret`';
2. If this is a '`call`', stack its address, jump to it and go to 1;
3. If this is a '`ret`', pop the last address from the stack, jump to it and go to 1.

Recursive Traversal

Introduce a partial support of one type of dynamic jump (`call`/`ret`) with almost no semantics support.

Recursive Traversal

1. Do linear sweep until encountering a '`call`' or a '`ret`';
2. If this is a '`call`', stack its address, jump to it and go to 1;
3. If this is a '`ret`', pop the last address from the stack, jump to it and go to 1.

Is It an Over-approximation ?

Recursive Traversal

Introduce a partial support of one type of dynamic jump (`call`/`ret`) with almost no semantics support.

Recursive Traversal

1. Do linear sweep until encountering a '`call`' or a '`ret`';
2. If this is a '`call`', stack its address, jump to it and go to 1;
3. If this is a '`ret`', pop the last address from the stack, jump to it and go to 1.

Is It an Over-approximation ?

Lets disassemble this piece of binary code:

```
0804846c: eb04          jmp  0x804846e+4
0804846e: efbeadde      dd   0xdeadbeef
08048472: a16e840408    mov  eax, [0x804846e]
08048477: 83c00a        add  eax, 0xa
```

Recursive Traversal

Introduce a partial support of one type of dynamic jump (`call`/`ret`) with almost no semantics support.

Recursive Traversal

1. Do linear sweep until encountering a '`call`' or a '`ret`';
2. If this is a '`call`', stack its address, jump to it and go to 1;
3. If this is a '`ret`', pop the last address from the stack, jump to it and go to 1.

Is It an Over-approximation ?

Lets disassemble this piece of binary code:

```
0804846c: eb04          jmp   0x804846e+4
0804846e: efbeadde      dd    0xdeadbeef
08048472: a16e840408    mov   eax, [0x804846e]
08048477: 83c00a        add   eax, 0xa
```

```
0804846c: eb04         jmp    0x804846e+4
```

Recursive Traversal

Introduce a partial support of one type of dynamic jump (`call`/`ret`) with almost no semantics support.

Recursive Traversal

1. Do linear sweep until encountering a '`call`' or a '`ret`';
2. If this is a '`call`', stack its address, jump to it and go to 1;
3. If this is a '`ret`', pop the last address from the stack, jump to it and go to 1.

Is It an Over-approximation ?

Lets disassemble this piece of binary code:

```
0804846c: eb04        jmp  0x804846e+4
0804846e: efbeadde    dd   0xdeadbeef
08048472: a16e840408  mov  eax, [0x804846e]
08048477: 83c00a      add  eax, 0xa
```

```
0804846c: eb04       jmp  0x804846e+4
0804846e: ef         out  dx, eax
```

Recursive Traversal

Introduce a partial support of one type of dynamic jump (`call`/`ret`) with almost no semantics support.

Recursive Traversal

1. Do linear sweep until encountering a '`call`' or a '`ret`';
2. If this is a '`call`', stack its address, jump to it and go to 1;
3. If this is a '`ret`', pop the last address from the stack, jump to it and go to 1.

Is It an Over-approximation ?

Lets disassemble this piece of binary code:

```
0804846c: eb04        jmp   0x804846e+4
0804846e: efbeadde    dd    0xdeadbeef
08048472: a16e840408  mov   eax, [0x804846e]
08048477: 83c00a      add   eax, 0xa
```

```
0804846c: eb04       jmp    0x804846e+4
0804846e: ef         out    dx, eax
0804846f: beaddea16e mov    esi, 0x6ea1dead
```

Recursive Traversal

Introduce a partial support of one type of dynamic jump (`call`/`ret`) with almost no semantics support.

Recursive Traversal

1. Do linear sweep until encountering a '`call`' or a '`ret`';
2. If this is a '`call`', stack its address, jump to it and go to 1;
3. If this is a '`ret`', pop the last address from the stack, jump to it and go to 1.

Is It an Over-approximation ?

Lets disassemble this piece of binary code:

```
0804846c: eb04         jmp   0x804846e+4
0804846e: efbeadde     dd    0xdeadbeef
08048472: a16e840408   mov   eax, [0x804846e]
08048477: 83c00a       add   eax, 0xa
```

```
0804846c: eb04        jmp    0x804846e+4
0804846e: ef          out    dx, eax
0804846f: beaddea16e  mov    esi, 0x6ea1dead
08048474: 840408      test   [eax+ecx], al
```

Recursive Traversal

Introduce a partial support of one type of dynamic jump (`call`/`ret`) with almost no semantics support.

Recursive Traversal

1. Do linear sweep until encountering a '`call`' or a '`ret`';
2. If this is a '`call`', stack its address, jump to it and go to 1;
3. If this is a '`ret`', pop the last address from the stack, jump to it and go to 1.

Is It an Over-approximation ?

Lets disassemble this piece of binary code:

```
0804846c: eb04        jmp   0x804846e+4
0804846e: efbeadde    dd    0xdeadbeef
08048472: a16e840408  mov   eax, [0x804846e]
08048477: 83c00a      add   eax, 0xa
```

```
0804846c: eb04        jmp   0x804846e+4
0804846e: ef          out   dx, eax
0804846f: beaddea16e  mov   esi, 0x6ea1dead
08048474: 840408      test  [eax+ecx], al
08048477: 83c00a      add   eax, 0xa
```

Recursive Traversal

Introduce a partial support of one type of dynamic jump (`call`/`ret`) with almost no semantics support.

Recursive Traversal

1. Do linear sweep until encountering a '`call`' or a '`ret`';
2. If this is a '`call`', stack its address, jump to it and go to 1;
3. If this is a '`ret`', pop the last address from the stack, jump to it and go to 1.

Is It an Over-approximation ?

Lets disassemble this piece of binary code:

```
0804846c: eb04         jmp   0x804846e+4
0804846e: efbeadde     dd    0xdeadbeef
08048472: a16e840408   mov   eax, [0x804846e]
08048477: 83c00a       add   eax, 0xa
```

```
0804846c: eb04        jmp   0x804846e+4
0804846e: ef          out   dx, eax
0804846f: beaddea16e  mov   esi, 0x6ea1dead
08048474: 840408      test  [eax+ecx], al
08048477: 83c00a      add   eax, 0xa
```

Recursive Traversal

Introduce a partial support of one type of dynamic jump (`call`/`ret`) with almost no semantics support.

Recursive Traversal

1. Do linear sweep until encountering a '`call`' or a '`ret`';
2. If this is a '`call`', stack its address, jump to it and go to 1;
3. If this is a '`ret`', pop the last address from the stack, jump to it and go to 1.

Is It an Over-approximation ?

Lets disassemble this piece of binary code:

```
0804846c: eb04         jmp   0x804846e+4
0804846e: efbeadde     dd    0xdeadbeef
08048472: a16e840408   mov   eax, [0x804846e]
08048477: 83c00a       add   eax, 0xa
```

```
0804846c: eb04        jmp   0x804846e+4
0804846e: ef          out   dx, eax
0804846f: beaddea16e  mov   esi, 0x6ea1dead
08048474: 840408      test  [eax+ecx], al
08048477: 83c00a      add   eax, 0xa
```

Yes!

Recursive Traversal

Introduce a partial support of one type of dynamic jump (`call`/`ret`) with almost no semantics support.

Recursive Traversal

1. Do linear sweep until encountering a '`call`' or a '`ret`';
2. If this is a '`call`', stack its address, jump to it and go to 1;
3. If this is a '`ret`', pop the last address from the stack, jump to it and go to 1.

Recursive Traversal

Introduce a partial support of one type of dynamic jump (`call`/`ret`) with almost no semantics support.

Recursive Traversal

1. Do linear sweep until encountering a '`call`' or a '`ret`';
2. If this is a '`call`', stack its address, jump to it and go to 1;
3. If this is a '`ret`', pop the last address from the stack, jump to it and go to 1.

Is It an Under-approximation ?

Recursive Traversal

Introduce a partial support of one type of dynamic jump (`call`/`ret`) with almost no semantics support.

Recursive Traversal

1. Do linear sweep until encountering a '`call`' or a '`ret`';
2. If this is a '`call`', stack its address, jump to it and go to 1;
3. If this is a '`ret`', pop the last address from the stack, jump to it and go to 1.

Is It an Under-approximation ?

Lets disassemble this piece of binary code:

```
00000000: b80003c1bb     inc  eax
00000005: b900000005     mov  $0, %eax
0000000a: 01c8           mov  $0, %ecx
0000000c: ff2502000000   jmp  *%eax
```

Recursive Traversal

Introduce a partial support of one type of dynamic jump (`call`/`ret`) with almost no semantics support.

Recursive Traversal

1. Do linear sweep until encountering a '`call`' or a '`ret`';
2. If this is a '`call`', stack its address, jump to it and go to 1;
3. If this is a '`ret`', pop the last address from the stack, jump to it and go to 1.

Is It an Under-approximation ?

Lets disassemble this piece of binary code:

```
00000000: b80003c1bb     inc   eax
00000005: b900000005     mov   $0, %eax
0000000a: 01c8           mov   $0, %ecx
0000000c: ff2502000000   jmp   *%eax
```

```
00000005: b900000005     mov   $0, %eax
```

Recursive Traversal

Introduce a partial support of one type of dynamic jump (call/ret) with almost no semantics support.

Recursive Traversal

1. Do linear sweep until encountering a 'call' or a 'ret';
2. If this is a 'call', stack its address, jump to it and go to 1;
3. If this is a 'ret', pop the last address from the stack, jump to it and go to 1.

Is It an Under-approximation ?

Lets disassemble this piece of binary code:

```
00000000: b80003c1bb     inc   eax
00000005: b900000005     mov   $0, %eax
0000000a: 01c8           mov   $0, %ecx
0000000c: ff2502000000   jmp   *%eax
```

```
00000005: b900000005     mov   $0, %eax
0000000a: 01c8           mov   $0, %ecx
```

Recursive Traversal

Introduce a partial support of one type of dynamic jump (`call`/`ret`) with almost no semantics support.

Recursive Traversal

1. Do linear sweep until encountering a '`call`' or a '`ret`';
2. If this is a '`call`', stack its address, jump to it and go to 1;
3. If this is a '`ret`', pop the last address from the stack, jump to it and go to 1.

Is It an Under-approximation ?

Lets disassemble this piece of binary code:

```
00000000: b80003c1bb     inc   eax
00000005: b900000005     mov   $0, %eax
0000000a: 01c8           mov   $0, %ecx
0000000c: ff2502000000   jmp   *%eax
```

```
00000005: b900000005     mov   $0, %eax
0000000a: 01c8           mov   $0, %ecx
0000000c: ff2502000000   jmp   *%eax
```

Recursive Traversal

Introduce a partial support of one type of dynamic jump (`call`/`ret`) with almost no semantics support.

Recursive Traversal

1. Do linear sweep until encountering a '`call`' or a '`ret`';
2. If this is a '`call`', stack its address, jump to it and go to 1;
3. If this is a '`ret`', pop the last address from the stack, jump to it and go to 1.

Is It an Under-approximation ?

Lets disassemble this piece of binary code:

```
00000000: b80003c1bb     inc   eax
00000005: b900000005     mov   $0, %eax
0000000a: 01c8           mov   $0, %ecx
0000000c: ff2502000000   jmp   *%eax
```

```
00000005: b900000005     mov   $0, %eax
0000000a: 01c8           mov   $0, %ecx
0000000c: ff2502000000   jmp   *%eax
```

Yes!

Recursive Traversal

Introduce a partial support of one type of dynamic jump (`call`/`ret`) with almost no semantics support.

Recursive Traversal

Incorrect

1. Do linear sweep until encountering a '`call`' or a '`ret`';
2. If this is a '`call`', stack its address, jump to it and go to 1;
3. If this is a '`ret`', pop the last address from the stack, jump to it and go to 1.

Is It an Under-approximation ?

Lets disassemble this piece of binary code:

```
00000000: b80003c1bb     inc   eax
00000005: b900000005     mov   $0, %eax
0000000a: 01c8           mov   $0, %ecx
0000000c: ff2502000000   jmp   *%eax
```

```
00000005: b900000005     mov   $0, %eax
0000000a: 01c8           mov   $0, %ecx
0000000c: ff2502000000   jmp   *%eax
```

Yes!

About Syntaxic-Based Disassemblers

UNIVERSITÉ DE BORDEAUX

Proposition

Having no knowledge of the semantics (or partial knowledge), will always lead to an incorrect disassembler.

Sketch of Proof:

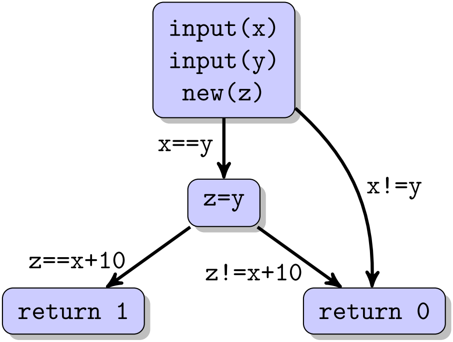
- **Over-approximation**: An "always false" statement will be always followed.
- **Under-approximation**: Dynamic jumps will never be followed.

Thus, a disassembler always need to know about the semantics of the instructions.

Overview

SMT-based Symbolic Exploration

```
1  int f(int x, int y)
2  {
3    int z;
4    z = y;
5
6    if (x == y)
7      if (z == x + 10)
8        return 1;
9
10   return 0;
11 }
```



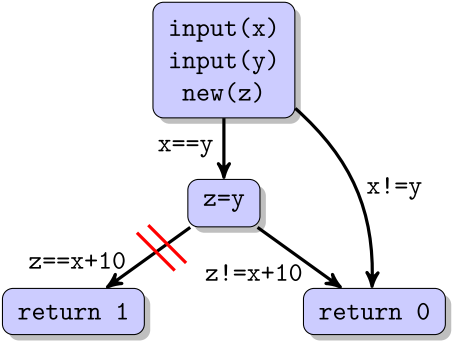
- **line 4**: $(x = y)$
- **line 8**: $(x = y) \wedge (y = x + 10)$ (**UNSAT**)
- **line 10** (path1): $(x \neq y)$
- **line 10** (path2): $(x = y) \wedge (y \neq x + 10)$

Algorithm

Explore the program and ask the SMT-solver at each program point if the path is feasible.

SMT-based Symbolic Exploration

```
1  int f(int x, int y)
2  {
3    int z;
4    z = y;
5
6    if (x == y)
7      if (z == x + 10)
8        return 1;
9
10   return 0;
11 }
```



- **line 4**: $(x = y)$
- **line 8**: $(x = y) \wedge (y = x + 10)$ (**UNSAT**)
- **line 10** (path1): $(x \neq y)$
- **line 10** (path2): $(x = y) \wedge (y \neq x + 10)$

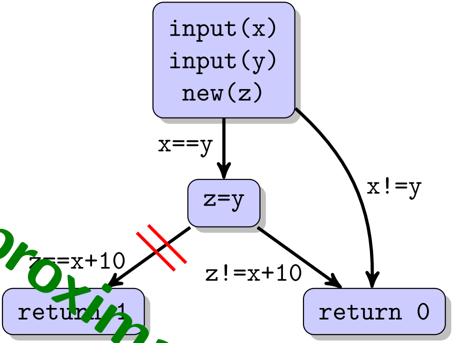
Algorithm

Explore the program and ask the SMT-solver at each program point if the path is feasible.

SMT-based Symbolic Exploration

```
1  int f(int x, int y)
2  {
3    int z;
4    z = y;
5
6    if (x == y)
7      if (z == x + 10)
8        return 1;
9
10   return 0;
11 }
```

Under-approximation



- **line 4**: $(x = y)$
- **line 8**: $(x = y) \wedge (y = x + 10)$ (**UNSAT**)
- **line 10** (path1): $(x \neq y)$
- **line 10** (path2): $(x = y) \wedge (y \neq x + 10)$

Algorithm

Explore the program and ask the SMT-solver at each program point if the path is feasible.

Directed Automated Random Exploration

UNIVERSITÉ DE BORDEAUX

DARE

1. First run the program on random inputs and get a trace;
2. Get each possible branching inside the previous trace and ask the SMT-solver to solve it.
3. If the SMT-solver fail, try to generate a random input to reach the untouched branches.

- **Original idea (2005)**:
  DART (Directed Automated Random Testing) [GKS05];
- **First applied to binary analysis (2008)**:
  Inside the OSMOSE software by CEA List [BH08]

Directed Automated Random Exploration

DARE

1. First run the program on random inputs and get a trace;
2. Get each possible branching inside the previous trace and ask the SMT-solver to solve it.
3. If the SMT-solver fail, try to generate a random input to reach the untouched branches.

Under-approximation

- **Original idea (2005)**:
  DART (Directed Automated Random Testing) [GKS05];
- **First applied to binary analysis (2008)**:
  Inside the OSMOSE software by CEA List [BH08]

Abstract Interpretation-Based CFG Recovery

Using an abstract interpretation framework on the CFG recovery problem is difficult because of the '*chicken-and-egg*' problem.

Abstract Interpretation-Based CFG Recovery [KZV09]

In '*An abstract interpretation-based framework for control flow reconstruction from binaries*' by Johannes Kinder, Florian Zuleger, and Helmut Veith (VMCAI 2009).

- Use a double abstract domain: CFG $\times$ Data-flow analysis;
- Recovery of the CFG is part of part of the process for reaching the fix-point.
- Data-flow analysis help on the way for the fix-point.
- The abstract domain of the data-flow analysis is a parameter of the framework. It can be anything as long as it match usual hypothesis of abstract domain (Galois connection, monotonicity, ...)
- Possible domains to use: k-sets, (stridded) intervals or VSA (Value-Set Analysis) [BR04].

Abstract Interpretation-Based CFG Recovery

Using an abstract interpretation framework on the CFG recovery problem is difficult because of the '*chicken-and-egg*' problem.

Abstract Interpretation-Based CFG Recovery [KZV09]

In '*An abstract interpretation-based framework for control flow reconstruction from binaries*' by Johannes Kinder, Florian Zuleger, and Helmut Veith (VMCAI 2009).

- Use a double abstract domain: CFG $\times$ Data-flow analysis;
- Recovery of the CFG is part of part of the process for reaching the fix-point.
- Data-flow analysis help on the way for the fix-point.
- The abstract domain of the data-flow analysis is a parameter of the framework. It can be anything as long as it match usual hypothesis of abstract domain (Galois connection, monotonicity. . .)
- Possible domains to use: k-sets, (stridded) intervals or VSA (Value-Set Analysis) [BR04].

Over-approximation

Alternating CFG Recovery

The previous framework lead very often to $\top$ (top) when recovering the CFG. Building very coarse over-approximation of the original CFG. The idea here is to alternate between under-approximation ('*trace collecting*' approach) and over-approximation ('*abstract-interpretation framework*').

Alternating CFG Recovery [KK12]

In '*Alternating control flow reconstruction*, by Kinder, Johannes and Kravchenko, Dmitry (VMCAI'12).

The semantics of the analyzed program is parametrized, three are given:

- Concrete semantics: A symbolic execution with full semantics;
- Under-approximation semantics: Build bounded traces of the program;
- Over-approximation semantics: Any abstract domain cited previously.

Then an '*Alternation framework* is defined that will decide when to use one semantics or the other.
The problem with that technique is that it is not clear what is obtained at the end. It is something between under-approximation and over-approximation.

CFG Recovery Methods: Summary

Syntax-Based Disassembler	Accuracy
Linear Sweep	Incorrect
Recursive Traversal	Incorrect

- All methods are just incorrect in all cases.

Semantics-Based Disassembler	Accuracy
SMT-based Symbolic Exploration	Under-approximation
Directed Automated Random Exploration	Under-approximation
Abstract Interpretation CFG Recovery	Over-approximation
Alternating CFG Reconstruction	?

- **Symbolic Exploration** and **Directed Automated Random Exploration** are of the same kind and provide under-approximation. They are useful for reverse-engineering.
- **Abstract-Interpretation framework** can be used for verification purpose.
- And, **Alternating CFG Reconstruction** is yet difficult to classify (need more work).

Overview

Insight Overview

- 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

- Targeting **UNIXish** platforms (should work with **Cygwin** but untested).
- Programmed in **C++ language**.
- Available under a 2-clause **BSD license** (Summer 2012).
- Essentially meant to be a **research tool** to ease experiments and techniques comparisons.
- Yet, we do care about **usability** and **users** (*eg*. use **GNU Autotools** build-system for build and install: `configure && make && make install`).

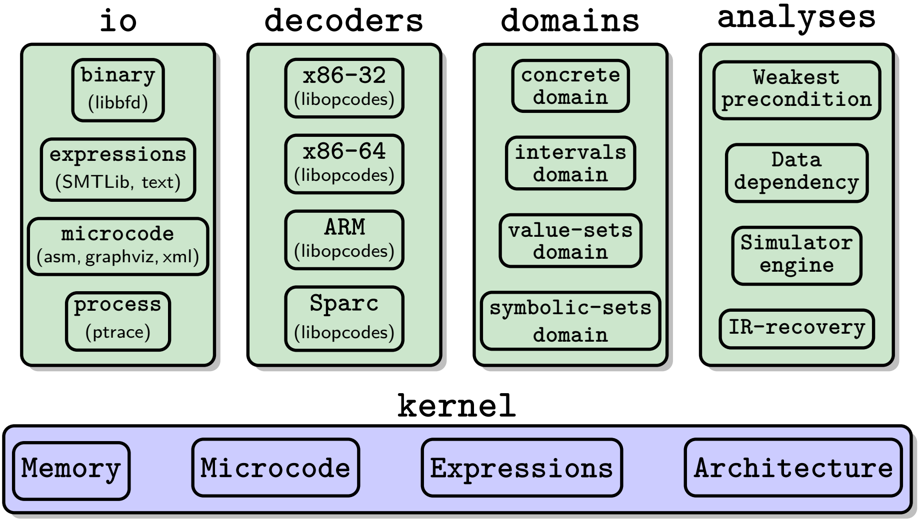
Insight Overview

- 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

- Targeting **UNIXish** platforms (should work with **Cygwin** but untested).
- Programmed in **C++ language**.
- Available under a 2-clause **BSD license** (Summer 2012).
- Essentially meant to be a **research tool** to ease experiments and techniques comparisons.
- Yet, we do care about **usability** and **users** (*eg*. use **GNU Autotools** build-system for build and install: `configure && make && make install`).

- But, we lack of time...

Insight Architecture



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 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.

Microcode expressions

- Operate on bitvectors arithmetic;
- Are used in instructions and guards;
- Are very expressive:
  - Arithmetic operators;
  - Operate on registers and immediate values;
  - 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 - 4`)

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

Microcode example

UNIVERSITÉ DE BORDEAUX

Assembly code:

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

Becomes the following microcode:

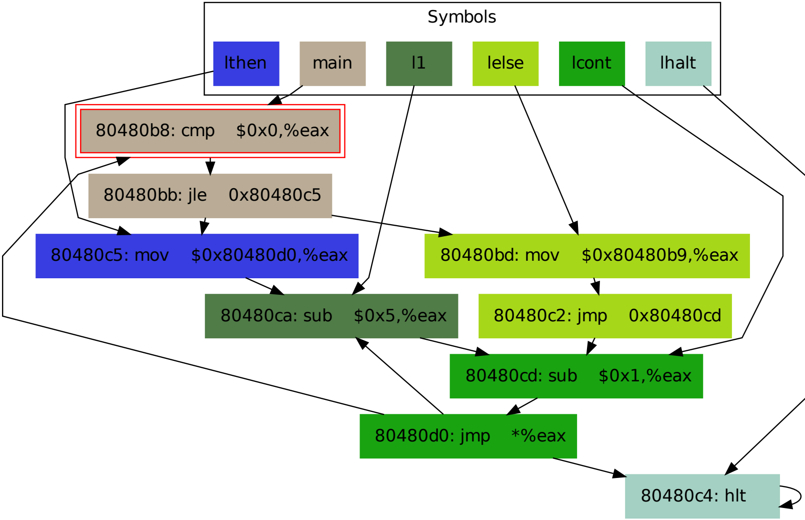
```
x8049284,0: %esp{0:32} := %esp{0:32} SUB 4{0:32}       -> x8049284,1
x8049284,1: [%esp{0:32},4,le] := %eax{0:32}            -> x8049285,0
x8049285,0: %tmp{0:32} := %eax{0:32} AND %eax{0:32}    -> x8049285,1
x8049285,1: %pf{0:1} := %tmp{0:32} LT 0{0:32}          -> x8049285,2
x8049285,2: %zf{0:1}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

Note that we assigned a '*local address*' to some sub-instructions.

A Full Example (1/2)

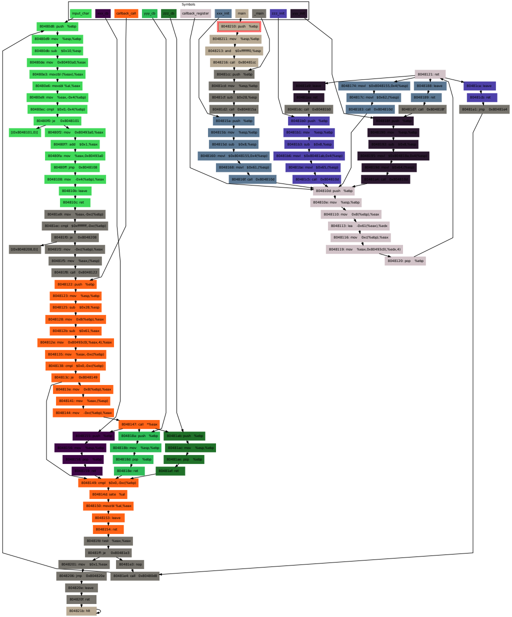
```
main:
    cmp $0, %eax
    jle lthen

lelse:
    mov $main + 1, %eax
    jmp lcont

lhalt:
    hlt

lthen:
    mov $l1 + 6, %eax

l1:
    sub $5, %eax

lcont:
    sub $1, %eax
    jmp *%eax
```

A Full Example (2/2)



jk_vmcai_demo1.bin

A Bigger Example (Callbacks)



Current & Future Work

Work on progress

- Add SPARC and amd64 architectures;
- Implement DARE and the abstract interpretation framework;
- Improve user interface (`cfgrecovery`);
- Make some more realistic case studies;
- Build a model-checker for microcode;
- Build a data-flow analyzer for microcode;
- Debug, debug, debug.

Future Work

- Build a complete UNIX and Microsoft Windows environment to simulate the execution properly;
- Recovery of high-level memory structures and types;
- Identification of procedures in the code;
- Handle self-modifying code;
- Automated de-obfuscation routines on microcode;
- ...

References I

Sébastien Bardin and Philippe Herrmann.
Structural testing of executables.
In *Proceedings of First International Conference on Software Testing, Verification, and Validation (ICST'2008)*, pages 22–31, Lillehammer, Norway, 2008. IEEE Computer Society.

Gogul Balakrishnan and Thomas Reps.
Analyzing memory access in x86 executables.
In *Proc. Int. Conf. on Compiler Construction*, pages 5–23, New York, NY, 2004. Springer.

Patrice Godefroid, Nils Klarlund, and Koushik Sen.
DART: directed automated random testing.
In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI'2005)*, pages 213–223, Chicago, IL, USA, 2005. ACM.

References II

Johannes Kinder and Dmitry Kravchenko.
Alternating control flow reconstruction.
In *Proceeding of 13th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'2012)*, volume 7148 of *Lecture Notes in Computer Science*, pages 267–282, Philadelphia, PA, 2012. Springer.

Johannes Kinder, Florian Zuleger, and Helmut Veith.
An abstract interpretation-based framework for control flow reconstruction from binaries.
In *Proceedings of 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'2009)*, volume 5403 of *Lecture Notes in Computer Science*, pages 214–228, Savannah, GA, USA, 2009. Springer.

Questions ?