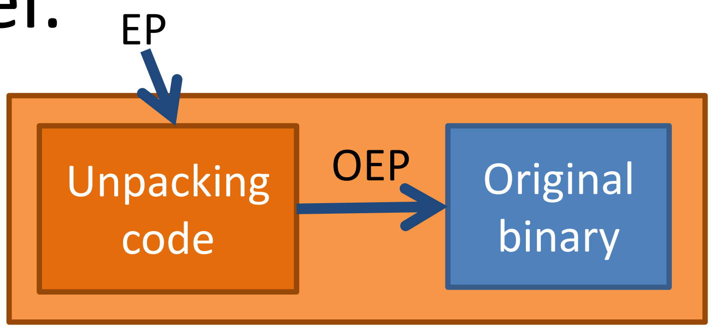
Tripoux: Reverse-Engineering Of Malware Packers For Dummies

Joan Calvet – j04n.calvet@gmail.com

Deepsec 2010

The Context (1)

- A lot of malware families use home-made packers to protect their binaries, following a standard model:



- The unpacking code is automatically modified for each new distributed binary.

The Context (2)

- Usually **people are only interested into the original binary**:

    1. It’s where the “real” malware behaviour is.

    2. It’s hard to understand packers.

The Context (3)

- **But developing an understanding of the unpacking code** helps to:
  - Get an easy access to the original binary (sometimes “generic unpacking algorithm” fails..!)
  - Build signatures (malware writers are lazy and there are often common algorithms into the different packer’s instances)
  - Find interesting pieces of code: checks against the environment, obfuscation techniques,...

The Question

Why the **human analysis** of such packers is difficult, especially for beginners ?

When trying to understand a packer, we can not just sit and observe the API calls made by the binary:

- This is only a small part of the packer code
- There can be useless API calls (to trick emulators,sandboxes...)

We have to dig into the assembly code, that brings the first problem...

Problem 1: x86 Semantic

- The x86 assembly language is pretty hard to learn and manipulate.

- Mainly because of inexplicit side-effects and different operation semantics depending on the machine state (operands, flags):

`MOVSB`

Read ESI, Read EDI, Read [ESI], Write [EDI]
If the DF flag is 0, the ESI and EDI register are **incremented**
If the DF flag is 1, the ESI and EDI register are **decremented**

Problem 1: x86 Semantic

- When playing with standard code coming from a compiler, you only have to be familiar with a small subset of the x86 instruction set.

- But we are in a different world...

Problem 1: x86 Semantic

Example : Win32.Waledac's packer

004040F5	0F48C9	CMOVS ECX,ECX
004040F8	0FBFC9	MOVSX ECX,CX
004040FB	81C7 490100	ADD EDI,149
00404101	0FBFC9	MOVSX ECX,CX
		FCOM EDI
		FCOMP EAX

00404136	81DB 4CA9AF	SBB EBX,32AFA94C
0040413C	81DB 18114A	SBB EBX,324A1118
00404142	C1DB E4	
00404145	C1DB B0	
00404148	FFF3	
0040414A	58	
0040414B	0F43C0	
0040414E	0F43C0	CMOVNB EAX,EAX
00404151	0FB6C0	MOVZX EAX,AL
00404154	0FB6C0	MOVZX EAX,AL
00404157	DCE0	FSUBR ST,ST
00404159	DCE2	FSUBR ST(2),ST
0040415B	DCE4	FSUBR ST(4),ST

004041FF	D6	SALC
00404200	6A F1	PUSH -0F
00404202	89F0	MOV EAX,ESI
00404204	DCC2	FADD ST(2),ST

Problem 2: Amount Of Information

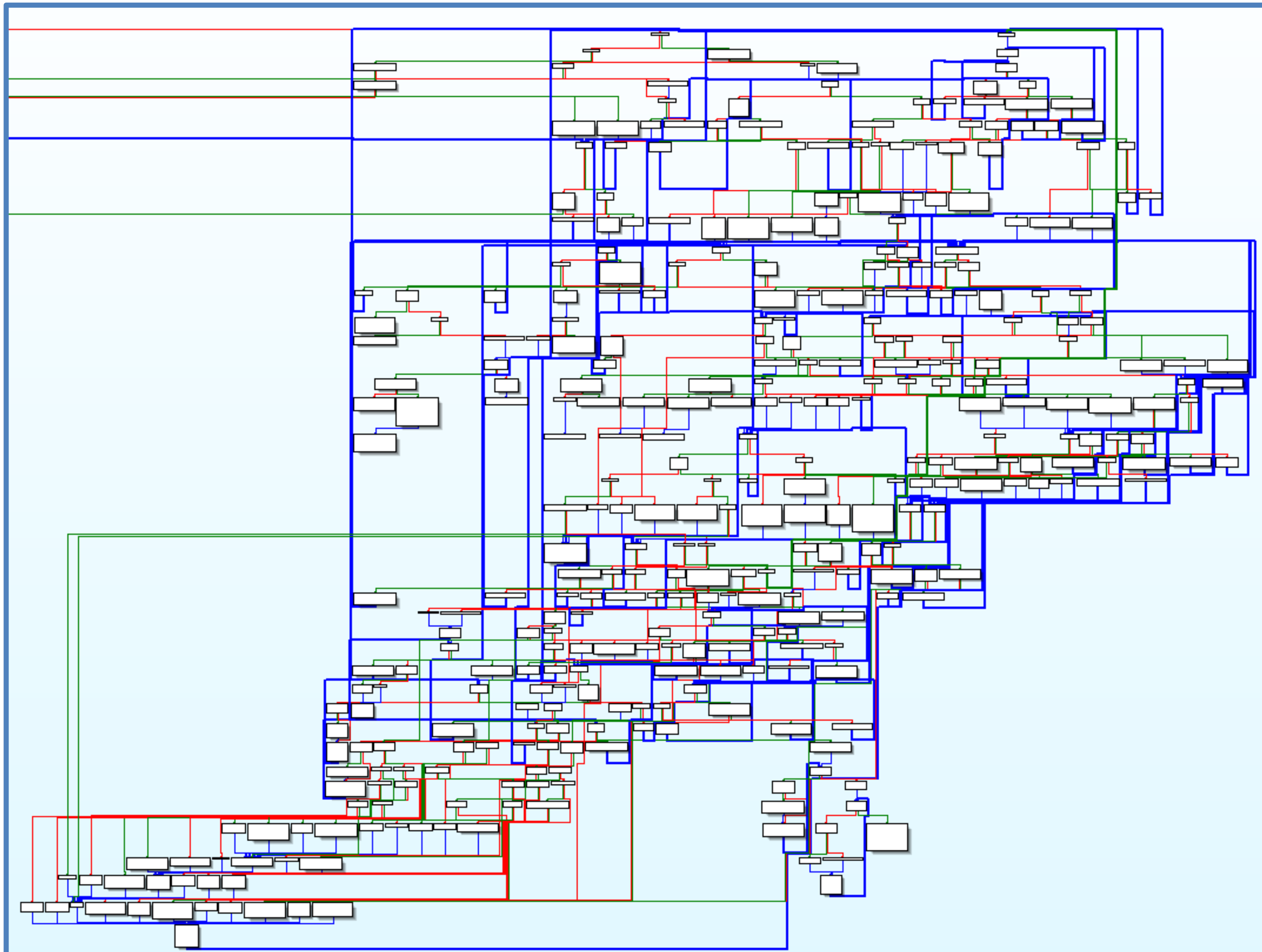
- Common packed binaries have several million instructions executed into the protection layers.

- Unlike standard code, we can not say that each of these line has a purpose.

- It's often very hard to choose the right abstraction level when looking at the packed binary:
"Should I really understand all these lines of code ?"

Problem 2: Amount Of Information

Example : Win32.Swizzor's packer



Problem 3: Absence Of (easily seen) High-Level Abstractions

- We like to “divide and conquer” complicated problems.
- In a standard binary:

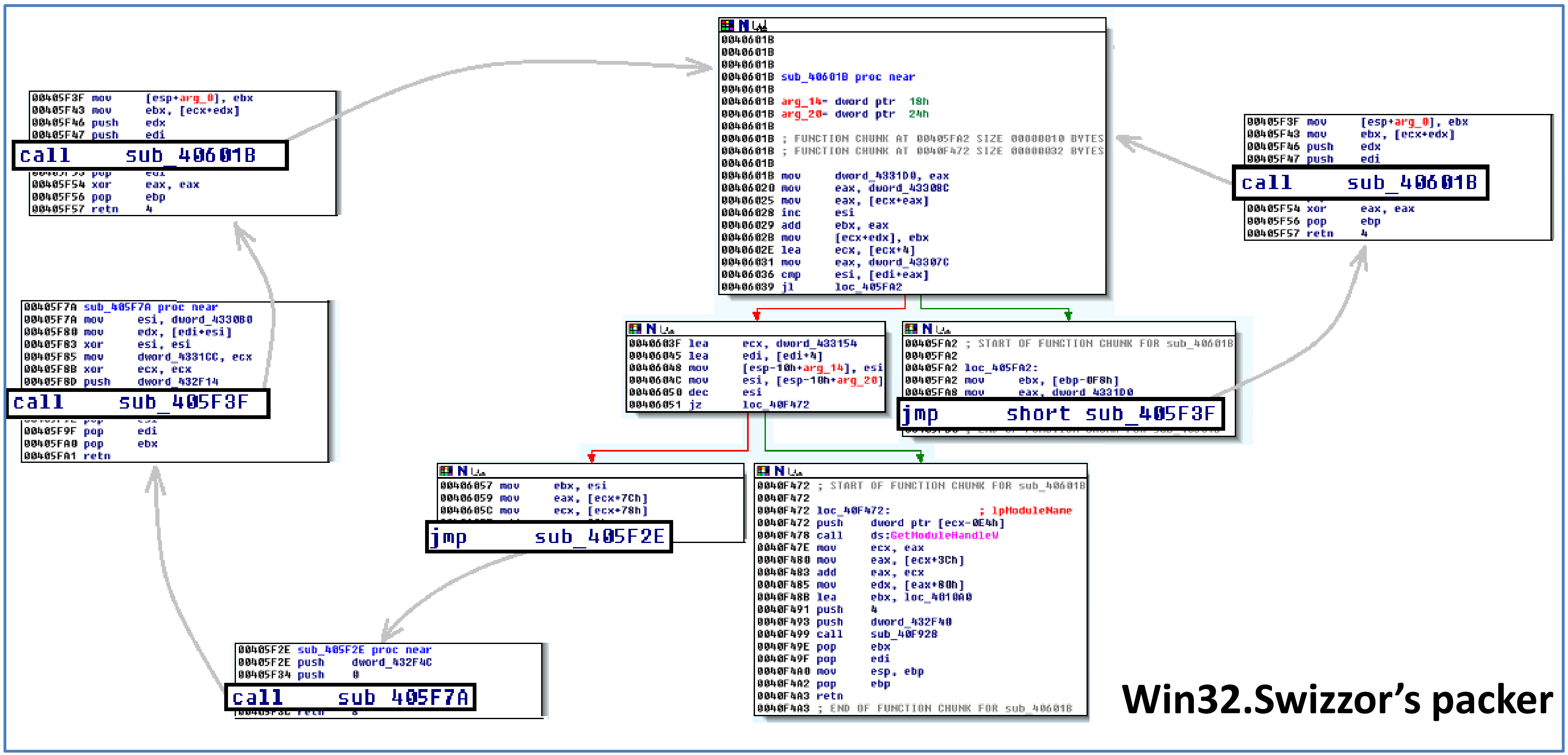
```
push    esi
push    dword ptr [eax]
call    sub_1011990
test    eax, eax
jnz     loc_1011C7B
```

```
sub_1011990 proc near

arg_0= dword ptr  8
arg_4= dword ptr  0Ch

push    ebp
mov     ebp, esp
mov     edx, [ebp+arg_0]
mov     ecx, [ebp+arg_4]
 ...
```

This is a function! We can thus consider the code inside it as a “block” that shares a common purpose

Problem 3: Absence Of (easily seen) High-Level Abstractions

- But in our world, we can have:



Problem 3: Absence Of (easily seen) High-Level Abstractions

- **No easy way left to detect functions** and thus divide our analysis in sub-parts.

- Also true for data: **no more high-level structures**, only a big array called memory.

The Good News

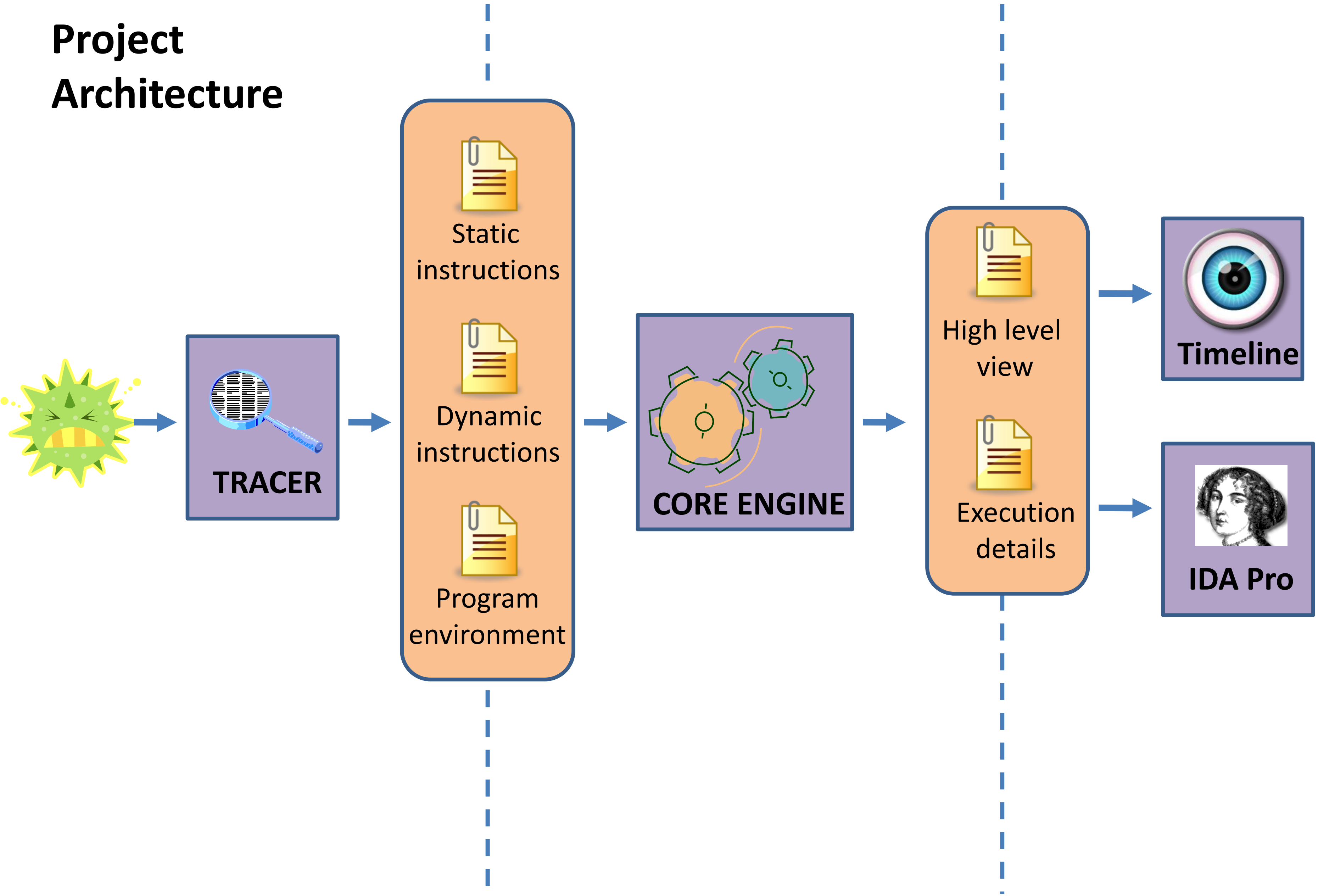
- Most of the time there is **only one "interesting" path inside the protection layers** (the one that actually unpacks the original binary).

- It's **pretty easy to detect that we have taken the "good" path**: suspicious behaviour (network packets, registry modifications...) that indicate a successful unpacking.

Proposed Solution

- Let's use this fact and adopt a **pure dynamic analysis approach**:
  - **Trace** the packed binary and collect the x86 side-effects (address problem 1)
  - Define an **intermediate representation** with some high level abstractions (address problem 3)
  - Build some **visualization tools** to easily navigate through the collected information (address problem 2)

Project Architecture



How to collect a maximum of information about the malware execution ?

STEP 1: THE TRACER

Tracing Engine (1)

- **Pin**: dynamic binary instrumentation framework:
  - Insert arbitrary code (C++) in the executable (JIT compiler)
  - Rich library to manipulate assembly instructions, basic blocks, library functions…
  - Deals with self-modifying code

- Check it at http://www.pintool.org/

- **But what information do we want to gather at run-time ?**

Tracing Engine (2)

1. Detailed description of the executed x86 instructions

- Binary code, address, size
- Instruction “type”:
  - (Un)Conditional branch
  - (In)Direct branch
  - Stack related
  - Throws an exception
  - API call
  - ...
- Data-flow information :
  - Memory access (@ + size)
  - Register access
- Flags access: read and possibly modified

Make post-analysis easier

Make side-effects explicit (Problem 1!)

Tracing Engine (3)

2. Interactions with the operating system:

– The “official” way: API function calls

- **We only trace the malware code** thanks to API calls detection (dynamically and statically linked libraries).
- **We dump the IN and OUT arguments of each API call**, plus the return value, thanks to the knowledge of the API functions prototypes.

– The “unofficial” way: **direct access to** user land Windows structures like **the PEB and the TEB**:

- We gather their base address at runtime (randomization!)

Tracing Engine (4)

3. Output:

1: Dynamic instructions file

Time	Address	Hash	Effects
1	0x40100a	0x397cb40	RR_ebx_eax WR_ebx
2	0x40100b	0x455e010	RM_419c51_1 RR_ebx
...			

2: Static instructions file

Hash	Length	Type	W Flags	R Flags	Binary code
0x397cb40	1	0	0	8D4	43
0x455e010	1	60	0	0	5E
...					

Tracing Engine (5)

3. Output:

3: Program environment

Type	Module name	Address
DOSH	ADVAPI32.DLL	77da0000
PE32H	ADVAPI32.DLL	77da00f0
PE32H	msvcrt.dll	77be00e8
DOSH	DNSAPI.dll	76ed0000
PEB	0	7ffdc000
TEB	0	7ffdf000
...		

STEP 2: THE CORE ENGINE

The Core Engine (1)

- Translate the tracer output into something usable.

- Set up some high-level abstractions onto the trace (Problem 3):
  - Waves
  - Loops

The Core Engine (2)

<u>1. Waves:</u>

- **Represent a subset of the trace where there is no self-modification code**:

  Two instructions i and j are in the same wave if i doesn't modify j and j doesn't modify i.

- Easy to detect in the trace:
  - Store the written memory by each instruction.
  - If we execute a written instruction: end of the current wave and start of a new wave.

The Core Engine (3)

2. Loops:

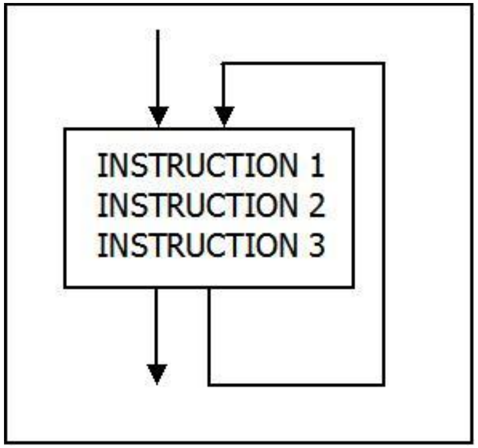
- **Instructions inside a loop have a common goal**: memory decryption, research of some specific information, anti-emulation...
- Thus they are good candidate for abstraction!

- **But how to detect loops ?**

The Core Engine (4)

2. Loops:

(SIMPLIFIED) STATIC POINT OF VIEW



TRACE POINT OF VIEW

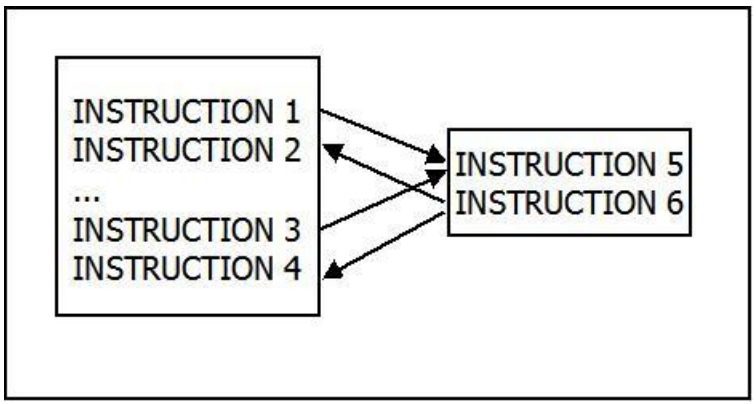
EXECUTED	TIME
INSTRUCTION1	1
INSTRUCTION2	2
INSTRUCTION3	3
INSTRUCTION1	4
INSTRUCTION2	5
…	…

When tracing a binary, can we just define a loop as the repetition of an instruction ?

The Core Engine (5)

2. Loops:

(SIMPLIFIED) STATIC POINT OF VIEW



TRACE POINT OF VIEW

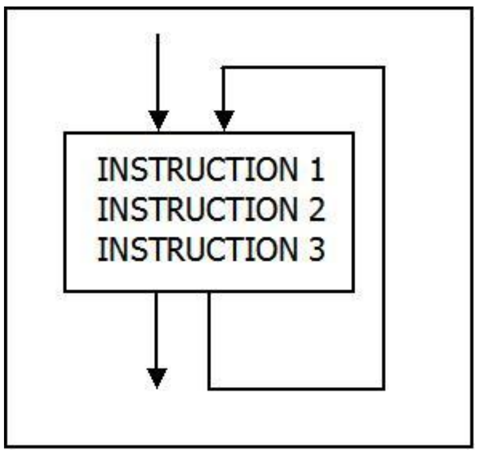
EXECUTED	TIME
INSTRUCTION1	1
INSTRUCTION5	2
INSTRUCTION6	3
INSTRUCTION2	4
…	…
INSTRUCTION3	5
INSTRUCTION5	6
INSTRUCTION6	7

This is not a loop ! So what's a loop ?

The Core Engine (6)

2. Loops:

(SIMPLIFIED) STATIC POINT OF VIEW



TRACE POINT OF VIEW

EXECUTED	TIME
INSTRUCTION1	1
INSTRUCTION2	2
INSTRUCTION3	3
INSTRUCTION1	4
INSTRUCTION2	5
INSTRUCTION3	6
INSTRUCTION1	7
…	…

What actually define the loop, is the back edge between instructions 3 and 1.

The Core Engine (7)

2. Loops:

- Thus we detect loops by looking for back edges inside the trace.

- Information collected about the loops:
  - Number of iterations
  - Read memory access
  - Write memory access
  - Multi-effects instructions (instructions with different effects at each loop turn)

Clusters (Read memory access, Write memory access)

The Core Engine (8)

- In addition to all the events gathered by the tracer (API calls, exceptions, system access...) the core engine also detects:
  - **Conditional or Indirect branch that always jump to the same target** (and that can thus be considered as <u>unconditional direct branch</u>)

The Core Engine (9)

Output:

1: High level view

```
[=> EVENT: API CALL <=]
[TIME: 36][@: 0x40121b]
[D_LoadLibraryA]
[A1:LPCSTR "shlwapi.dll"]
[RV:HMODULE 0x77f40000]

[=> EVENT: LOOP <=]
[START: 4cc620 - END: 4cc654]
[H: 0x21d21cd - T: 0x21d21ca]
| TURN : 2
| READ AREAS : [0x12feec-0x12fef3: 0x8 B]
| WRITE AREAS : [0x410992-0x410993: 0x2 B]
| DYNAMIC PROFILE : 0x21d21ed - 0x21d21ef
...
```

2: Full wave dumps

```
401070    55
401071    29d5
401073    4d
401074    89e5
...
```

How to avoid the Problem 2 and deal easily with all the collected information ?

STEP 3 : VISUALIZATION PART

High-Level View Of The Execution

- Provide a big picture of the trace, plus some analysis tools.
- Build with the “Timeline” widget from the MIT:

  http://www.simile-widgets.org/timeline/

DEMO 1

Low-Level View Of The Execution

- When you need to dig into the code.

- Use IDA Pro (and IDA Python) to display the output of the core engine with the information gathered dynamically (one wave at time!).

DEMO 2

Example : Win32.Swizzor's packer

```
00406650
00406650
00406650
0040665C sub_40665C proc near
0040665C
0040665C arg_0= dword ptr  4
0040665C arg_4= dword ptr  8
0040665C arg_8= dword ptr  0Ch
0040665C arg_C= dword ptr  10h
0040665C
0040665C push    384h
00406661 push    0
00406663 push    dword_432F14
00406669 push    3E8h
0040666E push    eax
0040666F call    sub_406574
```

```
00406674
00406674 loc_406674:
00406674 mov     [esp+arg_8], eax
00406678 mov     [esp+arg_0], edi
0040667C mov     dword_433114, edx
00406682 mov     dword_43312C, ecx
00406688 xor     edx, edx
0040668A add     edx, [ebp+0Ch]
0040668D mov     edi, [edx+0C4h]
00406693 mov     ecx, [edi+14h]
00406696 mov     eax, [ebx+ecx]
00406699 mov     ecx, [edi+120h]
0040669F mov     [esp+arg_C], eax
004066A3 mov     [esp+arg_4], ecx
004066A7 mov     eax, [edi+120h]
004066AD jmp     dword_432EE0
004066AD sub_40665C endp
004066AD
```

```
mov     [esp+18h], eax
lea     eax, ds:0[eax*2]
mov     dword_433110, eax
mov     eax, [esp+10h]
mov     edx, [esp+0Ch]
xor     eax, edx
mov     edi, ebx
lea     ebx, [ebx+4]
mov     [esp+1Ch], edi
mov     dword_43313C, eax
mov     ebp, [ebp+0Ch]
mov     eax, [ebp+0C4h]
mov     edi, [eax+14h]
xor     eax, eax
add     eax, [ebx+edi]
mov     [esp+20h], eax
mov     eax, dword_43312C
mov     ebx, [esp+20h]
xor     ebx, eax
mov     edi, 1
sub     edi, 1
mov     dword_4330F8, 2
mov     ecx, edi
jmp     dword_432F50
```

```
mov     eax, dword_4330F8
mov     dword_433144, eax
mov     ebp, esp


dec     eax
test    eax, eax
jnz     loc_406822
mov     edi, dword_43313C
mov     ecx, [esp+2DCh]
mov     [ebp+8], edi
mov     ecx, [ecx+0C4h]
jmp     dword_432F94
```

```
xor     edx, ebx
sub     dword_433158, edx
mov     dword_433128, eax
mov     eax, [esp+2DCh]
mov     ebx, [eax+0C4h]
mov     ecx, [ebx+7Ch]
mov     dword_433140, esi
xor     esi, esi
or      esi, dword_43313C
shr     esi, cl
mov     eax, dword_433110
xor     esi, eax
sub     dword_433158, esi
xor     edx, edx
or      edx, dword_433158
jmp     dword ptr [ebx+0E8h]
```

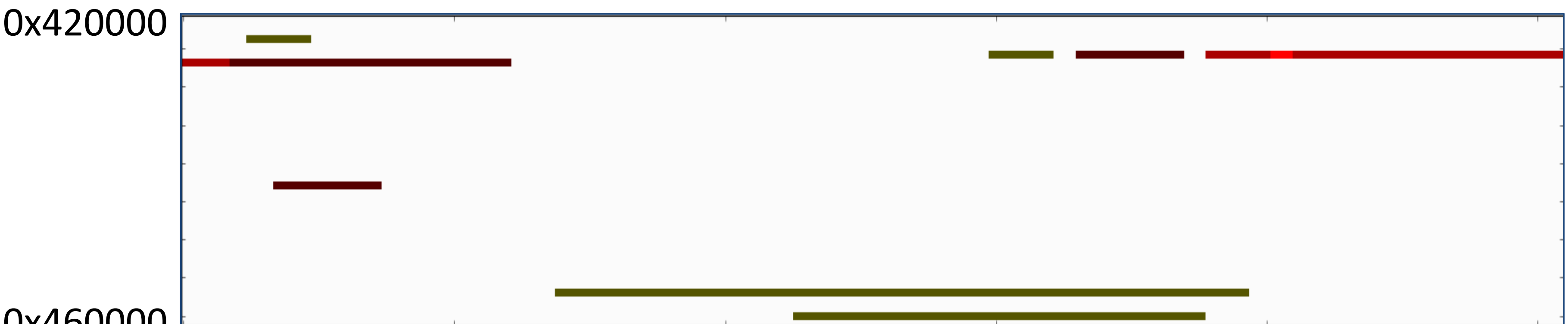
IDA fails to find all the JMP targets !

And so on for the next 6 basic blocs...

DEMO 3

Work In Progress (1)

- Address the lack of high level abstraction for data by dynamic typing:
  (#Read, #Write, #Execution) for each memory byte



A loop inside the Swizzor's packer

```
0x4284be - [5 0 0]
0x4284ca - [3 0 0]
0x4284ce - [2 0 0]
0x42850a - [1 0 0]
0x4284a8 - [1 0 0]
0x42951f - [1 0 0]
0x428212 - [1 1 0]
0x428499 - [1 1 0]
0x42a360 - [1 1 0]
0x42a678 - [1 1 0]
```

Allows some pretty efficient **heuristic rules**:

-The key is read 5 times because there are 5 decrypted areas by the loop.
- The decrypted areas are read 1 time and written 1 time.
-...

Work In Progress (2)

- Define a real framework for trace manipulation:
  - Slicing
  - Data Flow
  - De-obfuscation
  - …

- Allow the user to create his own abstractions on the trace (loops and waves are not always suitable!).

- Set up sandbox analysis to provide the visualization parts to the user ?

- Test, test, test.

Thanks!

- Source code and binaries are available here:

  http://code.google.com/p/tripoux/

- This is really a 0.1 version of the project, **any remark/advice is welcome** !

- If you are interested, follow the updates @joancalvet

- Thanks to: Pierre-Marc Bureau, Nicolas Fallière and Daniel Reynaud.