#CyberSBC2022

Advanced Malware Analysis Techniques

Cybersecurity Summer Bootcamp

LEÓN - 2022

Ricardo J. Rodríguez
University of Zaragoza

5 to 15 July 2022
León, Spain

incibe.es/summer-bootcamp

Organizers:

GOBIERNO DE ESPAÑA
VICEPRESIDENCIA PRIMERA DEL GOBIERNO
MINISTERIO DE ASUNTOS ECONÓMICOS Y TRANSFORMACIÓN DIGITAL
SECRETARÍA DE ESTADO DE DIGITALIZACIÓN E INTELIGENCIA ARTIFICIAL

incibe_
SPANISH NATIONAL CYBERSECURITY INSTITUTE

Partners:

OAS More rights for more people

Canada

AYUNTAMIENTO DE LEÓN

LEÓN CONGRESOS

universidad de león

Instructor

- **Ricardo J. Rodríguez**
  - PhD on Computer and Systems Engineering
  - Associate Professor (public servant) at the University of Zaragoza
  - **Researcher in cybersecurity issues**, especially in:
    - Program Binary Analysis
    - Digital forensics (in particular, in memory)
    - Security in systems based on RFID/NFC
  - **DisCo research group**
    - RME-DisCo: https://reversea.me
    - Follow us on Twitter and on Telegram! @reverseame
  - **E-mail**: rjrodriguez@unizar.es
    - Feel free to contact me if you have questions after the workshop!
  - **Personal website**: http://www.ricardojrodriguez.es

Departamento de Informática e Ingeniería de Sistemas
Universidad Zaragoza
1542

Organizers:

Partners:

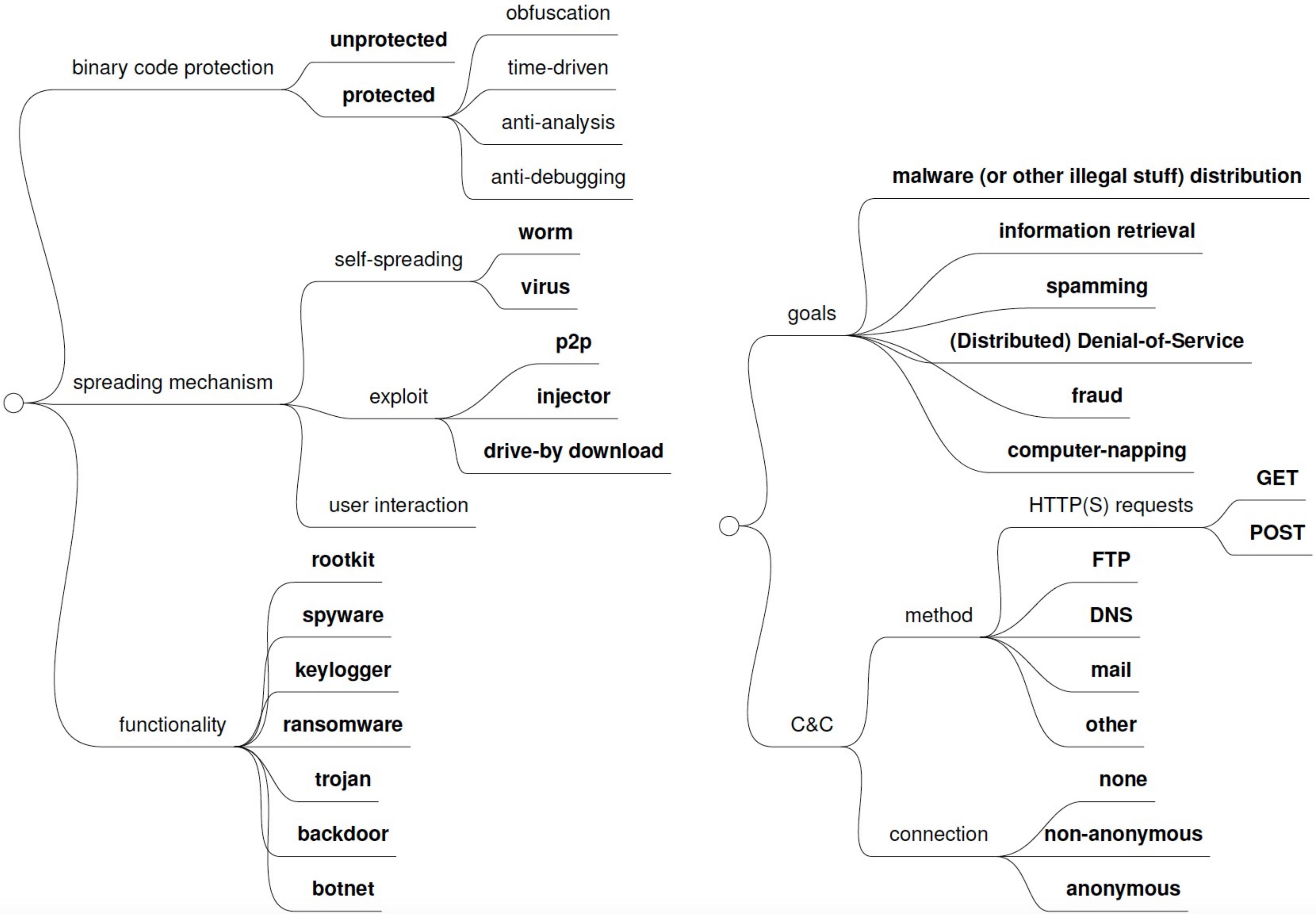
AGENDA

1. **Introduction**
   - What Is Malware?
   - Malware Analysis Methodology
   - Tools
2. **Previous Concepts**
   - Program Structure (PE Format)
   - WinAPIs and Malware

AGENDA

3. **Program Analysis Techniques: Control-Flow Graph**
   - Control-Flow Analysis
   - Terminology. Examples
4. **Program Analysis Techniques: Symbolic Execution**
   - History. Examples
   - Terminology
   - Challenges
5. **Program Analysis Techniques: Dynamic Binary Instrumentation**
   - DBI Advantages and Disadvantages
   - The Pin Framework
   - Examples

1. Introduction



- binary code protection
  - **unprotected**
  - **protected**
    - obfuscation
    - time-driven
    - anti-analysis
    - anti-debugging
- spreading mechanism
  - self-spreading
    - **worm**
    - **virus**
  - exploit
    - **p2p**
    - **injector**
    - **drive-by download**
  - user interaction
- functionality
  - **rootkit**
  - **spyware**
  - **keylogger**
  - **ransomware**
  - **trojan**
  - **backdoor**
  - **botnet**

- goals
  - **malware (or other illegal stuff) distribution**
  - **information retrieval**
  - **spamming**
  - **(Distributed) Denial-of-Service**
  - **fraud**
  - **computer-napping**
- C&C
  - method
    - HTTP(S) requests
      - **GET**
      - **POST**
    - **FTP**
    - **DNS**
    - **mail**
    - **other**
  - connection
    - **none**
    - **non-anonymous**
    - **anonymous**

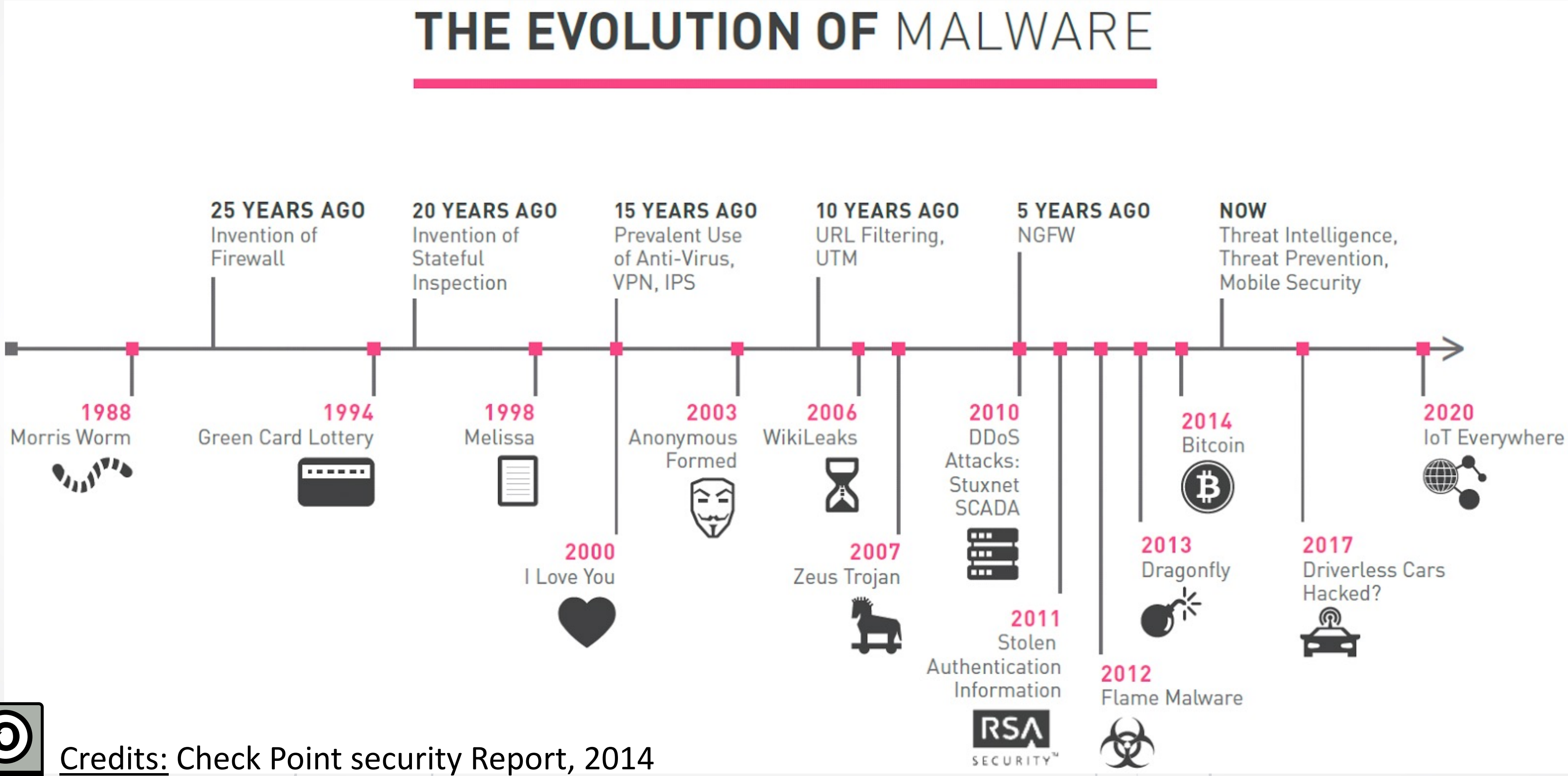
1. Introduction

Main Goal

https://www.fbi.gov/wanted/cyber/

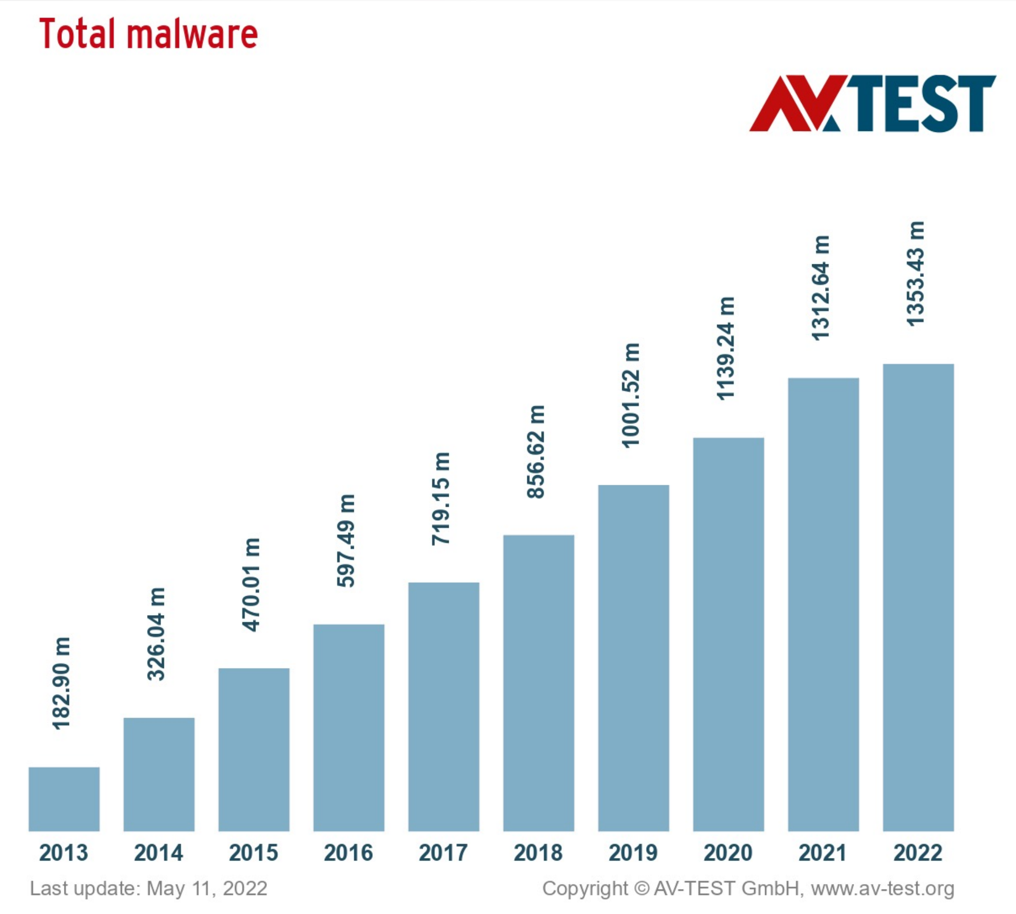
- Some numbers. . .
    - **ZeuS**: over $100M (acknowledged)
    - **Citadel, Dridex**: estimated £20M in the UK, $10M in the US (2015 only)
        - Let me do the math for you: £1.66M/month, $833k/month

1. Introduction



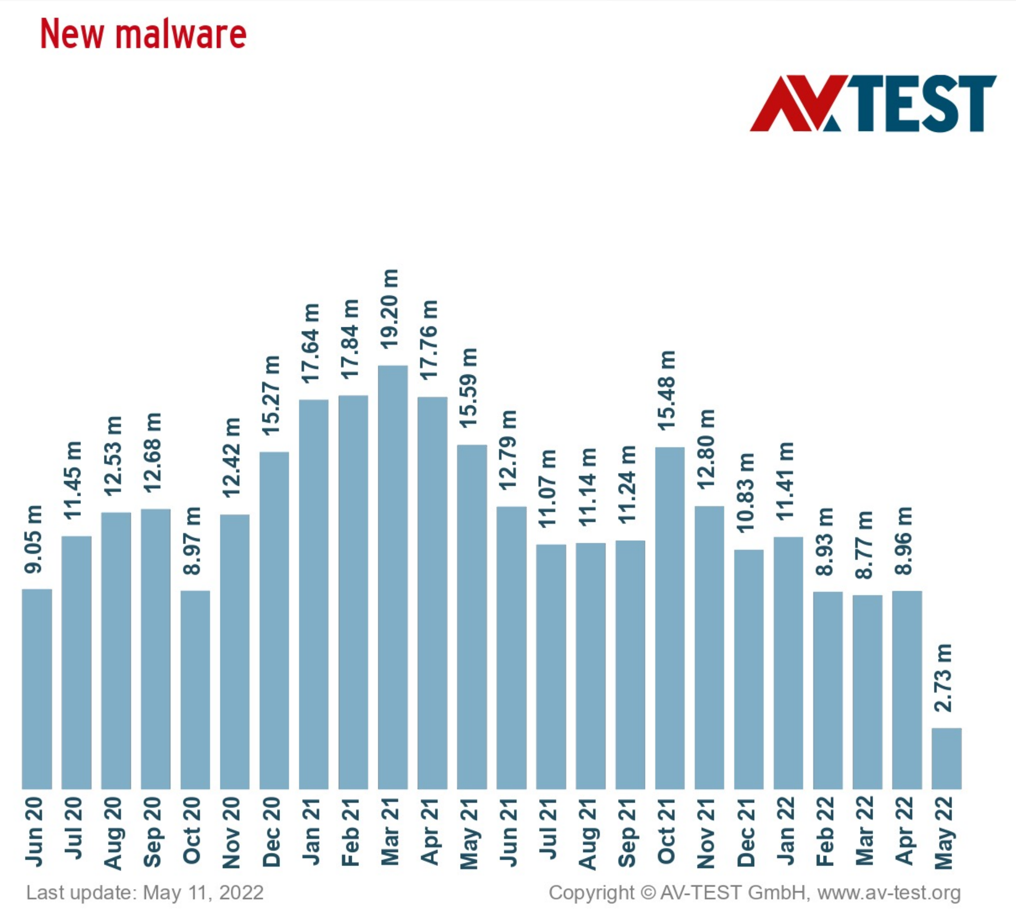
Credits: Check Point security Report, 2014

1. Introduction



Last update: May 11, 2022

Copyright © AV-TEST GmbH, www.av-test.org



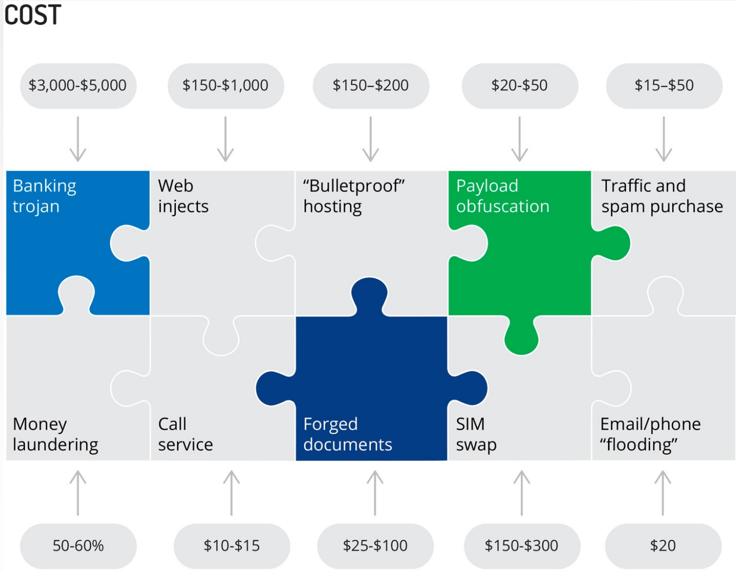
Last update: May 11, 2022

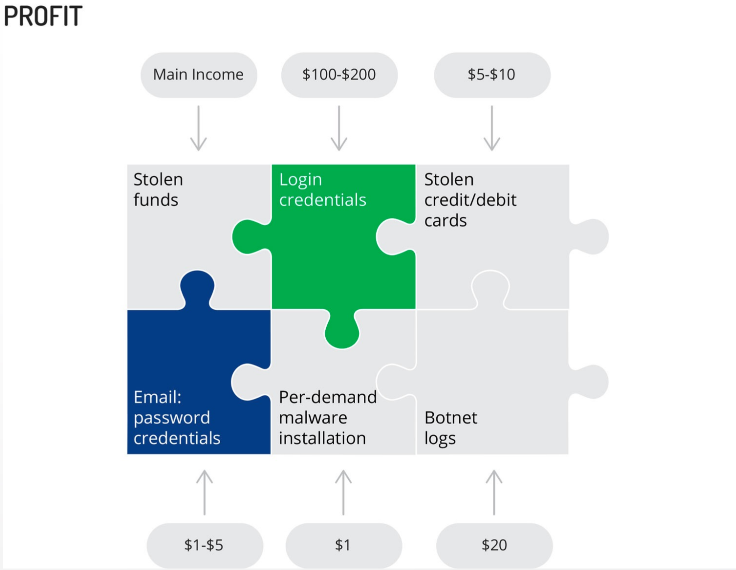
Copyright © AV-TEST GmbH, www.av-test.org

1. Introduction

Estimation of Cybercrime Costs and Benefits (2017)

Source: 2017 AT&T Global State of Cybersecurity survey





Credits: https://www.recordedfuture.com/cyber-operations-cost/

Organizers:

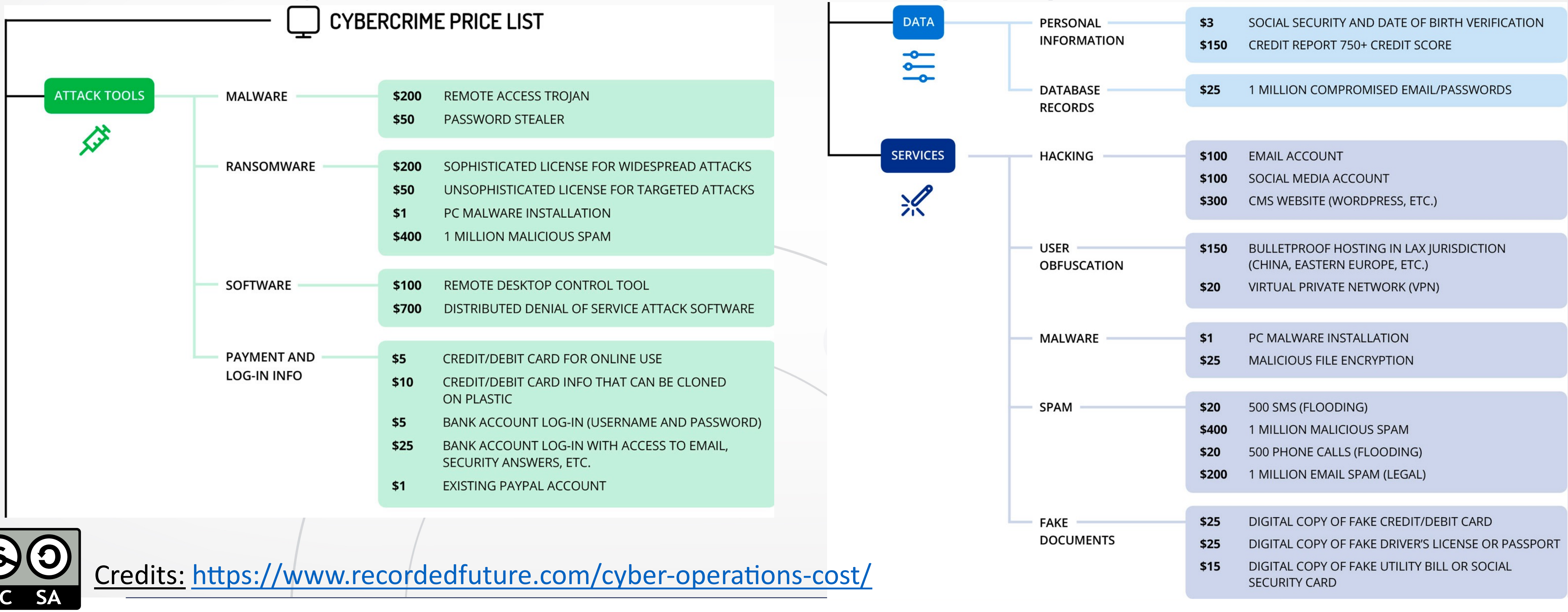
GOBIERNO DE ESPAÑA
VICEPRESIDENCIA PRIMERA DEL GOBIERNO
MINISTERIO DE ASUNTOS ECONÓMICOS Y TRANSFORMACIÓN DIGITAL
SECRETARÍA DE ESTADO DE DIGITALIZACIÓN E INTELIGENCIA ARTIFICIAL
incibe_
SPANISH NATIONAL CYBERSECURITY INSTITUTE

Partners:

OAS More rights for more people
Canada
AYUNTAMIENTO DE LEÓN
LEÓN CONGRESOS
universidad de león

1. Introduction

Estimation of Cybercrime Costs and Benefits (2017)



CYBERCRIME PRICE LIST

ATTACK TOOLS

MALWARE	$200	REMOTE ACCESS TROJAN
	$50	PASSWORD STEALER
RANSOMWARE	$200	SOPHISTICATED LICENSE FOR WIDESPREAD ATTACKS
	$50	UNSOPHISTICATED LICENSE FOR TARGETED ATTACKS
	$1	PC MALWARE INSTALLATION
	$400	1 MILLION MALICIOUS SPAM
SOFTWARE	$100	REMOTE DESKTOP CONTROL TOOL
	$700	DISTRIBUTED DENIAL OF SERVICE ATTACK SOFTWARE
PAYMENT AND LOG-IN INFO	$5	CREDIT/DEBIT CARD FOR ONLINE USE
	$10	CREDIT/DEBIT CARD INFO THAT CAN BE CLONED ON PLASTIC
	$5	BANK ACCOUNT LOG-IN (USERNAME AND PASSWORD)
	$25	BANK ACCOUNT LOG-IN WITH ACCESS TO EMAIL, SECURITY ANSWERS, ETC.
	$1	EXISTING PAYPAL ACCOUNT

DATA

PERSONAL INFORMATION	$3	SOCIAL SECURITY AND DATE OF BIRTH VERIFICATION
	$150	CREDIT REPORT 750+ CREDIT SCORE
DATABASE RECORDS	$25	1 MILLION COMPROMISED EMAIL/PASSWORDS

SERVICES

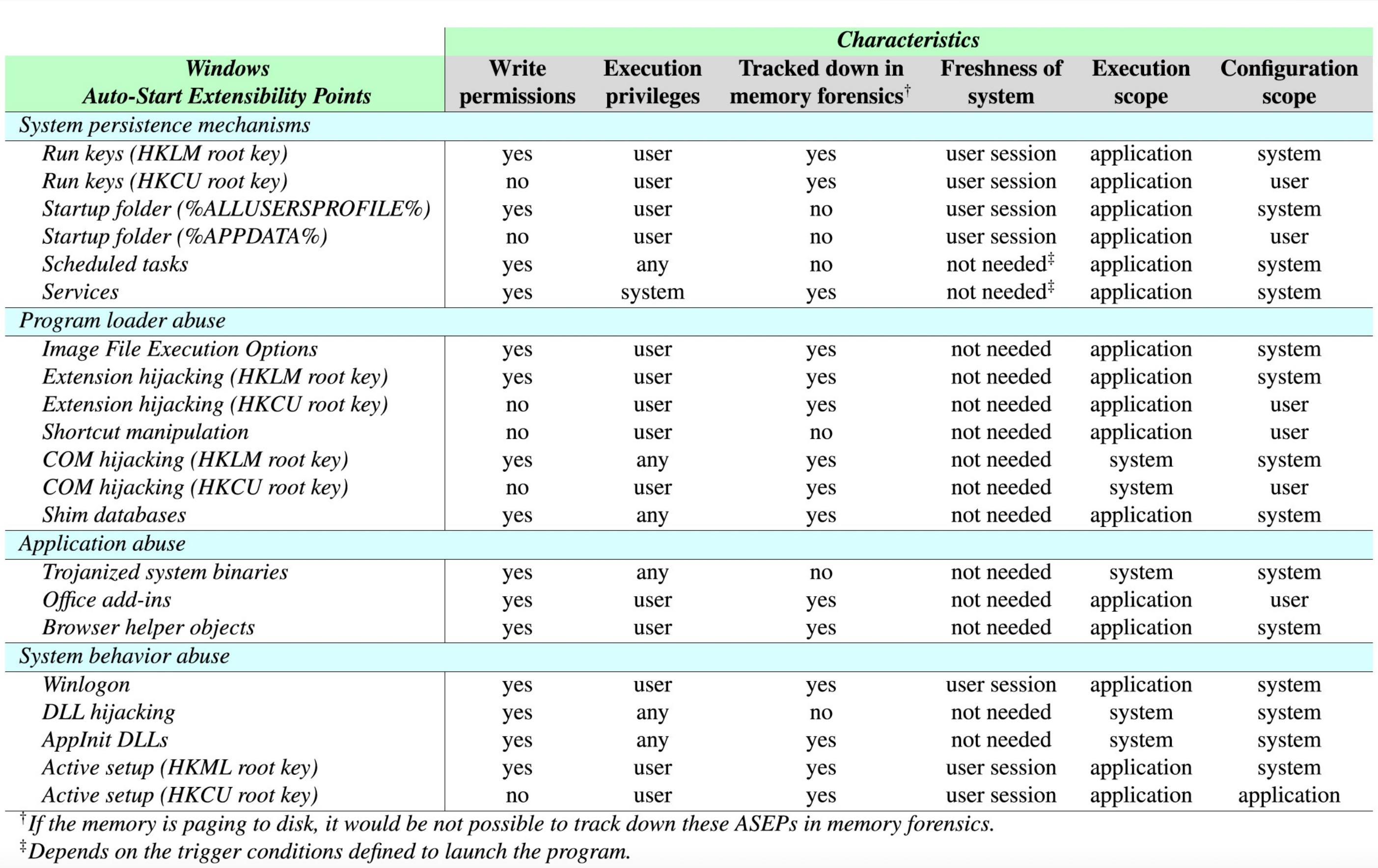
HACKING	$100	EMAIL ACCOUNT
	$100	SOCIAL MEDIA ACCOUNT
	$300	CMS WEBSITE (WORDPRESS, ETC.)
USER OBFUSCATION	$150	BULLETPROOF HOSTING IN LAX JURISDICTION (CHINA, EASTERN EUROPE, ETC.)
	$20	VIRTUAL PRIVATE NETWORK (VPN)
MALWARE	$1	PC MALWARE INSTALLATION
	$25	MALICIOUS FILE ENCRYPTION
SPAM	$20	500 SMS (FLOODING)
	$400	1 MILLION MALICIOUS SPAM
	$20	500 PHONE CALLS (FLOODING)
	$200	1 MILLION EMAIL SPAM (LEGAL)
FAKE DOCUMENTS	$25	DIGITAL COPY OF FAKE CREDIT/DEBIT CARD
	$25	DIGITAL COPY OF FAKE DRIVER'S LICENSE OR PASSPORT
	$15	DIGITAL COPY OF FAKE UTILITY BILL OR SOCIAL SECURITY CARD

Credits: https://www.recordedfuture.com/cyber-operations-cost/

1. Introduction

Malware

- ***Lifecycle***
    1. Initial compromise (social engineering
    2. Persistence
    3. Communication with C&C servers
    4. Lateral movement
    5. Data exfiltration / malicious activity



Windows Auto-Start Extensibility Points	*Characteristics*					
	Write permissions	**Execution privileges**	**Tracked down in memory forensics**[†]	**Freshness of system**	**Execution scope**	**Configuration scope**
System persistence mechanisms						
Run keys (HKLM root key)	yes	user	yes	user session	application	system
Run keys (HKCU root key)	no	user	yes	user session	application	user
Startup folder (%ALLUSERSPROFILE%)	yes	user	no	user session	application	system
Startup folder (%APPDATA%)	no	user	no	user session	application	user
Scheduled tasks	yes	any	no	not needed[‡]	application	system
Services	yes	system	yes	not needed[‡]	application	system
Program loader abuse						
Image File Execution Options	yes	user	yes	not needed	application	system
Extension hijacking (HKLM root key)	yes	user	yes	not needed	application	system
Extension hijacking (HKCU root key)	no	user	yes	not needed	application	user
Shortcut manipulation	no	user	no	not needed	application	user
COM hijacking (HKLM root key)	yes	any	yes	not needed	system	system
COM hijacking (HKCU root key)	no	user	yes	not needed	system	user
Shim databases	yes	any	yes	not needed	application	system
Application abuse						
Trojanized system binaries	yes	any	no	not needed	system	system
Office add-ins	yes	user	yes	not needed	application	user
Browser helper objects	yes	user	yes	not needed	application	system
System behavior abuse						
Winlogon	yes	user	yes	user session	application	system
DLL hijacking	yes	any	no	not needed	system	system
AppInit DLLs	yes	any	yes	not needed	system	system
Active setup (HKML root key)	yes	user	yes	user session	application	system
Active setup (HKCU root key)	no	user	yes	user session	application	application

[†] *If the memory is paging to disk, it would be not possible to track down these ASEPs in memory forensics.*

[‡] *Depends on the trigger conditions defined to launch the program.*

More details: Uroz, D. & Rodríguez, R. J. **Characteristics and Detectability of Windows Auto-Start Extensibility Points in Memory Forensics**. Digital Investigation, 2019, 28, S95-S104, Elsevier. https://doi.org/10.1016/j.diin.2019.01.026

1. Introduction

Malware Analysis Methodology

- **Static program analysis** (also called *dead code* or *cold analysis*)
  - **The program does not run**
  - You should take a look at…
    - PE properties
    - Import functions (which APIs are used?)
    - Hash computation (e.g., MD5, SHA1)
    - Retrieve strings from the binary file: strings
  - **Disadvantage:**
    - All possible execution paths are explored (*state explosion problem*)
      - You might be analyzing infeasible code

1. Introduction

Malware Analysis Methodology

- **Dynamic program analysis** (also called *live code* or *hot analysis*)
  - **The program does run**
  - You should take a look at…
    - Interaction with the OS: at the filesystem, process, and Windows Registry levels
    - Interaction with the Internet: connections to domain names or IPs, network data transmitted
  - Helps find out their (malicious?) behaviour
  - **Disadvantage:**
    - Only one of the possible execution paths is explored
      - It may depend on the current execution conditions (environment variables, datime, etc.)

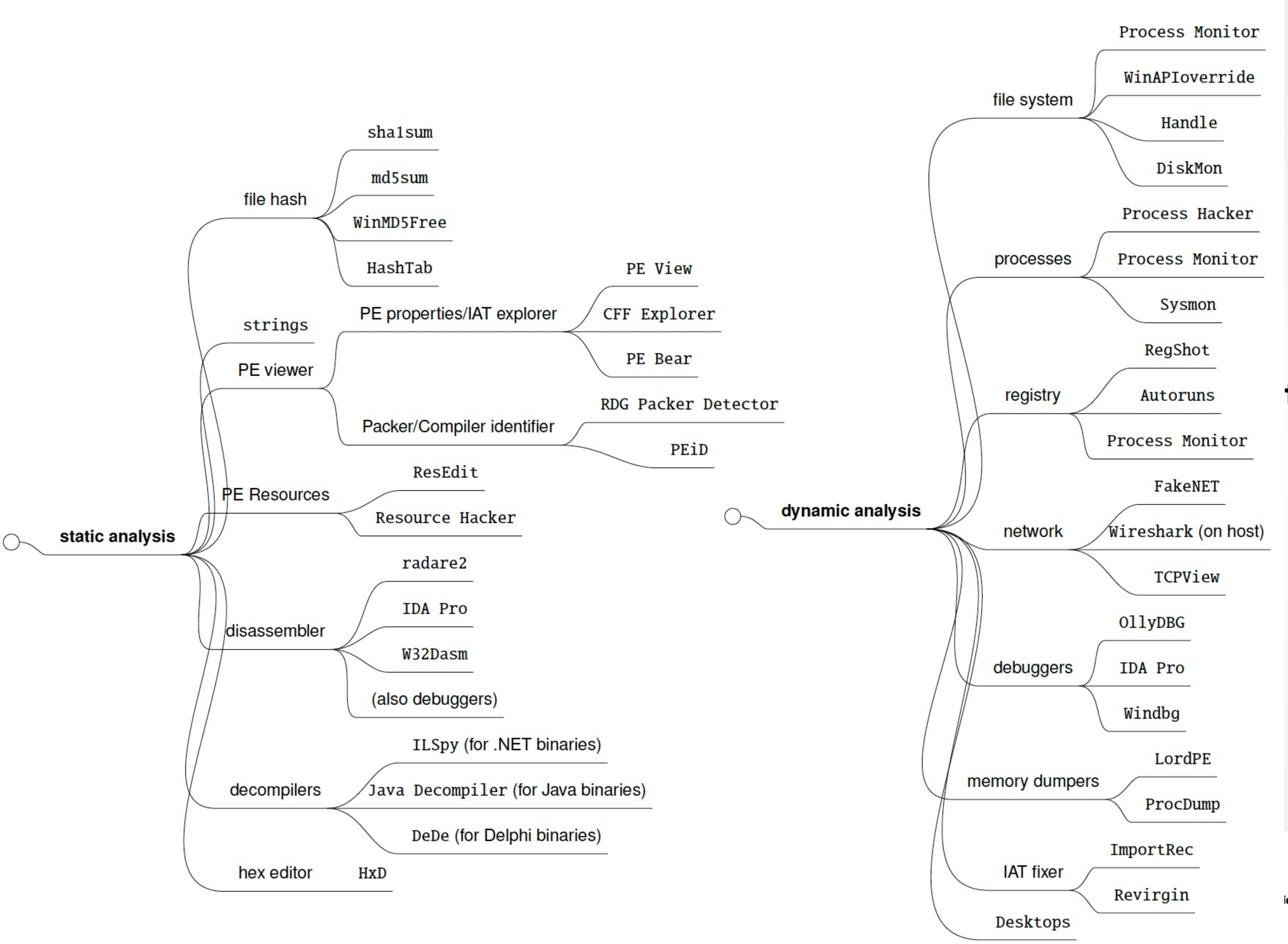
1.

An

- I

tivity

- D



1. Introduction

Attack patterns

- **Downloaders**
    - It is usually the first stage of a successful infection
    - It can install registry keys to automatically run on next reboot/login! (persistence via ASEPs)
- **Information retrievals**
    - Iterate through files looking for/mask/extensions/specific files...
- **Process memory explorers**
    - Read the memory of other processes and extract information of interest
- **Ransomware**
    - Iterate through directories and files, open, read and write them

1. Introduction

Attack patterns

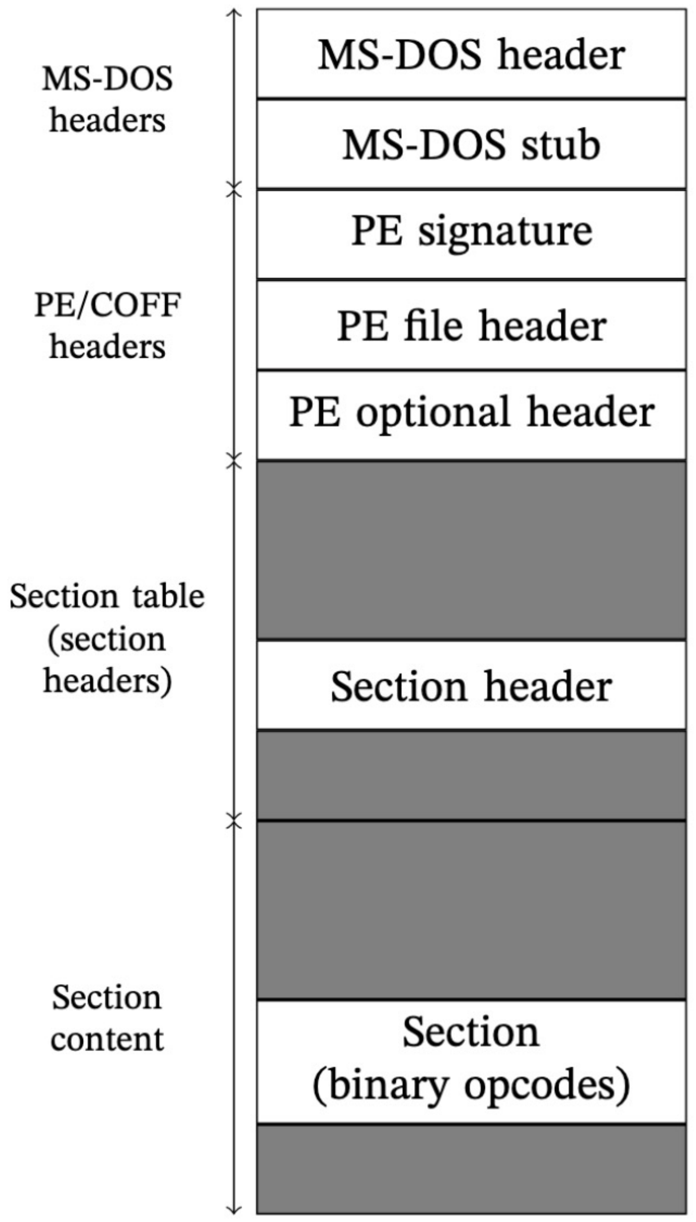
- **Keyloggers**
  - Set a hook function, either thread-specific or global
  - Remember that Windows is built on the event-driven paradigm
    - WH_CALLWNDPROC, WH_CBT, WH_DEBUG, WH_GETMESSAGE, WH_KEYBOARD, WH_MOUSE, WH_MSGFILTER
- **Code injection**
  - Inject code into the memory of another process and execute it
  - Three methods: remote DLL loading, hook function, raw code
- **Connection to C&C**
  - Winsocks (similar to psockets, but require calling WSAStartup first)
  - Wininet: HTTP and FTP session management made easy for developers

2. Previous Concepts

2. Previous Concepts

Program Structure

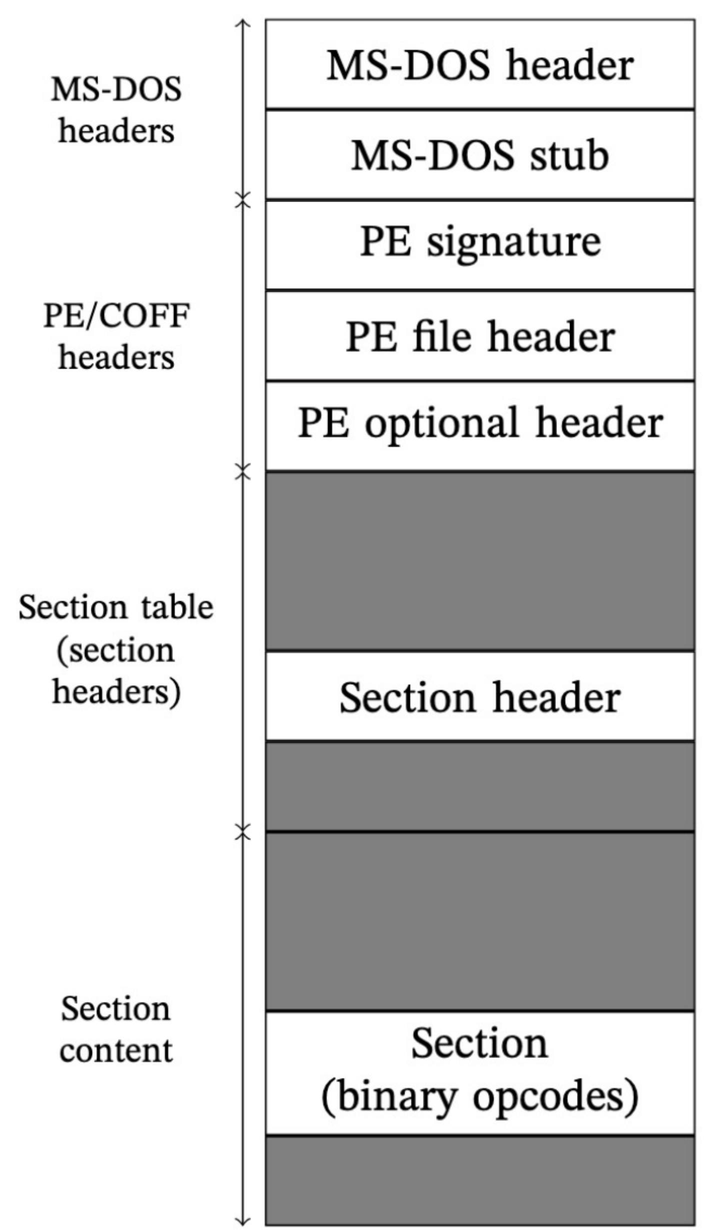
- Since Windows NT 3.1
- **PE: Portable Executable**
    - Data structure defined in WinNT.h (Microsoft Windows SDK)
    - Three parts: MS-DOS headers, PE/COFF headers, Section headers
    - https://docs.microsoft.com/en-us/windows/win32/debug/pe-format
- **MS-DOS headers**
    - First 64 bytes
    - e_magic: MZ (Mark Zbikowski)
    - e_lfanew: offset to PE/COFF headers



2. Previous Concepts

Program Structure

- PE/COFF headers
  - **PE signature** (“PE\0\0”)
  - **PE file header**
    - Define target machine, number of sections, characteristics, etc.
  - **PE optional header**
    - Optional for some object files
    - Fields of interest: ImageBase, BaseOfCode, AddressOfEntryPoint
    - DataDirectory: Directory table. Each entry has a meaning
- Section headers
  - IMAGE_SECTION_HEADER structure
  - Common sections: .text/.code, .rdata/.rodata, .data, .reloc, …



2. Previous Concepts

Use of WinAPIs

- **Static import**
  - Windows APIs invoked by the binary
  - They are present in the DataDirectory section, visible with any PE viewing tool
  - Function identified by string name or ordinal position (in EAT)
- **Dynamic import**
  - Windows API is resolved on execution
  - Different ways to dynamically import a function
    - Usually, LoadLibrary (loads a DLL) + GetProcAddress (gets the address of the function)
    - Can also be dynamically resolved by ordinal position (in EAT) instead of function name

2. Previous Concepts

Brief Summary of WinAPIs Used by Malware

- **Processes and IPCs (kernel32.dll)**
  - CreateProcessA, OpenProcess, CreateThread, CreatePipe, CreateNamedPipe, CreateMutex, OpenMutex, CreateToolhelp32Snapshot, CreateRemoteThread, ...
- **Files (kernel32.dll)**
  - CreateFile, WriteFile, ReadFile, CopyFile, ...
- **Registry (advap32i.dll)**
  - RegOpenKey, RegEnumKey, RegEnumValue, RegDeleteKey, RegQueryInfoKey, …
- **Network (ws2_32.dll) – Winsocks**
  - WSAStartup, WSASocket, socket, connect, accept, bind, recv, send, htons, …
  - urlmon.dll: URLDownloadToFile, …

2. Previous Concepts

Basic Malware Analysis

LAB SESSION 1

- **Additional files for *Lab session 1***
  - https://webdiis.unizar.es/~ricardo/sbc-2022/advanced-malware-analysis/laboratories/additional_files/lab1_malware_files.7z
- Follow the laboratory workbook provided on the workshop's website: https://webdiis.unizar.es/~ricardo/sbc-2022/advanced-malware-analysis/laboratories/lab1_intro_malware_analysis.pdf

3. Program Analysis Techniques: Control-Flow Graph

3. Program Analysis: Control-Flow Graph

Control-Flow Analysis

- Static program analysis technique
- Goal: determine the order of execution of the program statements
- Allows us to understand the structure of the Control-Flow Graph (CFG)
    - Low-level representation of control flow
- CFG: directed graph
    - **Nodes**: statements (or instructions)
    - **Edges**: control flow

A CFG specifies ALL possible paths of execution of a program

3. Program Analysis: Control-Flow Graph

History of Control-Flow Analysis

- American computer scientist
- Pioneer in the field of compiler optimization
- Fundamental work on compilers, code optimization, and parallelization
- First female IBM fellow in 1989
- First female Turing Award in 2006
    - Her 1970 papers, “Control Flow Analysis” and “A Basis for Program Optimization” established “intervals” as the context for efficient and effective data flow analysis and optimization

Frances Elizabeth Allen
(1932-2020)

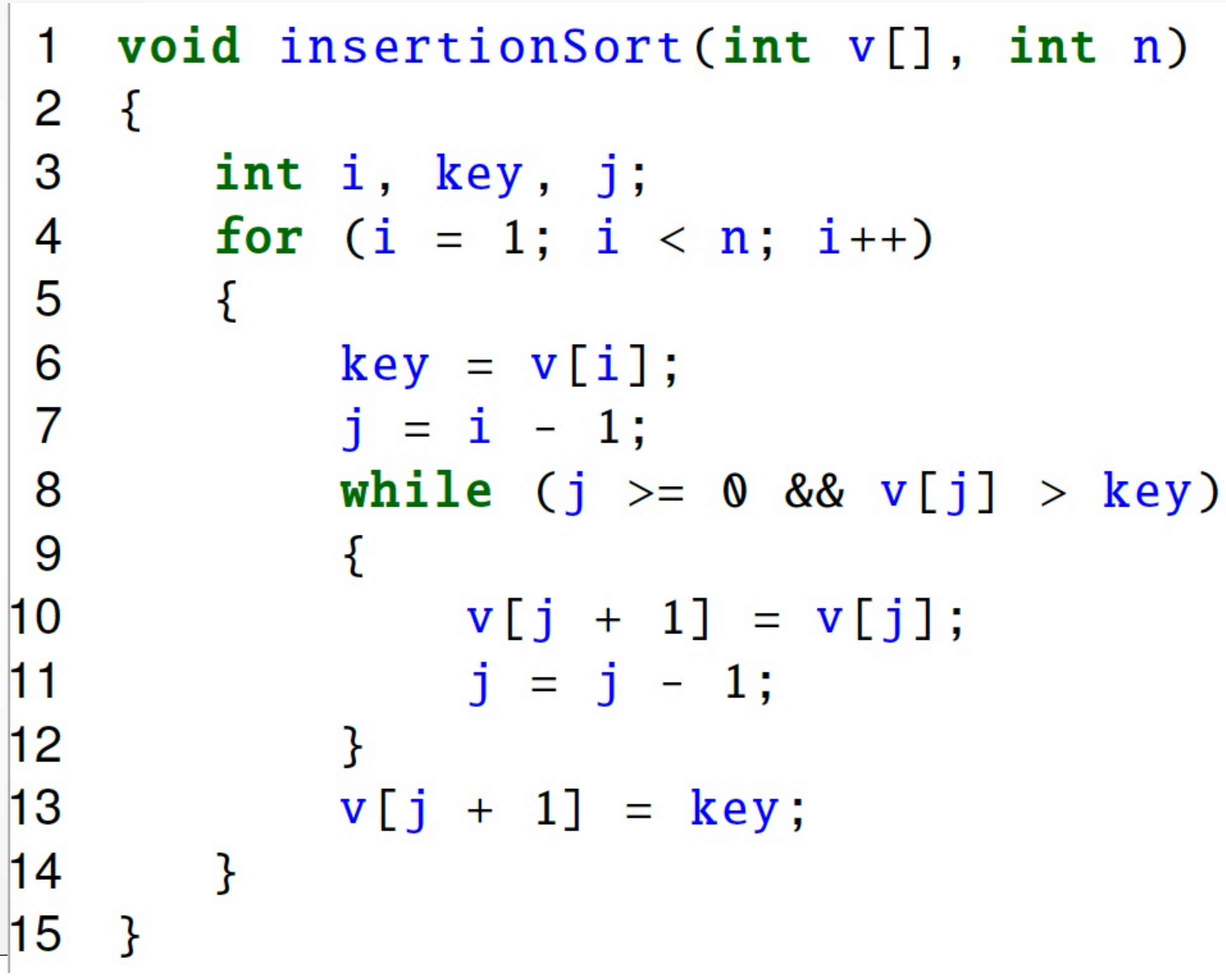
3. Program Analysis: Control-Flow Graph

Terminology

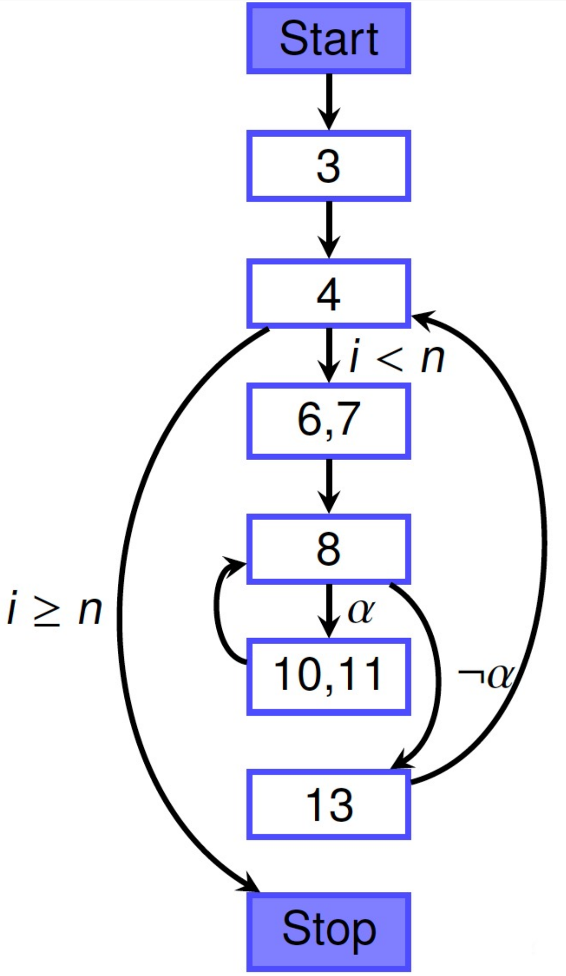
- Basic block:
  - (Linear) **sequence of consecutive program instructions that have an entry point** (first instruction executed) **and an exit point** (last instruction executed)
    - **Control enters only at the beginning of the sequence**
    - **Control leaves only at the end of the sequence**
    - **No branching in or out in the middle of the basic blocks**
- Path:
  - Sequence of nodes (static view), including an entry node and an exit node
  - **Path sequence**: subsequence of nodes along the path
- Trace:
  - Sequence of instructions executed during program execution (dynamic view)

3. Program Analysis: Control-Flow Graph

Examples



```
 1  void insertionSort(int v[], int n)
 2  {
 3      int i, key, j;
 4      for (i = 1; i < n; i++)
 5      {
 6          key = v[i];
 7          j = i - 1;
 8          while (j >= 0 && v[j] > key)
 9          {
10              v[j + 1] = v[j];
11              j = j - 1;
12          }
13          v[j + 1] = key;
14      }
15  }
```



$$\alpha = j \geq 0 \wedge v[j] > key$$

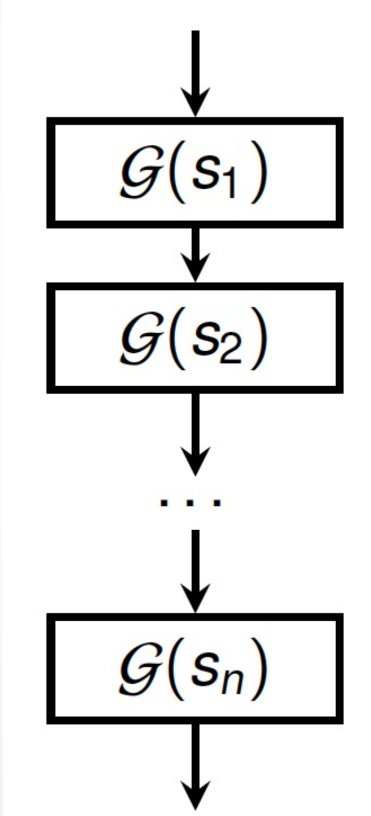
3. Program Analysis: Control-Flow Graph

Types of basic blocks

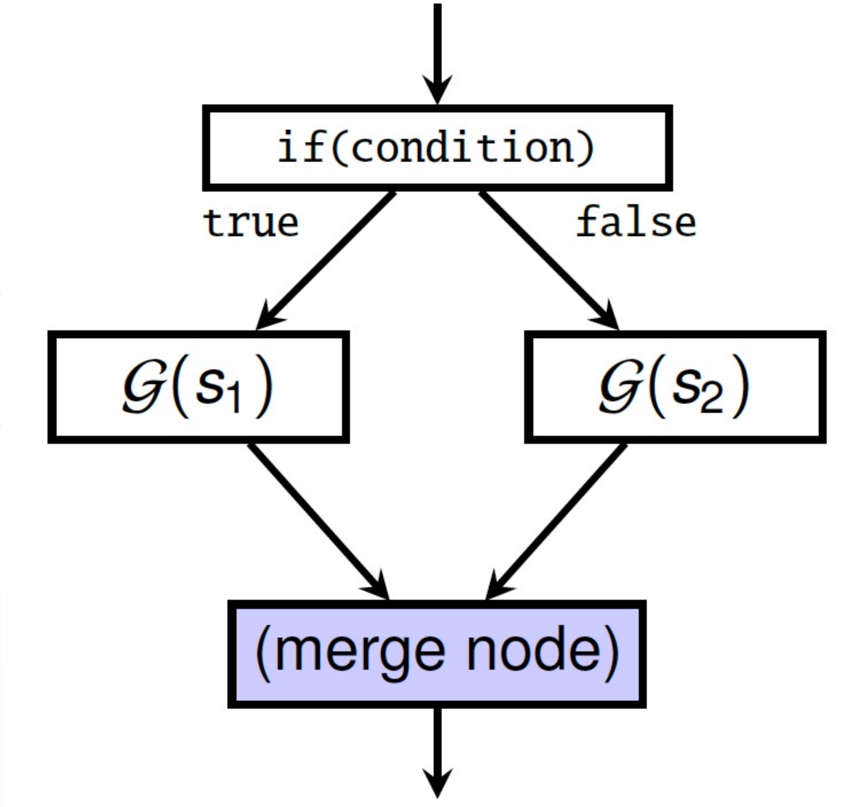
- **Entry node**
- **Exit node**
- **Decision node**: contains a conditional statement
    - Creates at least two branches
- **Merge node**:
    - Optional node
    - Point where multiple control branches merge
- **Statement node**: sequence of statements

3. Program Analysis: Control-Flow Graph

$\mathcal{G}(s_1; s_2; \ldots; s_n)$

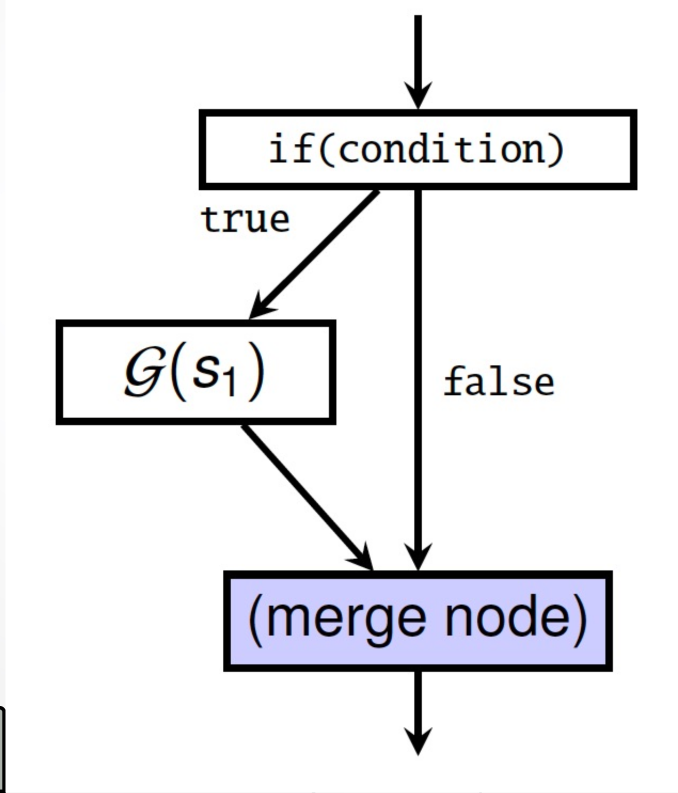


$\mathcal{G}(\texttt{if(condition) \textbf{then} s\_1 else s\_2})$

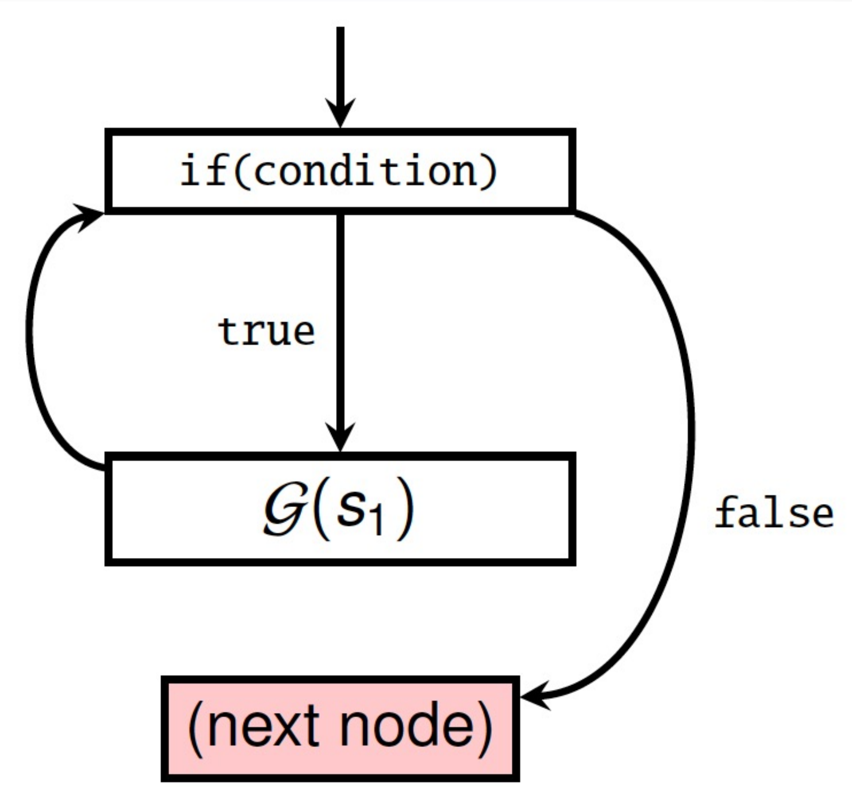


3. Program Analysis: Control-Flow Graph

$\mathcal{G}(\texttt{if(condition)}\ \textbf{then}\ \texttt{s\_1})$

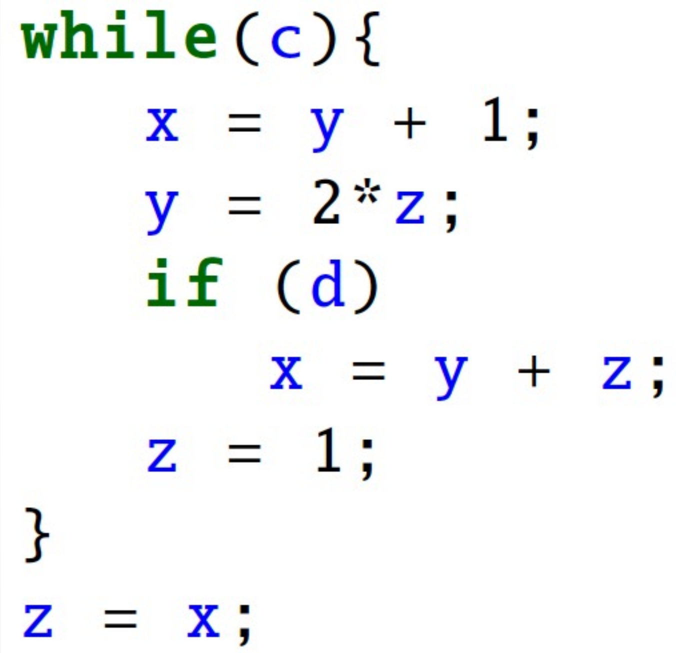


$\mathcal{G}(\texttt{while(condition)}\ \texttt{s\_1})$

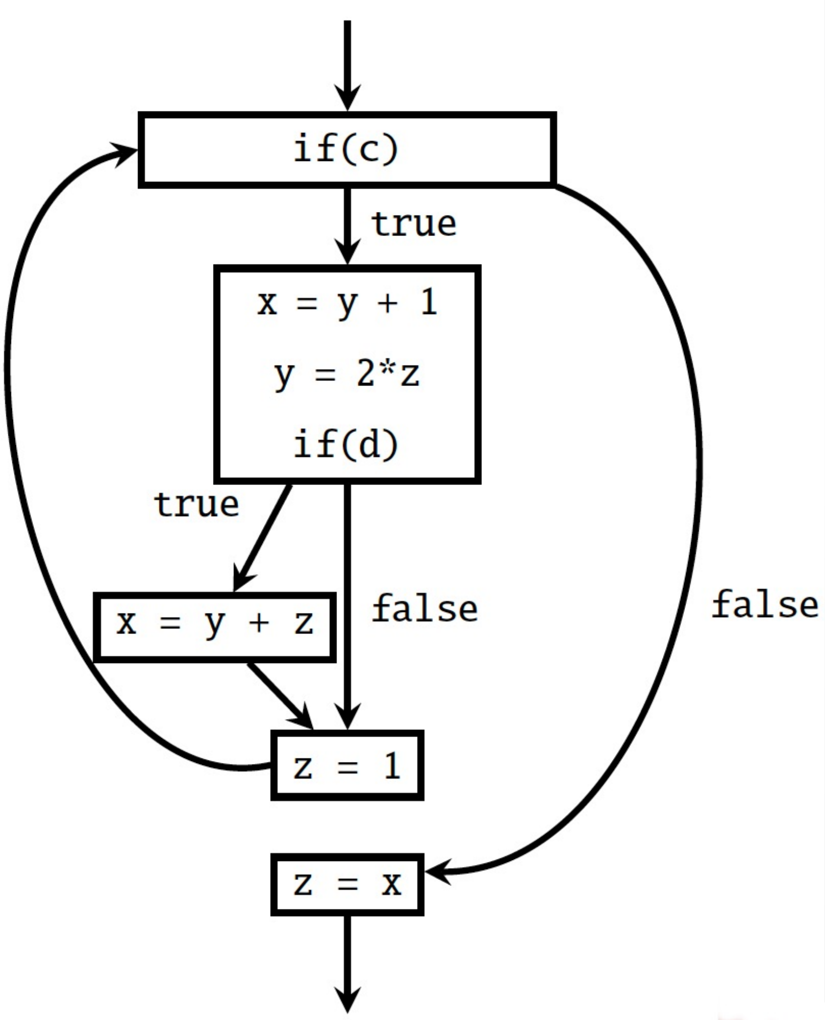


3. Program Analysis: Control-Flow Graph

Efficient CFG

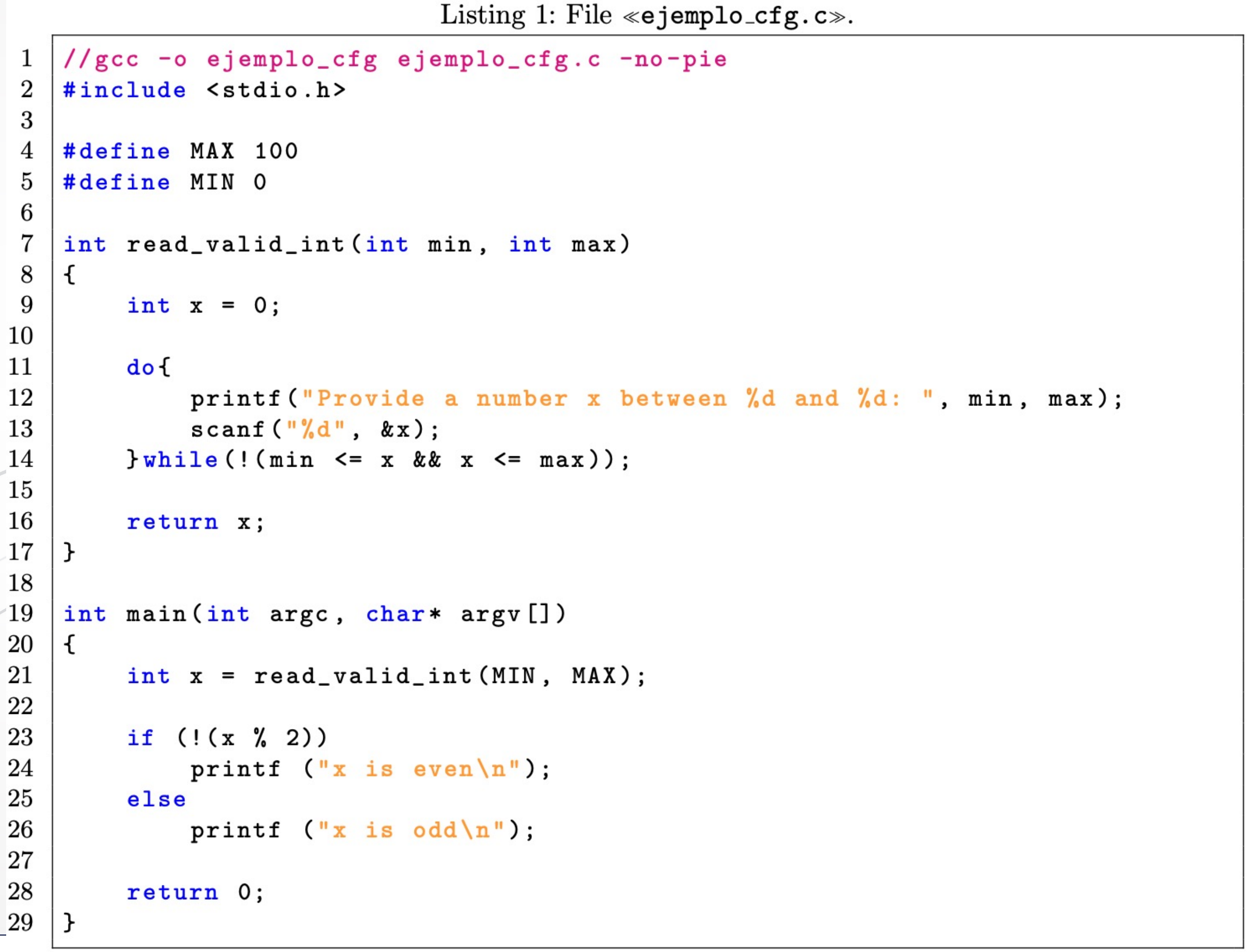


```
while(c){
    x = y + 1;
    y = 2*z;
    if (d)
        x = y + z;
    z = 1;
}
z = x;
```



3. Program Analysis: Control-Flow Graph

Another example

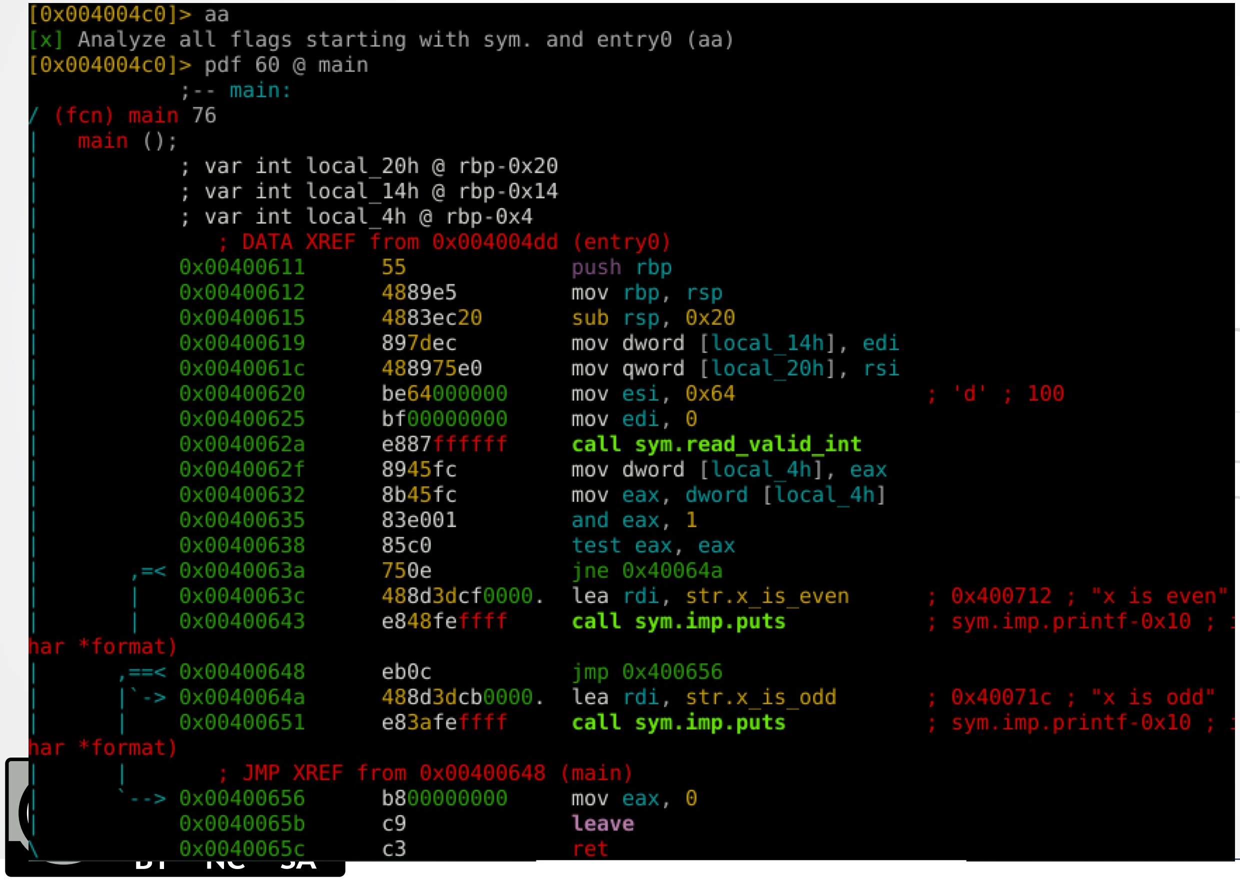


Listing 1: File «ejemplo_cfg.c».

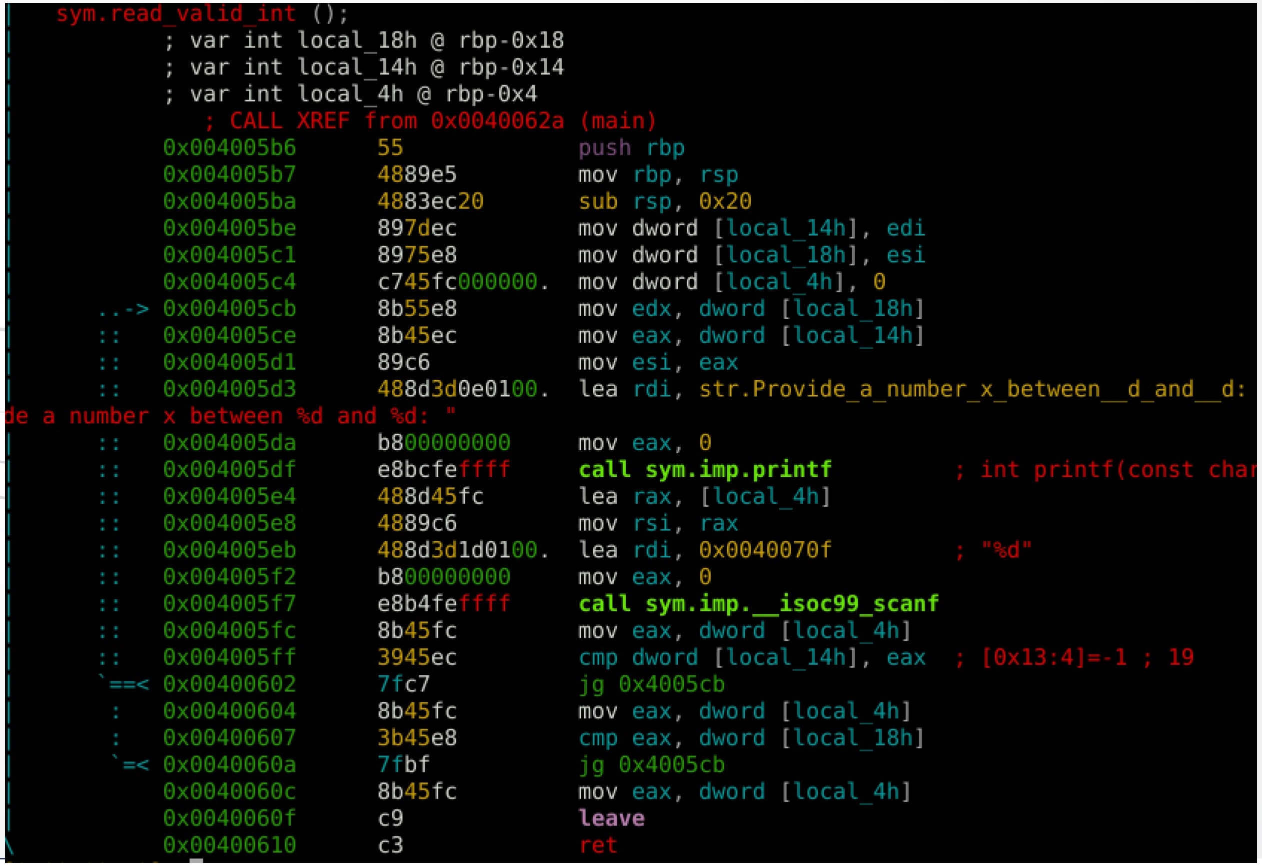
```c
//gcc -o ejemplo_cfg ejemplo_cfg.c -no-pie
#include <stdio.h>

#define MAX 100
#define MIN 0

int read_valid_int(int min, int max)
{
    int x = 0;

    do{
        printf("Provide a number x between %d and %d: ", min, max);
        scanf("%d", &x);
    }while(!(min <= x && x <= max));

    return x;
}

int main(int argc, char* argv[])
{
    int x = read_valid_int(MIN, MAX);

    if (!(x % 2))
        printf ("x is even\n");
    else
        printf ("x is odd\n");

    return 0;
}
```

3. Program Analysis: Control-Flow Graph

Another example



```
[0x004004c0]> aa
[x] Analyze all flags starting with sym. and entry0 (aa)
[0x004004c0]> pdf 60 @ main
            ;-- main:
/ (fcn) main 76
|   main ();
|           ; var int local_20h @ rbp-0x20
|           ; var int local_14h @ rbp-0x14
|           ; var int local_4h @ rbp-0x4
|              ; DATA XREF from 0x004004dd (entry0)
|           0x00400611      55             push rbp
|           0x00400612      4889e5         mov rbp, rsp
|           0x00400615      4883ec20       sub rsp, 0x20
|           0x00400619      897dec         mov dword [local_14h], edi
|           0x0040061c      488975e0       mov qword [local_20h], rsi
|           0x00400620      be64000000     mov esi, 0x64               ; 'd' ; 100
|           0x00400625      bf00000000     mov edi, 0
|           0x0040062a      e887ffffff     call sym.read_valid_int
|           0x0040062f      8945fc         mov dword [local_4h], eax
|           0x00400632      8b45fc         mov eax, dword [local_4h]
|           0x00400635      83e001         and eax, 1
|           0x00400638      85c0           test eax, eax
|       ,=< 0x0040063a      750e           jne 0x40064a
|       |   0x0040063c      488d3dcf0000.  lea rdi, str.x_is_even      ; 0x400712 ; "x is even"
|       |   0x00400643      e848feffff     call sym.imp.puts           ; sym.imp.printf-0x10 ;
har *format)
|      ,==< 0x00400648      eb0c           jmp 0x400656
|      |`-> 0x0040064a      488d3dcb0000.  lea rdi, str.x_is_odd       ; 0x40071c ; "x is odd"
|      |    0x00400651      e83afeffff     call sym.imp.puts           ; sym.imp.printf-0x10 ;
har *format)
|      |       ; JMP XREF from 0x00400648 (main)
|      `--> 0x00400656      b800000000     mov eax, 0
|           0x0040065b      c9             leave
\           0x0040065c      c3             ret
```



```
|   sym.read_valid_int ();
|           ; var int local_18h @ rbp-0x18
|           ; var int local_14h @ rbp-0x14
|           ; var int local_4h @ rbp-0x4
|              ; CALL XREF from 0x0040062a (main)
|           0x004005b6      55             push rbp
|           0x004005b7      4889e5         mov rbp, rsp
|           0x004005ba      4883ec20       sub rsp, 0x20
|           0x004005be      897dec         mov dword [local_14h], edi
|           0x004005c1      8975e8         mov dword [local_18h], esi
|           0x004005c4      c745fc000000.  mov dword [local_4h], 0
|       ..-> 0x004005cb     8b55e8         mov edx, dword [local_18h]
|       ::   0x004005ce     8b45ec         mov eax, dword [local_14h]
|       ::   0x004005d1     89c6           mov esi, eax
|       ::   0x004005d3     488d3d0e0100.  lea rdi, str.Provide_a_number_x_between__d_and__d:
de a number x between %d and %d: "
|       ::   0x004005da     b800000000     mov eax, 0
|       ::   0x004005df     e8bcfeffff     call sym.imp.printf         ; int printf(const char
|       ::   0x004005e4     488d45fc       lea rax, [local_4h]
|       ::   0x004005e8     4889c6         mov rsi, rax
|       ::   0x004005eb     488d3d1d0100.  lea rdi, 0x0040070f         ; "%d"
|       ::   0x004005f2     b800000000     mov eax, 0
|       ::   0x004005f7     e8b4feffff     call sym.imp.__isoc99_scanf
|       ::   0x004005fc     8b45fc         mov eax, dword [local_4h]
|       ::   0x004005ff     3945ec         cmp dword [local_14h], eax  ; [0x13:4]=-1 ; 19
|       `==< 0x00400602     7fc7           jg 0x4005cb
|        :   0x00400604     8b45fc         mov eax, dword [local_4h]
|        :   0x00400607     3b45e8         cmp eax, dword [local_18h]
|        `=< 0x0040060a     7fbf           jg 0x4005cb
|            0x0040060c     8b45fc         mov eax, dword [local_4h]
|            0x0040060f     c9             leave
\            0x00400610     c3             ret
```

Organizers:

Partners:

3. Program A -Flow Graph

Another example

```
0x400611 (0x400611) main
0x00400611:  push rbp
0x00400612:  mov  rbp, rsp
0x00400615:  sub  rsp, 0x20
0x00400619:  mov  dword ptr [rbp - 0x14], edi
0x0040061c:  mov  qword ptr [rbp - 0x20], rsi
0x00400620:  mov  esi, 0x64
0x00400625:  mov  edi, 0
0x0040062a:  call 0x4005b6
```

```
0x4005b6 (0x4005b6) read_valid_int
0x004005b6:  push rbp
0x004005b7:  mov  rbp, rsp
0x004005ba:  sub  rsp, 0x20
0x004005be:  mov  dword ptr [rbp - 0x14], edi
0x004005c1:  mov  dword ptr [rbp - 0x18], esi
0x004005c4:  mov  dword ptr [rbp - 4], 0
0x004005cb:  mov  edx, dword ptr [rbp - 0x18]
0x004005ce:  mov  eax, dword ptr [rbp - 0x14]
0x004005d1:  mov  esi, eax
0x004005d3:  lea  rdi, qword ptr [rip + 0x10e]
0x004005da:  mov  eax, 0
0x004005df:  call 0x4004a0
```

```
0x4005e4 (0x4005b6) read_valid_int+0x2e
0x004005e4:  lea  rax, qword ptr [rbp - 4]
0x004005e8:  mov  rsi, rax
0x004005eb:  lea  rdi, qword ptr [rip + 0x11d]
0x004005f2:  mov  eax, 0
0x004005f7:  call 0x4004b0
```

```
0x4005fc (0x4005b6) read_valid_int+0x46
0x004005fc:  mov eax, dword ptr [rbp - 4]
0x004005ff:  cmp dword ptr [rbp - 0x14], eax
0x00400602:  jg  0x4005cb
```

```
0x400604 (0x4005b6) read_valid_int+0x4e
0x00400604:  mov eax, dword ptr [rbp - 4]
0x00400607:  cmp eax, dword ptr [rbp - 0x18]
0x0040060a:  jg  0x4005cb
```

```
0x4005cb (0x4005b6) read_valid_int+0x15
0x004005cb:  mov  edx, dword ptr [rbp - 0x18]
0x004005ce:  mov  eax, dword ptr [rbp - 0x14]
0x004005d1:  mov  esi, eax
0x004005d3:  lea  rdi, qword ptr [rip + 0x10e]
0x004005da:  mov  eax, 0
0x004005df:  call 0x4004a0
```

```
0x40060c (0x4005b6) read_valid_int+0x56
0x0040060c:  mov   eax, dword ptr [rbp - 4]
0x0040060f:  leave
0x00400610:  ret
```

```
0x40062f (0x400611) main+0x1e
0x0040062f:  mov  dword ptr [rbp - 4], eax
0x00400632:  mov  eax, dword ptr [rbp - 4]
0x00400635:  and  eax, 1
0x00400638:  test eax, eax
0x0040063a:  jne  0x40064a
```

```
0x40063c (0x400611) main+0x2b
0x0040063c:  lea  rdi, qword ptr [rip + 0xcf]
0x00400643:  call 0x400490
```

```
0x400648 (0x400611) main+0x37
0x00400648:  jmp 0x400656
```

```
0x40064a (0x400611) main+0x39
0x0040064a:  lea  rdi, qword ptr [rip + 0xcb]
0x00400651:  call 0x400490
```

```
0x400656 (0x400611) main+0x45
0x00400656:  mov   eax, 0
0x0040065b:  leave
0x0040065c:  ret
```

3. Program Analysis Techniques: Symbolic Execution

3. Program Analysis: Symbolic Execution

- **Static program analysis technique**

- **Goal:** test all possible program execution paths instead of a single execution path
- Concrete execution vs. Symbolic execution
  - Symbolic execution generalizes tests
  - Allows unknown symbolic variables in the evaluation
  - **Check the feasibility of the program paths**

3. Program Analysis: Symbolic Execution

History

- **1976**
  - L. A. Clarke, *A System to Generate Test Data and Symbolically Execute Programs*, in IEEE Transactions on Software Engineering, vol. SE-2, no. 3, pp. 215-222, Sept. 1976. https://doi.org/10.1109/TSE.1976.233817
  - James C. King, *Symbolic execution and program testing*, Commun. ACM vol. 19, no. 7, pp. 385-394, Jul. 1976. https://doi.org/10.1145/360248.360252
  - **Problems**:
    - Not scalable: the program state has many bits, there are many program paths
    - Cannot make loops or library calls
    - Constraint solver is slow and not capable to handle advanced constraints

3. Program Analysis: Symbolic Execution

History

- **2005-2006:**
  - DART project (Godefroid and Sen, PLDI 2005)
    - Dynamic information for symbolic execution
  - EXE (Cadar, Ganesh, Pawlowski, Dill, and Engler, CCS 2006)
    - Powerful constraint solver that handles arrays
  - **Nowadays, we have**:
    - More powerful computers and clusters
    - Mixing techniques of concrete and symbolic executions
    - Powerful constraint solvers

3. Program Analysis: Symbolic Execution

Example

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

Concrete

1: $i = 1$
3: $i = 1, j = 2$
4: $i = 2, j = 2$
5: $i = 4, j = 2$
8: $i = 4$

Symbolic

1: i_{in}
3: $i = i_{in}, j = 2i_{in}$
4: $i = i_{in} + 1, j = 2i_{in}$
5: $i = (i_{in} + 1)2i_{in}, j = 2i_{in}$

True branch:

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

False branch:

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

3. Program Analysis: Symbolic Execution

Example: bug finding

```
int bar(int i)
{
      int j = 2*i;
      i++;
      i = i*j;
      if (i < 1)
            i = -i;

      i = j/i;
      return i;
}
```

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 creates problems…

- False branch is always safe

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

- What about the true branch?

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

3. Program Analysis: Symbolic Execution

Terminology

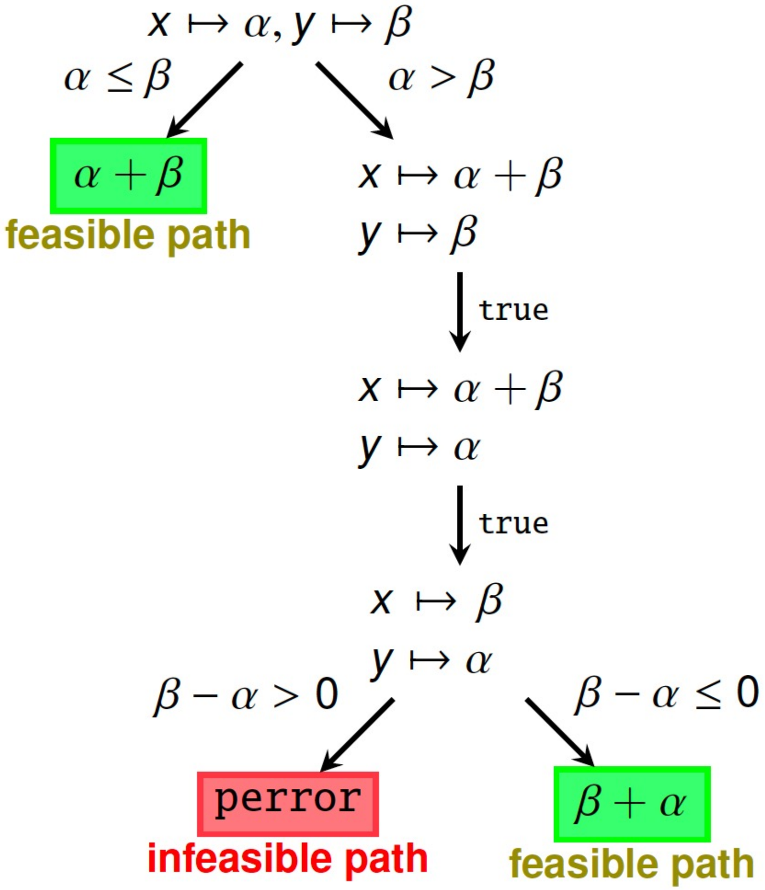
- **Path:** a path in the program's (interprocedural) CFG
- **Feasible path:** if there is an entry to the program that covers the path
- **Infeasible path**
  - If there is no entry to the program that covers the path
  - Infeasible path does NOT imply dead code. However, dead code implies an infeasible path
  - In real software, a large part of the paths are infeasible
    - Escalation problem when it is necessary to cover a large number of infeasible paths

- **Path condition:**
  - Quantifier-free formula on symbolic inputs that encodes all branch decisions made so far
- **Execution tree**: shows all the feasible and infeasible paths in the program

3. Program Analysis: Symbolic Execution

Another example

```
int f(int x, int y)
{
    if(x > y)
    {
        x = x + y;
        y = x - y;
        x = x - y;
        if(x - y > 0)
            perror("Error!");
    }

    return x + y;
}
```

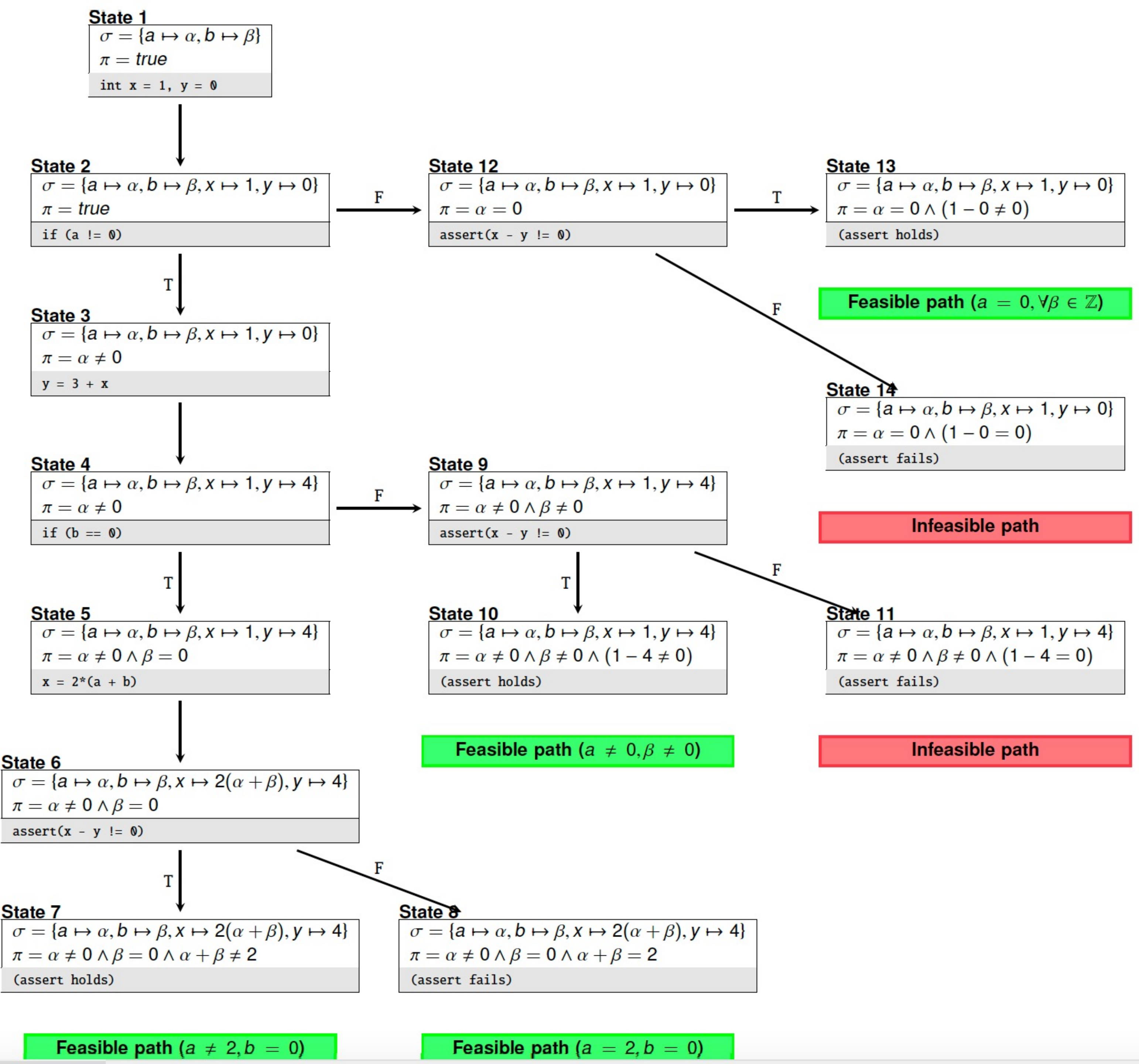


3. Program Analysis: Symbolic Execution

Terminology

- **State** of a symbolic execution engine: **(stmt; σ; π)**
- **stmt: next statement to evaluate**
- **σ : symbolic store**
  - Associates program variables with expressions of concrete values or symbolic values
- **π: path constraint**
  - Set of assumptions about the symbols due to the branches taken at execution to reach stmt
  - At the beginning, $\pi =$ true
- At any point, the symbolic state is described as the conjunction of these formulas
- **No need to keep track of infesiable paths during symbolic execution**

3. Pro cution



State 1
$\sigma = \{a \mapsto \alpha, b \mapsto \beta\}$
$\pi = true$
`int x = 1, y = 0`

State 2
$\sigma = \{a \mapsto \alpha, b \mapsto \beta, x \mapsto 1, y \mapsto 0\}$
$\pi = true$
`if (a != 0)`

T

State 3
$\sigma = \{a \mapsto \alpha, b \mapsto \beta, x \mapsto 1, y \mapsto 0\}$
$\pi = \alpha \neq 0$
`y = 3 + x`

State 4
$\sigma = \{a \mapsto \alpha, b \mapsto \beta, x \mapsto 1, y \mapsto 4\}$
$\pi = \alpha \neq 0$
`if (b == 0)`

T

State 5
$\sigma = \{a \mapsto \alpha, b \mapsto \beta, x \mapsto 1, y \mapsto 4\}$
$\pi = \alpha \neq 0 \wedge \beta = 0$
`x = 2*(a + b)`

State 6
$\sigma = \{a \mapsto \alpha, b \mapsto \beta, x \mapsto 2(\alpha + \beta), y \mapsto 4\}$
$\pi = \alpha \neq 0 \wedge \beta = 0$
`assert(x - y != 0)`

T

State 7
$\sigma = \{a \mapsto \alpha, b \mapsto \beta, x \mapsto 2(\alpha + \beta), y \mapsto 4\}$
$\pi = \alpha \neq 0 \wedge \beta = 0 \wedge \alpha + \beta \neq 2$
`(assert holds)`

Feasible path ($a \neq 2, b = 0$)

F

State 8
$\sigma = \{a \mapsto \alpha, b \mapsto \beta, x \mapsto 2(\alpha + \beta), y \mapsto 4\}$
$\pi = \alpha \neq 0 \wedge \beta = 0 \wedge \alpha + \beta = 2$
`(assert fails)`

Feasible path ($a = 2, b = 0$)

F

State 9
$\sigma = \{a \mapsto \alpha, b \mapsto \beta, x \mapsto 1, y \mapsto 4\}$
$\pi = \alpha \neq 0 \wedge \beta \neq 0$
`assert(x - y != 0)`

T

State 10
$\sigma = \{a \mapsto \alpha, b \mapsto \beta, x \mapsto 1, y \mapsto 4\}$
$\pi = \alpha \neq 0 \wedge \beta \neq 0 \wedge (1 - 4 \neq 0)$
`(assert holds)`

Feasible path ($a \neq 0, \beta \neq 0$)

F

State 11
$\sigma = \{a \mapsto \alpha, b \mapsto \beta, x \mapsto 1, y \mapsto 4\}$
$\pi = \alpha \neq 0 \wedge \beta \neq 0 \wedge (1 - 4 = 0)$
`(assert fails)`

Infeasible path

F

State 12
$\sigma = \{a \mapsto \alpha, b \mapsto \beta, x \mapsto 1, y \mapsto 0\}$
$\pi = \alpha = 0$
`assert(x - y != 0)`

T

State 13
$\sigma = \{a \mapsto \alpha, b \mapsto \beta, x \mapsto 1, y \mapsto 0\}$
$\pi = \alpha = 0 \wedge (1 - 0 \neq 0)$
`(assert holds)`

Feasible path ($a = 0, \forall \beta \in \mathbb{Z}$)

F

State 14
$\sigma = \{a \mapsto \alpha, b \mapsto \beta, x \mapsto 1, y \mapsto 0\}$
$\pi = \alpha = 0 \wedge (1 - 0 = 0)$
`(assert fails)`

Infeasible path

$, b \mapsto \beta\}$

$= 0$

Organizers:

GOBIERNO DE ESPAÑA
VICEPRESIDENCIA PRIMERA DEL GOBIERNO
MINISTERIO DE ASUNTOS ECONÓMICOS Y TRANSFORMACIÓN DIGI

universidad de león

3. Program Analysis: Symbolic Execution

Challenges

- **Path explosion:**
  - State space explosion
- **Modeling statements and environments:**
  - Interactions in the software stack
  - Handling of pointers, arrays, and other complex objects
- **Constraint solving:**
  - Complex combinations of constraints
  - Non-linear arithmetic

3. Program Analysis: Symbolic Execution

CFG + Symbolic Execution

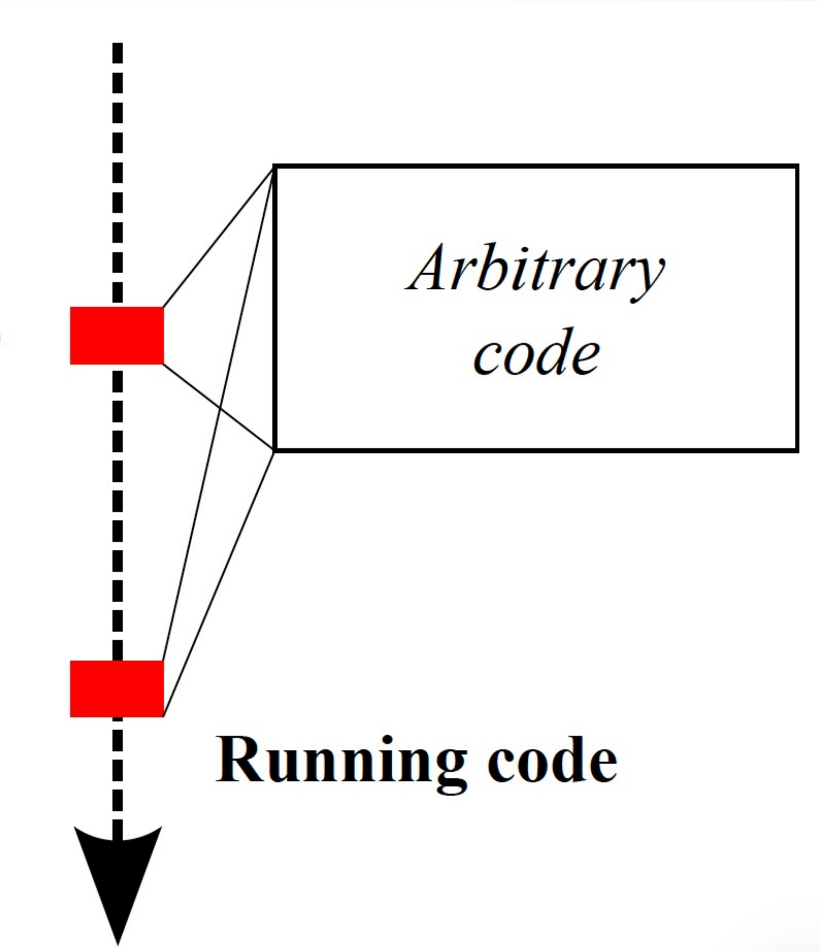
LAB SESSION 2

- **Additional files for *Lab session 2***
  - https://webdiis.unizar.es/~ricardo/sbc-2022/advanced-malware-analysis/laboratories/additional_files/lab2_malware_files.7z
- Follow the laboratory workbook provided on the workshop's website: https://webdiis.unizar.es/~ricardo/sbc-2022/advanced-malware-analysis/laboratories/lab2_cfg_symexec_analysis.pdf

3. Program Analysis Techniques: Dynamic Binary Instrumentation

3. Program Analysis: Dynamic Binary Instrumentation

- **Dynamic program analysis technique**
- __Goal:__ add arbitrary code during the execution of a program
    - **Instrumentation function**: what to do
    - **Instrumentation places**: where to do it
- **Different Dynamic Binary Instrumentation (DBI) engines**
    - Pin, Valgrind, DynamoRIO, etc…



3. Program Analysis: Dynamic Binary Instrumentation

DBI advantages and disadvantages

- **Advantages of binary instrumentation**
  - Programming language (totally) independent
  - Machine-mode vision
  - We can instrument proprietary software
- **Advantages of dynamic instrumentation**
  - No need to recompile/relink every time
  - Allow to find on-the-fly code
  - Dynamically generated code
  - Allow to instrument a process already running (attach)
- **Disadvantages:**
  - **Large overhead** (by instrumentation during execution)

3. Program Analysis: Dynamic Binary Instrumentation

The Pin Framework

- Developed by Intel, announced in 2005
- Supports Linux and Windows on 32-bit and 64-bit architectures
- Allows to insert **arbitrary C/C++ code in arbitrary places**
- **Components:**
  - **Pin:** instrumentation engine
  - **Pintool:** instrumentation tool
    - Uses the instrumentation engine to build something useful
    - Written in C/C++
    - Many examples shipped with Pin

3. Program Analysis: Dynamic Binary Instrumentation

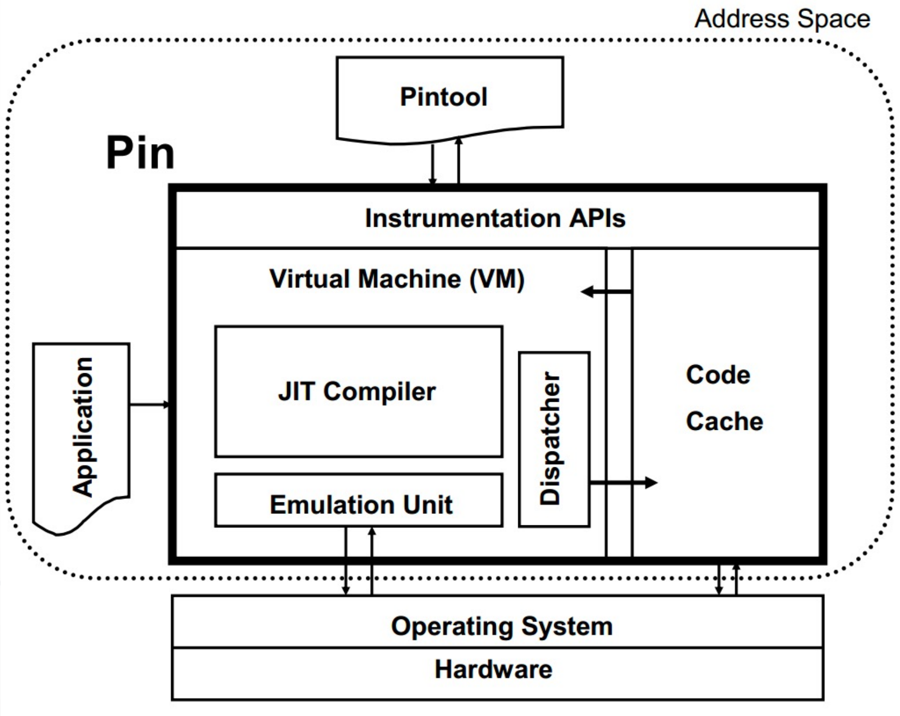
The Pin Framework: Types of APIs

- **Basic APIs**: independent of the architecture
    - Common functions (control-flow changes or memory accesses)
- **Architecture-specific API**: opcodes and operands
- **Call-based APIs**:
    - **Instrumentation routines**: defines WHERE the instrumentation is inserted
        - Only called on the first time
    - **Analysis routines**: defines WHAT to do when instrumentation is activated
        - Called every time the object is reached
    - **Callbacks routines**: called every time a certain event happens

3. Program Analysis: Dynamic Binary Instrumentation

The Pin Framework

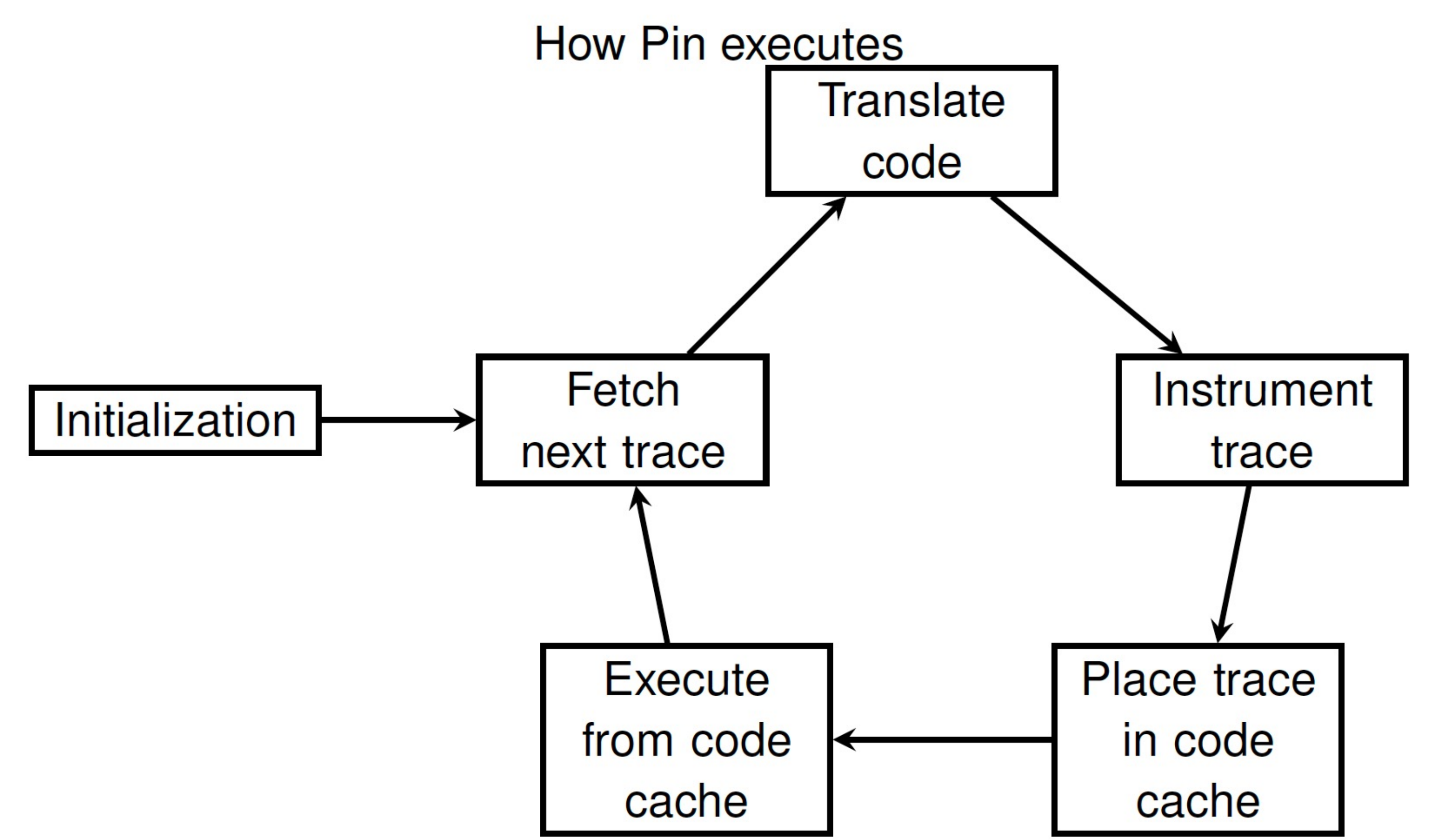
- JIT mode
    - Pin creates a modified copy of the application on-the-fly
    - Original code never executes
- Probe mode
    - Pin modifies the original application instructions
    - Inserts jumps to instrumentation code (trampolines)
    - Lower overhead, but less flexible approach



Organizers:
GOBIERNO DE ESPAÑA
VICEPRESIDENCIA PRIMERA DEL GOBIERNO
MINISTERIO DE ASUNTOS ECONÓMICOS Y TRANSFORMACIÓN DIGITAL
SECRETARÍA DE ESTADO DE DIGITALIZACIÓN E INTELIGENCIA ARTIFICIAL
incibe_
SPANISH NATIONAL CYBERSECURITY INSTITUTE
OAS More rights for more people
Partners:
Canada
AYUNTAMIENTO DE LEÓN
LEÓN CONGRESOS
universidad de león

3. Program Analysis: Dynamic Binary Instrumentation

The Pin Framework



3. Program Analysis: Dynamic Binary Instrumentation

The Pin Framework: Granularity

- **Low-level view**
  - Instruction (INS)
  - Basic block (BBL)
  - Trace (TRACE; also called Super basic block): single entry point, multiple exit points
- **Program-level view**
  - Routine (RTN)
  - Section (SEC)
  - Image (IMG)
- **System-level view**
  - Process
  - Thread
  - Exception
  - Syscalls

3. Program Analysis: Dynamic Binary Instrumentation

The Pin Framework: Instrumentation Points

- **IPOINT_BEFORE**
  - Insert a call before an instruction or routine
- **IPOINT_AFTER**
  - Insert a call on the fall through path of an instruction or return path of a routine
- **IPOINT_ANYWHERE**
  - Insert a call anywhere inside a trace or a BBL
- **IPOINT_TAKEN_BRANCH**
  - Insert a call on the edge taken of a branch, the side effects of the branch are visible

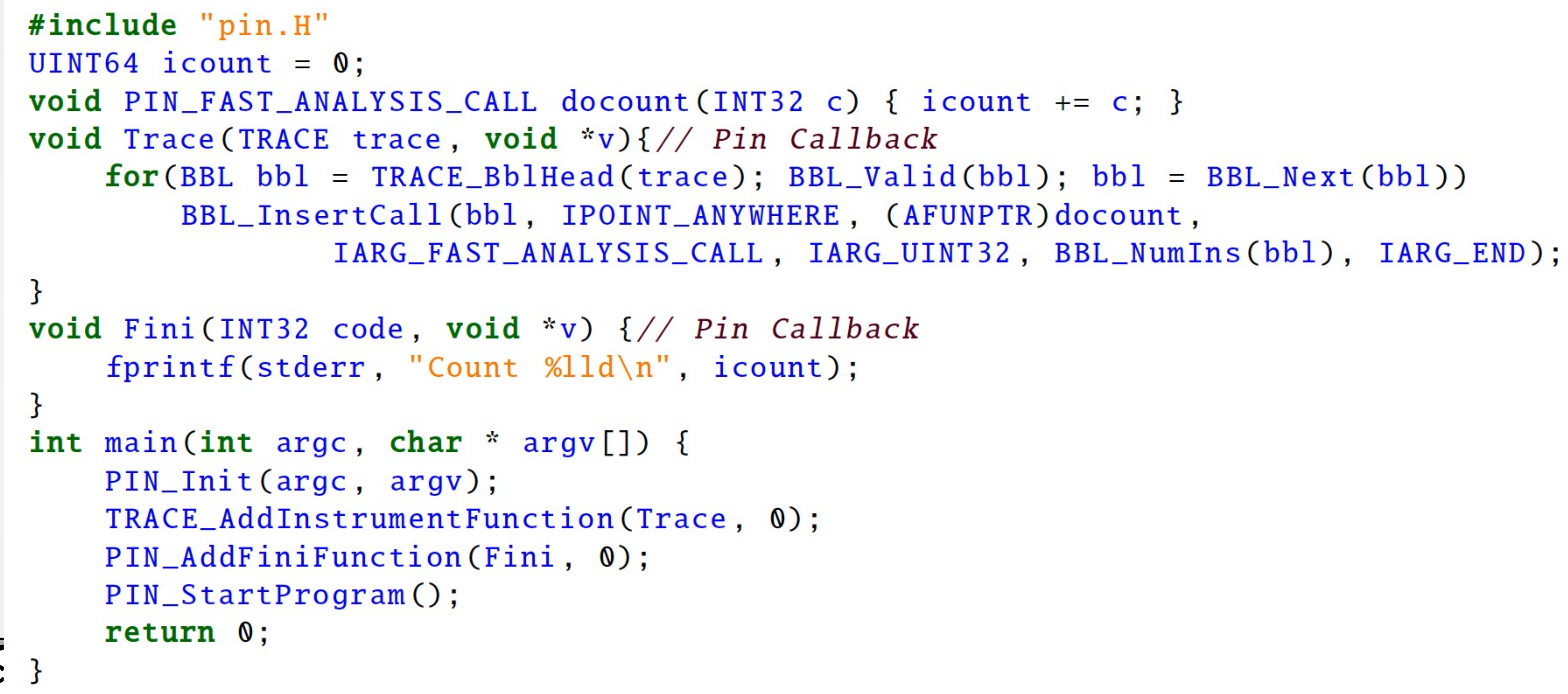
https://software.intel.com/sites/landingpage/pintool/docs/98484/Pin/html/group__INST__ARGS.html#ga707ea08e31f44f4a81e2a7766123bad7

3. Program Analysis: Dynamic Binary Instrumentation

The Pin Framework: Analysis Arguments

- **IARG_INST_PTR**: instruction pointer (program counter) value
- **IARG_UINT32 <value>**: **an integer value**
- **IARG_REG_VALUE <register name>**: value of the specified register
- **IARG_BRANCH_TARGET_ADDR**: target address of the instrumented branch
- **IARG_MEMORY_READ_EA**: effective address of a memory read
- These are just a few examples, **check the manual for all the possibilities**!

https://software.intel.com/sites/landingpage/pintool/docs/98484/Pin/html/group__INST__ARGS.html#ga089c27ca15e9ff139dd3a3f8a6f8451d



```
#include "pin.H"
UINT64 icount = 0;
void PIN_FAST_ANALYSIS_CALL docount(INT32 c) { icount += c; }
void Trace(TRACE trace, void *v){// Pin Callback
    for(BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl = BBL_Next(bbl))
        BBL_InsertCall(bbl, IPOINT_ANYWHERE, (AFUNPTR)docount,
                IARG_FAST_ANALYSIS_CALL, IARG_UINT32, BBL_NumIns(bbl), IARG_END);
}
void Fini(INT32 code, void *v) {// Pin Callback
    fprintf(stderr, "Count %lld\n", icount);
}
int main(int argc, char * argv[]) {
    PIN_Init(argc, argv);
    TRACE_AddInstrumentFunction(Trace, 0);
    PIN_AddFiniFunction(Fini, 0);
    PIN_StartProgram();
    return 0;
}
```

3. Program Analysis: Dynamic Binary Instrumentation

LAB SESSION 3

- **Additional files for *Lab session 3***
  - https://webdiis.unizar.es/~ricardo/sbc-2022/advanced-malware-analysis/laboratories/additional_files/lab3_malware_files.7z
- Follow the laboratory workbook provided on the workshop's website: https://webdiis.unizar.es/~ricardo/sbc-2022/advanced-malware-analysis/laboratories/lab3_dbi_analysis.pdf

#CyberSBC2022

5 to 15 July 2022
León, Spain

incibe.es/summer-bootcamp

Organizers:

GOBIERNO DE ESPAÑA
VICEPRESIDENCIA PRIMERA DEL GOBIERNO
MINISTERIO DE ASUNTOS ECONÓMICOS Y TRANSFORMACIÓN DIGITAL
SECRETARÍA DE ESTADO DE DIGITALIZACIÓN E INTELIGENCIA ARTIFICIAL

incibe_
SPANISH NATIONAL CYBERSECURITY INSTITUTE

OAS More rights for more people

Partners:

Canadä

AYUNTAMIENTO DE LEÓN

LEÓN CON GRE SOS

universidad de león