Optimize binary tracing: example with an ECDSA implementation

Guillaume VINET

19th May 2019

WhibOx 2019

- White-Box Cryptography (WBC) security analyses are becoming more mature
- Tracing the binary execution is now part of the state of the art
- Tracing can be very powerful if the WBC code is not adequatly protected.
- Nowadays, a state-of-the-art analysis requires to:
  - Optimize the data tracing to overcome the data size, disk space and heavy processing issues
  - Focus the analyses to specific instructions or parts of the code
  - Cover a large space of different attacks

How can this be achieved?

eShard

esDynamic

localhost:8000/lab

ESHARD | News | Android | Divers | Reading | TO_SEE | Ebook | hardening | Tools | awesome | LPC1769 | Crypto | Latex | NFC | Vendeurs | AfterEffect | Conferences | EBIOS

esDynamic

› eshard-notebooks › sva

Name	Last Modified
0-SimulatedVulnerabilityAnalysis	2 hours ago
1-TracerFrameworks	2 hours ago
2-FaulterFrameworks	2 hours ago
3-SpecialUseCases	a month ago
HOW	2 hours ago
What is inside.ipynb	a month ago
SVAModuleQuickStart.pdf	a month ago
SVANotebooksList.pdf	a month ago

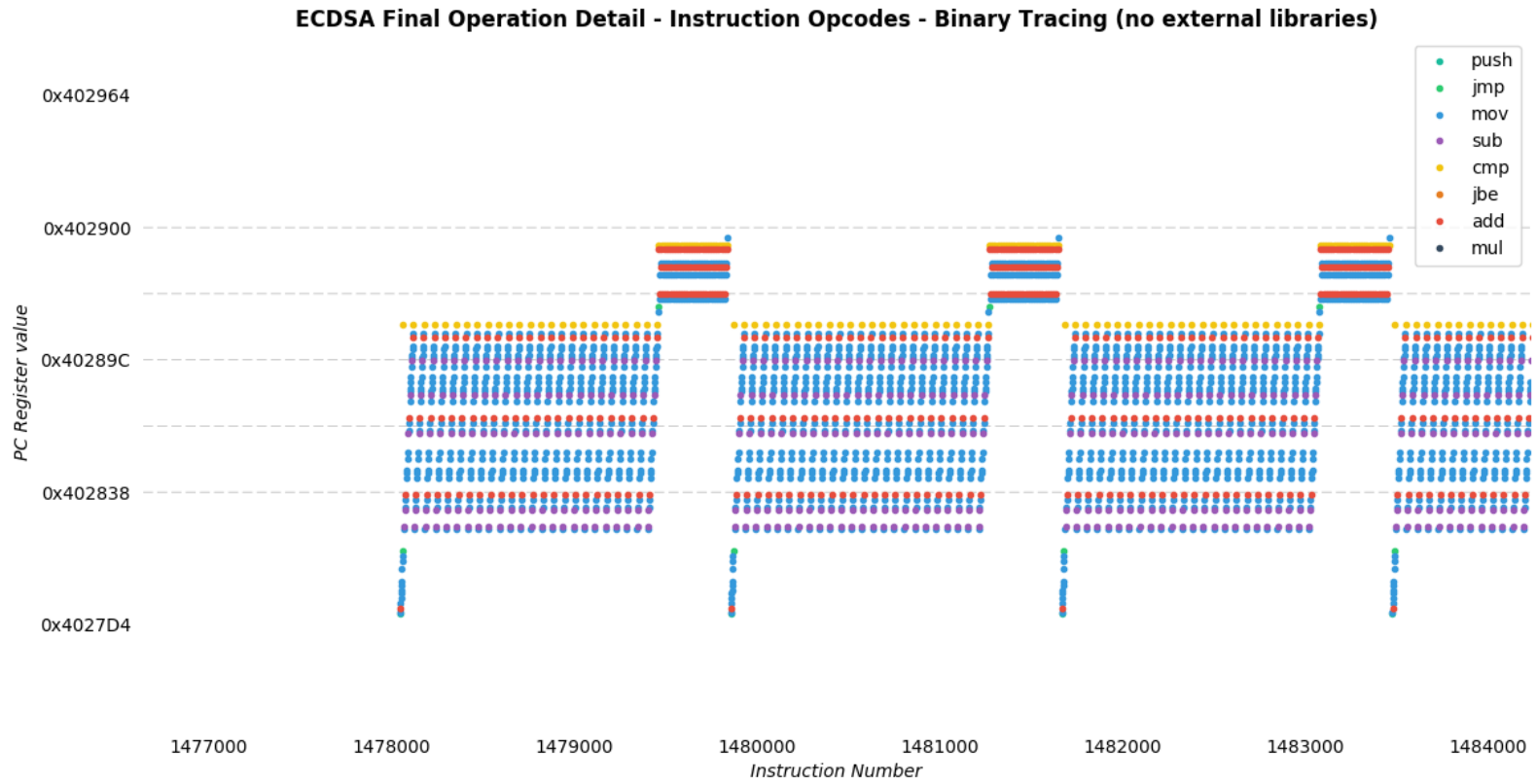
File Edit View Run Kernel Tabs Settings Help

tracer_presentation.ipynb

Code

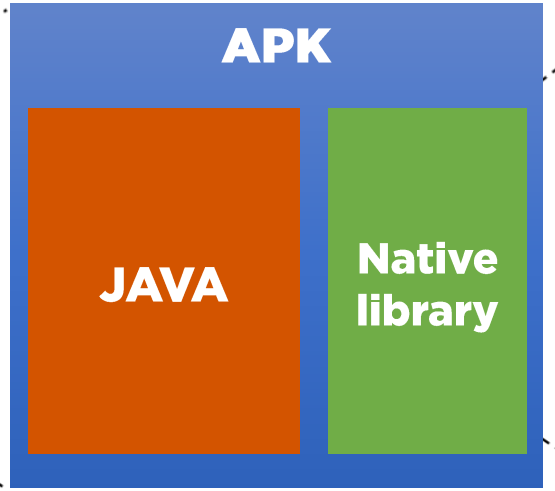
Python 3

Optimize your binary tracing: an example with an ECDSA implementation



WHERE IS THE WHITE-BOX?

Target the application → **Reverse Engineering** → **White-Box Extraction**



Secure functions ("secure VM")

White-Box Cryptography Library

Native binary file (assembly code)

NO SOURCE FILES!

It is not possible to add a printf or comment a line to see what happens

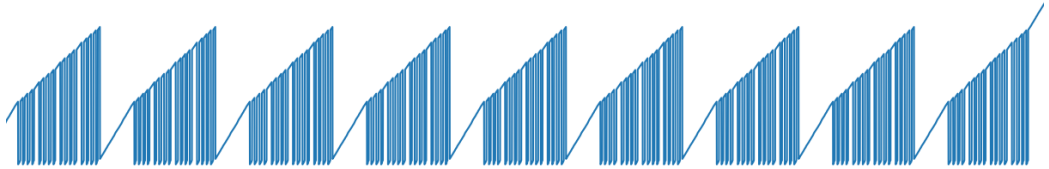
Which protections?

eShard

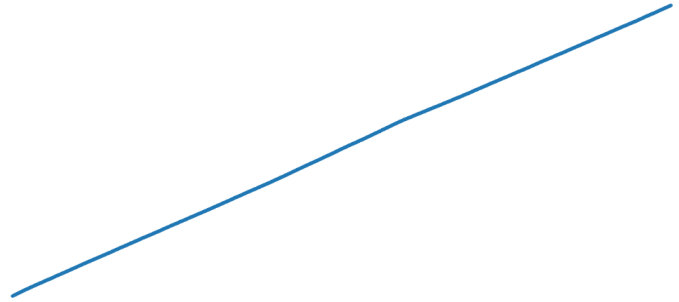
WHICH PROTECTIONS?

Native
binary file

Algorithm level
obfuscation



AES trace acquisition
with visible rounds



Another AES trace acquisition
but with no pattern

eShard

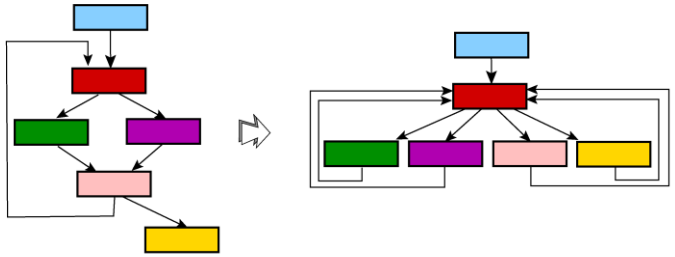
WHICH PROTECTIONS?

Native binary file

Algorithm level obfuscation

Program level obfuscation

- Control flow
- Data obfuscation
- Preventive transformations



Control flow flattening

Illustration: http://tigress.cs.arizona.edu/transformPage/docs/flatten/index.html

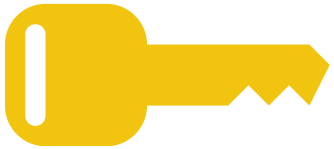
WHICH PROTECTIONS?

Native binary file

Algorithm level obfuscation

Program level obfuscation

HOW TO GET THE SECRET KEY?

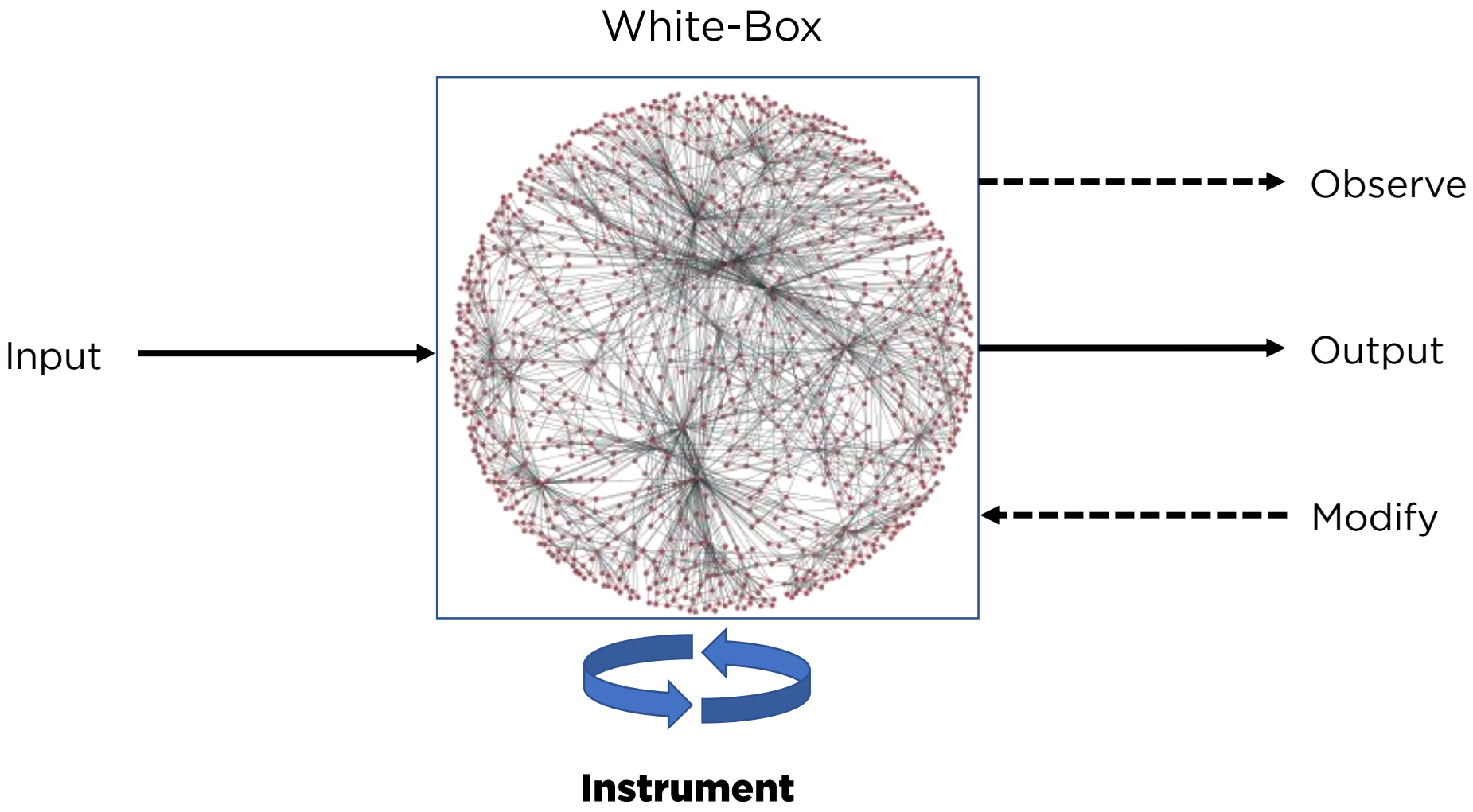


eShard

How to attack a White-Box

eShard

HOW TO ATTACK A WHITE-BOX?



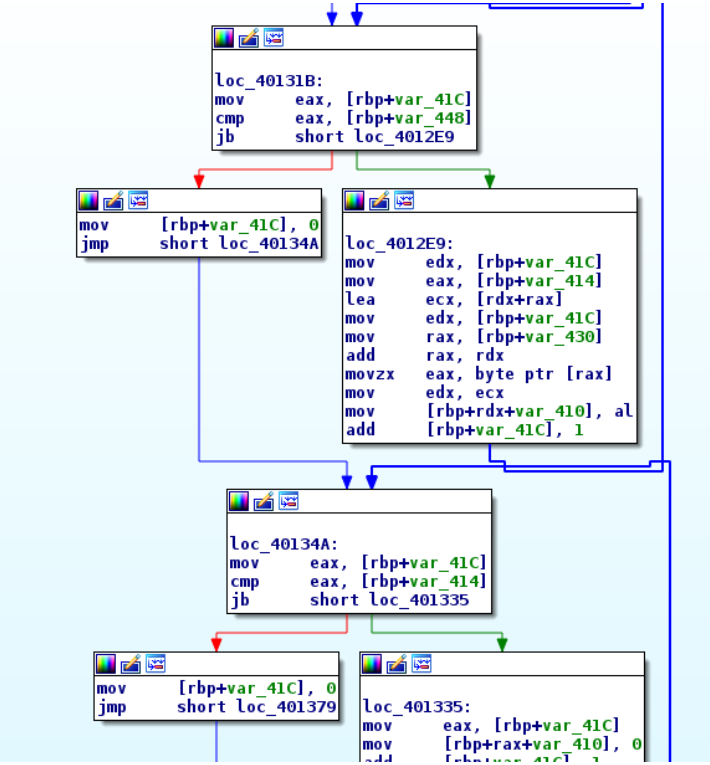
eShard

HOW TO ATTACK A WHITE-BOX?

Option 1: Reverse engineering

Assets:

- White-Box algorithm recovery (Industrial Property)



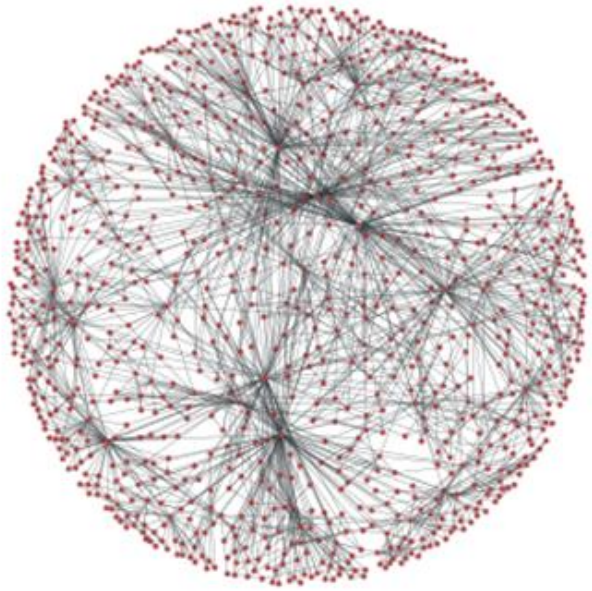
Drawbacks:

- Elapsed time: from several weeks to several months (if they are good protections)
- Expertise:
  - Multiple experts: reverse engineering & cryptography

eShard

HOW TO ATTACK A WHITE-BOX?

Option 2: White-Box Instrumentation

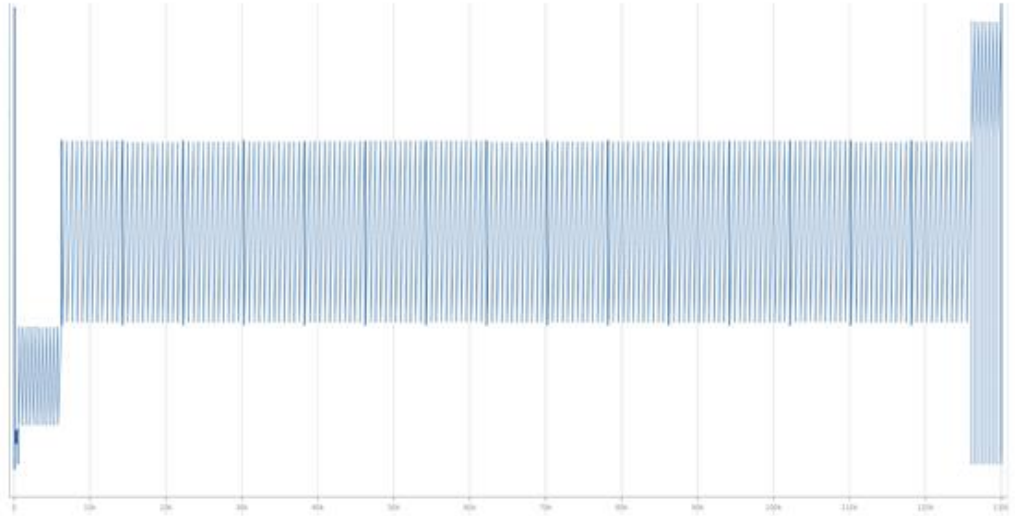


White-Box

Dynamic Binary Instrumentation Tool

→

to generate acquisition trace



Key recovery with statistical tools
Differential Computation Analysis (DCA)

Unboxing the White-Box - Practical attacks against Obfuscated Ciphers
Eloi Sanfelix, Cristofaro Mune and Job de Haas
Black Hat 2015

Differential Computation Analysis Hiding your White Box Designs is Not Enough
Joppe W. Bos, Charles Hubain, Wil Michiels and Philippe Teuwen
CHES 2016

eShard

HOW TO ATTACK A WHITE-BOX?

Option 2: White-Box Instrumentation

Assets:

- Elapsed time: from several hours to several weeks
- Expertise:
  - Expert: cryptography

Drawbacks (coming from binary obfuscation):

- White-Box algorithm not recovered
- Big trace size
- Time of acquisition

How to execute the White-Box?
What to monitor?

eShard

THE CHALLENGE

- We trace directly the White-Box without reverse engineering.
- We will obtain big trace size

THE TRACE SIZE
Is-it possible to obtain directly small traces without losing leakage information?

How to execute the White-Box?
What to monitor?

eShard

File Edit View Run Kernel Tabs Settings Help

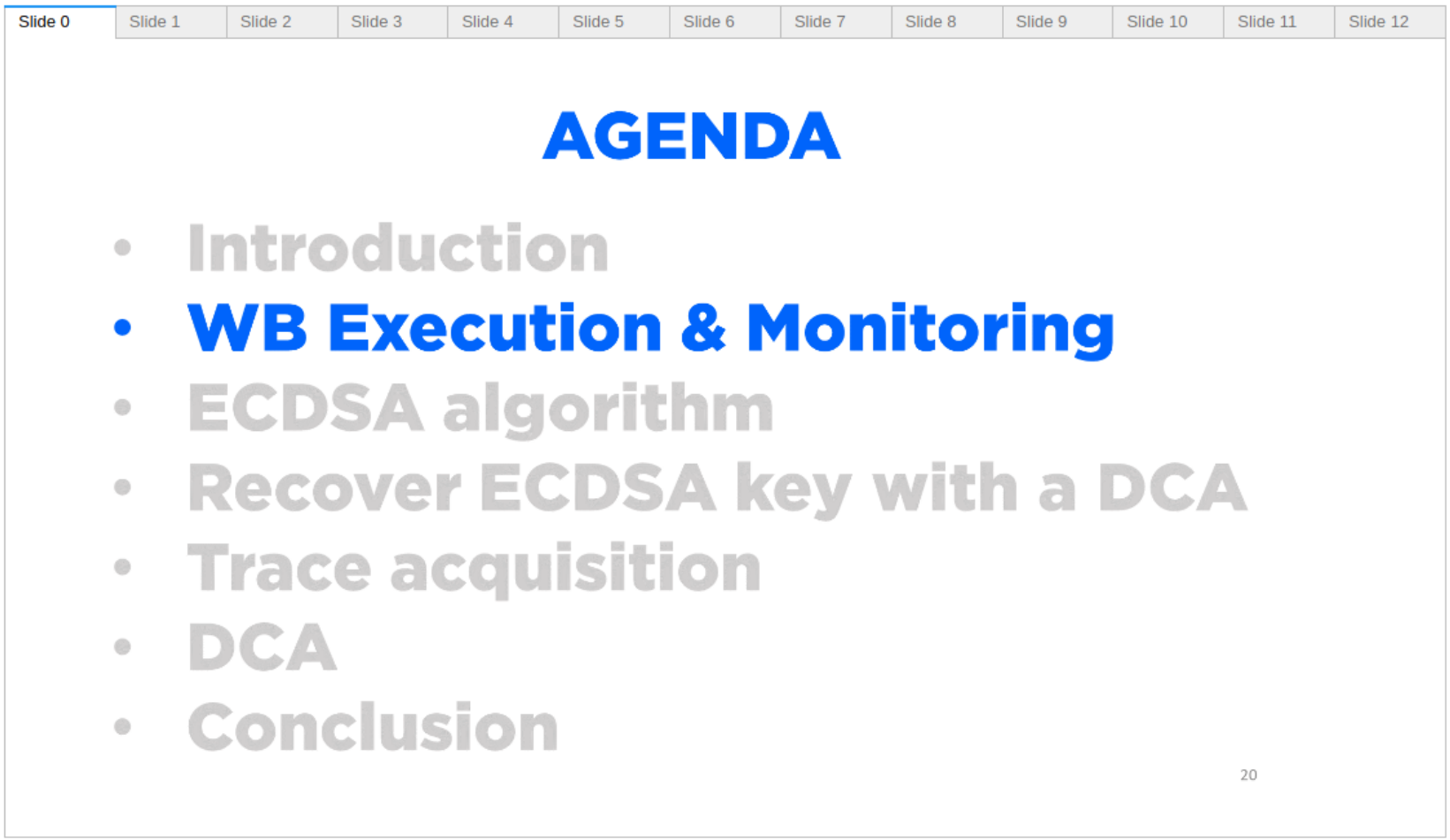
Code Python 3

White-Box Execution & Monitoring

[3]:
```
slideshow('WB Execution & Monitoring',
          './img/presentation_tracer',
          ['Diapositive%d.PNG' % hit for hit in list(range(20,33))],
          slide_width)
```

▾ **WB Execution & Monitoring**

Slide 0 | Slide 1 | Slide 2 | Slide 3 | Slide 4 | Slide 5 | Slide 6 | Slide 7 | Slide 8 | Slide 9 | Slide 10 | Slide 11 | Slide 12



eShard

EXECUTION & MONITORING

Tracer

included in

Side Channel Marvels Framework

https://github.com/SideChannelMarvels

Memory Access monitoring:

- Read/Write value
- Program Counter
- Kind of operation

Pin

X86, x86_64

Valgrind

X86, x86_64, ARM support

eShard

EXECUTION & MONITORING

Tracer

instruction

Reading

Writing

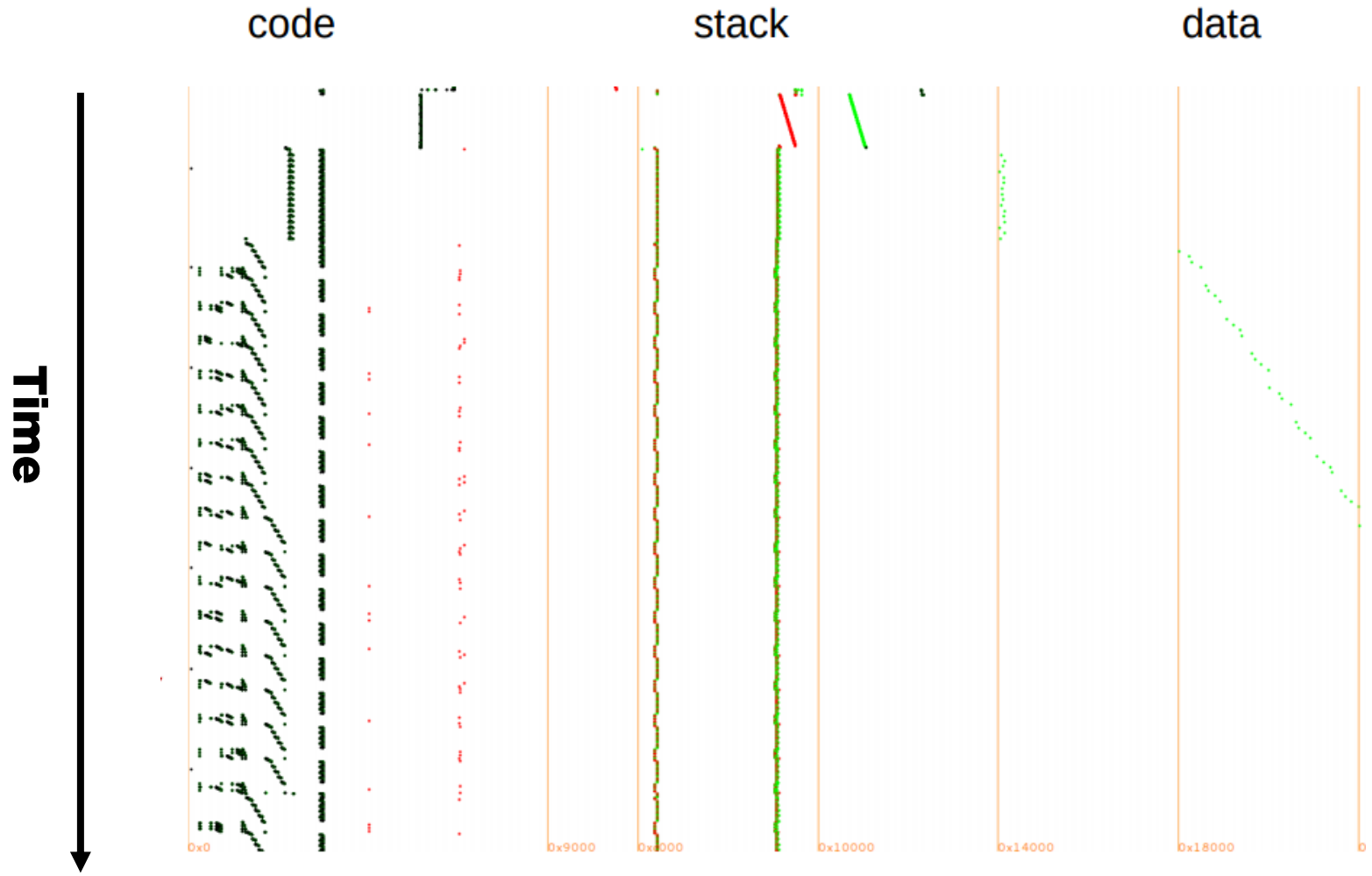


Illustration: https://www.sstic.org/media/SSTIC2016/SSTIC-actes/design_de_cryptographie_white-box_et_a_la_fin_c_es/SSTIC2016-Slides-design_de_cryptographie_white-box_et_a_la_fin_c_est_kerckhoffs_qui_gagne-hubain_teuwen_1.pdf

EXECUTION & MONITORING

Tracer

Assets:

- Binary can be traced directly:

  valgrind --tool=tracergrind --output=ls.trace ls

  - valgrind tracer
  - trace filename
  - binary to trace

eShard

EXECUTION & MONITORING

Tracer

Assets:

- Executables can be traced directly: no reverse engineering skill required
- Open Source

Drawbacks:

- Only memory access tracing
- Filtering based only on PC address/Memory address range
- To trace a library, a launcher must be created

eShard

EXECUTION & MONITORING

Ledger

Rainbow

- Memory Access monitoring:
  - Read/Write value
  - Program Counter
  - Kind of operation
- Register monitoring

Unicorn

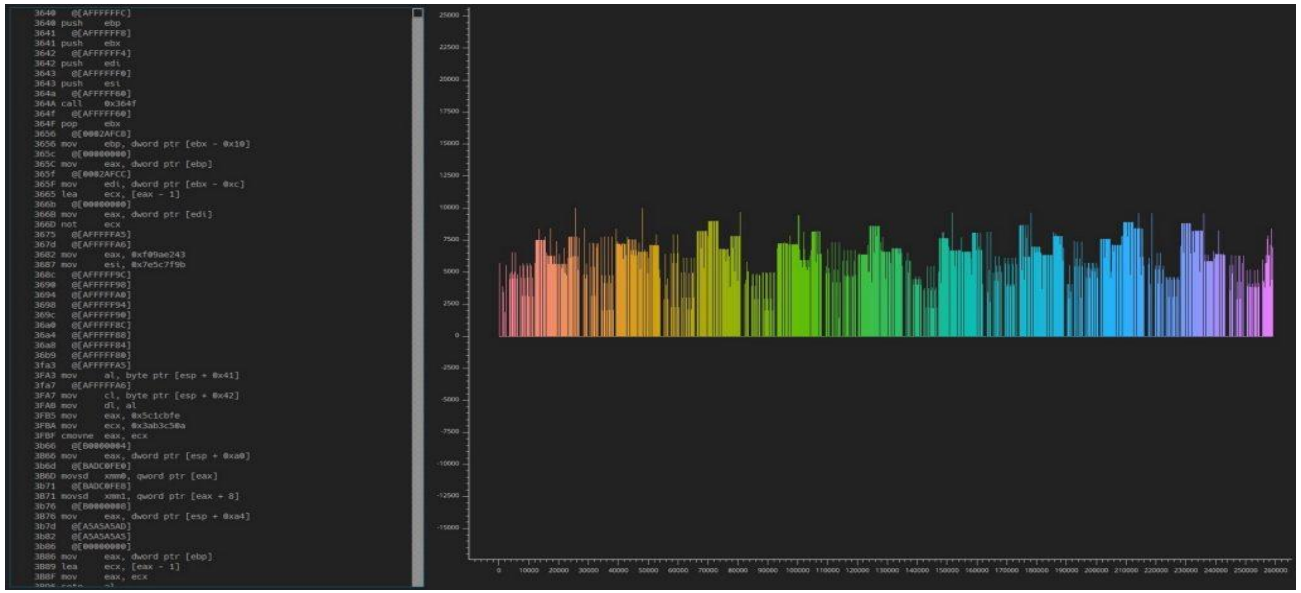


Illustration https://www.ledger.com/2019/02/26/introducing-rainbow-donjons-side-channel-analysis-simulation-tool/

eShard

EXECUTION & MONITORING

Assets:

- Open source
- Use the powerful Unicorn Engine…

Rainbow

eShard

EXECUTION & MONITORING

Ledger

Rainbow

```
# To transmit the input to the execution
# one way among others, we can hook the call
# to 'strtol' to dispatch one byte at a time
def strtol(em):
    em["eax"] = next(inp)
    return True

# Tell the emulator that we redefined this function
e.stubbed_functions["strtol"] = strtol

# In order to attack this binary efficiently,
# we need the 'rand' function to output 0 in the first
# part of the emulation to disable the random masking,
# and to output '7' during scheduling to force the
# execution of the 'correct' AES among the dummy ones
# randval = 0

def rand(em):
    # em["rax"] = randval
    em["rax"] = 0
    return True

e.stubbed_functions["rand"] = rand
```

Call to external libraries must be implemented

Source https://github.com/Ledger-Donjon/rainbow/blob/master/examples/ledger_ctf2/ripped.py

eShard

EXECUTION & MONITORING

Ledger

Rainbow

Assets:

- Open source
- Use the powerful Unicorn Engine…

Drawbacks:

- … that might need reverse engineering
- Executable/Library must be instrumented by a script
- The Unicorn emulation is slower than Valgrind/PIN

eShard

EXECUTION & MONITORING

eShard

esTracer

included in

esDynamic Platform

- Memory Access monitoring:
  - Read/Write value
  - Program Counter
  - Kind of operation
- Register monitoring

QEMU

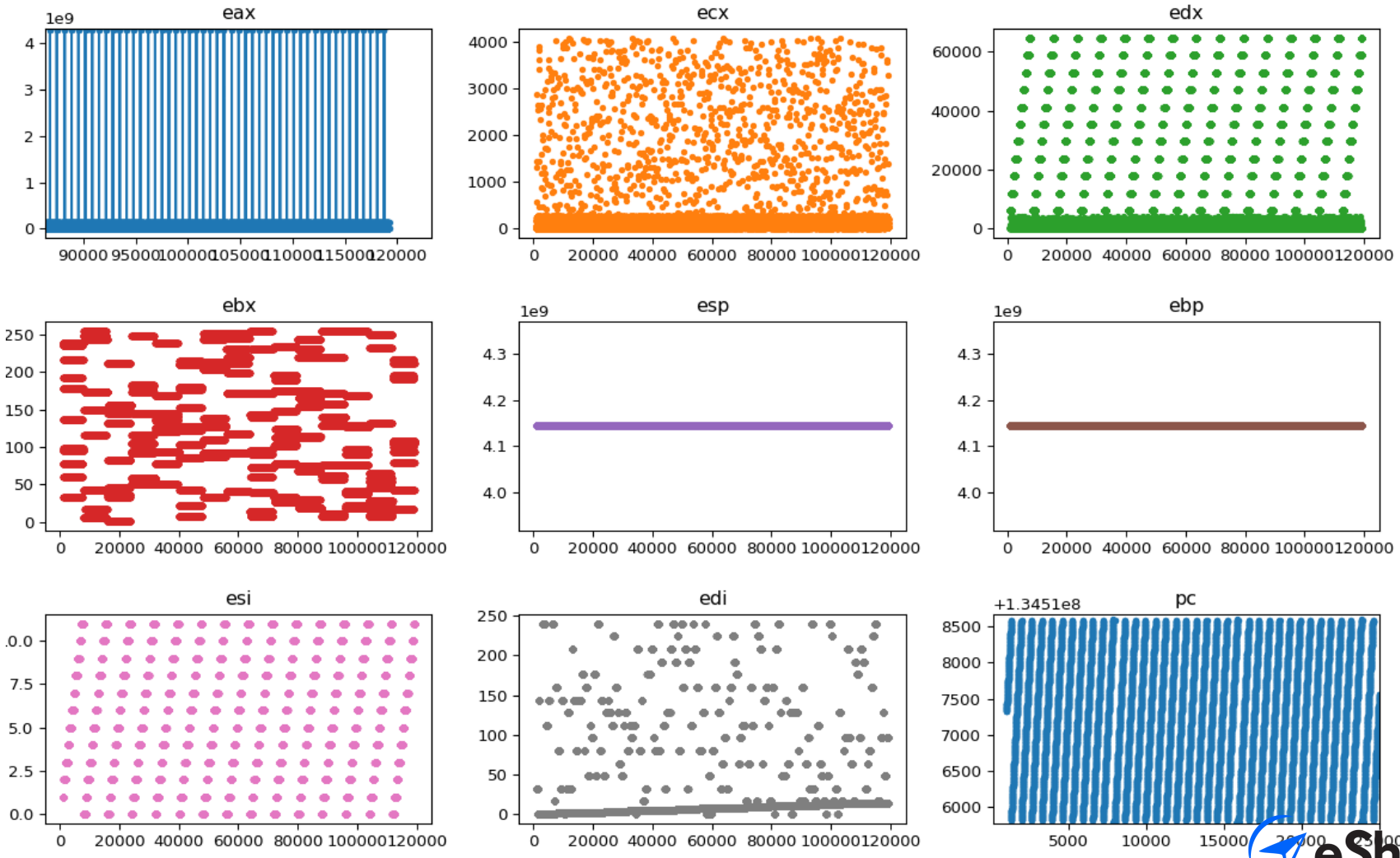
X86, x86_64,
ARM support

eShard

EXECUTION & MONITORING

eShard

esTracer



EXECUTION & MONITORING

Assets:

- Faster than Tracer and Rainbow
- Executables can be traced directly
- A lot of filtering options

Drawbacks:

- Not open source
- To trace a library, a launcher must be created

File Edit View Run Kernel Tabs Settings Help

Code Python 3

ECDSA algorithm

[4]:
```
slideshow('ECDSA algorithm',
        './img/presentation_tracer',
        ['Diapositive%d.PNG' % hit for hit in list(range(33,38))],
        slide_width)
```

▾ **ECDSA algorithm**

Slide 0 | Slide 1 | Slide 2 | Slide 3 | Slide 4

AGENDA

- Introduction
- WB Execution & Monitoring
- **ECDSA algorithm**
- Recover ECDSA key with a DCA
- Trace acquisition
- DCA
- Conclusion

33

THE CHALLENGE

- We trace directly the White-Box without reverse engineering.
- Configuration:
  - CPU i7-7560U, 2.4GHz dual core
  - 16 GB of RAM (we not need so much)
  - SSD NVMe
- We can only use Side Channel Marvels Tracer or esTracer
- We will obtain big trace size

eShard

THE CHALLENGE

- We trace directly the White-Box without reverse engineering. We can only use Side Channel Marvels Tracer or esTracer
- We will obtain big trace size

THE TRACE SIZE
Is-it possible to obtain directly small traces without losing leakage information?

eShard

ECDSA ALGORITHM

Input:

- message to sign m ,
- elliptic curve parameters p , a , b , n , $G = (G_x , G_y)$,
- secret key d .

Output:

- signature (r , s)

- Generate randomly the secret scalar k in [1 , $n - 1$]
- Compute the scalar multiplication: $Q = (Q_x , Q_y) = [k] . G$
- Compute $r = Q_x \bmod n$
- Compute $s = [r \times d + \text{Hash}(m)] \times k^{-1} \bmod n$
- return (r , s)

eShard

ECDSA ALGORITHM

Input:
- message to sign m ,
- elliptic curve parameters p , a , b , n , $G = (G_x , G_y)$,
- secret key d .

Output:
- signature (r , s)

- Generate randomly the secret scalar k in [1 , $n - 1$]
- Compute the scalar multiplication: $Q = (Q_x , Q_y) = [k] . G$
- Compute $r = Q_x \bmod n$
- Compute $s = [r \times d + \text{Hash}(m)] \times k^{-1} \bmod n$
- return (r , s)

Our use case:
attack the multiplication with a DCA

eShard

File Edit View Run Kernel Tabs Settings Help

Code Python 3

Recover ECDSA key with a DCA

[5]:
```
slideshow('Recover ECDSA key with a CDA',
          './img/presentation_tracer',
          ['Diapositive%d.PNG' % hit for hit in list(range(38,43))],
          slide_width)
```

▾ **Recover ECDSA key with a CDA**

Slide 0 | Slide 1 | Slide 2 | Slide 3 | Slide 4

AGENDA

- Introduction
- WB Execution & Monitoring
- ECDSA algorithm
- **Recover ECDSA key with a DCA**
- Trace acquisition
- DCA
- Conclusion

38

eShard

HOW TO RECOVER THE KEY WITH DCA

Example with 32-bits r & d

- $s = [\ r \times d + \text{Hash}(m)\ ] \times k^{-1} \bmod n$
- r is known

$$
\begin{array}{rcccccccc}
 & & & & r_3 & r_2 & r_1 & r_0 \\
 & & & & \times & d_3 & d_2 & d_1 & d_0 \\
\hline
c_7 & c_6 & c_5 & c_4 & c_3 & c_2 & c_1 & c_0
\end{array}
$$

We will not guess directly 32 bits but 8 bits by 8 bits

eShard

HOW TO RECOVER THE KEY WITH DCA

Example with 32-bits r & d

- $s = [\ r \times d + Hash(m)\ ] \times k^{-1} \bmod n$
- r is known

$$
\begin{array}{cccccccc}
 & & & & r_3 & r_2 & r_1 & r_0 \\
 & & & & \times & d_3 & d_2 & d_1 & d_0 \\
\hline
c_7 & c_6 & c_5 & c_4 & c_3 & c_2 & c_1 & c_0
\end{array}
$$

- Guess d_0 and correlate 8 bits information
- Intermediate value is:
  - $c_0 = r_0 \times d_0 \bmod 2^8$

eShard

HOW TO RECOVER THE KEY WITH DCA

Example with 32-bits r & d

- $s = [\ r \times d + Hash(m)\ ] \times k^{-1} \bmod n$
- r is known

$$
\begin{array}{rcccccccc}
 & & & & r_3 & r_2 & r_1 & r_0 \\
 & & & \times & d_1 & d_2 & d_1 & d_0 \\
\hline
c_7 & c_6 & c_5 & c_4 & c_3 & c_2 & c_1 & c_0
\end{array}
$$

- Guess d_1 and correlate 16 bits using the best candidates from d_0
- Intermediate value is:
  - $c_1 c_0 = (r_1 r_0 \times d_1 d_0) \bmod 2^{16}$

And so on for d_1, d_0 …

eShard

File Edit View Run Kernel Tabs Settings Help

Markdown Python 3

What is our target?

```
[6]: slideshow('Introduction',
             './img/presentation_tracer',
             ['Diapositive%d.PNG' % hit for hit in range(43,44)],
             slide_width)
```

▾ **Introduction**

Slide 0

OUR BINARY TARGET

ECDSA
X86-64 architecture

./binary/ECDSA_DoubleAdd
188da80eb03090f67cbf20eb43a18800f4ff0afd82ff1012
07192b95ffc8da78631011ed6b24cdd573f977a11e794811
c71acc89646e021a441830cbb36336cc63c0ae0b74c701c1

r = EFC26C4BF34CD5B90CD157199E3DFD11CB8E377F6BCA49B5
s = FFE27CA311855D0ACF38A510932609500CB9F223B2188B09
End.

Output

43

Our target is a x86-64 architecture executable

File Edit View Run Kernel Tabs Settings Help

Markdown Python 3

Our target is a x86-64 architecture executable.

[7]:
```
!file ./binary/ECDSA_DoubleAdd
```

```
./binary/ECDSA_DoubleAdd: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=1051ce0abf73dfe08e0b0116d0d0a1c79c7b8fb9, stripped, with debug_info
```

We need to give it three parameters:

- the input message as a 48 hexadecimal character string
- the scalar k as a 48 hexadecimal character string
- the secret key d as a 48 hexadecimal character string

[8]:
```
!./binary/ECDSA_DoubleAdd 00447E3732E84D0E6794713A06527541C3F10F1B9F96155D 214436CD8471252983E092E510F3E9EF92FC93C5622A
```

```
r = 428D1F6314348DE54F5CCFAAFEC9ADC40072D09AD2DA8ED5
s = 9A1354E4394B0662EE1DE3DF441D5E5C6153A44079779CB7
End.
```

The output displays the r and s parameters.

We understood how the binary works, let see now how to use esTracer.

File Edit View Run Kernel Tabs Settings Help

Markdown Python 3

Before using esTracer, we configure the binary handler configuration object. It eases the management of the binary to attack.

[9]:
```
from essva import binary_handler

bin_hand = binary_handler.BinaryHandlerConfiguration()
bin_hand.algo = "ECDSA_DoubleAdd"
bin_hand.cmd = "./binary/ECDSA_DoubleAdd"
bin_hand.input_byte_size = 48
bin_hand.output_byte_size = 48
```

We need to indicate how to give a correct input to the program, and as well how to interpret its response.

[10]:
```
def process_input(block):
    msg = block[:48]
    scalar = block[48:]
    secret_key_d = "c71acc89646e021a441830cbb36336cc63c0ae0b74c701c1"
    return "%s %s %s" % (msg, scalar, secret_key_d)

def process_output(output):
    r, s = re.compile("[A-Fa-f0-9]{48}").findall(output)
    return r + s

bin_hand.process_input = process_input
bin_hand.process_output = process_output
```

We are ready to use esTracer!!!

eShard

File Edit View Run Kernel Tabs Settings Help

Markdown Python 3

Trace acquisition

[11]:
```
slideshow('Introduction',
        './img/presentation_tracer',
        ['Diapositive%d.PNG' % hit for hit in range(44,46)],
        slide_width)
```

Introduction

Slide 0 | Slide 1

AGENDA

- Introduction
- WB Execution & Monitoring
- ECDSA algorithm
- Recover ECDSA key with a DCA
- **Trace acquisition**
- DCA
- Conclusion

44

eShard

FIRST STEP

What must be traced?

- only the binary itself, not external system libraries

How to know where to trace?

- Trace memory access or registers
- Display them to see distinguishable patterns
- Program Counter (PC), address of executed instruction, tracing is a good start

eShard

File Edit View Run Kernel Tabs Settings Help

Markdown Python 3

PC trace

We trace the program scanning only the PC register.

We import the esTracer object, and link it to the binary handler configuration.

[12]:
```
from essva.tracer import Tracer
tracer = Tracer(bin_hand)
```

By default, it will trace everything, so we configure it:

- we want to trace only the PC register.
- we focus the tracing on the binary itself, the PC range value can be retrieved for instance with readelf.

[13]:
```
tracer.registers = ["pc"]
tracer.filt_pre.in_pc.ranges = [[0x400568, 0x40AF9C]]
```

We indicate where the traces will be generated.

[14]:
```
tracer.directory_out = "trace_pc"
```

Also, we configure esTracer to generate trace filename with a number (0.bin, 1.bin ...). The index.txt file will be generated, and each of this line will contain the hexadecimal string representing the plaintext and the ciphertext. In that way, we will not have big file name.

[15]:
```
tracer.trace_gen_info = "ASCII_FILE"
```

We generate one trace. esTracer can:

- either automatically generate a random input,
- either use a list of user input. We will use this option.

[16]:
```
msg = "188da80eb03090f67cbf20eb43a18800f4ff0afd82ff1012 07192b95ffc8da78631011ed6b24cdd573f977a11e794811"
```

[17]:
```
tracer.generate_traces([msg])
```

```
Generating 1 trace(s). Please wait...
```

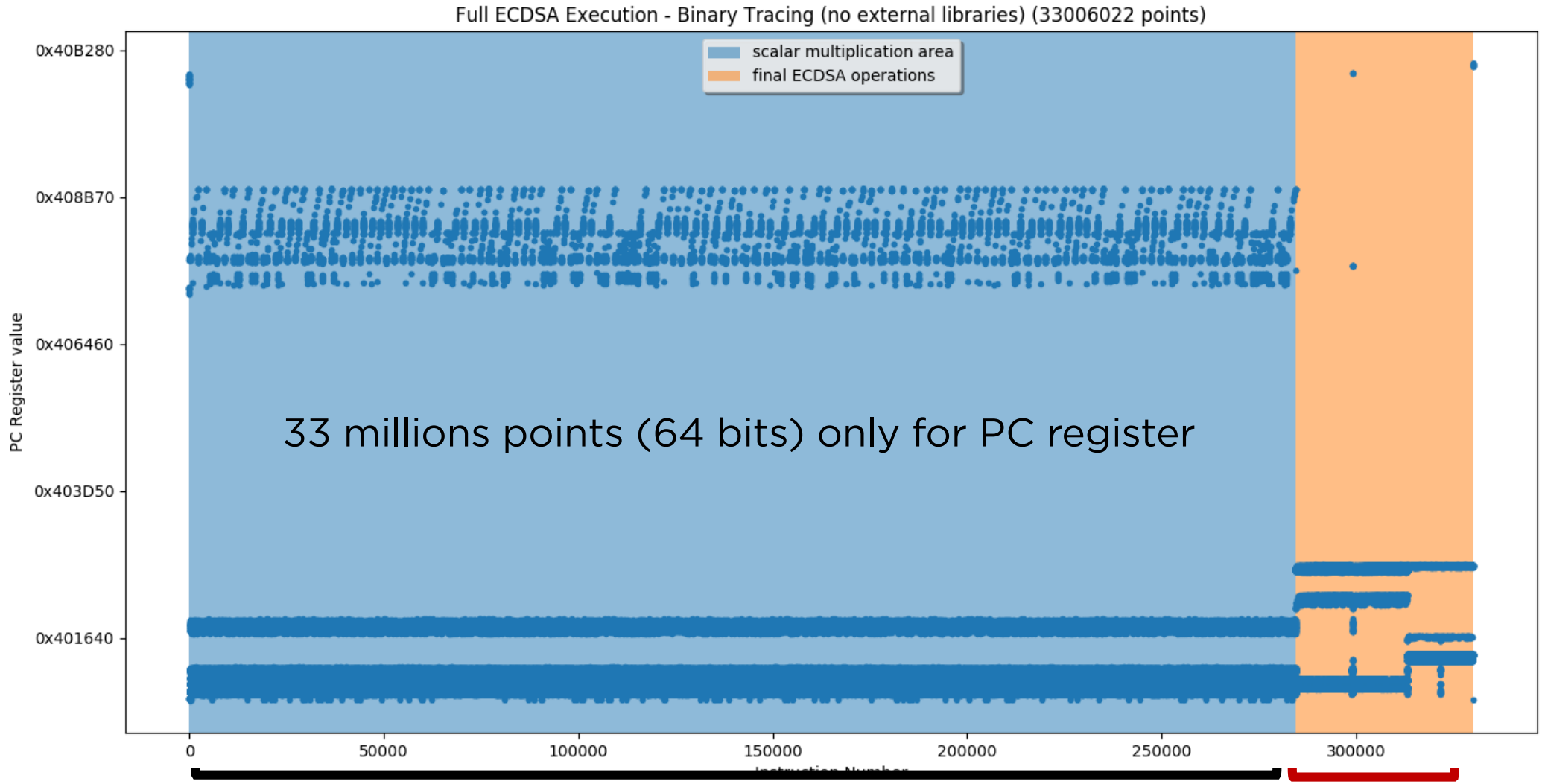
Processing 100.0 % Elapsed: 00:00:00 Remaining: 00:00:00

eShard

File Edit View Run Kernel Tabs Settings Help

Launcher | tracer_presentation.ipynb | tracer_presentation.ipynb

Markdown | Python 3



Double & Add but not our use case

Our use case: r x d

Let start the tracing!

eShard

File Edit View Run Kernel Tabs Settings Help

Code Python 3

Ok. So we are going to attack the final ECDSA operation area indicated in the first figure.

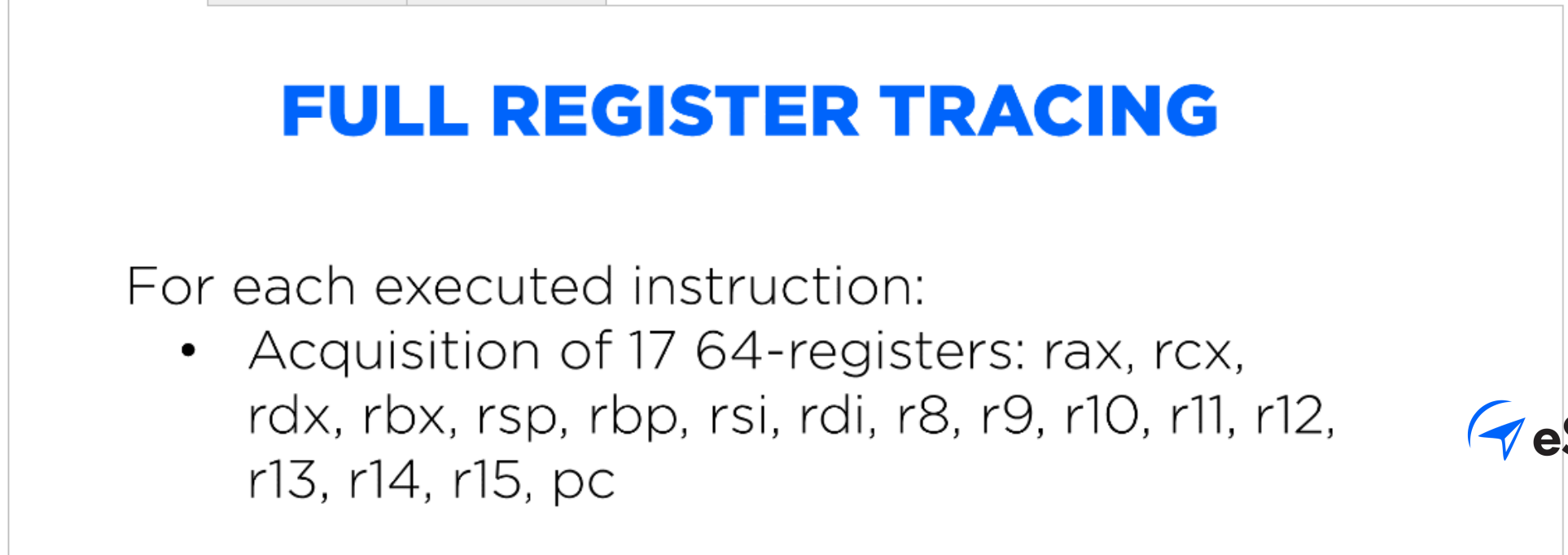
Now, let test different ways of tracing!

Full register tracing

[22]:
```
slideshow('Full register tracing',
          './img/presentation_tracer',
          ['Diapositive%d.PNG' % hit for hit in range(47,50)],
          slide_width)
```

Full register tracing

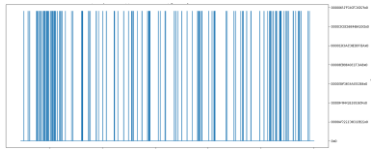
Slide 0 | Slide 1 | Slide 2

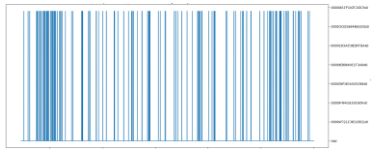


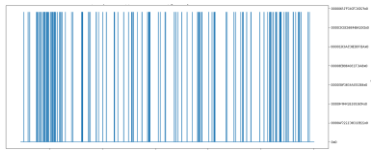
eShard

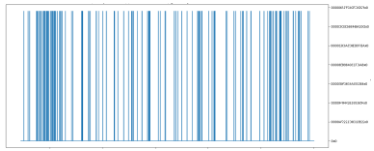
FULL REGISTER TRACING

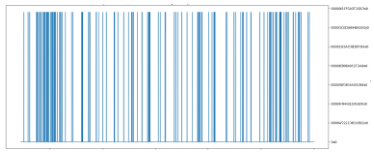
esTracer

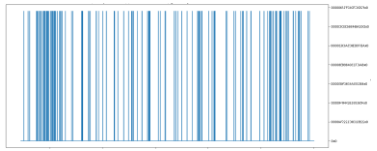

~4.510 MB

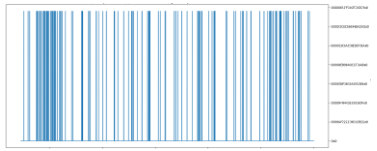

~4.446 MB

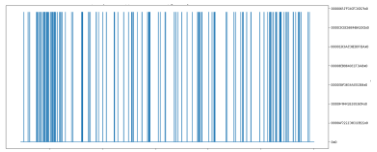

~4.509 MB

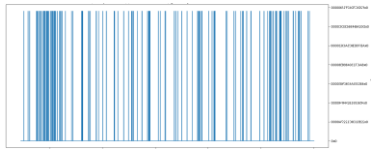

~4.667 MB

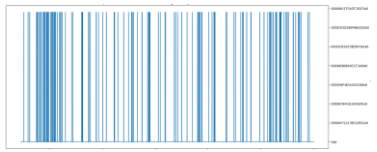

~4.287 MB


~4.287 MB


~4.805 MB


~4.661 MB


~4.437 MB


~4.677 MB

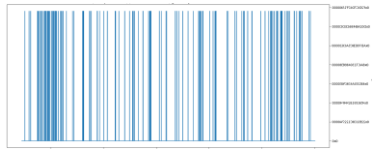
Full tracing time: 127 sec
Full traces size: 47.000 MB

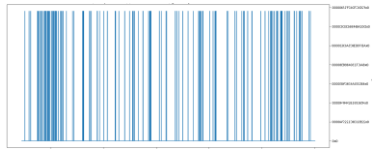
eShard

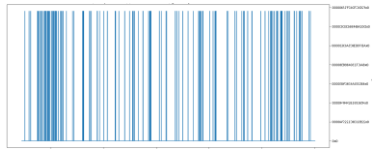
FULL REGISTER TRACING

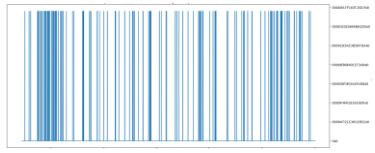
2 problems
- Different trace size
- Big trace size

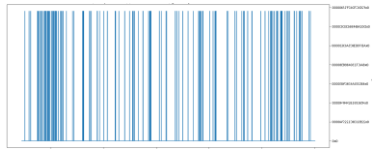
How to resolve these problems?


~4.510 MB


~4.446 MB


~4.509 MB


~4.667 MB


~4.287 MB


~4.287 MB


~4.805 MB


~4.661 MB


~4.437 MB


~4.677 MB

eShard

File Edit View Run Kernel Tabs Settings Help

Markdown Python 3

To trace all the registers, we simply set an empty list.

[23]:
```
tracer.registers = []
```

And we update the directory that will contain the traces.

[24]:
```
tracer.directory_out = "trace_reg_all"
```

We indicate a list of 10 input messages that will be used to generate the traces.

[25]:
```
msg_lst = ['68EC053994AAE7128D85130302100612492D63F6D32367811C20E7DEB538DD97995616A40653015D933B3049D8ECEC0D',
           '7180A1C02C0E9C7D03A3CF1496555ACE9A4DAF777A8DA94063B72748329CED4D8A41BBA6293A18ADF726848E577C0170',
           '23F4A4BFACF050288124AFD09A1CFFAD2FAA4816DB74785E6530B43F91DA0B8F1572BD81E3A8D098E58AC1C6F6D47C78',
           '369375D308EE338D35481602B1C89033114FEE02999625D3863D2578F45727B16FFCD33E0A5F7B504B9C087E5936396D',
           '9B3DFC37BB5F002361ADE5EF5662E4B735BC696C5AC9E200C048E7C3ABD20724C40F30DA0232CE597D1E33CB2841C209',
           '9B3DFC37BB5F002361ADE5EF5662E4B735BC696C5AC9E200C048E7C3ABD20724C40F30DA0232CE597D1E33CB2841C209',
           'F78AD30AEC0A91FDF2F14D4FF7BB0C68C640189E82CA4E5E3F9CCD96CFAFAD64C874BBE15D773293A29EDA968E7FAFD8',
           '7BC02B61391D2CE20925831C60839C8631C50CC025B74BBDC174118FF82795DC304B92F975FD53F776C1E186E88DA8E7',
           '7214574836B33DF33BDB0EEE24472A67CDFCCC92978A7A60470B11C8C2AE70957B33789384B48B824B39E5EF06F34CC8',
           'A636A91E1B280321806AE5A56FF68FAE8F835AEB5312F6F6F5BCACC99BE0E7E0A5BD3ABBCD88DB099A7CC4FFA6F20291']
```

Let's go!

[40]:
```
tracer.generate_traces(msg_lst)
```

```
Generating 10 trace(s). Please wait...
```

Processing 100.0 % Elapsed: 00:02:05 Remaining: 00:00:00

eShard

File Edit View Run Kernel Tabs Settings Help

Code

Python 3

[44]:
```
slideshow('Resynchronisation & Filtering',
          './img/presentation_tracer',
          ['Diapositive%d.PNG' % hit for hit in range(50,59)],
          slide_width)
```

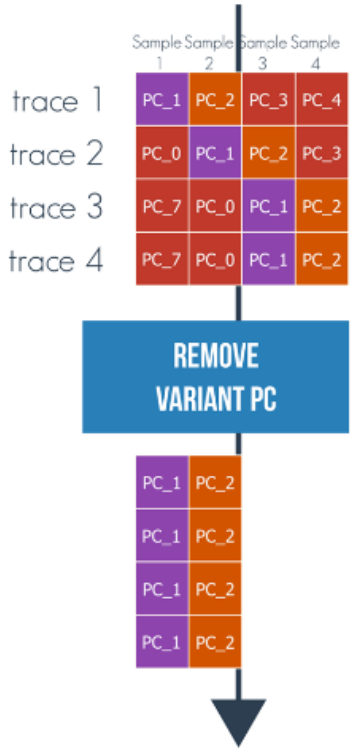
Resynchronisation & Filtering

Slide 0 | Slide 1 | Slide 2 | Slide 3 | Slide 4 | Slide 5 | Slide 6 | Slide 7 | Slide 8

RESYNCHRONISATION

Problem 1 - Different trace size

- Why?
  - ECDSA algorithm
- How defeat it?
  - Remove variant PC

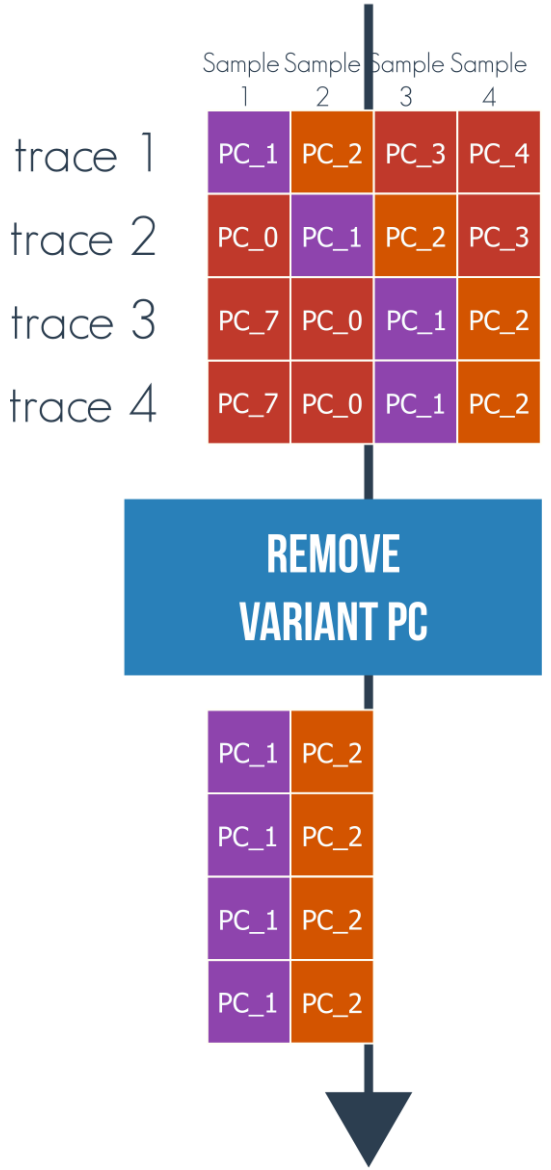


50

RESYNCHRONISATION

Problem 1 - Different trace size

- Why?
  - ECDSA algorithm
- How defeat it?
  - Remove variant PC

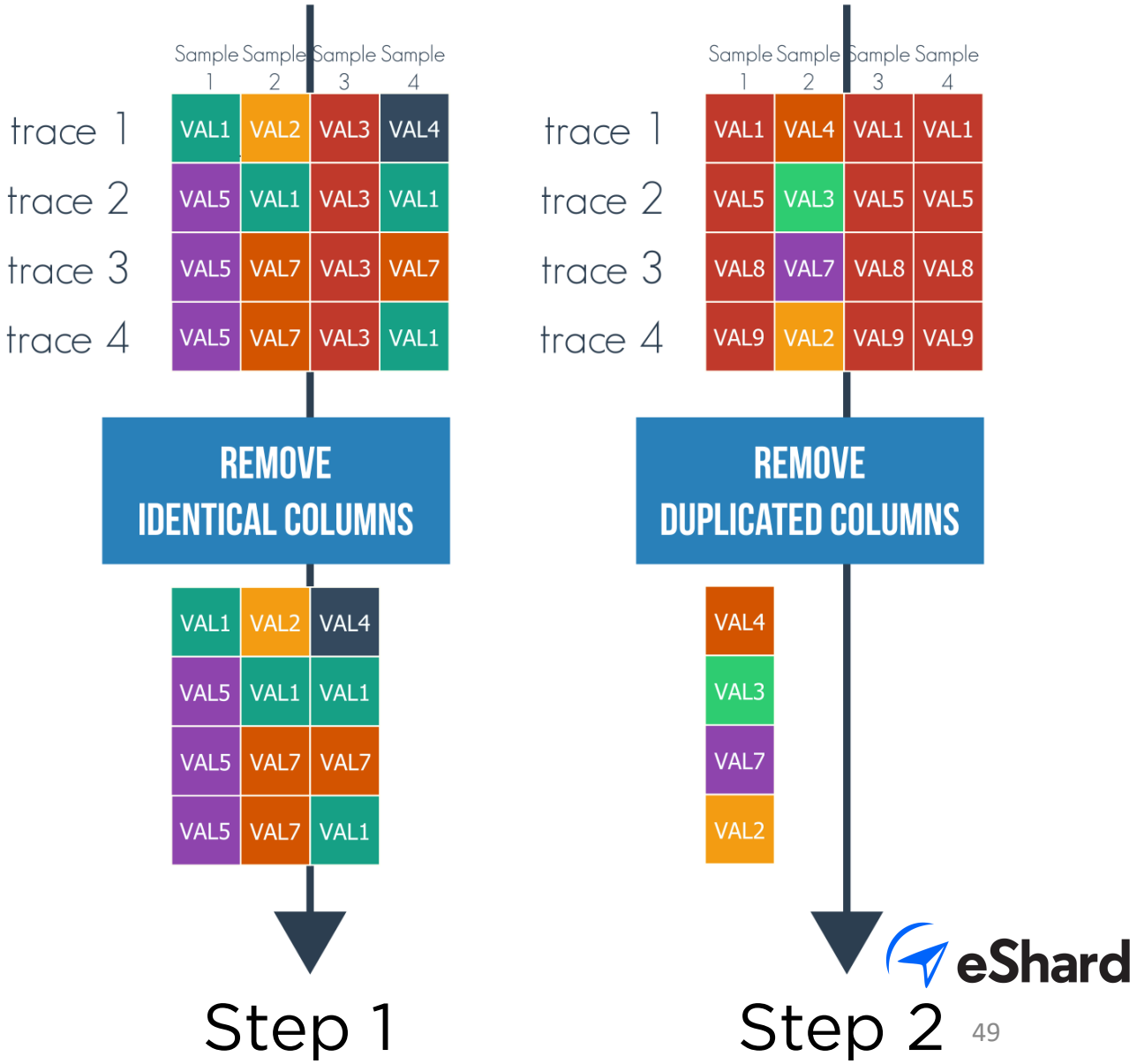


eShard

FILTERING

Problem 2 – Big trace size

- Why?
  - Unvariant registers
- How defeat it?
  - Step 1: remove identical colums
  - Step 2: remove duplicatec columns



RESYNCHRONISATION & FILTERING

Drawbacks:

- Post-processing:
  - Problem 1: space disk. We obtain big traces and transform them in small traces.
  - Problem 2: time. We lost a lot of time to generate them, and filter them.

Can we resolve these two problems directly during the trace generation?

eShard

RESYNCHRONISATION & FILTERING

Drawbacks:

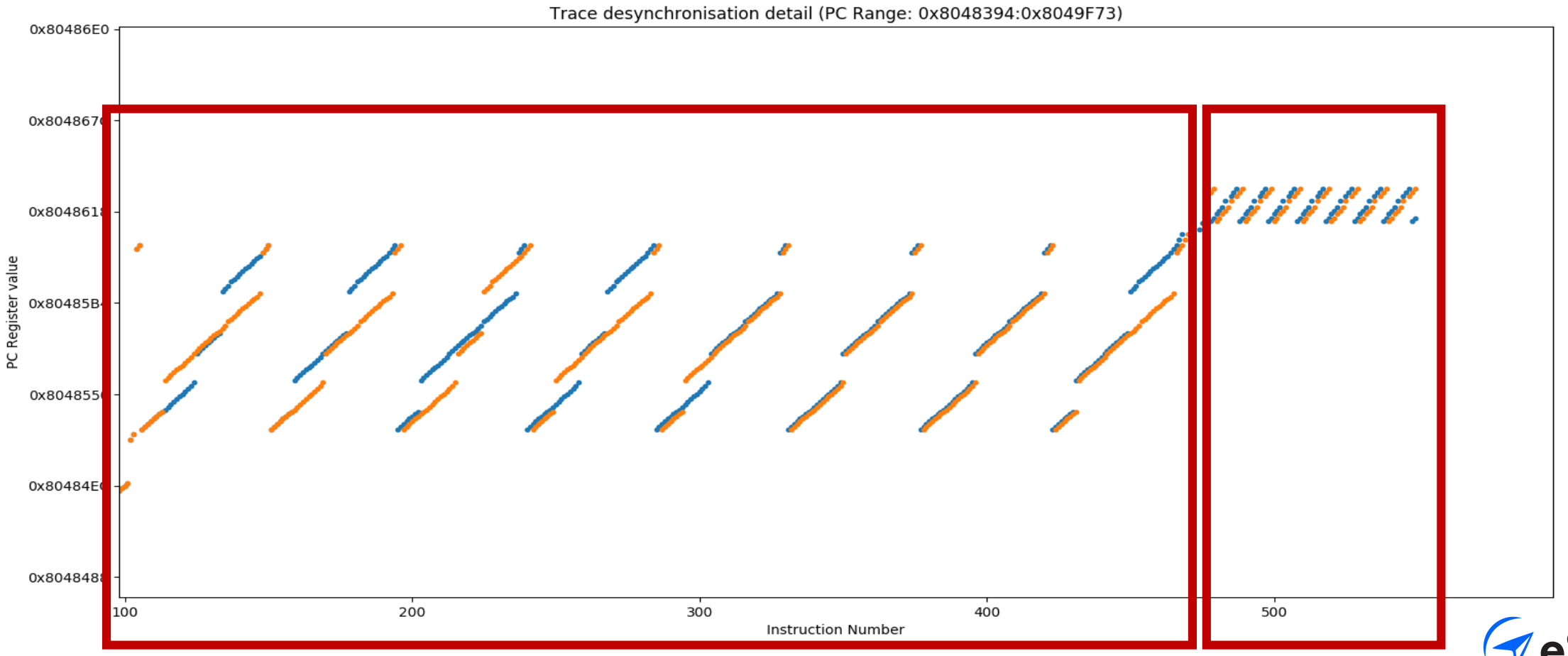
- Post-processing:
  - Problem 1: space disk. We obtain big traces and transform them in small traces.
  - Problem 2: time. We lost a lot of time to generate them, and filter them.
  - Pattern Detector & Accurate register tracing

Can we resolve these two problems directly during the trace generation?

eShard

PATTERN DETECTOR

Example of desynchronisation with 2 PC traces



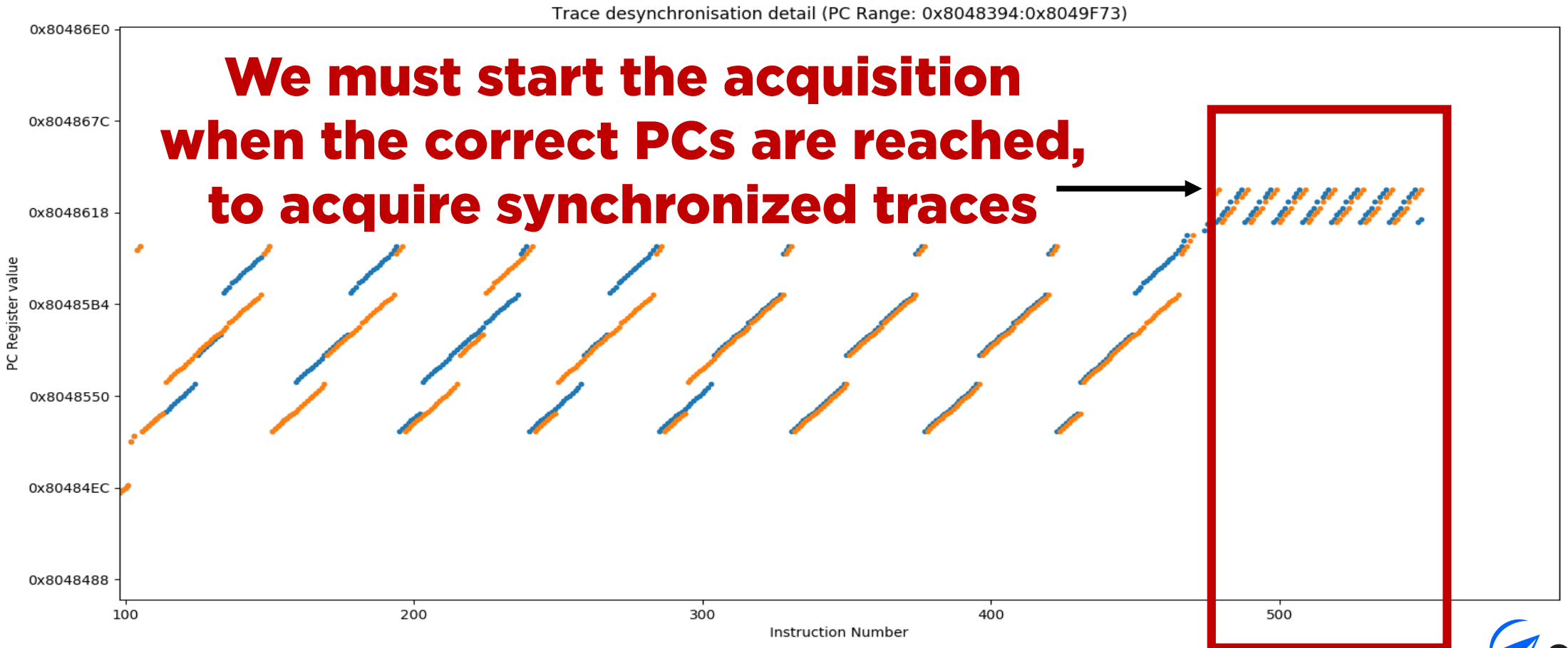
Traces totally different

Traces lightly shifted

eShard

PATTERN DETECTOR

Example of desynchronisation with 2 PC traces



Traces totally different

Traces lightly shifted

eShard

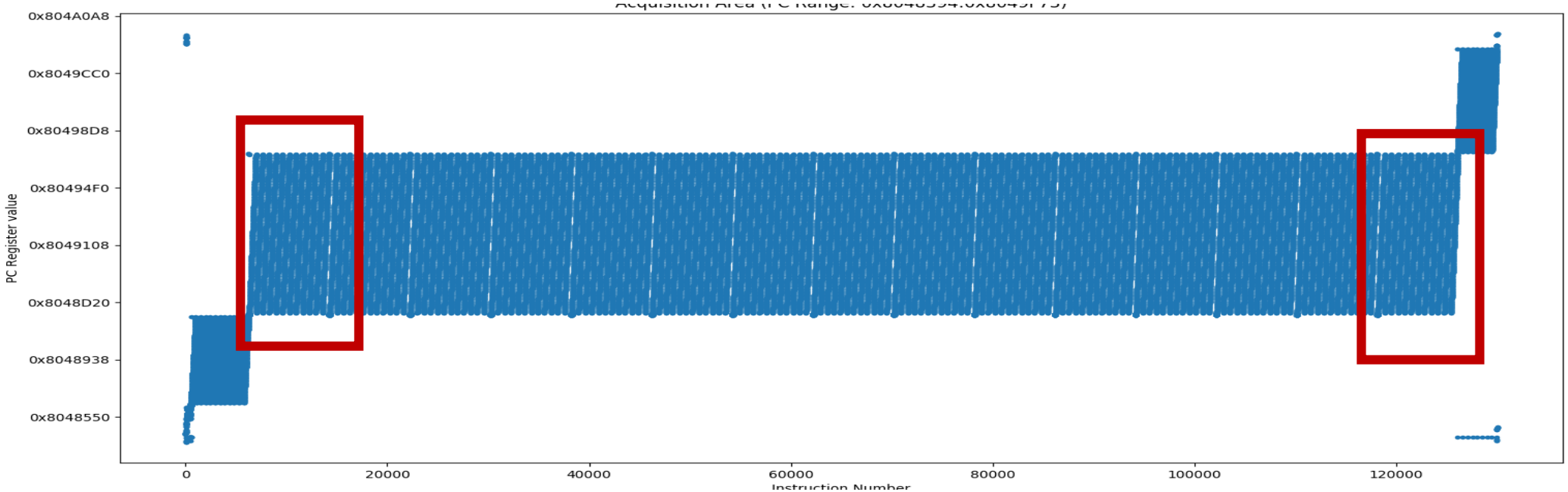
PATTERN DETECTOR

Trig&Act:

- Trigger: pattern detector
- Action: start/stop acquisition, stop program

Trig&Act chaining:

- Trace only first & last rounds
- Defeat several synchronisations



ACCURATE REGISTER ACQUISITION

cmp al, [rbp+var_2C]

- No modification in

rax, rcx, rdx, rbx, rsp, rbp, rsi, rdi, r8, r9, r10, r11, r12, r13, r14, r15, pc

- Do not trace this instruction

eShard

ACCURATE REGISTER ACQUISITION

```
sub     edx, eax
```

- Only edx is written and eax/edx read
- Useless to acquire rcx, rbx, rsp, rbp, rsi, rdi, r8, r9, r10, r11, r12, r13, r14, r15, pc
- Acquire only read/written registers or both

eShard

File Edit View Run Kernel Tabs Settings Help

Code

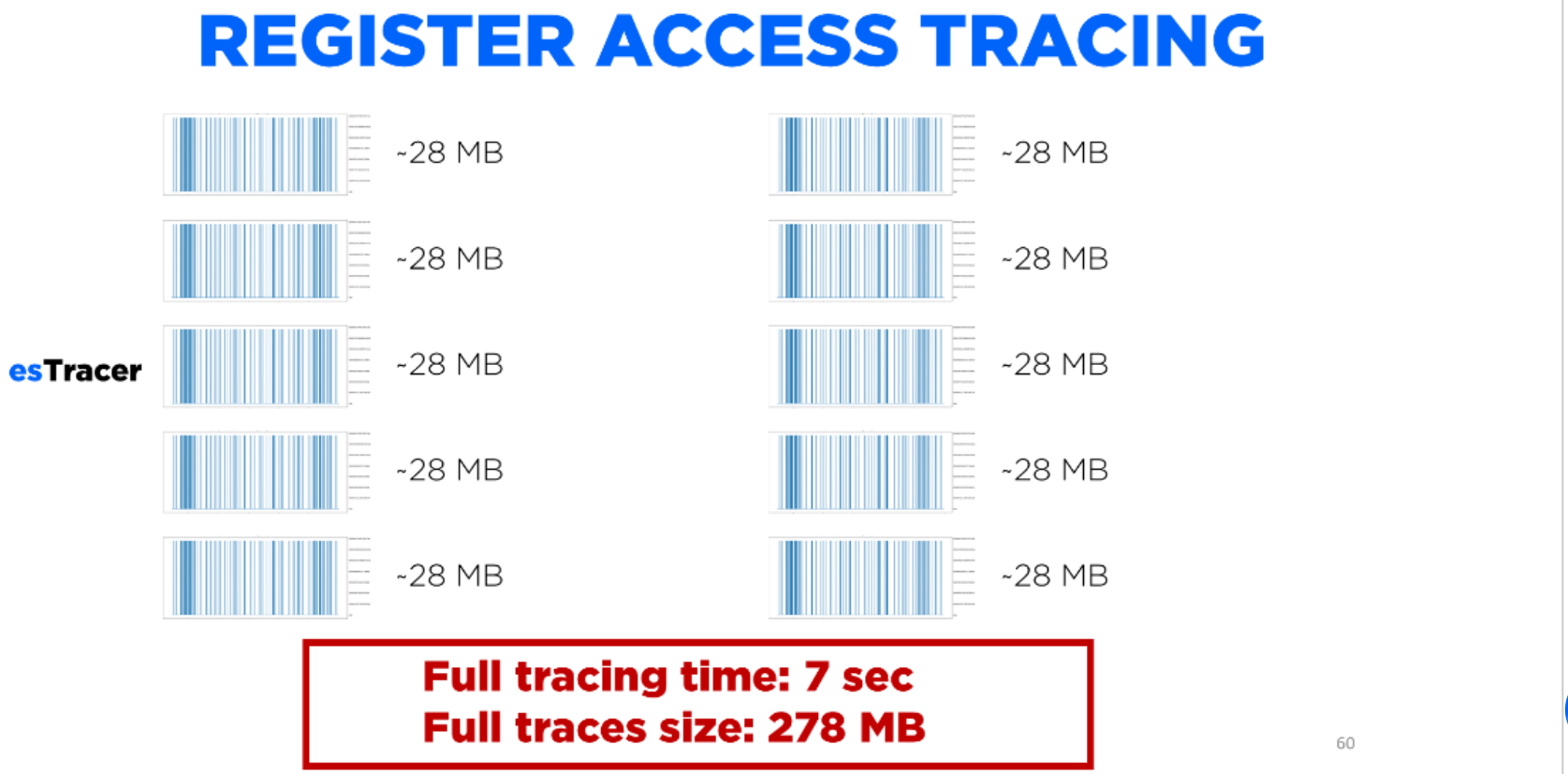
Python 3

Register access tracing with trig&act

[45]:
```
slideshow('Register access tracing with trig&act',
          './img/presentation_tracer',
          ['Diapositive%d.PNG' % hit for hit in range(59,63)],
          slide_width)
```

Register access tracing with trig&act

Slide 0 | Slide 1 | Slide 2 | Slide 3



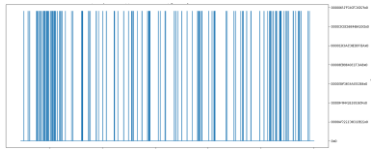
REGISTER ACCESS TRACING

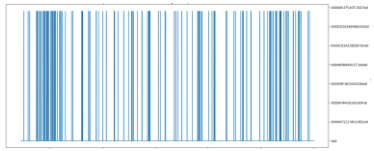
- Trig&act to get synchronized traces
- Trace only written registers

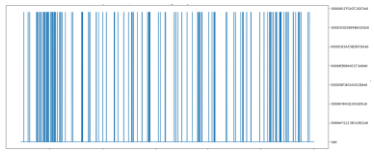
eShard

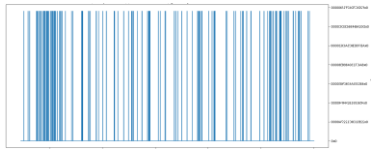
REGISTER ACCESS TRACING

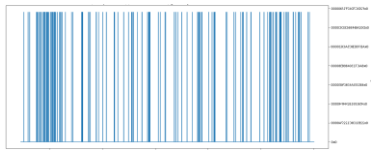
esTracer

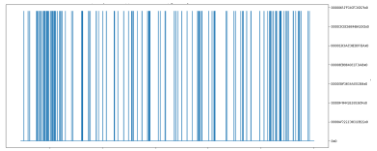

~28 MB

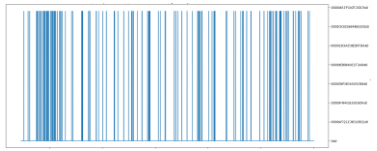

~28 MB

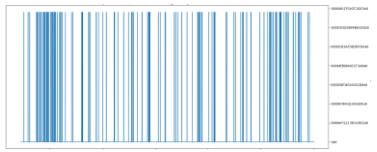

~28 MB

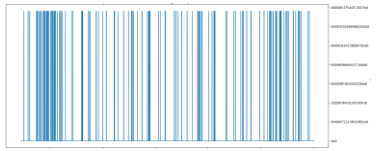

~28 MB

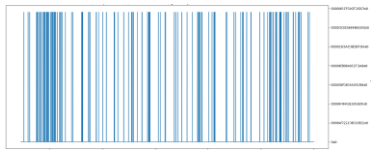

~28 MB


~28 MB


~28 MB


~28 MB


~28 MB


~28 MB

Full tracing time: 7 sec
Full traces size: 278 MB

eShard

TRACING SUMMARY

10 traces generated	Full Register Tracing	Register Access (Write) with Trig & act
Tool	**esTracer**	**esTracer**
Time	**127 sec**	**7 sec**
Full size	**47.000 MB**	**277 MB**
Synchronized traces	**No**	**Yes**

Tracing time: 18 time faster
Trace size: 170 time smaller

eShard

TRACING SUMMARY

10 traces generated	Full Register Tracing	Register Access (Write) with Trig & act
Tool	**esTracer**	**esTracer**
Time	**127 sec**	**7 sec**
Full size	**47.000 MB**	**277 MB**
Synchronized traces	**No**	**Yes**

Can we get smaller traces only with memory access?

eShard

We are going to reduce the trace size with two tricks:

- first, for each executed instruction we want to trace only the updated registers,
- then, we want to acquire directly synchronized traces with a trig&act.

Trig&Act configuration

After analyzing the trace, we find correct parameters to remove the desynchronisation:

- Our pattern detector (it means our trigger): when the PC value 0x401e17 was called 3 times.
- Our action: as soon as this pattern is detected, we start the acquisition.

[46]:
```
nb_inst_start = 284567*100
pc_start = int(pc_np[nb_inst_start])
pc_start_count = len(np.where(pc_np[:nb_inst_start+1] == pc_start)[0])
```

We create the trig&act. It is a list composed of:

- the PC value,
- the number of time it must be called
- and the action that must be triggered once this pattern was found.

[53]:
```
import essva.sva_constants as const
trig_act_1 = [pc_start,
              pc_start_count,
              const.TAA_T_START_ACQUISITION]
```

The trig&act configuration is simple: it is a list of trig&act.

[54]:
```
tracer.trig_and_acts = [ trig_act_1 ]
```

File Edit View Run Kernel Tabs Settings Help

Code Python 3

Register access configuration

We use the instruction analyzer to ease the register access configuration.

It scans a PC trace to know for each executed instruction (in association with its PC value):

- which registers are read or written,
- what is the kind of operation.

[ ]:
```
from essva.tools import inst_analyzer
from essva import sva_constants as const
analyzer = inst_analyzer.InstAnalyzer(bin_hand.cmd)
mne_dict, pc_info_dict = analyzer.update(pc_np)
```

This information is returned in two dictionaries. A helper function exits to ease esTracer configuration. In our case, we want:

- to trace all the registers
- and more especially only the registers that are updated (written) by the instruction

[56]:
```
cfg = analyzer.get_state_filter_cfg(
    pc_trace_np=pc_np,
    registers=[],
    operation="W",
    sva_reg_mode=True,
    nb_inst_start=None,
    nb_inst_end=None)
```

We set this configuration in esTracer. In a next version, the register access configuration could be done automatically by esTracer, without using the instruction analyzer.

[57]:
```
tracer.filt_post.in_states.cfg = cfg
```

And, we are ready to generate once again 10 traces.

[58]:
```
tracer.directory_out = "register_access"
tracer.generate_traces(msg_lst)
```

```
Generating 10 trace(s). Please wait...
```

Processing 100.0 % Elapsed: 00:00:06 Remaining: 00:00:00

File Edit View Run Kernel Tabs Settings Help

Code Python 3

Memory Access Tracing with TracerGrind from the Side Channel Marvels

[60]:
```
slideshow('Introduction',
        './img/presentation_tracer',
        ['Diapositive%d.PNG' % hit for hit in range(63,65)])
```

Introduction

Slide 0 | Slide 1

MEMORY ACCESS TRACING

For each executed memory access instruction:

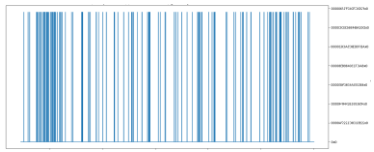
- Acquire the read/written address and value

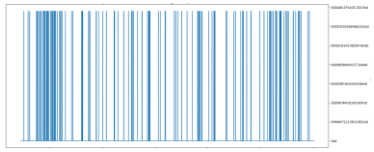
63

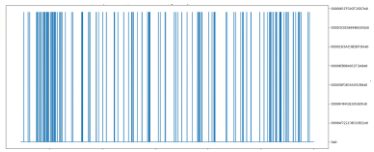
eShard

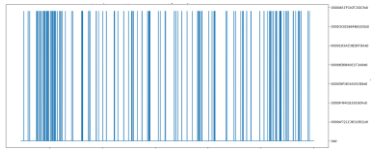
MEMORY ACCESS TRACING

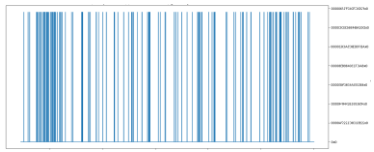
Tracer
Valgrind

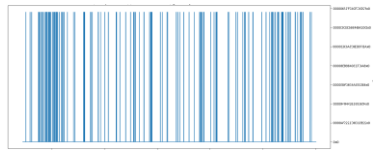

~203 MB

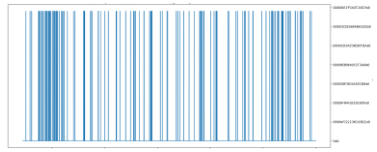

~204 MB

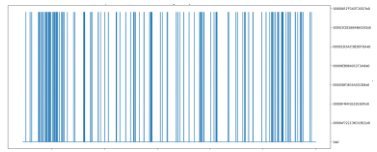

~207 MB

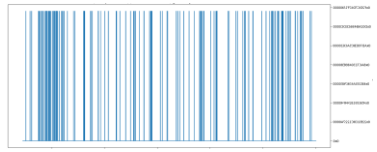

~214 MB

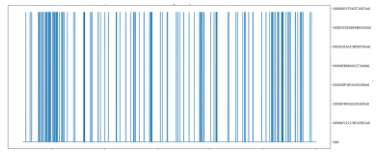

~196 MB


~196 MB


~220 MB


~213 MB


~203 MB


~214 MB

Full tracing time: 200 sec
Full traces size: ~2.073 MB

eShard

File Edit View Run Kernel Tabs Settings Help

Code Python 3

This time, we are going to trace the memory access (value and addresses) with TracerGrind from the Side Channel Marvels.

[ ]:
```
import struct

tac = 0
for index, hit in enumerate(msg_lst):
    cmd = "valgrind --tool=tracergrind --filter=0x400568-0x40AF9C --trace-instr=no --output=tracergrind/0.bin ./binary/
    tic = time.time()
    out = subprocess.getoutput(cmd % (index, process_input(hit)))
    tac += time.time() - tic
    subprocess.check_output("texttrace ./tracergrind/0.bin >(grep '^.M' > ./tracergrind/tmp)", shell=True, executable='

    extract = { 1 : "<B", 2 : "<H", 4 : "<I", 8 : "<Q"}
    length = 0
    with open("./tracergrind/tmp", 'r') as trace:
        for line in iter(trace.readline, ''):
            res = re.match(\".*START_ADDRESS: (.*) LENGTH: (.) MODE.*DATA: (.*)\\n\", line)
            mem_addr= int(res.group(1), 16)
            mem_size=int(res.group(2), 16)
            mem_data=int(res.group(3), 16)
            length += mem_size
            length += 8
    print("Lenght %2d" % length)
print(tac)
```

eShard

File Edit View Run Kernel Tabs Settings Help

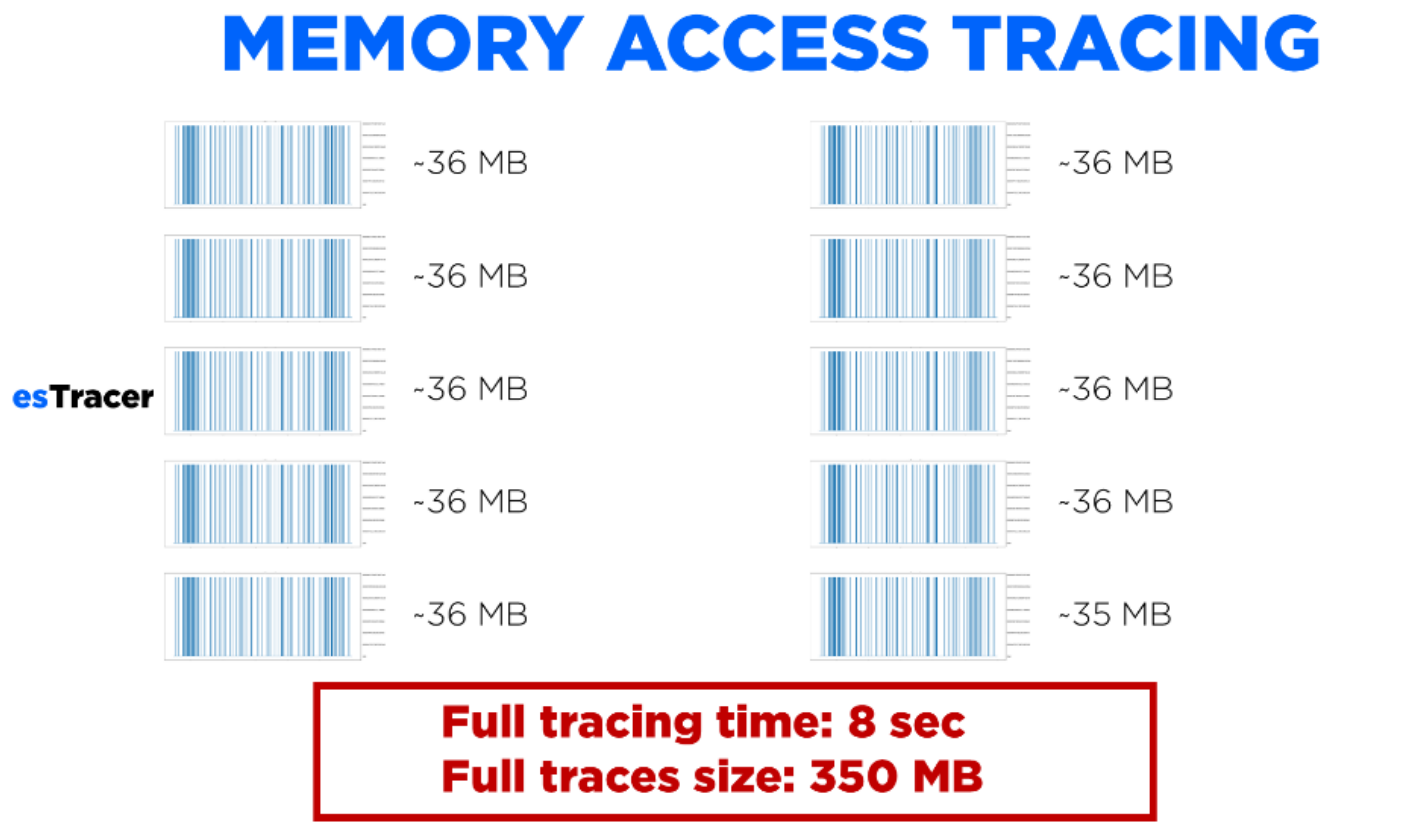
Code Python 3

Memory Access Tracing with esTracer

[61]:
```
slideshow('Introduction',
          './img/presentation_tracer',
          ['Diapositive%d.PNG' % hit for hit in range(65, 68)],
          slide_width)
```

▾ **Introduction**

Slide 0 | Slide 1 | Slide 2



eShard

TRACING SUMMARY

10 traces generated	Full Register Tracing	Register Access (Write) with Trig & act	Memory Access (value and address) with trig&act	Memory Access (value and address)
Tool	**esTracer**	**esTracer**	**esTracer**	**Tracer Valgrind**
Time	**127 sec**	**7 sec**	**8 sec**	**200 sec**
Full size	**47.000 MB**	**277 MB**	**350 MB**	**2.073 MB**
Synchronized traces	**No**	**Yes**	**Yes**	**No**

Memory access tracing is not better

With trig&act, we can skip the point multiplication (very big). Without it, we would have the same trace size as with Tracer Valgrind.

eShard

TRACING SUMMARY

10 traces generated	Full Register Tracing	Register Access (Write) with Trig & act	Memory Access (value and address) with trig&act	Memory Access (value and address)
Tool	**esTracer**	**esTracer**	**esTracer**	**Tracer Valgrind**
Time	**127 sec**	**7 sec**	**8 sec**	**200 sec**
Full size	**47.000 MB**	**277 MB**	**350 MB**	**12.818 MB**
Synchronized traces	**No**	**Yes**	**Yes**	**No**

Another idea?

eShard

File Edit View Run Kernel Tabs Settings Help

Code Python 3

This time, we trace:

- the memory access (value and addresses) with esTracer,
- we keep our trig&act.

Instead of configuring the registers options, we just need to configure the mem_access_states options.

We trace only the manipulated value and address, since we are not interesting in the PC value or the kind of operation.

[62]:
```
tracer.mem_access_states = [ "val", "add"]
```

[63]:
```
tracer.directory_out = "mem_access_estracer"
tracer.verbose_mode = False
```

[64]:
```
tracer.generate_traces(msg_lst)
```

```
Generating 10 trace(s). Please wait...
```

Processing 100.0 % Elapsed: 00:00:07 Remaining: 00:00:00

eShard

File Edit View Run Kernel Tabs Settings Help

Markdown Python 3

Mult Instruction Tracing

[66]:
```
slideshow('Introduction',
          './img/presentation_tracer',
          ['Diapositive%d.PNG' % hit for hit in range(68, 72)],
          slide_width)
```

Introduction

Slide 0 | Slide 1 | Slide 2 | Slide 3

MULT INSTRUCTION TRACING

We attack a multiplication... so we could focus on instruction related to it.

For each executed mult instruction:

- Acquire the read/written registers

68

MULT INSTRUCTION TRACING

We attack a multiplication… so we could focus on instruction related to it.

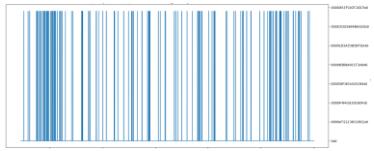
For each executed mult instruction:

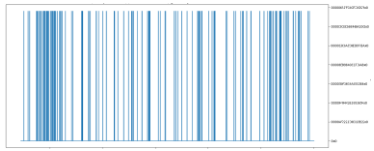
- Acquire the read/written registers

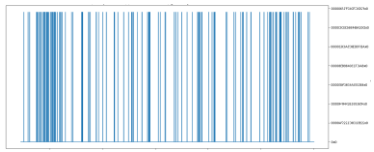
eShard

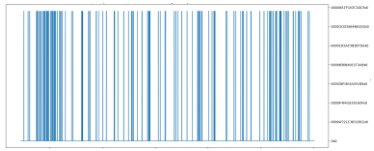
MULT INSTRUCTION TRACING

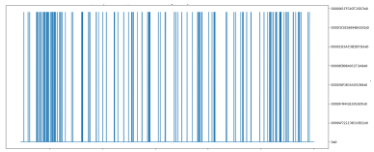
esTracer

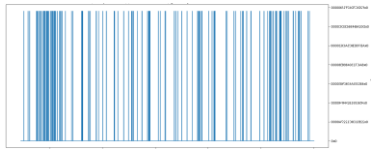

~0.043 MB

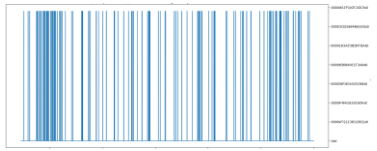

~0.043 MB

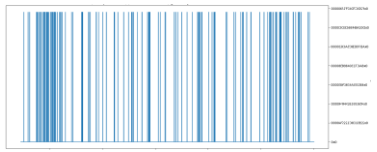

~0.043 MB

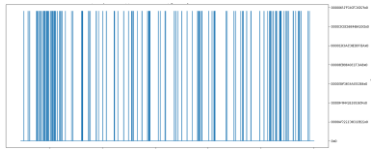

~0.043 MB

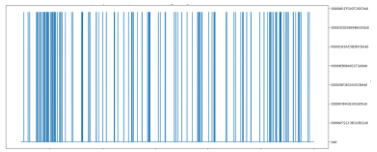

~0.043 MB


~0.043 MB


~0.043 MB


~0.043 MB


~0.043 MB


~0.043 MB

Full tracing time: 0’06’
Full traces size: 0.092 MB

eShard

TRACING SUMMARY

10 traces generated	Full Register Tracing	Register Access (Write) with Trig & act	Memory Access (value and address) with trig&act	Mult instruction with trig&act
Tool	**esTracer**	**esTracer**	**esTracer**	**esTracer**
Time	**127 sec**	**7 sec**	**8 sec**	**6 sec**
Full size	**47.000 MB**	**277 MB**	**350 MB**	**0.092 MB**
Synchronized traces	**No**	**Yes**	**Yes**	**Yes**

Register access versus Mult Tracing:

- **Time: ~same**
- **Trace size: 3010 time smaller**

eShard

TRACING SUMMARY

10 traces generated	Full Register Tracing	Register Access (Write) with Trig & act	Memory Access (value and address) with trig&act	Mult instruction with trig&act
Tool	esTracer	esTracer	esTracer	esTracer
Time	127 sec	7 sec	8 sec	6 sec
Full size	47.000 MB	277 MB	350 MB	0.092 MB
Synchronized traces	No	Yes	Yes	Yes

Are we sure there are still leakage points?

eShard

File Edit View Run Kernel Tabs Settings Help

Markdown Python 3

This time, we trace:

- only the mult instruction
- only the register handled by this instruction: rax, rbx and rdx
- and we use a trig&act to get synchronized traces

We deactivate the previous configuration.

[67]:
```
tracer.filt_post.in_states.cfg = []
```

We indicate that we want to trace only the PC related to the mult instruction. One of this dictionary created with the instruction analyzer enables to do it easily. In a next version, the kind of instruction tracing could be done automatically by esTracer, without using the instruction analyzer.

[68]:
```
tracer.filt_pre.in_pc.ranges = []
tracer.filt_pre.in_pc.pts = mne_dict["mul"]["pc"]
```

We indicate which registers must be traced.

[69]:
```
tracer.registers = ["rax", "rbx", "rdx"]
```

We are ready to generate the traces.

[70]:
```
tracer.directory_out = "trace_mult_pc"
tracer.generate_traces(msg_lst)
```

```
Generating 10 trace(s). Please wait...
```

Processing 100.0 % Elapsed: 00:00:06 Remaining: 00:00:00

[71]:
```
!ls -al ./trace_mult_pc/
```

```
total 92
drwxr-xr-x  2 user users 4096 Jun  4 12:03 .
drwxr-xr-x 12 user users 4096 Jun  4 12:03 ..
-rw-r--r--  1 user users 4320 Jun  4 12:03 0.bin
-rw-r--r--  1 user users 4320 Jun  4 12:03 1.bin
-rw-r--r--  1 user users 4320 Jun  4 12:03 2.bin
-rw-r--r--  1 user users 4320 Jun  4 12:03 3.bin
-rw-r--r--  1 user users 4320 Jun  4 12:03 4.bin
-rw-r--r--  1 user users 4320 Jun  4 12:03 5.bin
-rw-r--r--  1 user users 4320 Jun  4 12:03 6.bin
-rw-r--r--  1 user users 4320 Jun  4 12:03 7.bin
-rw-r--r--  1 user users 4320 Jun  4 12:03 8.bin
-rw-r--r--  1 user users 4320 Jun  4 12:03 9.bin
```

File Edit View Run Kernel Tabs Settings Help

Markdown Python 3

DCA Attack on mult traces

[72]:
```
slideshow('DCA',
          './img/presentation_tracer',
          ['Diapositive%d.PNG' % hit for hit in range(72, 82)],
          slide_width)
```

▾ DCA

Slide 0 | Slide 1 | Slide 2 | Slide 3 | Slide 4 | Slide 5 | Slide 6 | Slide 7 | Slide 8 | Slide 9

AGENDA

- Introduction
- WB Execution & Monitoring
- ECDSA algorithm
- Recover ECDSA key with a DCA
- Trace acquisition
- **DCA**
- Conclusion

72

eShard

KEY RECOVERY

Example with 32-bits r & d

- $s = [\ r \times d + Hash(m)\ ] \times k^{-1} \bmod n$
- r is known

$$
\begin{array}{rrrrrrrr}
 & & & r_3 & r_2 & r_1 & r_0 \\
 & & \times & d_3 & d_2 & d_1 & d_0 \\
\hline
c_7 & c_6 & c_5 & c_4 & c_3 & c_2 & c_1 & c_0
\end{array}
$$

- Guess d_0 and correlate 8 bits information
- Intermediate value is:
  - $c_0 = r_0 \times d_0 \bmod 2^8$

eShard

KEY RECOVERY

Example with 32-bits r & d

- $s = [\ r \times d + Hash(m)\ ] \times k^{-1} \bmod n$
- r is known

$$
\begin{array}{cccccccc}
 & & & & r_3 & r_2 & r_1 & r_0 \\
 & & & & \times\ d_3 & d_2 & d_1 & d_0 \\
\hline
c_7 & c_6 & c_5 & c_4 & c_3 & c_2 & c_1 & c_0
\end{array}
$$

- Guess d_1 and correlate 16 bits using the best candidates from d_0.
- Intermediate value is:
  - $c_1 c_0 = (r_1 r_0 \times d_1 d_0) \bmod 2^{16}$

eShard

KEY RECOVERY

Problem 1: How many best guesses for each byte?

- d_0 : 5 best guesses
- d_1 : 5 best guesses
- d_2 : 5 best guesses
- d_3 : 1 best guesses

eShard

KEY RECOVERY

How many key words to recover ?

- $d_3\ d_2\ d_1\ d_0$:$word_0$
- $d_7\ d_6\ d_5\ d_4$:$word_0$
- $d_{11}\ d_{10}\ d_9\ d_8$:$word_0$
- $d_{15}\ d_{14}\ d_{13}\ d_{12}$:$word_0$
- $d_{19}\ d_{18}\ d_{17}\ d_{16}$:$word_0$
- $d_{23}\ d_{22}\ d_{21}\ d_{20}$:$word_0$

- $word_0$
- $word_1$
- $word_2$
- $word_3$
- $word_4$
- $word_5$

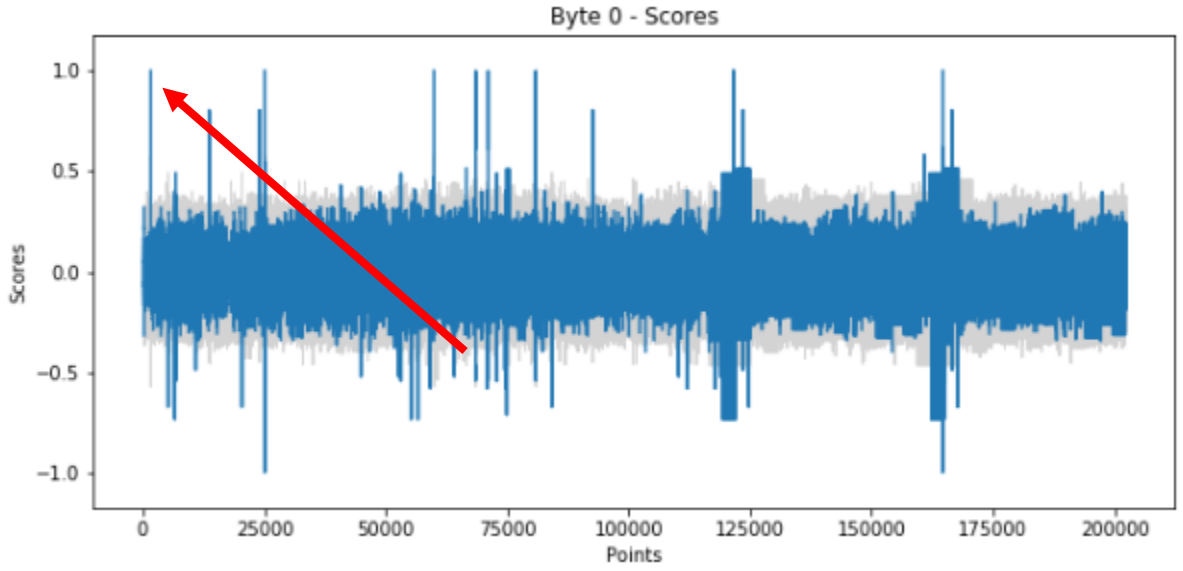
eShard

KEY RECOVERY

Problem 2: The importance of attack frame

$$
\begin{array}{rcccccccc}
 & r_7 & r_6 & r_5 & r_4 & r_3 & r_2 & r_1 & r_0 \\
\times & d_7 & d_6 & d_5 & d_4 & d_3 & d_2 & d_1 & d_0 \\
\hline
 & & & & & & & & \\
\ldots & c_7 & c_6 & c_5 & c_4 & c_3 & c_2 & c_1 & c_0
\end{array}
$$

$word_0$ recovery



eShard

KEY RECOVERY

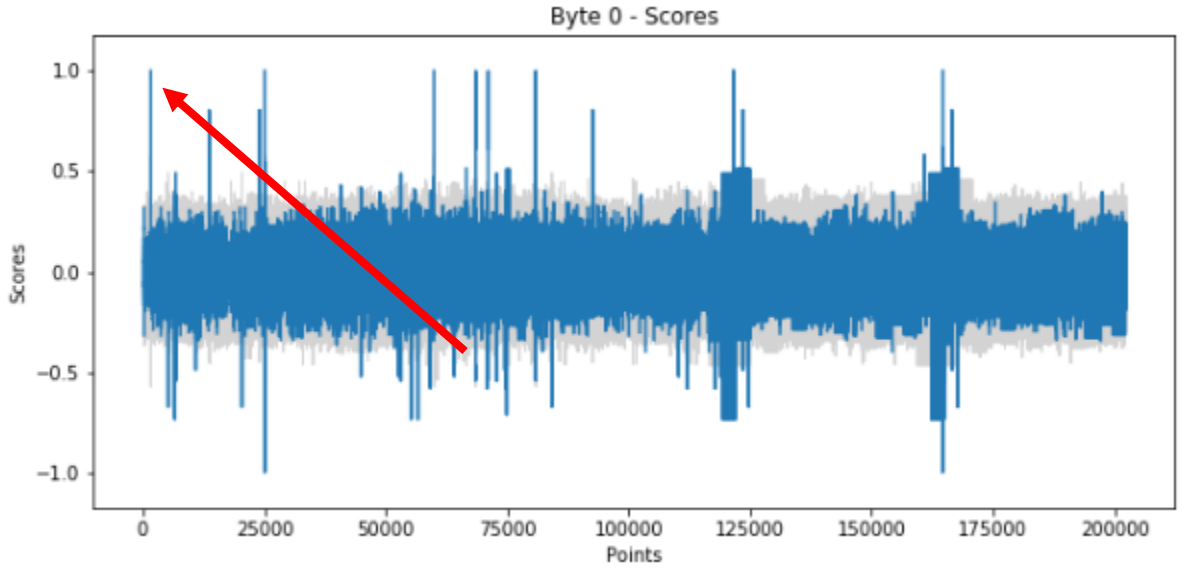
Problem 2: The importance of attack frame

$$
\begin{array}{rcccccccc}
 & r_7 & r_6 & r_5 & r_4 & r_3 & r_2 & r_1 & r_0 \\
\times & d_7 & d_6 & d_5 & d_4 & d_3 & d_2 & d_1 & d_0 \\
\hline
 & & & & & c_3 & c_2 & c_1 & c_0
\end{array}
$$

$word_0$ recovery

$$
\begin{array}{rcccccccc}
 & r_7 & r_6 & r_5 & r_4 & r_3 & r_2 & r_1 & r_0 \\
\times & d_7 & d_6 & d_5 & d_4 & d_3 & d_2 & d_1 & d_0 \\
\hline
 & c_7 & c_6 & c_5 & c_4 & . & . & . & .
\end{array}
$$

$word_1$ recovery



If we attack $word_1$ in the same frame, we might correlate $word_0$ with d_7 d_6 d_5 d_4

We must adjust the frame

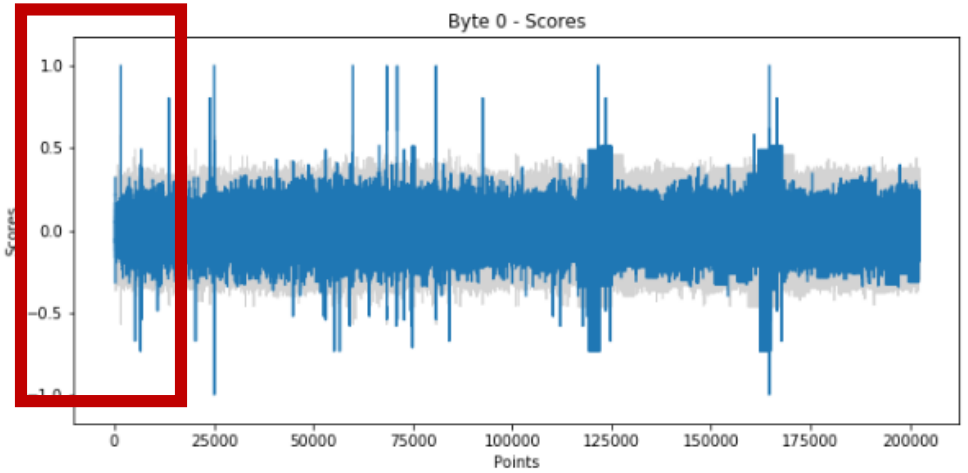
eShard

KEY RECOVERY

Problem 2: The importance of attack frame

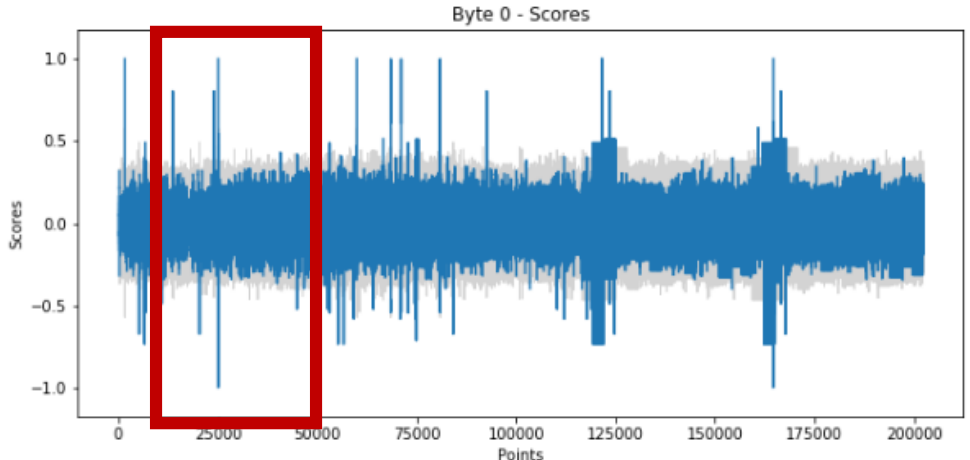
$$
\begin{array}{rcccccccc}
 & r_7 & r_6 & r_5 & r_4 & r_3 & r_2 & r_1 & r_0 \\
\times & d_7 & d_6 & d_5 & d_4 & d_3 & d_2 & d_1 & d_0 \\
\hline
 & & & & & c_3 & c_2 & c_1 & c_0
\end{array}
$$

$word_0$ recovery



$$
\begin{array}{rcccccccc}
 & r_7 & r_6 & r_5 & r_4 & r_3 & r_2 & r_1 & r_0 \\
\times & d_7 & d_6 & d_5 & d_4 & d_3 & d_2 & d_1 & d_0 \\
\hline
 & c_7 & c_6 & c_5 & c_4 & . & . & . & .
\end{array}
$$

$word_1$ recovery



eShard

DCA ATTACK

- 500 traces: 407 sec & 4.1 MB
- DCA Attack:
  - 48 sec
  - Value Model

eShard

File Edit View Run Kernel Tabs Settings Help

Code Python 3

First, we decide to acquire 500 traces:

- we focus on the binary itself
- we trace only the registers updated by the multiplication instructions (rax", rbx, rdx)

[73]:
```
tracer.directory_out = "./test_attack"
tracer.generate_traces(500)
```

Now, we are ready to make the CDA.

We get the traces, and for each of them we recover their plaintext.

[74]:
```
secret_key_d = "c71acc89646e021a441830cbb36336cc63c0ae0b74c701c1"
metadatas = {'plaintext': estraces.bin_extractor.PatternExtractor(r"([A-Fa-f0-9]{48})", num=2),
             'key': estraces.bin_extractor.DirectValue(secret_key_d)}
```

[75]:
```
ths64 = estraces.read_ths_from_bin_filenames_pattern(filename_pattern="./test_attack/*.bin",
                                                     dtype='uint64',
                                                     offset=0,
                                                     metadatas_parsers=metadatas)
```

eShard

We perform the attack.

For each word, we use this configuration to keep the best guesses:

- d0: 5 best guesses
- d1: 5 best guesses
- d2: 5 best guesses
- d3: 1 best guess

This configuration was choosen after several attack execution, and it is specific to our binary.

[76]:
```
nbToKeep = [5, 5, 5, 1]

tic = time.time()
res = multiplication.attack_interm_mults(ths=ths64,
                                         bits=32,
                                         bytesToAttack=range(0, 24),
                                         nbToKeep=nbToKeep,
                                         sel_func_builder=multiplication.selection_function_interm_mults_32bits)
print(time.time() - tic)
```

```
 #### Results attack on byte 23 ####

Good candidate at position 1

Best keys kept:
C7 1A CC 89 64 6E 02 1A 44 18 30 CB B3 63 36 CC 63 C0 AE 0B 74 C7 01 C1          Correlation: 1.00000000000000

Key to recover is:
C7 1A CC 89 64 6E 02 1A 44 18 30 CB B3 63 36 CC 63 C0 AE 0B 74 C7 01 C1
41.52753496170044
```

The attack succeeded!

File Edit View Run Kernel Tabs Settings Help

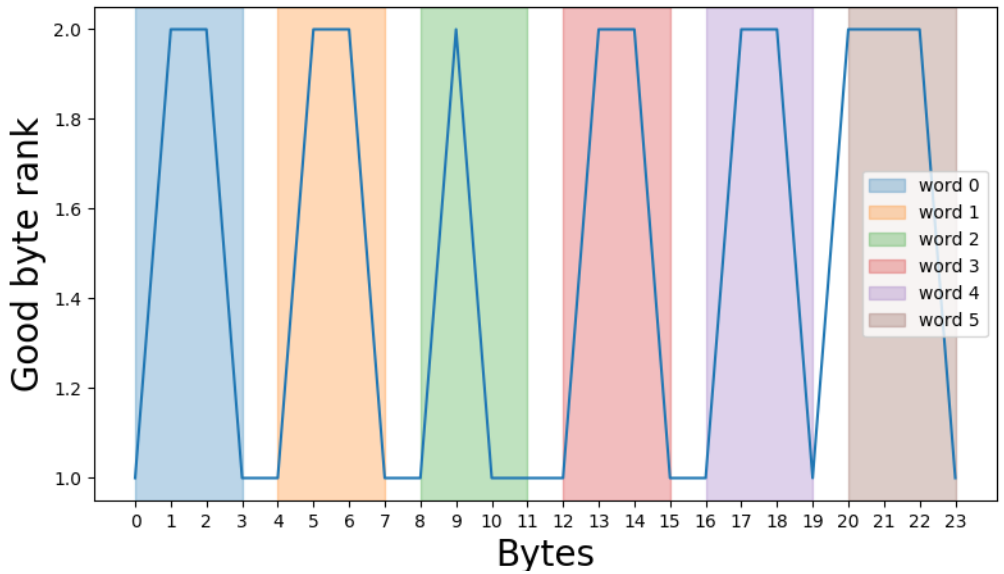
Markdown Python 3

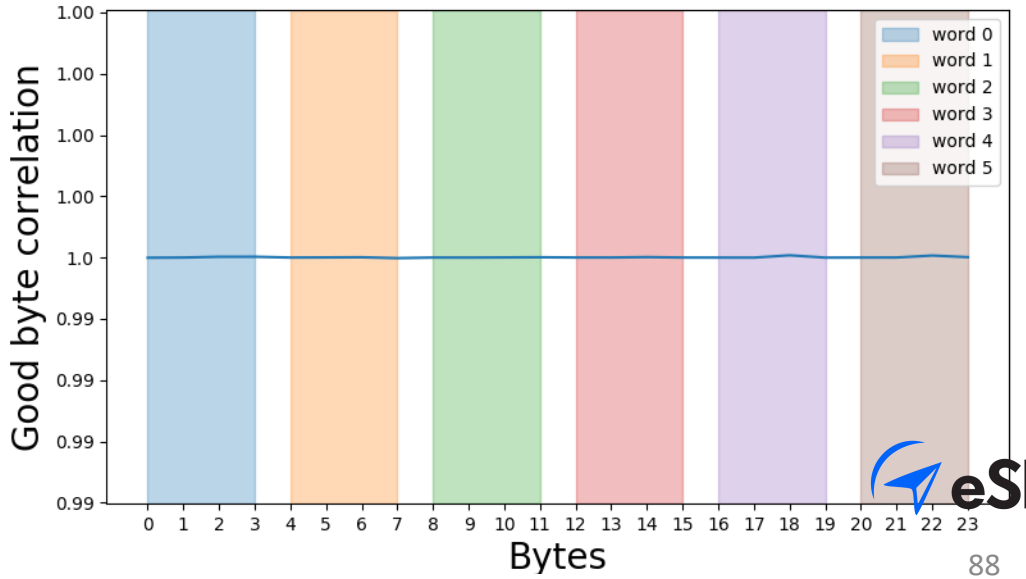
The attack succeeded!

We can plot the key byte ranking and correlation scores:

- For each word, we see the guess ranking on one, two, three and four bytes:
  - Nearly for each word, the correct key guesses on two and three bytes show up only the second place. As we take the 5 best guesses for these positions, this is not a problem,
  - When the key guess is done on four bytes, the best guess is ranked at the first place each time.
- We have always correlation score to 1 since we use the exact values to make the correlation (and not their hamming weight for instance)

[77]:
```
plot_results(ranks=res[0], correl=res[1], nbBits=32)
```





File Edit View Run Kernel Tabs Settings Help

Code Python 3

Conclusion

[78]:
```
slideshow('Introduction',
          './img/presentation_tracer',
          ['Diapositive%d.PNG' % hit for hit in range(82, 85)],
          slide_width)
```

▾ **Introduction**

Slide 0 | Slide 1 | Slide 2

AGENDA

- Introduction
- WB Execution & Monitoring
- ECDSA algorithm
- Recover ECDSA key with a DCA
- Trace acquisition
- DCA
- **Conclusion**

82

[ ]:

CONCLUSION

Tracing a White-Box must be focused on the binary.

Why trace directly without reverse engineering?

Fast

But:

Big size traces

Post treatment required & time consuming

eShard

CONCLUSION

Strategies to defeat these issues:

Focus tracing only on memory access or register access

Pattern detector to trace only interesting area

Mult eax, edx Depending on the algorithm, focus on specific instructions

In that way, it is possible to obtain small traces that still contain leakage points.

eShard