Exploiting Software Vulnerabilities

Program Binary Analysis

Universidad Zaragoza

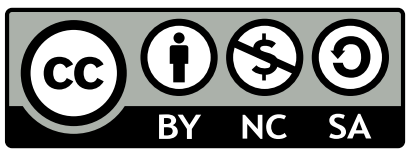
Dept. of Computer Science and Systems Engineering
University of Zaragoza, Spain

Course 2020/2021

Master's Degree in Informatics Engineering

UNIVERSITY OF ZARAGOZA

Google Meet



Outline

Universidad Zaragoza

Outline

Universidad Zaragoza

Introduction

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("hello world!\n");
    return 0;
}
```

```
push    ebp
mov     ebp, esp
and     esp, -16
sub     esp, 16
call    ___main
mov     DWORD PTR [esp], OFFSET FLAT:LC0
call    _puts
mov     eax, 0
leave
ret
```

- **Programs are written in text**
  - Both source code and assembly!
  - **Sequences of characters (bytes)**
  - Hard to work with (for humans, not for machines)
  - We need some **structured representation**

Universidad Zaragoza

Introduction

Program Analysis

Automatically reasoning and deriving properties about the behavior of computer programs

Approaches

- **Static Program Analysis**
  - Without executing the program
  - Abstract model of the program is obtained and (symbolically) executed
  - Analysis done through the abstract model
  - **Examples**: CFA, DFA, concolic execution, ...
- **Dynamic Program Analysis**
  - Executing the program on chosen inputs
  - Traces are collected and then analyzed
  - Analysis done through these concrete executions
  - **Examples**: software testing, taint analysis, ...

Universidad Zaragoza 1542

Introduction

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 and human-readable information about the program (variables, types, functions, etc.)
- **Bytecode**: may vary depending on the bytecode considered, but keep track of fewhigh-level information about the program such as types and functions. Programs are unstructured, through
- **Binary file**: only keep track of the instructions in an unstructured way (no for-loop, no clear argument passing in procedures, etc). No type, no naming. Binary file may enclose meta-data that might be helpful for analysis (symbols, debug, etc.)
- **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!

Introduction

Binary code vs. source code

What you code is not what you execute!

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

Universidad Zaragoza

Introduction

Motivations

Why analyze binary programs?

- 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 other “hostile” programs)
- Technology forecasting
- Mistrust in the Compilation Chain
- C compiler possibly buggy
- Checking low-level bugs (e.g., exploitability of a stack buffer-overflow)
- Bugs with a strong interconnection with hardware

Universidad Zaragoza 1542

Introduction

Understanding papers on Program Analysis

> *For those who keep track of such things, checkers in the research system typically traverse program paths (flow-sensitive) in a forward direction, going across function calls (inter-procedural) while keeping track of call-site-specific information (context-sensitive) and toward the end of the effort had some of the support needed to detect when a path was infeasible (path-sensitive).*

Note these terms

- **Flow-(in)sensitive**
- **Inter-(intro)procedural**
- **Context-(in)sensitive**
- **Path-(in)sensitive**

Further reading: *A few billion lines of code later: using static analysis to find bugs in the real world.* Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, Dawson Engler. *Communications of the ACM*, vol. 53, iss. 2, pp. 66-75 (February 2010). doi: 10.1145/1646353.1646374

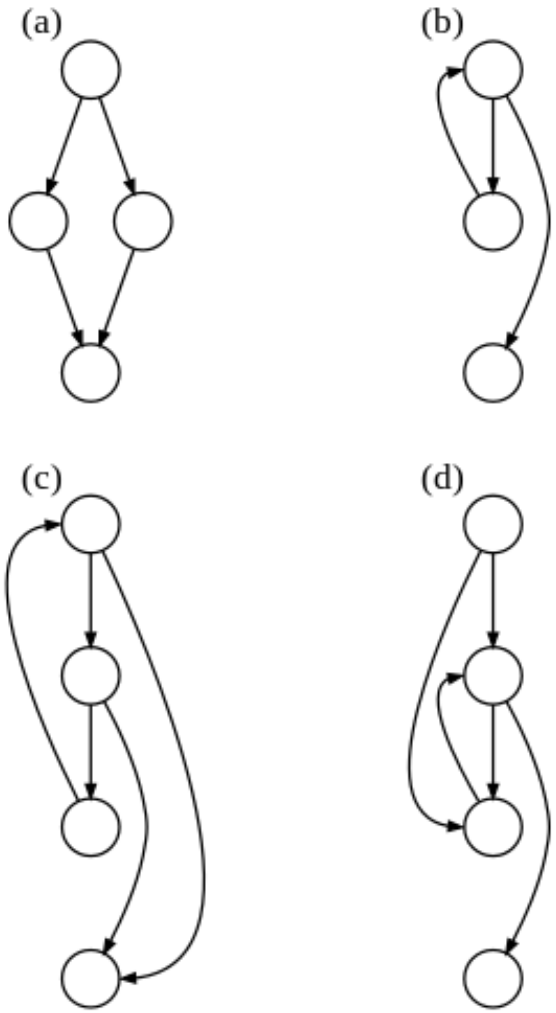
Universidad Zaragoza 1542

Outline

Universidad Zaragoza

Static Analysis Techniques

Control-Flow Graphs



- **Flow of control inside a function**
- **Nodes: basic blocks**
  - Sequence of consecutive program instructions having an entry point (first instruction executed) and one exit point (last instruction executed)
  - Entry and exit blocks
- **Edge**: control flows from A to B

Applications

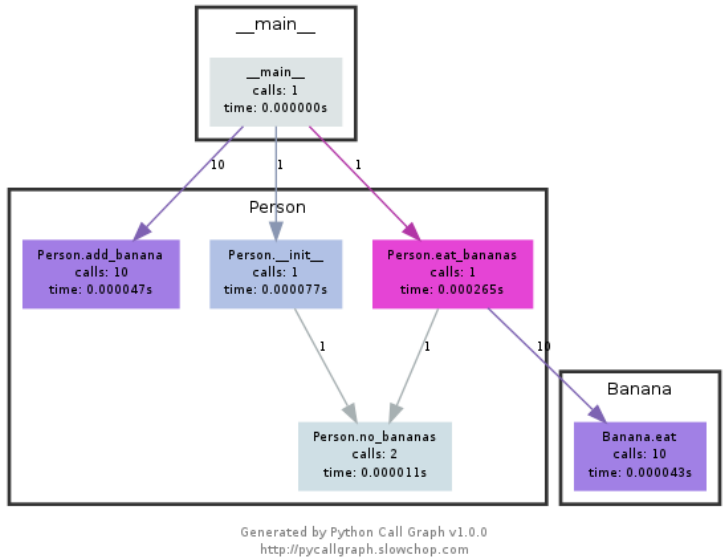
- Compiler optimizations
- Data-flow analysis (taint analysis)
- Behavioral-based monitors

Credits: `https://en.wikipedia.org/wiki/Control_flow_graph`

Universidad Zaragoza

Static Analysis Techniques

Call Graphs

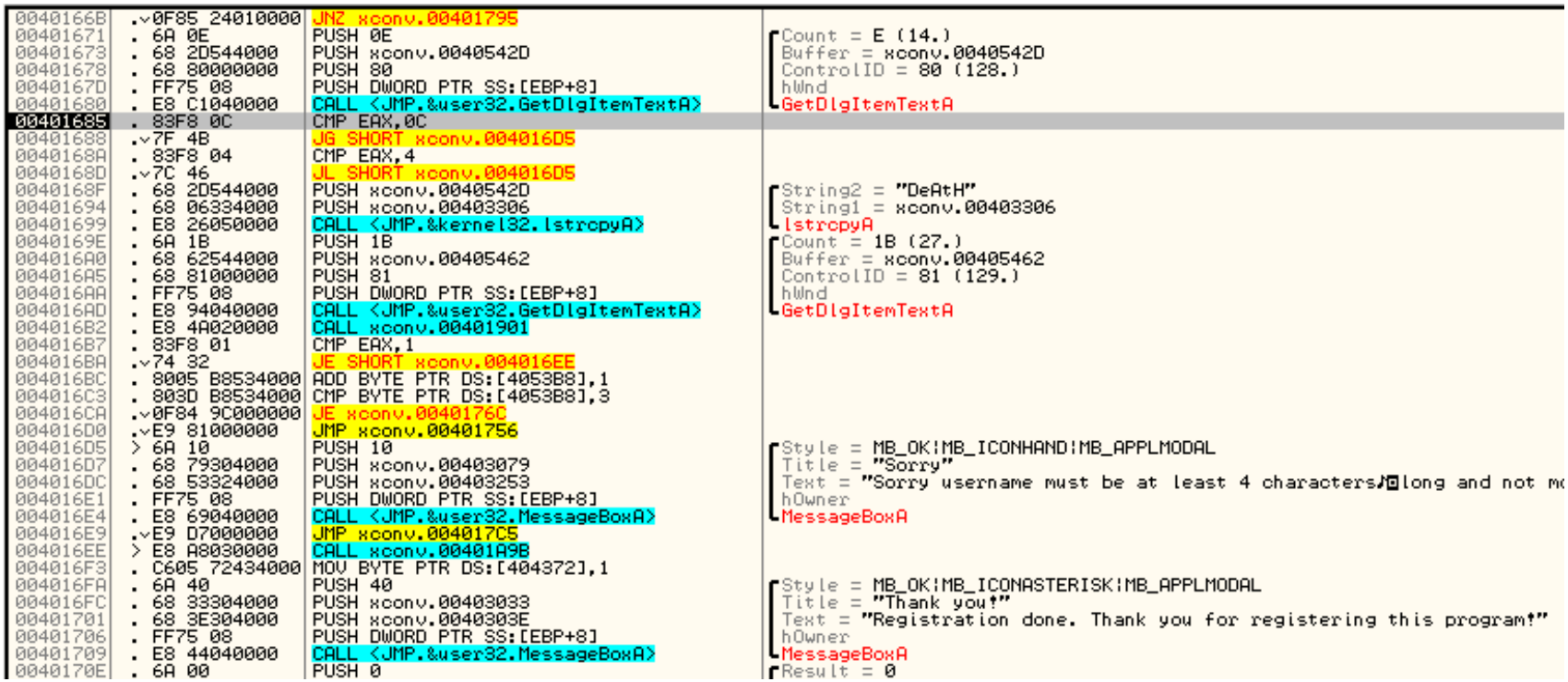


- **Interprocedural CFG**. **Information flow between functions**
- **Nodes**: functions
- **Edge**: A might call B
- Types: static, dynamic (record of a program execution)
- Application: find never-called procedures
- **Tools available for automatic call-graph generation**

Credits: `https://en.wikipedia.org/wiki/Call_graph`

Static Analysis Techniques

Disassembling



- Roughly speaking, **read** `PUSH EAX` **instead of** `0x50`
- **Lot of tools** see `https://en.wikibooks.org/wiki/X86_Disassembly/Disassemblers_and_Decompilers`
  - Win32Dasm
  - OllyDBG (also debugger)
  - IDA Pro (also debugger)
  - r2 (also debugger)

Universidad Zaragoza

Static Analysis Techniques

Disassembling

Main challenges

- **Variable length instruction sets**: overlapping instructions
- **Data and code mixed**: misclassify data as instructions
- **Indirect jumps**: Any location could be the start of an instruction!
- **Start of functions**: when calls are indirect
- **End of functions**: when no dedicated return instruction exists
  - Handwritten assembly code may not conform the standard calling conventions
- **Code compression**: code of two functions overlaps
- **Self-modifying code**

Universidad Zaragoza

Static Analysis Techniques

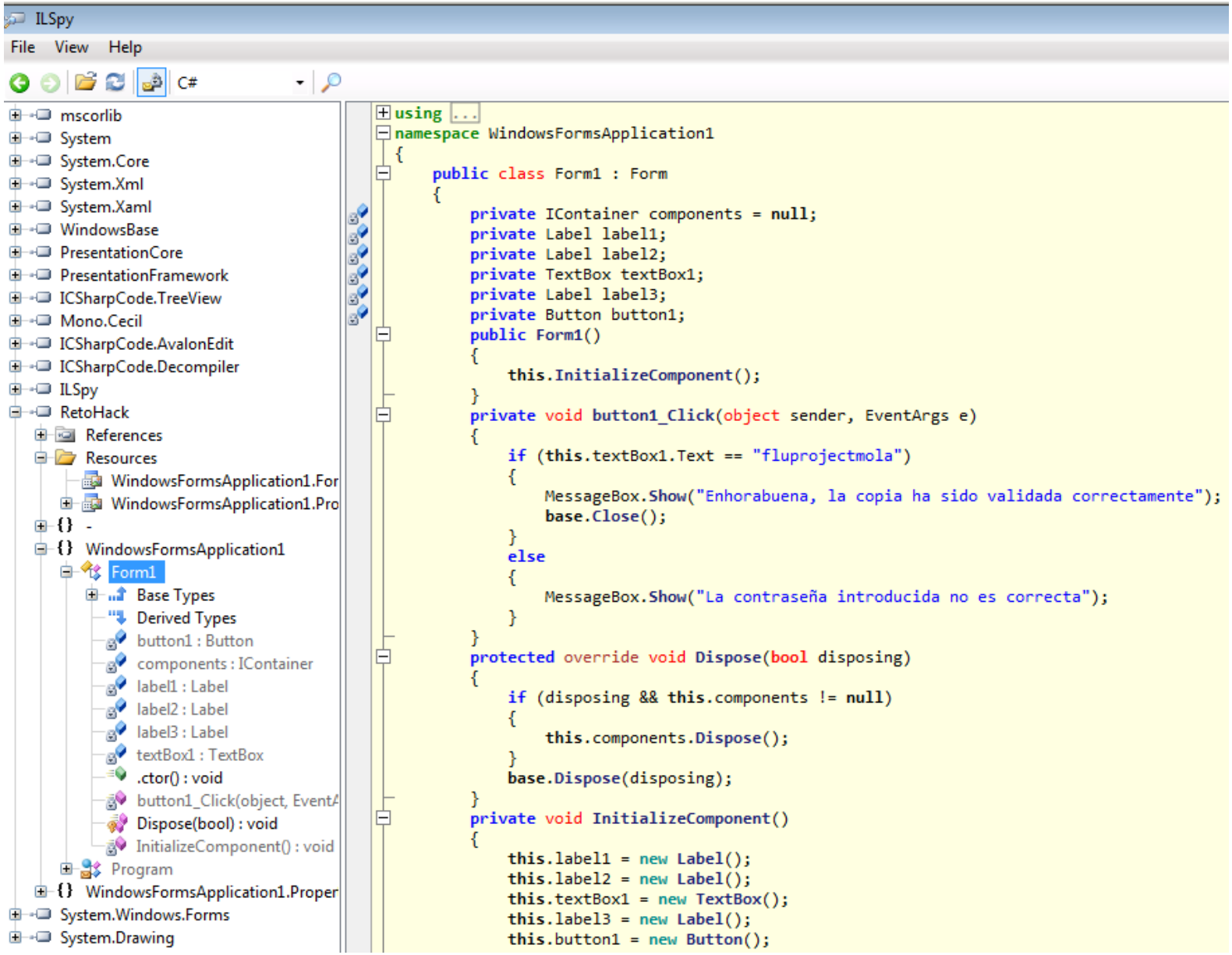
Decompilation – example

```
int __stdcall sub_40162C(HWND hDlg, int a2, int a3, int a4){
  HICON v4; // eax@2
  UINT v5; // eax@5

  switch ( a2 )  {
    case 272:
      v4 = LoadIconA(hInstance, (LPCSTR)0x64);
      SendMessageA(hDlg, 0x80u, 1u, (LPARAM)v4);
      break;
    case 273:
      if ( a3 == 126 ) {
        v5 = GetDlgItemTextA(hDlg, 128, dword_40542D, 14);
        if ( (signed int)v5 > 12 || (signed int)v5 < 4 ) {
          MessageBoxA(hDlg,"Sorry username must be at least 4
characters\r\nlong
and not more than 12 characters.", "Sorry", 0x10u);
        } else {
          lstrcpyA(dword_403306, dword_40542D);
          GetDlgItemTextA(hDlg, 129, byte_405462, 27);
          if ( sub_401901() == 1 ) {
            sub_401A9B();
            byte_404372 = 1;
            MessageBoxA(hDlg, "Registration done. Thank you for registering
this
program!", "Thank you!", 0x40u);
            EndDialog(hDlg, 0);
            EnableWindow(dword_403363, 0);
            SetWindowTextA(
              dword_4054A7,
              "X-Convertor v1.0  2005 by TDC and BoR0\r\n\n
Coded by\t: TDC and BoR0\r\nVersion\t\t: 1.0\r\nRelease
date\t: 18-08-2005\r\n \r\nX-Convertor converts up to 4KB
each convert.\r\n \r\nRegistered version. Thank you.\r\n");
            lstrcatA(byte_403330, dword_403306);
            SetWindowTextA(dword_4054AB, byte_403330);
          }
        else {
          ++byte_4053B8;
          if ( byte_4053B8 == 3 ) {
            MessageBoxA(hDlg, "Your serial is not correct",
                        "Sorry", 0x10u);
            byte_4053B8 = 0;
            EndDialog(hDlg, 0);
          } else {
            MessageBoxA(hDlg, "Your serial is not correct",
                        "Sorry", 0x10u);
          }
        }
      } else {
        if ( a3 == 127 ) {
          byte_4053B8 = 0;
          EndDialog(hDlg, 0);
        }
      }
      break;
    case 16:
      byte_4053B8 = 0;
      EndDialog(hDlg, 0);
      break;
  }
  return 0;
}
```

Static Analysis Techniques

Decompilation



Universidad Zaragoza

Static Analysis Techniques

Decompilation

Main challenges

- **Disassembly**: **first step of any decompiler!**
- **Target language**: assembly code may not correspond to any legal source code
- **Library functions**
- **Instruction compiler-dependent equivalences**
  - `int a= 0` → `mov eax, [a]; xor eax, eax`
- **Artefacts of the target architecture**: unnecessary jumps-to-jumps
- **Structured control-flow**
- **Compiler optimizations**: loop unrolling, shifts, adds, ...
- **Loads/stores**: operations on arrays, records, pointers, and objects
- **Self-modification code**: normally, segment code shall be unchanged, although there exist programs that modify themselves!

Universidad Zaragoza 1542

Static Analysis Techniques

Data Flow Analysis

- Analyze the **effect of each basic block**
- Compose effects of basic blocks to derive information at basic block boundaries
- Framework to **provide facts about programs**. Based on all paths through program (including also infeasible paths)
- **Derives information about the dynamic behavior of a program by only examining the static code**

Useful for...

- **Program debugging**: which definitions (of variables) may reach a program point?
- **Program optimizations**: constant folding, copy propagation, common sub-expression elimination, etc.

Universidad Zaragoza
1542

Static Analysis Techniques

Data Flow Analysis

Consider the statement $a = b + c$

Statement effects

- **Uses variables** (b, c)
- **"Kills" a previous definition** (previous value of a)
- **New definition** (a)

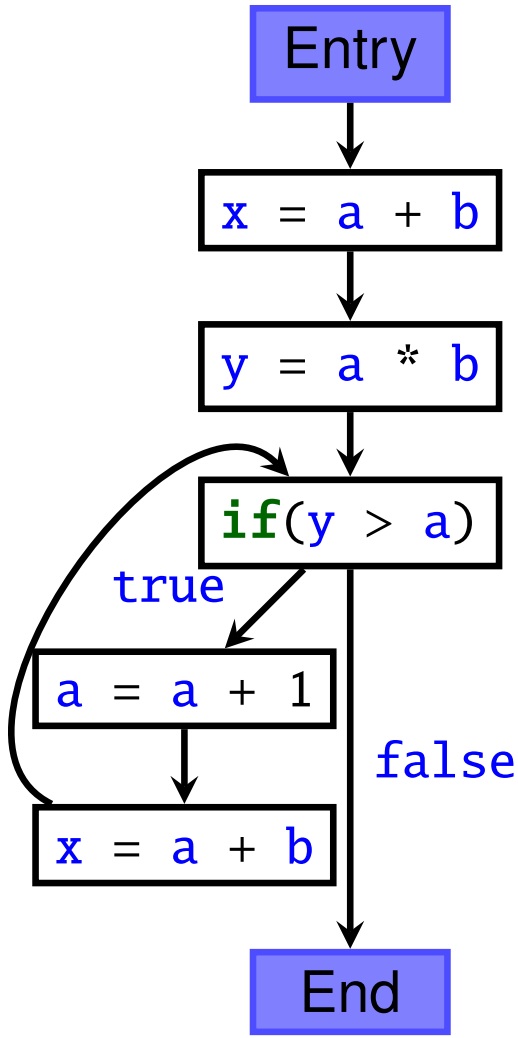
- **Compose effect of statements → effect of a basic block**
  - *Locally exposed use*: use of a data item which is not preceded in the basic block by a definition of the data item
  - Any definition of a data item kills all definitions of the same data item reaching the basic block
  - *Locally available definition*: last definition of data item in basic block

Universidad Zaragoza

Static Analysis Techniques

Data Flow Analysis

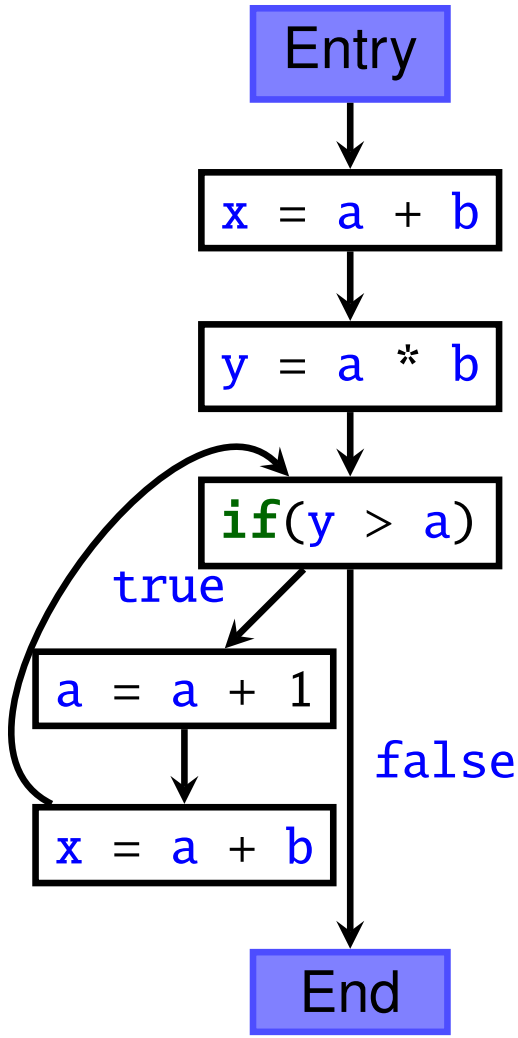
- **Facts**
  - $a + b$ is available
  - $a * b$ is available
  - $a + 1$ is available
- Let's compute for each program point the facts that holds!



Static Analysis Techniques

Data Flow Analysis

Statement	Gen	Kill
`x = a + b`	$a + b$	
`y = a * b`	$a * b$	
`y > a`		
`a = a + 1`		$a + b$
		$a * b$
		$a + 1$



Static Analysis Techniques

Data Flow Analysis

- *Forward* vs. *backward*: data flow from in to out (vs. out to in)
- *Must* vs. *may*: at joint points, only keep the facts that hold on all paths (vs. any path) that are joined

	Must	**May**
Forward	Available expressions	Reaching definitions
Backward	Very busy expressions	Live variables

Limitations

- **Data-Flow Analysis is good at analyzing local variables**
  - **What about values stored in the heap?**
  - Not modeled in traditional data flow
- In general, **it is hard to analyze pointers**. Suppose `*x = p`
  - *Assume all data flow facts are killed*
  - Or assume that write through `x` may affect any variable whose address has been taken

Static Analysis Techniques

Symbolic Execution

- **Enables us to scale and model all possible program executions**
- Concrete vs. symbolic execution
  - **Testing works, but each test only explores one possible execution path**
- **Generalizes testing**
  - Allows unknown symbolic variables in evaluation
  - **Checks feasibility of program paths**

Main challenges

- **Path explosion**
- **Modeling statements and environments**
- **Constraint solving**

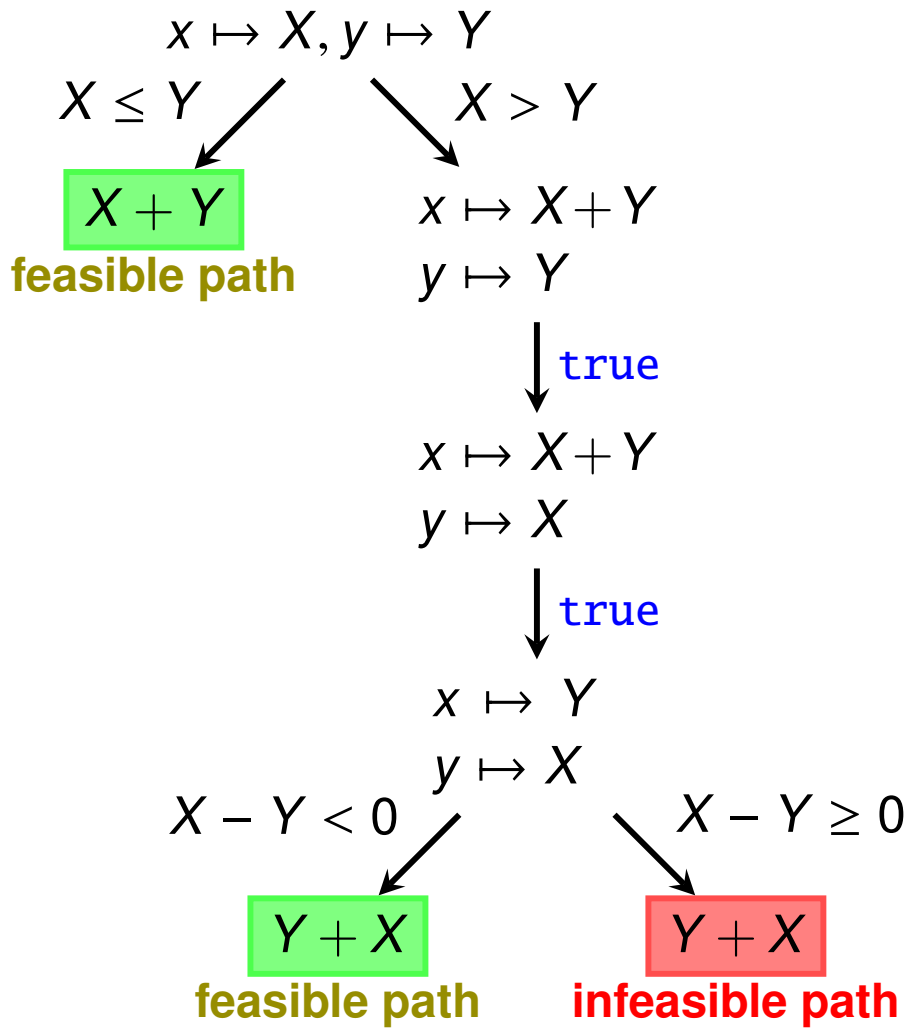
Further reading: Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. *A Survey of Symbolic Execution Techniques*. ACM Comput. Surv. 51, 3, Article 50 (July 2018), 39 pages. doi: 10.1145/3182657

Universidad Zaragoza

Static Analysis Techniques

Symbolic Execution

```
1  int f(int x, int y)
2  {
3      if(x > y)
4      {
5          x = x + y;
6          y = x - y;
7          x = x - y;
8          if(x - y > 0)
9              perror("Error!");
10     }
11
12     return x + y;
13 }
```



How to decide which branches are feasible?

Combine path condition with branch condition and ask a SMT solver!

Static Analysis Techniques

Symbolic Execution – example: bug finding

Spot the vulnerability! Which value triggers it?

```
1  int bar(int i)
2  {
3      int j = 2*i;
4      i++;
5      i = i*j;
6      if (i < 1)
7          i = -i;
8
9      i = j/i;
10     return i;
11 }
```

False branch condition $i = (i_{in} + 1)2i_{in}$

$(i_{in} + 1)2i_{in} \geq 1$

True branch condition $i = -(i_{in} + 1)2i_{in}$

$(i_{in} + 1)2i_{in} < 1$

Universidad Zaragoza

Static Analysis Techniques

Symbolic Execution – example: bug finding

Spot the vulnerability! Which value triggers it?

```
1  int bar(int i)
2  {
3      int j = 2*i;
4      i++;
5      i = i*j;
6      if (i < 1)
7          i = -i;
8
9      i = j/i;
10     return i;
11 }
```

False branch condition

$$i = (i_{in} + 1)2i_{in}$$
$$(i_{in} + 1)2i_{in} \geq 1$$

True branch condition

$$i = -(i_{in} + 1)2i_{in}$$
$$(i_{in} + 1)2i_{in} < 1$$

Division by zero arises problems...

Universidad Zaragoza

Static Analysis Techniques

Symbolic Execution – example: bug finding

Spot the vulnerability! Which value triggers it?

```
1  int bar(int i)
2  {
3      int j = 2*i;
4      i++;
5      i = i*j;
6      if (i < 1)
7          i = -i;
8
9      i = j/i;
10     return i;
11 }
```

False branch condition $i = (i_{in} + 1)2i_{in}$

$(i_{in} + 1)2i_{in} \geq 1$

True branch condition $i = -(i_{in} + 1)2i_{in}$

$(i_{in} + 1)2i_{in} < 1$

Division by zero arises problems...

False branch is always safe

$(i > 0, \forall i_{in} | (i_{in} + 1)2i_{in} \geq 1)$

What happens with the true branch?

Universidad Zaragoza

Static Analysis Techniques

Symbolic Execution – example: bug finding

Spot the vulnerability! Which value triggers it?

```
1  int bar(int i)
2  {
3      int j = 2*i;
4      i++;
5      i = i*j;
6      if (i < 1)
7          i = -i;
8
9      i = j/i;
10     return i;
11 }
```

False branch condition $i = (i_{in} + 1)2i_{in}$

$(i_{in} + 1)2i_{in} \geq 1$

True branch condition $i = -(i_{in} + 1)2i_{in}$

$(i_{in} + 1)2i_{in} < 1$

Division by zero arises problems...

False branch is always safe

$(i > 0, \forall i_{in} | (i_{in} + 1)2i_{in} \geq 1)$

What happens with the true branch?

$-(i_{in} + 1)2i_{in} = 0$

Universidad Zaragoza

Static Analysis Techniques

Symbolic Execution – example: bug finding

Spot the vulnerability! Which value triggers it?

```
1  int bar(int i)
2  {
3      int j = 2*i;
4      i++;
5      i = i*j;
6      if (i < 1)
7          i = -i;
8
9      i = j/i;
10     return i;
11 }
```

False branch condition $i = (i_{in} + 1)2i_{in}$

$(i_{in} + 1)2i_{in} \geq 1$

True branch condition $i = -(i_{in} + 1)2i_{in}$

$(i_{in} + 1)2i_{in} < 1$

Division by zero arises problems...

False branch is always safe

$(i > 0, \forall i_{in} | (i_{in} + 1)2i_{in} \geq 1)$

What happens with the true branch?

$-(i_{in} + 1)2i_{in} = 0 \rightarrow i_{in} = -1, i_{in} = 0$

Universidad Zaragoza

Outline

Universidad Zaragoza

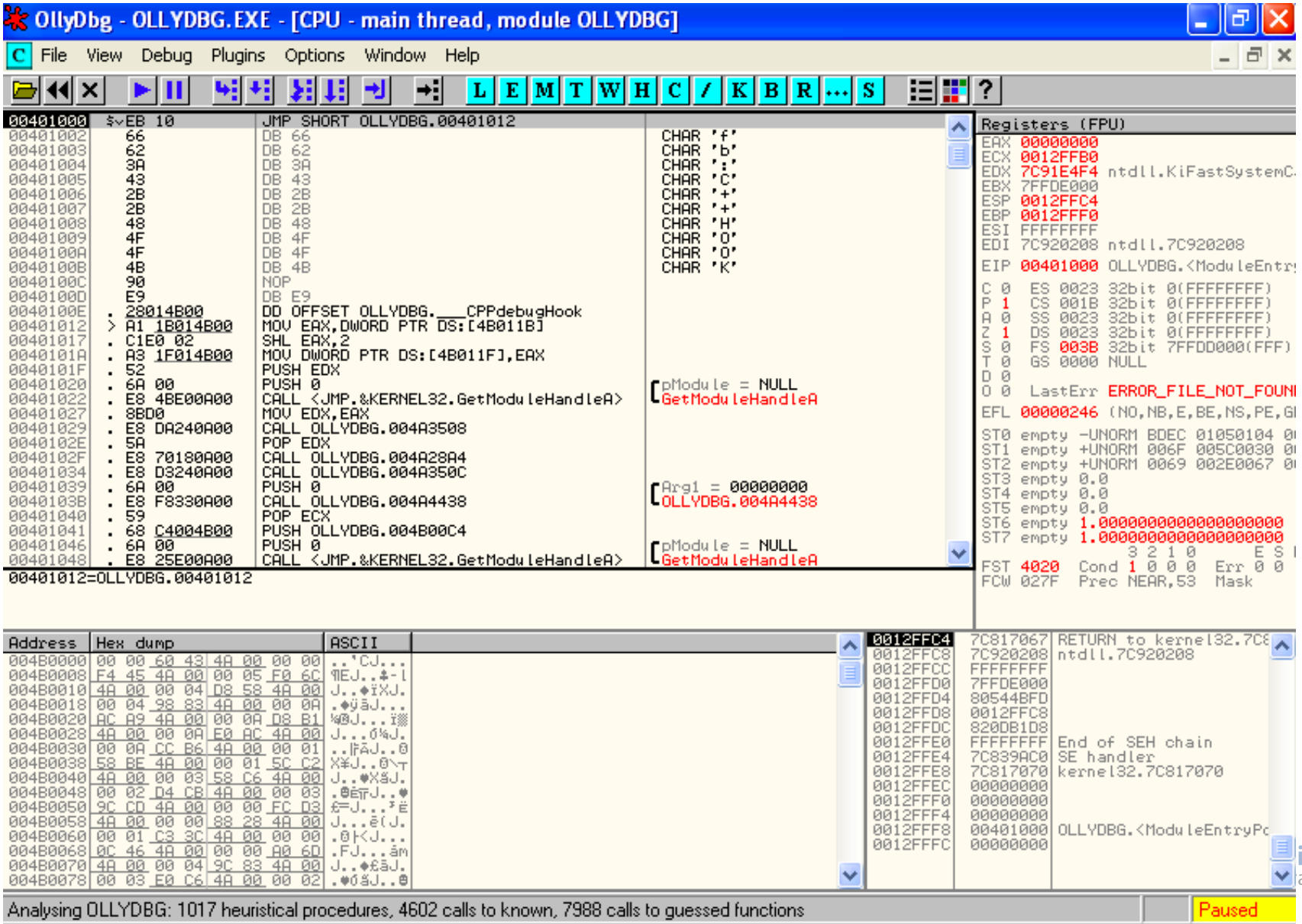
Dynamic Analysis Techniques
Debugging

- **Execute the program instructions with special software**: *debuggers*
  - We can see the values of every CPU register, the stack, the memory, etc.
- Source code vs. binary debugging
- **Breakpoints**: stops the execution when it is reached
  - Software (memory) breakpoints
  - Hardware breakpoints
  - On execution, reading, or writing operations
- Step into / step onto

Universidad Zaragoza

Dynamic Analysis Techniques

Debugging (example: OllyDBG)



Dynamic Analysis Techniques
Fuzzing

Roughly speaking, “fuzzing means. . . ” (quoting Iñaki Rodríguez-Gastón)

Dynamic Analysis Techniques
Fuzzing

Roughly speaking, “fuzzing means. . . ” (quoting Iñaki Rodríguez-Gastón)

- **Black-box approach** (at the beginning): no previous knowledge of program internals
  - **Evolved to white-box approach**: state-of-the-art fuzzers “learn” from program behavior
- **Many anomalous (unexpected, invalid, or random data) inputs are given to the application**
- **Application is monitored for any sign of error**
  - Unexpected behavior
  - Crashes
    - Buffer overflow
    - Integer overflow
    - Memory corruption errors
    - Format string bugs

Universidad Zaragoza
1542

Dynamic Analysis Techniques
Fuzzing

Charlie Miller's "five lines of Python" dumb fuzzer

- **Found vulnerabilities in PDF readers and MS Powerpoint**

```
numwrites = random.randrange(math.ceil((float(len (buf)) / FuzzFactor))) + 1
for j in range (numwrites):
      rbyte = random.randrange(256)
      rn = random.randrange(len(buf))
      buf[rn] = "%c"%(rbyte);
```

Universidad Zaragoza

Dynamic Analysis Techniques

Fuzz Testing

A simple example: HTTP GET requests

- A standard HTTP GET request: `GET /index.html HTTP/1.1`
- **Anomalous requests**
  - `AAAAAA...AAAA /index.html HTTP/1.1`
  - `GET ///////index.html HTTP/1.1`
  - `GET %n%n%n%n%n%n.html HTTP/1.1`
  - `GET /AAAAAAAAAAAAA.html HTTP/1.1`
  - `GET /index.html HTTTTTTTTTTTTP/1.1`
  - `GET /index.html HTTP/1.1.1.1.1.1.1.1`
  - etc.

Types of fuzzers

- **Mutation-based fuzzing**
- **Generation-based fuzzing**

Universidad Zaragoza

Dynamic Analysis Techniques

Taint analysis

Measure what is the influence of the input data into the application

- Data come from tainted sources (any external input) and end in tainted sinks
- *Flow* from X to Y: **an operation that uses X to derive a value Y**
- **Tainted value: if the source of the value X is untrustworthy** (e.g., user-supplied string)

Taint Propagation

- **Object X tainted the object Y**
- **Taint operator** t: $X \mapsto t(Y)$
- **A taint operator is transitive**: $X \mapsto t(Y)$ and $Y \mapsto t(Z)$, then $X \mapsto t(Z)$

Dynamic Analysis Techniques

Taint analysis

Main challenges

- **Tainted addresses**
  - Distinguishing between memory addresses and cells is not always appropriate
  - Taint granularity is important (bit, byte, word, etc.)
- **Undertainting**
  - Dynamic taint analysis does not properly handle some types of information flow
- **Overtainting**
  - Deciding when to introduce taint is often easier than deciding when to remove taint
- **Time of detection vs. time of attack**
  - When used for attack detection, dynamic taint analysis may raise an alert too late

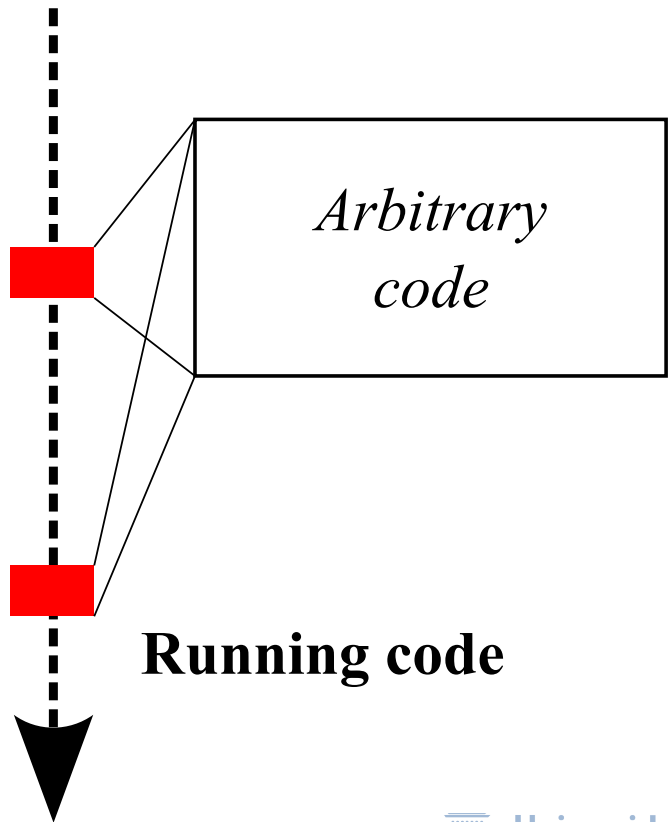
Universidad Zaragoza
1542

Dynamic Analysis Techniques

Dynamic Binary Instrumentation

Arbitrary code addition during the execution of a binary

- What do I insert? → **instrumentation function**
- Where? → **addition places**



Dynamic Analysis Techniques

Dynamic Binary Instrumentation

Arbitrary code addition during the execution of a binary

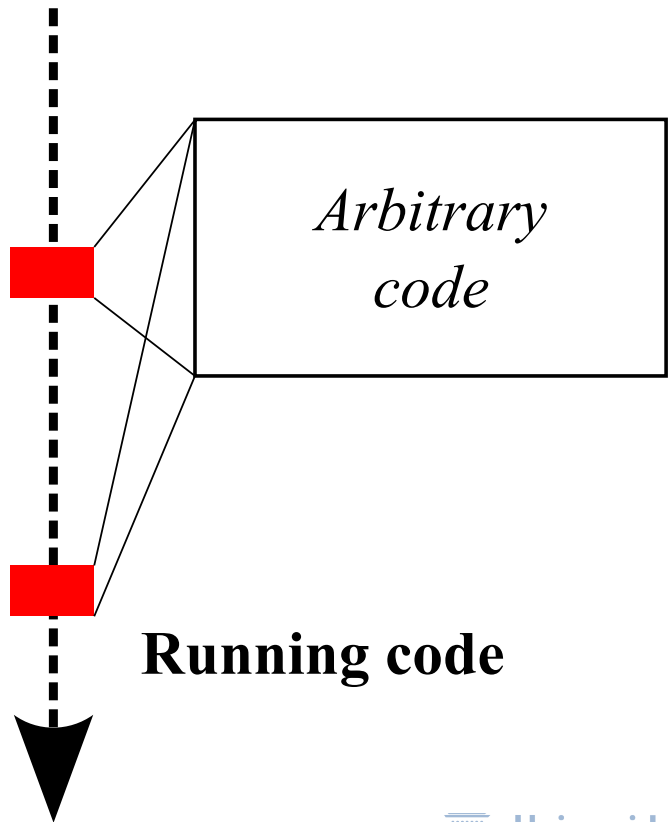
- What do I insert? → **instrumentation function**
- Where? → **addition places**

Advantages

- Independent of the programming language
- We can instrument proprietary software
- No need to recompile/relink each time
- Allows to instrument a process already in execution

Main disadvantage

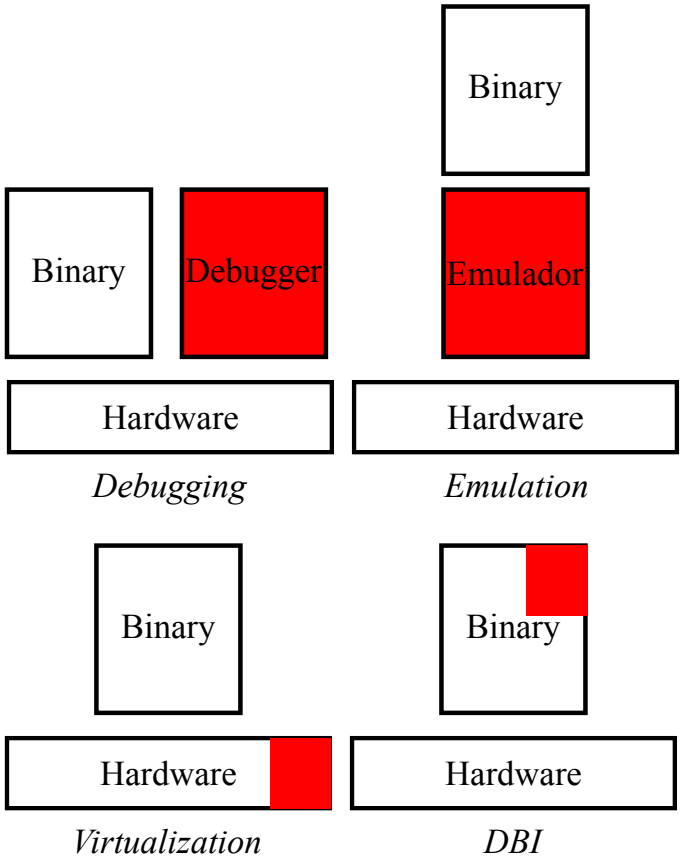
- **Overhead ⇒⇓ performance**



Universidad Zaragoza

Dynamic Analysis Techniques

Placing DBI in the context of dynamic analysis



- **Executable transformation**
- **Total control over execution**
- **No need of architectural support**

Credits: J-Y. Marion, D. Reynaud *Dynamic Binary Instrumentation for Deobfuscation and Unpacking*. *DeepSec*, 2009

Exploiting Software Vulnerabilities

Program Binary Analysis

Universidad Zaragoza

Dept. of Computer Science and Systems Engineering
University of Zaragoza, Spain

Course 2020/2021

Master's Degree in Informatics Engineering

UNIVERSITY OF ZARAGOZA

Google Meet