Binary Code Analysis: Concepts and Perspectives

Emmanuel Fleury
<emmanuel.fleury@u-bordeaux.fr>

LaBRI, Université de Bordeaux, France

May 12, 2016

Overview

Overview

Why Looking at Binary Code?

- Analysis of legacy/off-the-shelf/proprietary software;
- Software reverse-engineering on malware (or others);
- Analysis of software generated with untrusted compiler;
- To capture many low-level security issues;
- Analysis of low-level interactions (hardware/OS).
- Optimize a binary without the sources (recompilation).

What we mean by "*Binary Programs*"?

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 usually **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.

What we mean by "*Binary Programs*"?

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 usually **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.

What we mean by "*Binary Programs*"?

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**.

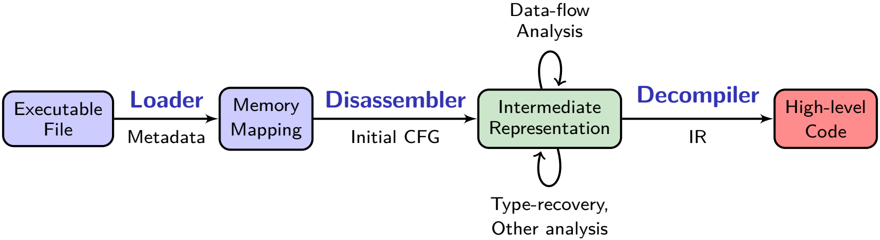
Bytecode: May vary depending on the bytecode considered, but keep track of few high-level information about the program such as **types** and **functions**. But, programs are usually **unstructured**.

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.

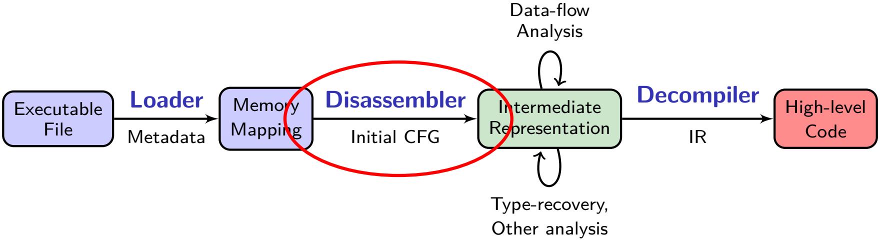
Binary code is the closest format of what will be executed!

Binary Analysis Pipeline



- **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** at an address in memory, translate it into an **intermediate representation** which will be analyzed afterward.
- **Disassembler**: Combination of a **decoder** and a **strategy** to browse through the memory in order to recover all the control-flow of the program.
- **Decompiler**: Translate the assembly code into a high-level language with variables, types, functions and more (modules, objects, classes, . . .).
- **Verificator**: Take the high-level representation of the program and check it against formally specified properties.

Binary Analysis Pipeline



- **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** at an address in memory, translate it into an **intermediate representation** which will be analyzed afterward.
- **Disassembler**: Combination of a **decoder** and a **strategy** to browse through the memory in order to recover all the control-flow of the program.
- **Decompiler**: Translate the assembly code into a high-level language with variables, types, functions and more (modules, objects, classes, . . .).
- **Verificator**: Take the high-level representation of the program and check it against formally specified properties.

Practical and Theoretical Challenges

- Trustable reconstruction of the program control-flow;
- "*As much as we can*" automation of recovery of the control-flow;
- Scaling the analysis from small to big binary software;
- Performing automatic and correct, but partial, decompilation;
- Verification of few accessibility properties on real binary programs;

Practical and Theoretical Challenges

- Trustable reconstruction of the program control-flow;
- "*As much as we can*" automation of recovery of the control-flow;
- Scaling the analysis from small to big binary software;
- Performing automatic and correct, but partial, decompilation;
- Verification of few accessibility properties on real binary programs;

**It does not seems to be a lot,
but it is already quite tricky!**

Overview

1. Introducing to Binary Code Analysis
2. **Why Is Binary Analysis Special?**
    - Unstructured Programming
    - Architectural Model
3. Low-level Programs Formal Model
4. Control-flow Recovery
5. Current and Future Trends

Unstructured Programming

No Advanced Programming Constructs and Types

- No variable (only registers and memory accesses)
- No advanced types (only: Value, Pointer or Instructions);
- No advanced control-flow constructs (`if-then-else`, `for`, `while`, `...`);

Jump-based Programming

- Static Jumps: `jmp 0x12345678`
- Dynamic Jumps: `jmp *%eax`

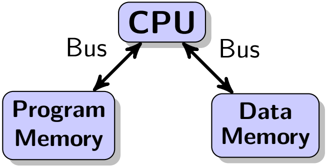
No Function Facilities

- No Function Type or Definition;
- No Argument Passing Facilities;
- No Procedural Context Facilities;

Architectural Model

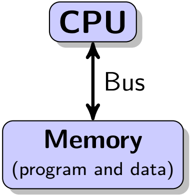
Harvard Architecture

- First implemented in the **Mark I** (1944).
- Keep program and data separated.
- Allows to fetch data and instructions in the same time.



Princeton Architecture (Von Neumann)

- First implemented in the **ENIAC** (1946).
- Allows **self-modifying code** and **entanglement of program and data**.

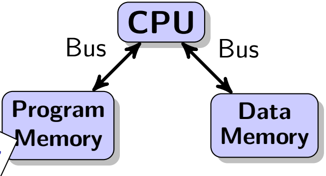


Architectural Model

Harvard Arc ture

- First implemented rk I (1944).
- Keep program and data se
- Allows to fetch data and instructi the same time.

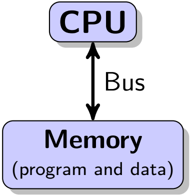
High-level programming



Princeton Ar tecture (Von Neumann)

- First implemented in th C (1946).
- Allows **self-modifying code** a **entanglement of program and dat**

Low-level programming



Overview

1. Introducing to Binary Code Analysis
2. Why Is Binary Analysis Special?
3. **Low-level Programs Formal Model**
4. Control-flow Recovery
5. Current and Future Trends

Why Another Execution Model?

- Semantics of low-level programs differ drastically from the usual models;
- Real execution models are optimized a lot which make them difficult to handle;
- A simpler model with the same expressivity make it easier to understand;
- A formalization is necessary to start thinking about proofs;

Memory Model

Memory

- $\mathbb{D} \subseteq \mathbb{N}$: A discrete numerical domain;
- $\mathbb{A} = \mathbb{D}$: Memory addresses (part of the numerical domain);
- $\mathbb{M} : \mathbb{A} \mapsto \mathbb{D}$: The set of all possible valuations of the memory;
- Notation: $m \in \mathbb{M}$, $m(addr) = val$.

Partially Initialized Memory

- $\mathbb{M}|_A : \mathbb{A} \mapsto \mathbb{D} \cup \{\perp\}$: The set of all partial valuations of $\mathbb{M}$, with $A \subseteq \mathbb{A}$ the initialized addresses such that $\forall a \in \mathbb{A} \setminus A$, $m(a) = \perp$.
- Notation: If $m \in \mathbb{M}|_A$, then $\mathbb{M}(m)$ denotes the set of all the fully initialized memories that can be spawned with m as generator.

Register(s)

- $\mathrm{pc} \in \mathbb{A}$: The program counter (the only register of the model);

Assembly Language

Instructions

- $\mathbb{I}$: A (finite) set of instructions;
- '**`load`** `value, addr`': Load the evaluation of '`value`' at '`addr`' in memory;
- '**`branch`** `cond, addr`': Jump to '`addr`' if the expression '`cond`' is zero;
- '**`halt`**': Stop program execution;

Expressions

Expressions are usual arithmetics (*e.g.* '`10*(5-7)/3`') with:

- **`[addr]`**$\in \mathbb{D}$: Access to the content of the address '`addr`'$\in \mathbb{A}$;

Operational Semantics

- $\mathbb{I} : \mathbb{M} \times \mathbb{A} \mapsto \mathbb{M} \times \mathbb{A}$ where $i \in \mathbb{I}$, $i(m,\mathrm{pc}) = (m',\mathrm{pc}')$;
- $[\![$**`load`** `value, addr`$]\!] =$ `([addr]:=value, pc':=pc+1)`
- $[\![$**`branch`** `cond, addr`$]\!] =$
  `([0]:=[0], if cond==0 then pc':=addr else pc':=pc+1)`
- $[\![$**`halt`**$]\!] =$ `([0]:=[0], pc':=pc)`

System Calls (optional)

- **`syscall read`** `addr`: Get an input (keyboard) and store it into '`addr`';
- **`syscall write`** `value`: Write '`value`' on the output (screen).

Low-level Programs

Decoding Instructions

- $\mathbb{I}$: A set of instructions as described before;
- $\delta : \mathbb{D} \mapsto \mathbb{I}$: A decoding function to map a value to an instruction.

Low-Level Program

A program $P = (m_{init}, \mathtt{pc}_0, \delta)$, is given by:

- An initial, partially initialized, memory $m_{init} \in \mathbb{M}|_A$ (with $A \subseteq \mathbb{A}$),
- An initial program counter $\mathtt{pc}_0 \in \mathbb{A}$,
- And a decoding function $\delta : \mathbb{D} \mapsto \mathbb{I}$.

Valid Run

$$(m_0, \mathtt{pc}_0) \xrightarrow{i_0(m_0,\mathtt{pc}_0)} (m_1, \mathtt{pc}_1) \xrightarrow{i_1(m_1,\mathtt{pc}_1)} \dots \xrightarrow{i_k(m_k,\mathtt{pc}_k)} (m_{k+1}, \mathtt{pc}_{k+1}) \dots$$

Where $m_0 \in \mathbb{M}(m_{init})$ and $\forall p \geq 0$, $i_p = \delta(m_p, \mathtt{pc}_p)$ and $(m_{p+1}, \mathtt{pc}_{p+1}) = i_p(m_p, \mathtt{pc}_p)$.

A First Full Example

- m_0 as below;
- $\mathrm{pc}_0 = 2$;
- δ: We already applied it to the memory when needed.

Addr	**Initial Content**
0x0	$\perp$
0x1	$\perp$
0x2	`syscall read 0`
0x3	`load [0], 1`
0x4	`load [0]*[1], 1`
0x5	`load [0]-1, 0`
0x6	`branch [0]!=0, 4`
0x7	`branch [1]!=0, 9`
0x8	`load 1, [1]`
0x9	`syscall write [1]`
0xa	`halt`

A First Full Example

- m_0 as below;
- $\mathrm{pc}_0 = 2$;
- δ: We already applied it to the memory when needed.

Addr	Initial Content	
0x0	$\perp$	`;; counter (var)`
0x1	$\perp$	`;; accumulator (var)`
0x2	`syscall read 0`	
0x3	`load [0], 1`	
0x4	`load [0]*[1], 1`	
0x5	`load [0]-1, 0`	
0x6	`branch [0]!=0, 4`	
0x7	`branch [1]!=0, 9`	
0x8	`load 1, [1]`	
0x9	`syscall write [1]`	
0xa	`halt`	

A First Full Example

- m_0 as below;
- $\mathrm{pc}_0 = 2$;
- δ: We already applied it to the memory when needed.

Addr	**Initial Content**
0x0	$\perp$
0x1	$\perp$
0x2	`syscall read 0`
0x3	`load [0], 1`
0x4	`load [0]*[1], 1`
0x5	`load [0]-1, 0`
0x6	`branch [0]!=0, 4`
0x7	`branch [1]!=0, 9`
0x8	`load 1, [1]`
0x9	`syscall write [1]`
0xa	`halt`

```
;; counter (var)
;; accumulator (var)
;; get initial value
```

A First Full Example

- m_0 as below;
- $\mathrm{pc}_0 = 2$;
- δ: We already applied it to the memory when needed.

Addr	Initial Content	
0x0	$\perp$	`;; counter (var)`
0x1	$\perp$	`;; accumulator (var)`
0x2	`syscall read 0`	`;; get initial value`
0x3	`load [0], 1`	`;; initialize accumulator`
0x4	`load [0]*[1], 1`	
0x5	`load [0]-1, 0`	
0x6	`branch [0]!=0, 4`	
0x7	`branch [1]!=0, 9`	
0x8	`load 1, [1]`	
0x9	`syscall write [1]`	
0xa	`halt`	

A First Full Example

- m_0 as below;
- $\mathrm{pc}_0 = 2$;
- δ: We already applied it to the memory when needed.

Addr	Initial Content	
0x0	⊥	;; counter (var)
0x1	⊥	;; accumulator (var)
0x2	`syscall read 0`	;; get initial value
0x3	`load [0], 1`	;; initialize accumulator
0x4	`load [0]*[1], 1`	;; compute next step
0x5	`load [0]-1, 0`	
0x6	`branch [0]!=0, 4`	
0x7	`branch [1]!=0, 9`	
0x8	`load 1, [1]`	
0x9	`syscall write [1]`	
0xa	`halt`	

A First Full Example

- m_0 as below;
- $\mathtt{pc}_0 = 2$;
- δ: We already applied it to the memory when needed.

Addr	Initial Content	
0x0	$\perp$	`;; counter (var)`
0x1	$\perp$	`;; accumulator (var)`
0x2	`syscall read 0`	`;; get initial value`
0x3	`load [0], 1`	`;; initialize accumulator`
0x4	`load [0]*[1], 1`	`;; compute next step`
0x5	`load [0]-1, 0`	`;; decrement counter`
0x6	`branch [0]!=0, 4`	
0x7	`branch [1]!=0, 9`	
0x8	`load 1, [1]`	
0x9	`syscall write [1]`	
0xa	`halt`	

A First Full Example

- m_0 as below;
- $\mathrm{pc}_0 = 2$;
- δ: We already applied it to the memory when needed.

Addr	Initial Content	
0x0	$\perp$	`;; counter (var)`
0x1	$\perp$	`;; accumulator (var)`
0x2	`syscall read 0`	`;; get initial value`
0x3	`load [0], 1`	`;; initialize accumulator`
0x4	`load [0]*[1], 1`	`;; compute next step`
0x5	`load [0]-1, 0`	`;; decrement counter`
0x6	`branch [0]!=0, 4`	`;; loop if counter is not zero`
0x7	`branch [1]!=0, 9`	
0x8	`load 1, [1]`	
0x9	`syscall write [1]`	
0xa	`halt`	

A First Full Example

- m_0 as below;
- $\mathrm{pc}_0 = 2$;
- δ: We already applied it to the memory when needed.

Addr	Initial Content	
0x0	$\perp$	`;; counter (var)`
0x1	$\perp$	`;; accumulator (var)`
0x2	`syscall read 0`	`;; get initial value`
0x3	`load [0], 1`	`;; initialize accumulator`
0x4	`load [0]*[1], 1`	`;; compute next step`
0x5	`load [0]-1, 0`	`;; decrement counter`
0x6	`branch [0]!=0, 4`	`;; loop if counter is not zero`
0x7	`branch [1]!=0, 9`	`;; check if result is not zero`
0x8	`load 1, [1]`	`;; if result was zero, set result to 1`
0x9	`syscall write [1]`	
0xa	`halt`	

A First Full Example

- m_0 as below;
- $\mathrm{pc}_0 = 2$;
- δ: We already applied it to the memory when needed.

Addr	Initial Content	
0x0	$\perp$	`;; counter (var)`
0x1	$\perp$	`;; accumulator (var)`
0x2	`syscall read 0`	`;; get initial value`
0x3	`load [0], 1`	`;; initialize accumulator`
0x4	`load [0]*[1], 1`	`;; compute next step`
0x5	`load [0]-1, 0`	`;; decrement counter`
0x6	`branch [0]!=0, 4`	`;; loop if counter is not zero`
0x7	`branch [1]!=0, 9`	`;; check if result is not zero`
0x8	`load 1, [1]`	`;; if result was zero, set result to 1`
0x9	`syscall write [1]`	`;; output result`
0xa	`halt`	

A First Full Example

- m_0 as below;
- $\mathrm{pc}_0 = 2$;
- δ: We already applied it to the memory when needed.

Addr	Initial Content	
0x0	$\perp$	`;; counter (var)`
0x1	$\perp$	`;; accumulator (var)`
0x2	`syscall read 0`	`;; get initial value`
0x3	`load [0], 1`	`;; initialize accumulator`
0x4	`load [0]*[1], 1`	`;; compute next step`
0x5	`load [0]-1, 0`	`;; decrement counter`
0x6	`branch [0]!=0, 4`	`;; loop if counter is not zero`
0x7	`branch [1]!=0, 9`	`;; check if result is not zero`
0x8	`load 1, [1]`	`;; if result was zero, set result to 1`
0x9	`syscall write [1]`	`;; output result`
0xa	`halt`	`;; halt program`

Dynamic Jumps

- m_0 as below;
- $\mathrm{pc}_0 = 1$;
- δ: We already applied it to the memory when needed.

Addr	Initial Content
0x0	$\perp$
0x1	`syscall read 0`
0x2	`branch 0<[1]<4, [1]*2+2`
0x3	`branch 0==0, 1`
0x4	`syscall write 10`
0x5	`halt`
0x6	`syscall write 42`
0x7	`halt`
0x8	`syscall write 1001`
0x9	`halt`

Dynamic Jumps

- m_0 as below;
- $\mathrm{pc}_0 = 1$;
- δ: We already applied it to the memory when needed.

Addr	Initial Content
0x0	$\perp$
0x1	`syscall read 0`
0x2	`branch 0<[1]<4, [1]*2+2`
0x3	`branch 0==0, 1`
0x4	`syscall write 10`
0x5	`halt`
0x6	`syscall write 42`
0x7	`halt`
0x8	`syscall write 1001`
0x9	`halt`

`;; input (var)`

Dynamic Jumps

- m_0 as below;
- $\mathrm{pc}_0 = 1$;
- δ: We already applied it to the memory when needed.

Addr	Initial Content	
0x0	$\perp$	;; input (var)
0x1	`syscall read 0`	;; get initial value
0x2	`branch 0<[1]<4, [1]*2+2`	
0x3	`branch 0==0, 1`	
0x4	`syscall write 10`	
0x5	`halt`	
0x6	`syscall write 42`	
0x7	`halt`	
0x8	`syscall write 1001`	
0x9	`halt`	

Dynamic Jumps

- m_0 as below;
- $\mathrm{pc}_0 = 1$;
- δ: We already applied it to the memory when needed.

Addr	Initial Content
0x0	$\perp$
0x1	`syscall read 0`
0x2	`branch 0<[1]<4, [1]*2+2`
0x3	`branch 0==0, 1`
0x4	`syscall write 10`
0x5	`halt`
0x6	`syscall write 42`
0x7	`halt`
0x8	`syscall write 1001`
0x9	`halt`

```
;; input (var)
;; get initial value
;; dynamic jump
```

Dynamic Jumps

- m_0 as below;
- $\mathrm{pc}_0 = 1$;
- δ: We already applied it to the memory when needed.

Addr	Initial Content
0x0	$\perp$
0x1	`syscall read 0`
0x2	`branch 0<[1]<4, [1]*2+2`
0x3	`branch 0==0, 1`
0x4	`syscall write 10`
0x5	`halt`
0x6	`syscall write 42`
0x7	`halt`
0x8	`syscall write 1001`
0x9	`halt`

```
;; input (var)
;; get initial value
;; dynamic jump
;; loop on wrong choice
```

Dynamic Jumps

- m_0 as below;
- $\mathrm{pc}_0 = 1$;
- δ: We already applied it to the memory when needed.

Addr	Initial Content	
0x0	$\perp$	;; input (var)
0x1	`syscall read 0`	;; get initial value
0x2	`branch 0<[1]<4, [1]*2+2`	;; dynamic jump
0x3	`branch 0==0, 1`	;; loop on wrong choice
0x4	`syscall write 10`	;; output 10 on 1
0x5	`halt`	
0x6	`syscall write 42`	;; output 42 on 2
0x7	`halt`	
0x8	`syscall write 1001`	;; output 1001 on 3
0x9	`halt`	

Self-modifying code

- m_0 as below;
- $\mathrm{pc}_0 = 2$;
- δ: We already applied it to the memory when needed but here are the rest:
  - 0 ↦ `branch [0]!=0, 4`
  - 1 ↦ `branch 0==0, 8`

Addr	Initial Content
0x0	$\perp$
0x1	0
0x2	`syscall read 0`
0x3	`load [1], 6`
0x4	`load [0], 1`
0x5	`load [0]-1, [0]`
0x6	`load [1], 0`
0x7	`branch 0==0, 3`
0x8	`halt`

Self-modifying code

- m_0 as below;
- $\mathrm{pc}_0 = 2$;
- δ: We already applied it to the memory when needed but here are the rest:
  - 0 ↦ `branch [0]!=0, 4`
  - 1 ↦ `branch 0==0, 8`

Addr	Initial Content
0x0	$\perp$
0x1	0
0x2	`syscall read 0`
0x3	`load [1], 6`
0x4	`load [0], 1`
0x5	`load [0]-1, [0]`
0x6	`load [1], 0`
0x7	`branch 0==0, 3`
0x8	`halt`

`;; input (var)`

`;; initialized data`

Self-modifying code

- m_0 as below;
- $\mathrm{pc}_0 = 2$;
- δ: We already applied it to the memory when needed but here are the rest:
  - 0 ↦ `branch [0]!=0, 4`
  - 1 ↦ `branch 0==0, 8`

	Addr	Initial Content	
	0x0	$\perp$	`;; input (var)`
	0x1	0	`;; initialized data`
⇒	0x2	`syscall read 0`	`;; get initial value`
	0x3	`load [1], 6`	
	0x4	`load [0], 1`	
	0x5	`load [0]-1, [0]`	
	0x6	`load [1], 0`	
	0x7	`branch 0==0, 3`	
	0x8	`halt`	

Self-modifying code

- m_0 as below;
- $\mathrm{pc}_0 = 2$;
- δ: We already applied it to the memory when needed but here are the rest:
  - 0 ↦ `branch [0]!=0, 4`
  - 1 ↦ `branch 0==0, 8`

	Addr	Initial Content	
	0x0	`n`	`;; input (var)`
	0x1	`0`	`;; initialized data`
⇒	0x2	`syscall read 0`	`;; get initial value`
	0x3	`load [1], 6`	
	0x4	`load [0], 1`	
	0x5	`load [0]-1, [0]`	
	0x6	`load [1], 0`	
	0x7	`branch 0==0, 3`	
	0x8	`halt`	

Self-modifying code

- m_0 as below;
- $\mathrm{pc}_0 = 2$;
- δ: We already applied it to the memory when needed but here are the rest:
  - 0 ↦ `branch [0]!=0, 4`
  - 1 ↦ `branch 0==0, 8`

	Addr	Initial Content	
	0x0	`n`	`;; input (var)`
	0x1	`0`	`;; initialized data`
	0x2	`syscall read 0`	`;; get initial value`
⇒	0x3	`load [1], 6`	`;; rewrite code ahead`
	0x4	`load [0], 1`	
	0x5	`load [0]-1, [0]`	
	0x6	`load [1], 0`	
	0x7	`branch 0==0, 3`	
	0x8	`halt`	

Self-modifying code

- m_0 as below;
- $\mathrm{pc}_0 = 2$;
- δ: We already applied it to the memory when needed but here are the rest:
  - 0 ↦ `branch [0]!=0, 4`
  - 1 ↦ `branch 0==0, 8`

	Addr	**Initial Content**	
	0x0	`n`	`;; input (var)`
	0x1	`0`	`;; initialized data`
	0x2	`syscall read 0`	`;; get initial value`
⇒	0x3	`load [1], 6`	`;; rewrite code ahead`
	0x4	`load [0], 1`	
	0x5	`load [0]-1, [0]`	
	0x6	`branch [0]!=0, 4`	
	0x7	`branch 0==0, 3`	
	0x8	`halt`	

Self-modifying code

- m_0 as below;
- $\mathrm{pc}_0 = 2$;
- δ: We already applied it to the memory when needed but here are the rest:
  - 0 ↦ `branch [0]!=0, 4`
  - 1 ↦ `branch 0==0, 8`

	Addr	Initial Content	
	0x0	`n`	`;; input (var)`
	0x1	`0`	`;; initialized data`
	0x2	`syscall read 0`	`;; get initial value`
	0x3	`load [1], 6`	`;; rewrite code ahead`
⇒	0x4	`load [0], 1`	`;; overwrite [1] with [0]`
	0x5	`load [0]-1, [0]`	
	0x6	`branch [0]!=0, 4`	
	0x7	`branch 0==0, 3`	
	0x8	`halt`	

Self-modifying code

- m_0 as below;
- $\mathrm{pc}_0 = 2$;
- δ: We already applied it to the memory when needed but here are the rest:
  - 0 ↦ `branch [0]!=0, 4`
  - 1 ↦ `branch 0==0, 8`

	Addr	**Initial Content**	
	0x0	`n`	`;; input (var)`
	0x1	`n`	`;; initialized data`
	0x2	`syscall read 0`	`;; get initial value`
	0x3	`load [1], 6`	`;; rewrite code ahead`
⇒	0x4	`load [0], 1`	`;; overwrite [1] with [0]`
	0x5	`load [0]-1, [0]`	
	0x6	`branch [0]!=0, 4`	
	0x7	`branch 0==0, 3`	
	0x8	`halt`	

Self-modifying code

- m_0 as below;
- $\mathrm{pc}_0 = 2$;
- δ: We already applied it to the memory when needed but here are the rest:
  - $0 \mapsto$ `branch [0]!=0, 4`
  - $1 \mapsto$ `branch 0==0, 8`

	Addr	Initial Content	
	0x0	`n`	`;; input (var)`
	0x1	`n`	`;; initialized data`
	0x2	`syscall read 0`	`;; get initial value`
	0x3	`load [1], 6`	`;; rewrite code ahead`
	0x4	`load [0], 1`	`;; overwrite [1] with [0]`
⇒	0x5	`load [0]-1, [0]`	`;; decrement [0]`
	0x6	`branch [0]!=0, 4`	
	0x7	`branch 0==0, 3`	
	0x8	`halt`	

Self-modifying code

- m_0 as below;
- $\mathrm{pc}_0 = 2$;
- δ: We already applied it to the memory when needed but here are the rest:
  - 0 ↦ `branch [0]!=0, 4`
  - 1 ↦ `branch 0==0, 8`

	Addr	Initial Content	
	0x0	`n-1`	`;; input (var)`
	0x1	`n`	`;; initialized data`
	0x2	`syscall read 0`	`;; get initial value`
	0x3	`load [1], 6`	`;; rewrite code ahead`
	0x4	`load [0], 1`	`;; overwrite [1] with [0]`
⇒	0x5	`load [0]-1, [0]`	`;; decrement [0]`
	0x6	`branch [0]!=0, 4`	
	0x7	`branch 0==0, 3`	
	0x8	`halt`	

Self-modifying code

- m_0 as below;
- $\mathrm{pc}_0 = 2$;
- δ: We already applied it to the memory when needed but here are the rest:
  - 0 ↦ `branch [0]!=0, 4`
  - 1 ↦ `branch 0==0, 8`

	Addr	Initial Content	
	0x0	`n-1`	`;; input (var)`
	0x1	`n`	`;; initialized data`
	0x2	`syscall read 0`	`;; get initial value`
	0x3	`load [1], 6`	`;; rewrite code ahead`
	0x4	`load [0], 1`	`;; overwrite [1] with [0]`
	0x5	`load [0]-1, [0]`	`;; decrement [0]`
⇒	0x6	`branch [0]!=0, 4`	`;; if not zero loop to 4`
	0x7	`branch 0==0, 3`	
	0x8	`halt`	

Self-modifying code

- m_0 as below;
- $\mathrm{pc}_0 = 2$;
- δ: We already applied it to the memory when needed but here are the rest:
  - 0 ↦ `branch [0]!=0, 4`
  - 1 ↦ `branch 0==0, 8`

	Addr	**Initial Content**	
	0x0	0	`;; input (var)`
	0x1	1	`;; initialized data`
	0x2	`syscall read 0`	`;; get initial value`
	0x3	`load [1], 6`	`;; rewrite code ahead`
	0x4	`load [0], 1`	`;; overwrite [1] with [0]`
	0x5	`load [0]-1, [0]`	`;; decrement [0]`
	0x6	`branch [0]!=0, 4`	`;; if not zero loop to 4`
⇒	0x7	`branch 0==0, 3`	`;; jump to 3`
	0x8	`halt`	

Self-modifying code

- m_0 as below;
- $\mathrm{pc}_0 = 2$;
- δ: We already applied it to the memory when needed but here are the rest:
  - 0 ↦ `branch [0]!=0, 4`
  - 1 ↦ `branch 0==0, 8`

	Addr	Initial Content	
	0x0	`0`	`;; input (var)`
	0x1	`1`	`;; initialized data`
	0x2	`syscall read 0`	`;; get initial value`
⇒	0x3	`load [1], 6`	`;; rewrite code ahead`
	0x4	`load [0], 1`	`;; overwrite [1] with [0]`
	0x5	`load [0]-1, [0]`	`;; decrement [0]`
	0x6	`branch [0]!=0, 4`	`;; if not zero loop to 4`
	0x7	`branch 0==0, 3`	`;; jump to 3`
	0x8	`halt`	

Self-modifying code

- m_0 as below;
- $\mathrm{pc}_0 = 2$;
- δ: We already applied it to the memory when needed but here are the rest:
  - 0 ↦ `branch [0]!=0, 4`
  - 1 ↦ `branch 0==0, 8`

	Addr	**Initial Content**	
	0x0	`0`	`;; input (var)`
	0x1	`1`	`;; initialized data`
	0x2	`syscall read 0`	`;; get initial value`
⇒	0x3	`load [1], 6`	`;; rewrite code ahead`
	0x4	`load [0], 1`	`;; overwrite [1] with [0]`
	0x5	`load [0]-1, [0]`	`;; decrement [0]`
	0x6	`branch 0==0, 8`	`;; jump to 8`
	0x7	`branch 0==0, 3`	`;; jump to 3`
	0x8	`halt`	

Self-modifying code

- m_0 as below;
- $\mathrm{pc}_0 = 2$;
- δ: We already applied it to the memory when needed but here are the rest:
  - 0 ↦ `branch [0]!=0, 4`
  - 1 ↦ `branch 0==0, 8`

	Addr	Initial Content	
	0x0	`0`	`;; input (var)`
	0x1	`1`	`;; initialized data`
	0x2	`syscall read 0`	`;; get initial value`
	0x3	`load [1], 6`	`;; rewrite code ahead`
⇒	0x4	`load [0], 1`	`;; overwrite [1] with [0]`
	0x5	`load [0]-1, [0]`	`;; decrement [0]`
	0x6	`branch 0==0, 8`	`;; jump to 8`
	0x7	`branch 0==0, 3`	`;; jump to 3`
	0x8	`halt`	

Self-modifying code

- m_0 as below;
- $\mathrm{pc}_0 = 2$;
- δ: We already applied it to the memory when needed but here are the rest:
  - 0 ↦ `branch [0]!=0, 4`
  - 1 ↦ `branch 0==0, 8`

	Addr	Initial Content	
	0x0	`0`	`;; input (var)`
	0x1	`0`	`;; initialized data`
	0x2	`syscall read 0`	`;; get initial value`
	0x3	`load [1], 6`	`;; rewrite code ahead`
⇒	0x4	`load [0], 1`	`;; overwrite [1] with [0]`
	0x5	`load [0]-1, [0]`	`;; decrement [0]`
	0x6	`branch 0==0, 8`	`;; jump to 8`
	0x7	`branch 0==0, 3`	`;; jump to 3`
	0x8	`halt`	

Self-modifying code

- m_0 as below;
- $\mathrm{pc}_0 = 2$;
- δ: We already applied it to the memory when needed but here are the rest:
  - $0 \mapsto$ `branch [0]!=0, 4`
  - $1 \mapsto$ `branch 0==0, 8`

	Addr	Initial Content	
	0x0	`0`	`;; input (var)`
	0x1	`0`	`;; initialized data`
	0x2	`syscall read 0`	`;; get initial value`
	0x3	`load [1], 6`	`;; rewrite code ahead`
	0x4	`load [0], 1`	`;; overwrite [1] with [0]`
⇒	0x5	`load [0]-1, [0]`	`;; decrement [0]`
	0x6	`branch 0==0, 8`	`;; jump to 8`
	0x7	`branch 0==0, 3`	`;; jump to 3`
	0x8	`halt`	

Self-modifying code

- m_0 as below;
- $\mathrm{pc}_0 = 2$;
- δ: We already applied it to the memory when needed but here are the rest:
  - $0 \mapsto$ `branch [0]!=0, 4`
  - $1 \mapsto$ `branch 0==0, 8`

	Addr	Initial Content	
	0x0	`-1`	`;; input (var)`
	0x1	`0`	`;; initialized data`
	0x2	`syscall read 0`	`;; get initial value`
	0x3	`load [1], 6`	`;; rewrite code ahead`
	0x4	`load [0], 1`	`;; overwrite [1] with [0]`
⇒	0x5	`load [0]-1, [0]`	`;; decrement [0]`
	0x6	`branch 0==0, 8`	`;; jump to 8`
	0x7	`branch 0==0, 3`	`;; jump to 3`
	0x8	`halt`	

Self-modifying code

- m_0 as below;
- $\mathrm{pc}_0 = 2$;
- δ: We already applied it to the memory when needed but here are the rest:
  - 0 ↦ `branch [0]!=0, 4`
  - 1 ↦ `branch 0==0, 8`

	Addr	Initial Content	
	0x0	`-1`	`;; input (var)`
	0x1	`0`	`;; initialized data`
	0x2	`syscall read 0`	`;; get initial value`
	0x3	`load [1], 6`	`;; rewrite code ahead`
	0x4	`load [0], 1`	`;; overwrite [1] with [0]`
	0x5	`load [0]-1, [0]`	`;; decrement [0]`
⇒	0x6	`branch 0==0, 8`	`;; jump to 8`
	0x7	`branch 0==0, 3`	`;; jump to 3`
	0x8	`halt`	

Self-modifying code

- m_0 as below;
- $\mathrm{pc}_0 = 2$;
- δ: We already applied it to the memory when needed but here are the rest:
  - 0 ↦ `branch [0]!=0, 4`
  - 1 ↦ `branch 0==0, 8`

	Addr	Initial Content	
	0x0	`-1`	`;; input (var)`
	0x1	`0`	`;; initialized data`
	0x2	`syscall read 0`	`;; get initial value`
	0x3	`load [1], 6`	`;; rewrite code ahead`
	0x4	`load [0], 1`	`;; overwrite [1] with [0]`
	0x5	`load [0]-1, [0]`	`;; decrement [0]`
	0x6	`branch 0==0, 8`	`;; jump to 8`
	0x7	`branch 0==0, 3`	`;; jump to 3`
⇒	0x8	`halt`	

Variable Size Instructions

A few real-world assembly languages have variable size instructions. This property is sometimes used to hide part of a program with a technique called "**instruction overlapping**". This property can be easily added to our model as follow.

Instructions

- $\mathbb{I}$: A (finite) set of instructions;
- '`load value, addr`': Load the evaluation of '`value`' at '`addr`' in memory
  Encoded in two memory cells, first for '`load value`' and second for '`address`';
- '`branch cond, addr`': Jump to '`addr`' if the expression '`cond`' is zero
  Encoded in two memory cells, first for '`branch cond`' and second for '`address`';
- '`halt`': Stop program execution. Encoded in one memory cell as before;

Operational Semantics

- $\mathbb{I} : \mathbb{M} \times \mathbb{A} \mapsto \mathbb{M} \times \mathbb{A}$ where $i \in \mathbb{I}$, $i(m,\mathrm{pc}) = (m',\mathrm{pc}')$;
- $[\![$`load value, addr`$]\!] =$ `([addr]:=value, pc':=pc+2)`
- $[\![$`branch cond, addr`$]\!] =$
  `([0]:=[0], if cond==0 then pc':=addr else pc':=pc+2)`
- $[\![$`halt`$]\!] =$ `([0]:=[0], pc':=pc)`

Overview

Control-Flow Recovery

- Control-flow recovery is prior to any other work because it aims at **recovering the semantics** of the program.
- The point is to **gather all the possible execution paths** of the binary program **for all possible inputs**.
- Because of dynamic jumps and self-modifying code, the gathering of all the possible runs **requires to perform data-analysis on a partial semantics of the program**.
- Most of the analysis techniques work only with the complete semantics of the program (**Chicken and Egg Problem**).
- Thus, **we need to come with new techniques. . .**

Types of Control-Flow Recovery

Correctness

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

Techniques

Syntax-based Recovery

- Linear Sweep
- Recursive Traversal

Semantics-based Recovery

- Concrete Execution
- Symbolic Execution

Undecidability of the General Problem

Theorem

Recovering the control-flow of a binary program is **undecidable** (for the general case).

Sketch of Proof

1. Lets, first, assume that the model we presented is equivalent to a Turing machine.
2. Recovering all the run would requires to collect all the possible values of pc.
3. Because of self-modifying code, the values pointed by the pc must also be recovered (which means that we need to track strictly more than one variable).
4. Thus, we can reduce any accessibility problem for a given program to a control-flow recovery problem by adding to the original program a conditional jump to an error state. And try to see if this extra program state is in the program control-flow.
5. Finally, as the accessibility problem is undecidable, the control-flow recovery problem is also undecidable for the general case.

Syntax-based: Linear Sweep

Linear Sweep

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

Syntax-based: Linear Sweep

Linear Sweep

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

Is it adding and missing execution paths?

Syntax-based: Linear Sweep

Linear Sweep

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

Is it adding and missing execution paths?

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
```

Syntax-based: Linear Sweep

Linear Sweep

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

Is it adding and missing execution paths?

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
```

Syntax-based: Linear Sweep

Linear Sweep

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

Is it adding and missing execution paths?

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
```

Syntax-based: Linear Sweep

Linear Sweep

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

Is it adding and missing execution paths?

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
```

Syntax-based: Linear Sweep

Linear Sweep

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

Is it adding and missing execution paths?

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
```

Syntax-based: Linear Sweep

Linear Sweep

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

Is it adding and missing execution paths?

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
```

Syntax-based: Linear Sweep

Linear Sweep

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

Is it adding and missing execution paths?

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, it is adding and missing execution paths!

Syntax-based: Linear Sweep

Linear Sweep

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

Incorrect

Is it adding and missing execution paths?

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, it is adding and missing execution paths!

Syntax-based: 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.

Syntax-based: 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.

What does it add to linear sweep?

Syntax-based: 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.

What does it add to linear sweep?

Lets disassemble this piece of binary code:

```
0804846c: e882feffff call 0x08048c00          08048c00: 83c00010 add eax, 0x1000
08048471: a16e840408 mov eax, [0x804846e]   08048c03: c3       ret
08048476: 83c00a     add eax, 0xa
...
```

Syntax-based: 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.

What does it add to linear sweep?

Lets disassemble this piece of binary code:

```
0804846c: e882feffff call 0x08048c00       08048c00: 83c00010 add eax, 0x1000
08048471: a16e840408 mov eax, [0x804846e]  08048c03: c3       ret
08048476: 83c00a     add eax, 0xa
...
```

```
0804846c: e882feffff  call  0x08048c00
```

Syntax-based: 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.

What does it add to linear sweep?

Lets disassemble this piece of binary code:

```
0804846c: e882feffff call 0x08048c00        08048c00: 83c00010 add eax, 0x1000
08048471: a16e840408 mov eax, [0x804846e]   08048c03: c3       ret
08048476: 83c00a     add eax, 0xa
...
```

```
0804846c: e882feffff  call  0x08048c00
08048c00: 83c00010    add eax, 0x1000
```

Syntax-based: 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.

What does it add to linear sweep?

Lets disassemble this piece of binary code:

```
0804846c: e882feffff call 0x08048c00      08048c00: 83c00010 add eax, 0x1000
08048471: a16e840408 mov eax, [0x804846e]  08048c03: c3       ret
08048476: 83c00a     add eax, 0xa
...
```

```
0804846c: e882feffff  call  0x08048c00
08048c00: 83c00010    add eax, 0x1000
08048c03: c3          ret
```

Syntax-based: 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.

What does it add to linear sweep?

Lets disassemble this piece of binary code:

```
0804846c: e882feffff call 0x08048c00       08048c00: 83c00010 add eax, 0x1000
08048471: a16e840408 mov eax, [0x804846e]  08048c03: c3       ret
08048476: 83c00a     add eax, 0xa
...
```

```
0804846c: e882feffff  call  0x08048c00
08048c00: 83c00010    add eax, 0x1000
08048c03: c3          ret
08048471: a16e840408  mov  eax, [0x804846e]
```

Syntax-based: 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.

What does it add to linear sweep?

Lets disassemble this piece of binary code:

```
0804846c: e882feffff call 0x08048c00        08048c00: 83c00010 add eax, 0x1000
08048471: a16e840408 mov eax, [0x804846e]   08048c03: c3       ret
08048476: 83c00a     add eax, 0xa
...
```

```
0804846c: e882feffff  call  0x08048c00
08048c00: 83c00010    add eax, 0x1000
08048c03: c3          ret
08048471: a16e840408  mov   eax, [0x804846e]
08048477: 83c00a      add   eax, 0xa
```

Syntax-based: 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.

What does it add to linear sweep?

Lets disassemble this piece of binary code:

```
0804846c: e882feffff call 0x08048c00        08048c00: 83c00010 add eax, 0x1000
08048471: a16e840408 mov eax, [0x804846e]   08048c03: c3       ret
08048476: 83c00a     add eax, 0xa
...
```

```
0804846c: e882feffff  call  0x08048c00
08048c00: 83c00010    add eax, 0x1000
08048c03: c3          ret
08048471: a16e840408  mov   eax, [0x804846e]
08048477: 83c00a      add    eax, 0xa ...
```

Syntax-based: 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.

What does it add to linear sweep?

Lets disassemble this piece of binary code:

```
0804846c: e882feffff call 0x08048c00        08048c00: 83c00010 add eax, 0x1000
08048471: a16e840408 mov eax, [0x804846e]   08048c03: c3       ret
08048476: 83c00a     add eax, 0xa
...
```

```
0804846c: e882feffff  call  0x08048c00
08048c00: 83c00010    add eax, 0x1000
08048c03: c3          ret
08048471: a16e840408  mov   eax, [0x804846e]
08048477: 83c00a      add    eax, 0xa ...
```

But, it is based on linear sweep, so. . .

Syntax-based: 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.

Incorrect

What does it add to linear sweep?

Lets disassemble this piece of binary code:

```
0804846c: e882feffff call 0x08048c00         08048c00: 83c00010 add eax, 0x1000
08048471: a16e840408 mov eax, [0x804846e]   08048c03: c3       ret
08048476: 83c00a     add eax, 0xa
...
```

```
0804846c: e882feffff  call  0x08048c00
08048c00: 83c00010    add eax, 0x1000
08048c03: c3          ret
08048471: a16e840408  mov   eax, [0x804846e]
08048477: 83c00a      add    eax, 0xa ...
```

But, it is based on linear sweep, so...

About Syntax-Based Disassemblers

What can we deduce from these examples?

Having partial knowledge of the semantics, will **always** lead to miss some behaviours and produce an incorrect control-flow.

About Syntax-Based Disassemblers

What can we deduce from these examples?

Having partial knowledge of the semantics, will **always** lead to miss some behaviours and produce an incorrect control-flow.

To be correct, a disassembler always need to know about the semantics of all the instructions!

Semantics-based: Concrete Execution

Concrete Execution

Given some chosen inputs, run the program several times and collect the traces. The collection of all the traces will give you the semantics of the program.

- Efficient and simple to settle down (by using Pin, for example).
- Quite fast for a run, even if you need to store all the traces.
- Can be automatized with random inputs (fuzzing).

But!

- There is, almost, no hope to reach full coverage of the program.
- Random input makes it very difficult to control the time needed to reach a good coverage.

Semantics-based: Concrete Execution

Concrete Execution

Given some chosen inputs, run the program several times and collect the traces. The collection of all the traces will give you the semantics of the program.

Under-approximation

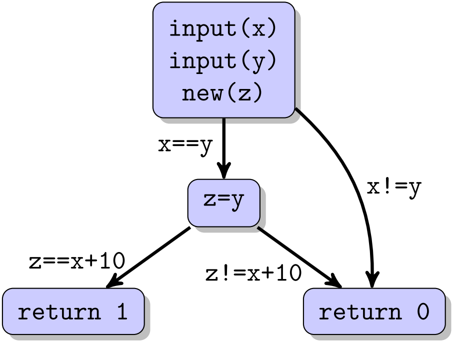
- Efficient and simple to settle down (by using Pin, for example).
- Quite fast for a run, even if you need to store all the traces.
- Can be automatized with random inputs (fuzzing).

But!

- There is, almost, no hope to reach full coverage of the program.
- Random input makes it very difficult to control the time needed to reach a good coverage.

Symbolic Execution

```
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 }
```



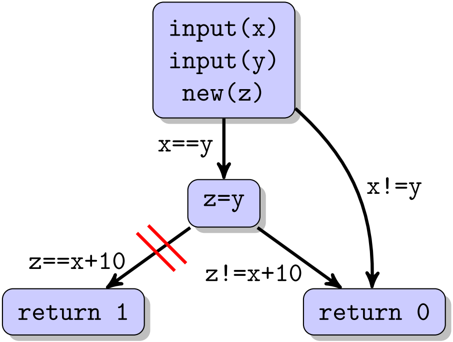
- **line 4**: $(x = y)$
- **line 8**: $(x = y) \wedge (y = x + 10)$ (**UNSAT**)
- **line 10** (path1): $(x \neq y)$
- **line 10** (path2): $(x = y) \wedge (y \neq x + 10)$

Algorithm (James King, 1976)

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

Symbolic Execution

```
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 }
```



input(x)
input(y)
new(z)
x==y
x!=y
z=y
z==x+10
z!=x+10
return 1
return 0

- **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 (James King, 1976)

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

Directed Automated Concrete Execution

Directed Automated Concrete Execution

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

- **Original idea (2005)**:
  DART (Directed Automated Random Testing) by Patrice Godefroid;
- **First applied to binary analysis (2008)**:
  Inside the OSMOSE software by CEA List.

Directed Automated Concrete Execution

Directed Automated Concrete Execution

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

Under-approximation

- **Original idea (2005)**:
  DART (Directed Automated Random Testing) by Patrice Godefroid;
- **First applied to binary analysis (2008)**:
  Inside the OSMOSE software by CEA List.

Full Symbolic Execution on Binary Code

Algorithm

1. Start at entry point;
2. Symbolically execute the current instruction;
3. If a dynamic jump or a test is encountered, run the SMT-solver on the conjunction of all previous paths and list possible outputs;
4. If the SMT-solver output an answer, follow the satisfiable paths and go to 2;
5. If the SMT-solver cannot answer, stop here.

A few limitations and challenges:

- Tool must be aware of the semantics of all the instructions;
- Context of the Operating System must be simulated;
- Under-approximation (efficiency depends upon the cleverness of SMT-solver);
- Loops are unfolded up to a certain limit to enforce termination;
- Detection of local context and scope helps to keep the formula small.

Full Symbolic Execution on Binary Code

Algorithm

1. Start at entry point;
2. Symbolically execute the current instruction;
3. If a dynamic jump or a test is encountered, run the SMT-solver on the conjunction of all previous paths and list possible outputs;
4. If the SMT-solver output an answer, follow the satisfiable paths and go to 2;
5. If the SMT-solver cannot answer, stop here.

Under-approximation

A few limitations and challenges:

- Tool must be aware of the semantics of all the instructions;
- Context of the Operating System must be simulated;
- Under-approximation (efficiency depends upon the cleverness of SMT-solver);
- Loops are unfolded up to a certain limit to enforce termination;
- Detection of local context and scope helps to keep the formula small.

Abstract Interpretation-Based 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

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

- Use a double abstract domain: CFG × 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 Value-Set Analysis.

Abstract Interpretation-Based 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

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

- Use a double abstract domain: CFG × 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, (strided) intervals or Value-Set Analysis.

Over-approximation

Control-Flow Recovery: Summary

Syntax-based Disassembler	Accuracy
Linear Sweep	**Incorrect**
Recursive Traversal	**Incorrect**

- All methods are just incorrect in all cases.

Semantics-Based Disassembler	Accuracy
Concrete Execution	**Under-approximation**
Directed Automated Concrete Execution	**Under-approximation**
Full Symbolic Execution	**Under-approximation**
Abstract Interpretation Recovery	**Over-approximation**

- **Symbolic Execution** and **Directed Automated Concrete Execution** are of the same kind and provide under-approximation. They are useful for reverse-engineering.
- **Abstract-Interpretation framework** are, most of the time, too imprecise.

Overview

1. Introducing to Binary Code Analysis
2. Why Is Binary Analysis Special?
3. Low-level Programs Formal Model
4. Control-flow Recovery
5. **Current and Future Trends**

Current and Future Trends

Current Trends

- Multiplication of tools and frameworks (reinventing the wheel).
- Clear split between academic and industry tools (complexity of use of academic tools is currently too high).
- Still some limitations to automatically recover control-flow of everyday-life binaries and to **scale**.

Future Trends

- A stable and flexible framework for binary analysis.
- Support for the main platforms (Windows, Linux, *BSD, MacOS).
- Deal with loops and variable size inputs in a more efficient way.

Questions?