Exploring different faulting techniques for stressing White-Box cryptography binaries

eShard

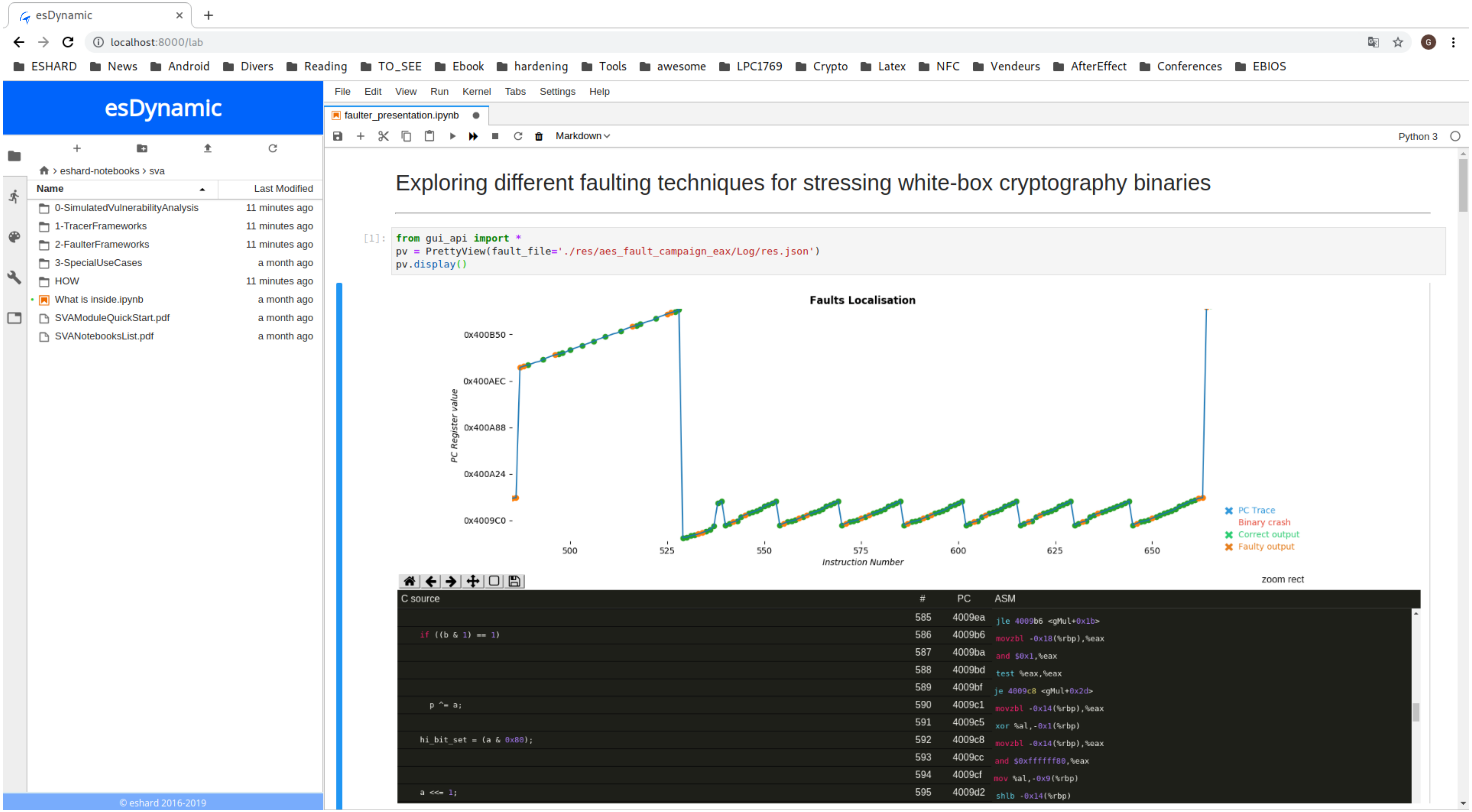
Guillaume VINET

19th May 2019

WhibOx 2019

- Binary analysis: dynamic fault injection is a powerful way to stress WBC-based solutions,
- Publications in this area remain modest, mostly due to challenging practical realisation,
- Registers, memory access can be changed in runtime leading to exploitable faulty computations,
- Nowadays, a state-of-the-art WBC security analysis must include:
  - static and dynamic fault injections
  - an efficient way to induce dynamic faulty computations: being precise and able to affect large range of instructions
  - a large range of public fault injection attacks exploiting single or multiple faults

How can this be achieved?



esDynamic
localhost:8000/lab
ESHARD News Android Divers Reading TO_SEE Ebook hardening Tools awesome LPC1769 Crypto Latex NFC Vendeurs AfterEffect Conferences EBIOS
esDynamic
File Edit View Run Kernel Tabs Settings Help
faulter_presentation.ipynb
Markdown
Python 3
eshard-notebooks › sva
Name
Last Modified
0-SimulatedVulnerabilityAnalysis 11 minutes ago
1-TracerFrameworks 11 minutes ago
2-FaulterFrameworks 11 minutes ago
3-SpecialUseCases a month ago
HOW 11 minutes ago
What is inside.ipynb a month ago
SVAModuleQuickStart.pdf a month ago
SVANotebooksList.pdf a month ago
© eshard 2016-2019
Exploring different faulting techniques for stressing white-box cryptography binaries
[1]:
from gui_api import *
pv = PrettyView(fault_file='./res/aes_fault_campaign_eax/Log/res.json')
pv.display()
Faults Localisation
0x400B50
0x400AEC
0x400A88
0x400A24
0x4009C0
PC Register value
500
525
550
575
600
625
650
Instruction Number
PC Trace
Binary crash
Correct output
Faulty output
zoom rect
C source
#
PC
ASM
585 4009ea jle 4009b6 <gMul+0x1b>
if ((b & 1) == 1)
586 4009b6 movzbl -0x18(%rbp),%eax
587 4009ba and $0x1,%eax
588 4009bd test %eax,%eax
589 4009bf je 4009c8 <gMul+0x2d>
p ^= a;
590 4009c1 movzbl -0x14(%rbp),%eax
591 4009c5 xor %al,-0x1(%rbp)
hi_bit_set = (a & 0x80);
592 4009c8 movzbl -0x14(%rbp),%eax
593 4009cc and $0xffffff80,%eax
594 4009cf mov %al,-0x9(%rbp)
a <<= 1;
595 4009d2 shlb -0x14(%rbp)

WHERE IS THE WHITE-BOX?

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

APK

- **JAVA**
- **Native library**

Secure functions ("secure VM")

White-Box Cryptography Library →

A5 D8 7E
5E DD 9F
BF A0 48
BIN

Native
binary file
(assembly code)

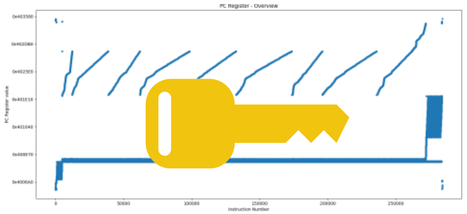
NO SOURCE FILES!

Why fault a white-box?

eShard

WHY FAULT A WHITE-BOX?

Native
binary file



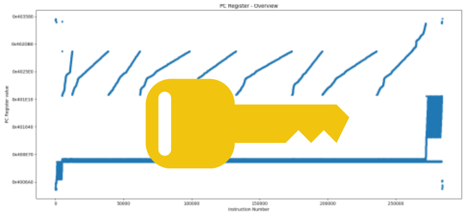
Cryptographic
attacks

- Differential Fault Analysis (DFA)
- Safe Error
- …

eShard

WHY FAULT A WHITE-BOX?

Native
binary file



Cryptographic
attacks

- Differential Fault Analysis (DFA)
- Safe Error
- ...

Security
downgrade

- Defeat integrity mechanisms
- Defeat algorithm protection.

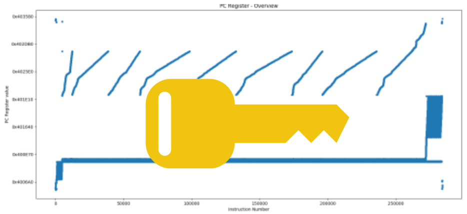
Stuck mask value to transform a 2nd order attack to a 1rst order

- ...

eShard

WHY FAULT A WHITE-BOX?

Native binary file



Cryptographic attacks

Security downgrade

How can we find fault parameters without a COMBINATORIAL COMPLEXITY?

eShard

File Edit View Run Kernel Tabs Settings Help

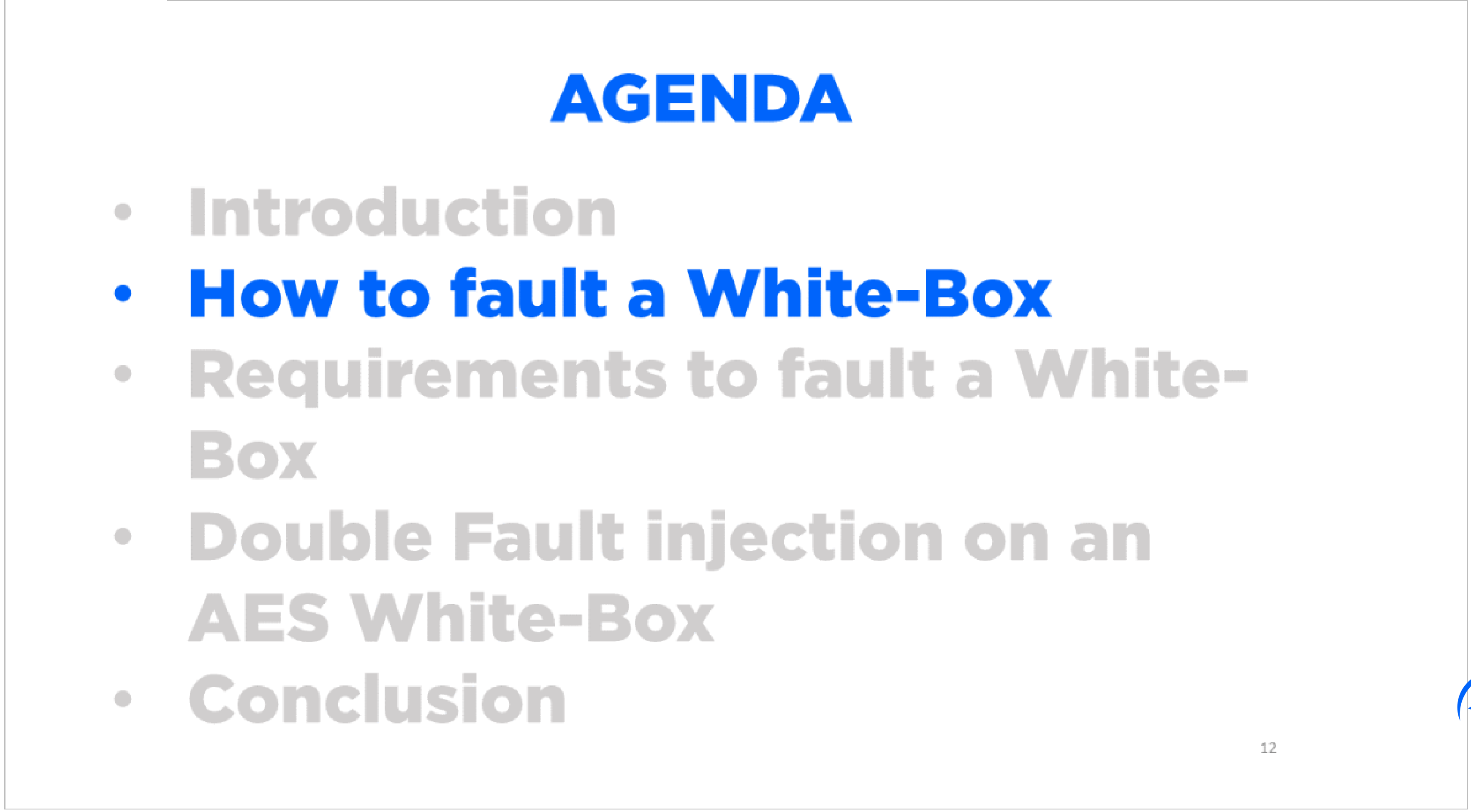
Code Python 3

How to fault a White-Box

[3]:
```
slideshow('How to fault a White-Box',
          './img/presentation_faulter',
          ['Diapositive%d.PNG' % hit for hit in range(12,21)],
          slide_width)
```

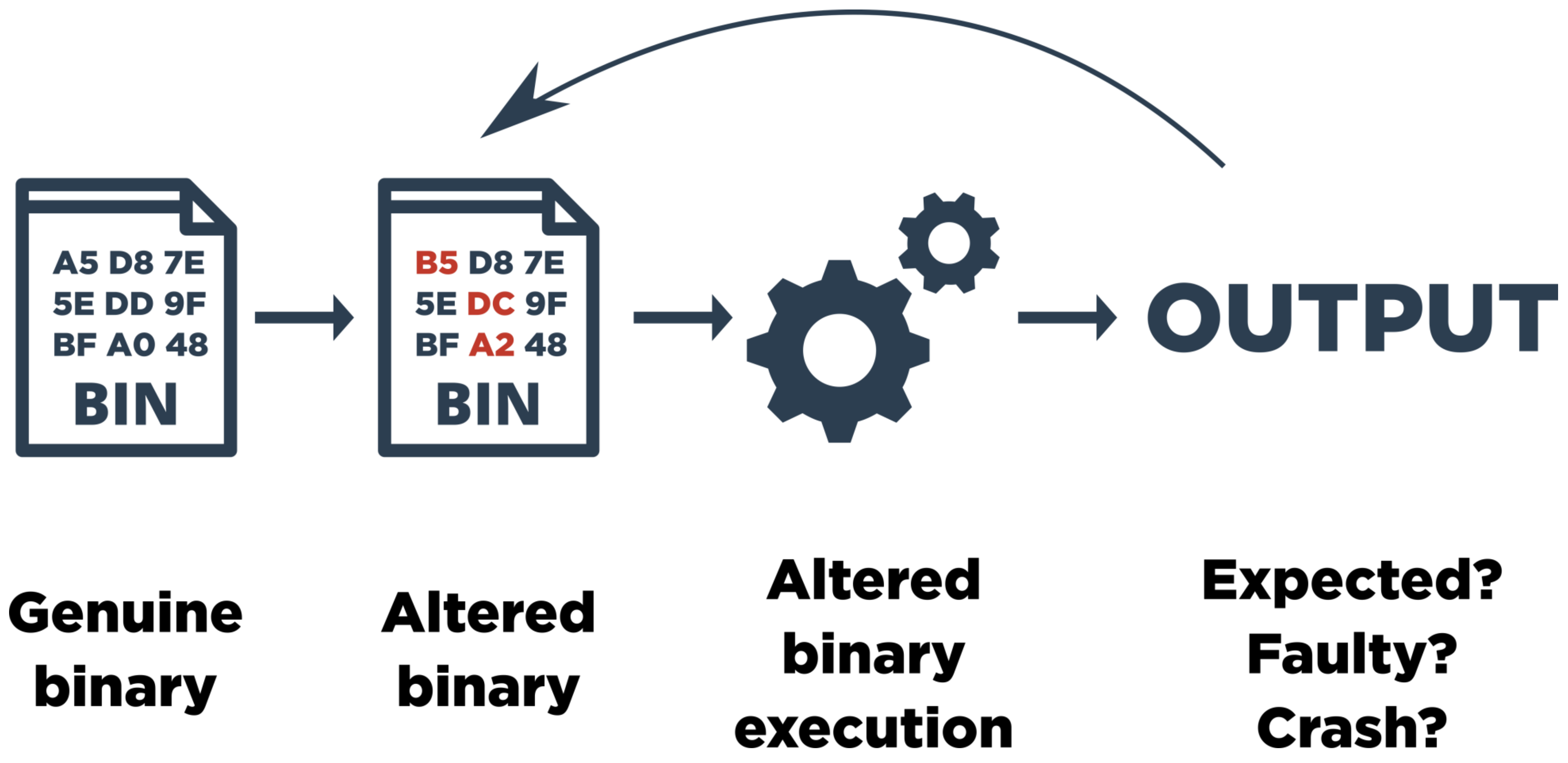
How to fault a White-Box

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



eShard

STATIC FAULT INJECTION



eShard

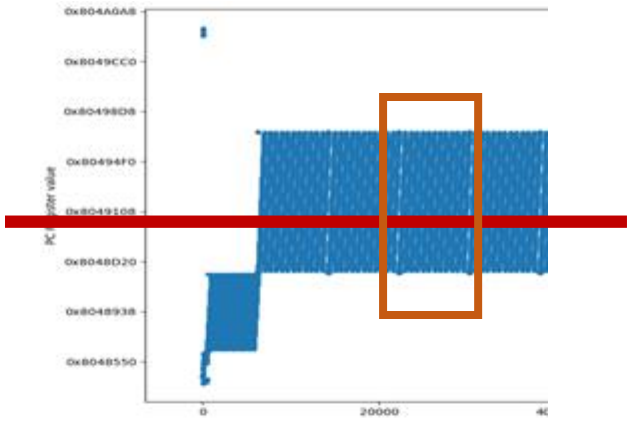
STATIC FAULT INJECTION

Assets:

- Easy to implement

Drawbacks:

- Speed: how to avoid combinatorial complexity with multiple fault injections?
- Accuracy: valuable to modify table value, but not disturbing operation execution
- Anti-Fault countermeasures: fault easily detected



eShard

STATIC FAULT TOOLS

Deadpool

included in

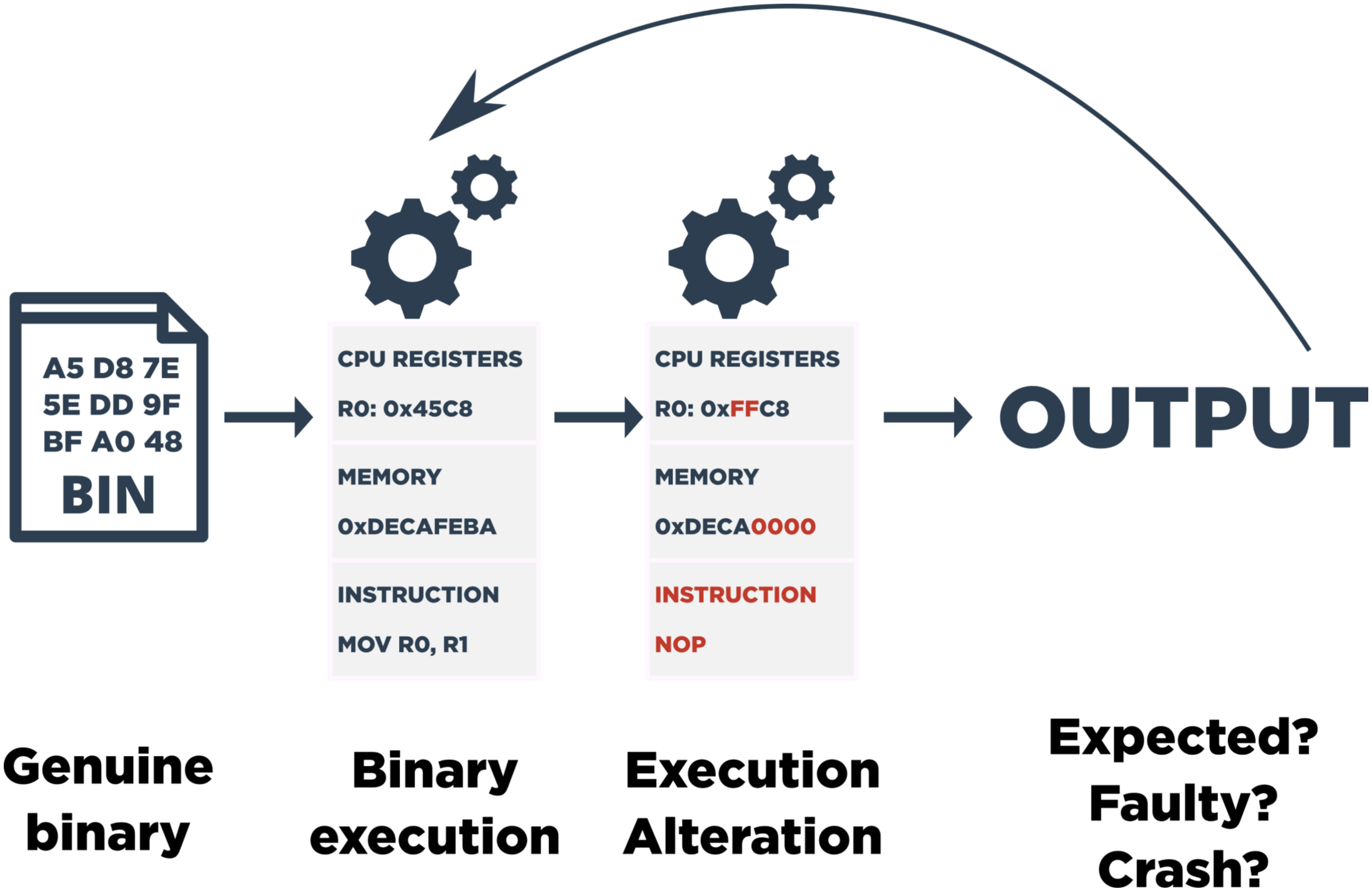
Side Channel Marvels Framework

https://github.com/SideChannelMarvels

- Python framework
- Tree strategy to inject the faults

eShard

DYNAMIC FAULT INJECTION



eShard

DYNAMIC FAULT INJECTION

Assets:

- Accuracy:
    - Alter registers, memory or instructions
    - Multiple fault injection to defeat security countermeasures

Drawbacks:

- Fault Model: which fault effects must be implemented
- Speed: how to avoid combinatorial complexity with multiple fault injections?

eShard

DYNAMIC FAULT INJECTION

Assets:
- Open source
- Use the powerful Unicorn Engine…

Unicorn

eShard

DYNAMIC FAULT INJECTION

```
# Patch printf and putchar .plt --> return
mu.mem_write(0x80483BC, "\xc3")
mu.mem_write(0x80483EC, "\xc3")
```

Call to external libraries must be implemented/patched

```
def hook_mem_access_fault(uc, access, address, size, value, user_data):
    global output, evtId, fault
    evtId += 1
    pc = uc.reg_read(UC_X86_REG_EIP)

    targetId = user_data[0]
    if access == UC_MEM_READ:
        value = u32(uc.mem_read(address, size))
        if should_fault(evtId, targetId, fault, address, size):
            print "FAULTING AT ", targetId
            # Already faulted this time
            fault = False
            # Random bit in this event
            bitfault = 1 << random.randint(0, size*8 -1)
            uc.mem_write(address, pack(value ^ bitfault, size))

    # At this PC the pushes a byte of output to the stack
    if pc == 0x08049e4f:
        output.append(value)
```

Know where to recover the ciphertext once the fault was injected

eShard

Source https://www.riscure.com/uploads/2017/09/eu-15-sanfelix-mune-dehaas-unboxing-the-white-box-wp_v1.1.pdf

DYNAMIC FAULT INJECTION

Unicorn

Assets:

- Open source
- Use the powerful Unicorn Engine…

Drawbacks:

- … that needs reverse engineering
- Executable/Library must be instrumented by a script
- The Unicorn emulation is slow

eShard

File Edit View Run Kernel Tabs Settings Help

Code Python 3

Requirements to fault a White-Box

[4]:

```
slideshow('Requirements to fault a White-Box',
          './img/presentation_faulter',
          ['Diapositive%d.PNG' % hit for hit in range(21,30)],
          slide_width)
```

Requirements to fault a White-Box

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

AGENDA

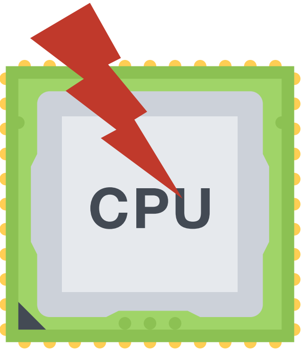
- Introduction
- How to fault a White-Box
- **Requirements to fault a White-Box**
- Double Fault injection on an AES White-Box
- Conclusion

21

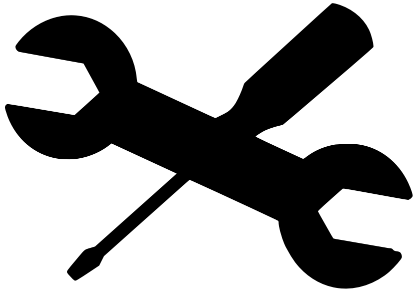
eShard

REQUIREMENTS TO FAULT

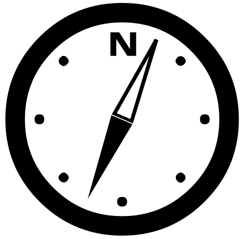
Dynamic fault injection



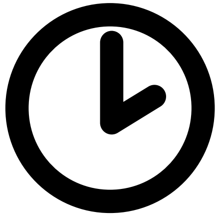
Relevant Fault Models



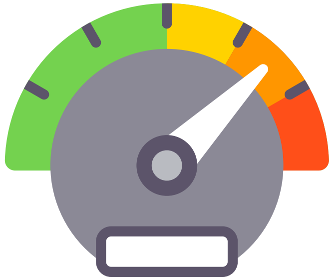
Configuration



Where?



When?



Speed

REQUIREMENTS TO FAULT

Faults models:

- Register modification

SET TO ZERO

ADD USER VALUE

XOR USER VALUE

SET USER VALUE

SET RANDOM

eShard

WHAT ARE WE LOOKING FOR?

Faults models:

- Register modification
  - Data Flow Modification

SET TO ZERO

ADD USER VALUE

XOR USER VALUE

SET USER VALUE

SET RANDOM

eShard

REQUIREMENTS TO FAULT

Faults models:

- Register modification
  - Data Flow Modification
  - Control Flow Modification with Program Counter Register

SET TO ZERO

ADD USER VALUE

XOR USER VALUE

SET USER VALUE

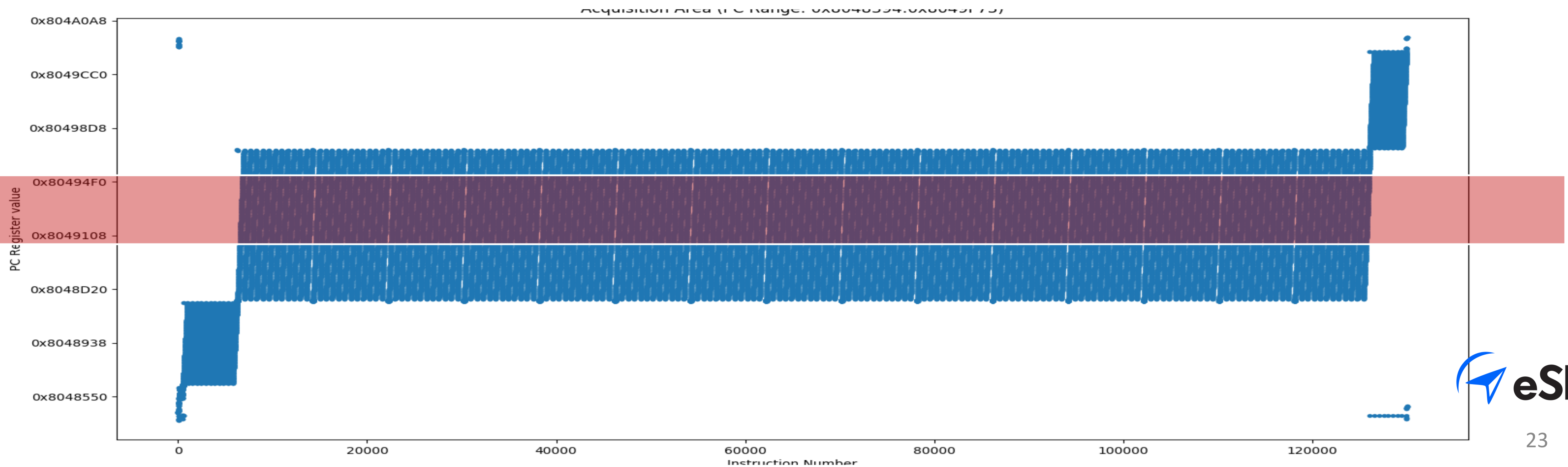
SET RANDOM

eShard

REQUIREMENTS TO FAULT

Where to fault:

- Filtering based on:
  - Program Counter
  - Kind of instruction: mov, add …

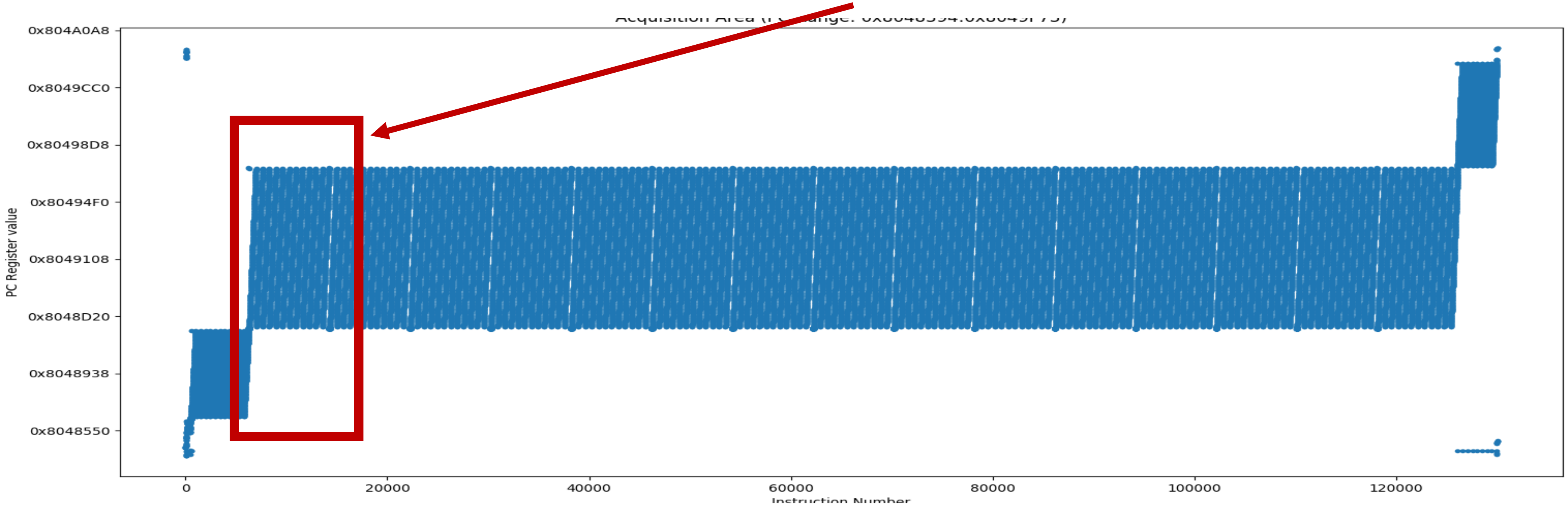


eShard

REQUIREMENTS TO FAULT

When to fault: pattern detector

- **We must start the faults when the correct PCs are reached …**
- **… and we also stop them in the same way**

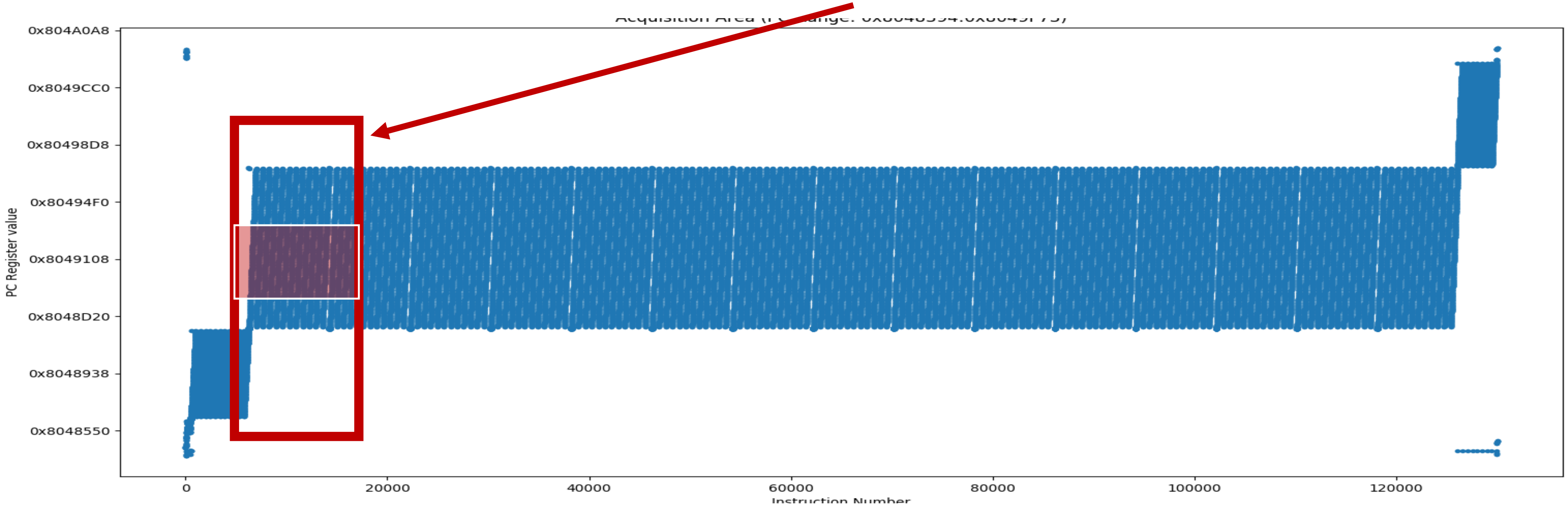


eShard

REQUIREMENTS TO FAULT

When to fault: pattern detector

- **It can be combined with a filtering based on Program Counter value or Instruction kind**



eShard

REQUIREMENTS TO FAULT

eShard

esFaulter

included in

esDynamic Platform

→

- Dynamic register modification
  - Data flow disturbance
  - Control flow disturbance

→

- Multi-fault injection
- Fault&trace capabilities
- Filtering and trig&act capabilities

QEMU

- X86, x86_64, ARM support
- Takes advantage of Qemu speed

eShard

File Edit View Run Kernel Tabs Settings Help

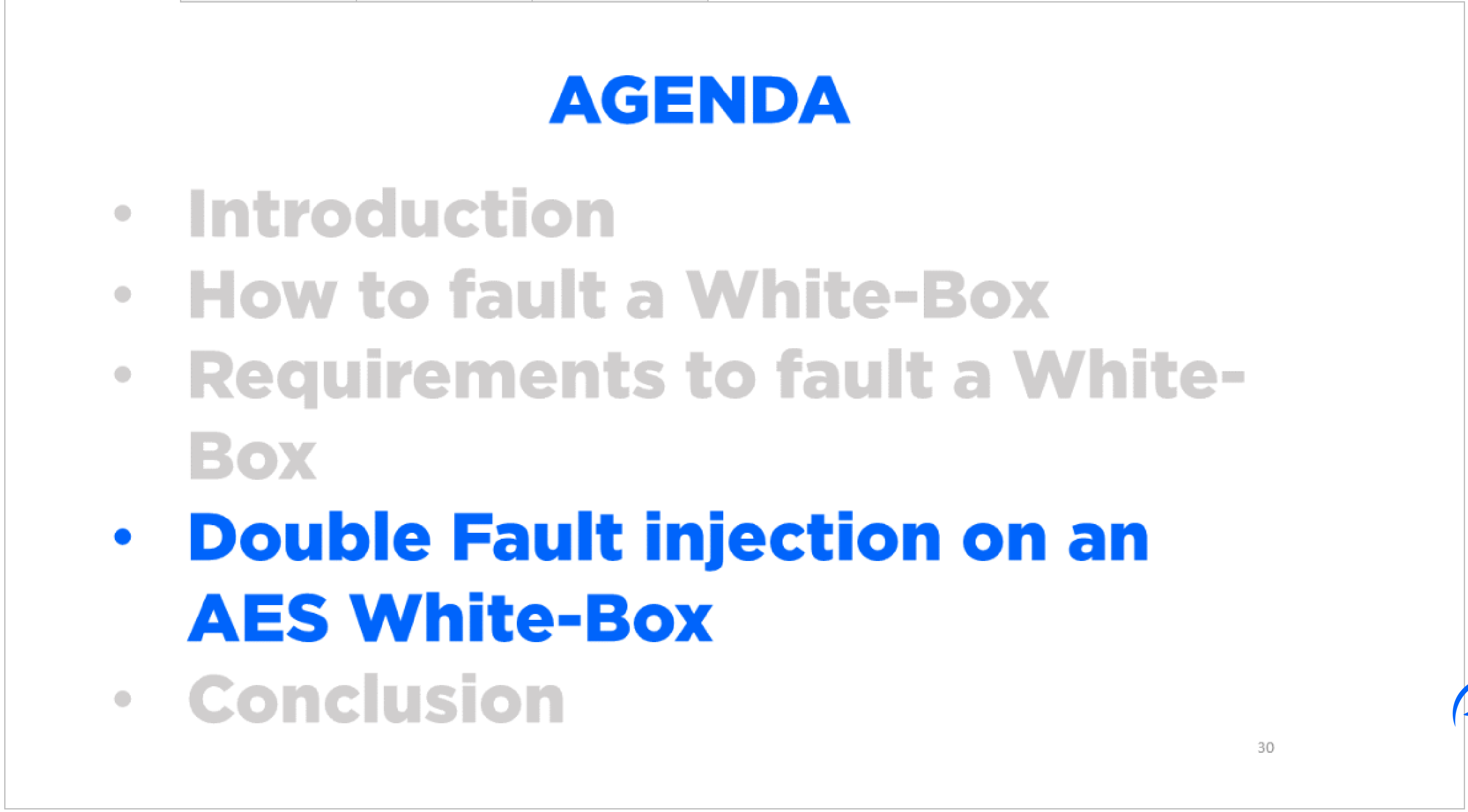
Code | Python 3

Double Fault injection on an AES White-Box

[5]:
```
slideshow('Introduction',
          './img/presentation_faulter',
          ['Diapositive%d.PNG' % hit for hit in range(30,34)],
          slide_width)
```

▾ **Introduction**

Slide 0 | Slide 1 | Slide 2 | Slide 3



eShard

THE CHALLENGE

- Attack an AES White-Box implementation
- Configuration:
  - CPU i7-7560U, 2.4GHz dual core
  - 16 GB of RAM (we not need so much)
  - SSD NVMe
- Double fault injection
- Key recovery from the faulty outputs with a DFA of Piret (with 4 modified bytes in a specific way)

eShard

THE CHALLENGE

- Attack an AES White-Box implementation
- Double fault injection
- Key recovery from the faulty outputs with a DFA of Piret (with 4 modified bytes in a specific way)

How can we find faults parameters without a COMBINATORIAL COMPLEXITY ?

nb_ins x nb_fault_model x nb_target x nb_input x nb_area

eShard

ILLUSTRATION

AES-128
X86-64 architecture

```
$./cipheraes 06 1F C9 F5 88 B2 F9 D2 00 19 86 82 2C 12 11 79
message: 061fc9f588b2f9d2001986822c121179
cipher: 14ed01ea7ce2a551c9791ae85c7cecf4
```

Differential Fault Analysis with a double fault injection attack:

- Data flow disturbance
- Control flow disturbance

eShard

File Edit View Run Kernel Tabs Settings Help

Markdown Python 3

Our target is a x86-64 architecture executable.

[6]:
```
!file ./binary/cipheraes
```

```
./binary/cipheraes: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-6
4.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=b4bdf121200bbae155c26bc03d474e9f29eb980f, stripped, with debug_info
```

Let's see how to use it.

[7]:
```
!./binary/cipheraes
```

```
cipheraes B01 B02 B03 B04 B05 B06 B07 B08 B09 B10 B11 B12 B13 B14 B15 B16, BXX is hexadecimal
arguments problem
```

We need to provide a 16-byte hexadecimal input, and then get the ciphered message.

[8]:
```
!./binary/cipheraes 061FC9F588B2F9D2001986822C121179
```

```
cipheraes B01 B02 B03 B04 B05 B06 B07 B08 B09 B10 B11 B12 B13 B14 B15 B16, BXX is hexadecimal
arguments problem
```

The input format must be carefully respected in order to get the ciphered message.

[9]:
```
!./binary/cipheraes 06 1F C9 F5 88 B2 F9 D2 00 19 86 82 2C 12 11 79
```

```
message: 061fc9f588b2f9d2001986822c121179
cipher: 14ed01ea7ce2a551c9791ae85c7cecf4
```

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

eShard

File Edit View Run Kernel Tabs Settings Help

Markdown Python 3

Where to attack?

[10]:
```
slideshow('Where to attack?',
          './img/presentation_faulter',
          ['Diapositive%d.PNG' % hit for hit in range(34,36)],
          slide_width)
```

Where to attack?

Slide 0 | Slide 1

FIRST STEP

Where inject faults?

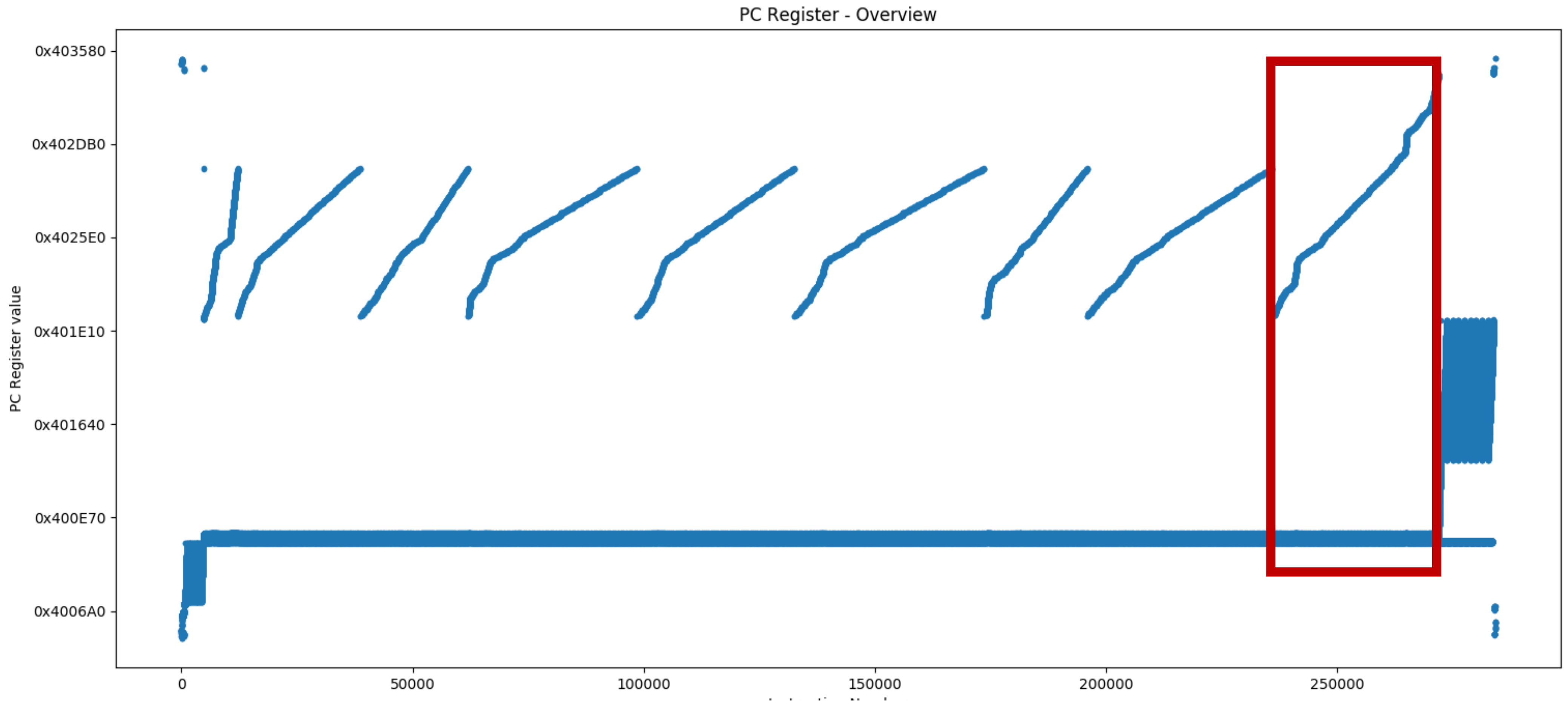
- only the binary itself, not external system libraries

How to know where to inject faults?

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

34

SIMPLE FAULT INJECTION



AES round patterns? Let's attack the last one

eShard

File Edit View Run Kernel Tabs Settings Help

Markdown Python 3

Before performing fault injections, we need to know where to inject them.

We are going to use esTracer to acquire the PC register evolution, and then, we analyze its graphical representation.

First, we configure the binary handler configuration object. It eases the management of the binary to attack.

[11]:
```
from essva.binary_handler import BinaryHandlerConfiguration
import re

bin_hand = BinaryHandlerConfiguration()

bin_hand.algo = "AES-128"
bin_hand.cmd = "./binary/cipheraes"
```

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

[12]:
```
def process_input(block):
    return " ".join("%02X" % hit for hit in bytes.fromhex(block)).lower()

def process_output(output):
    return re.match("message: [a-f,0-9]{32}\ncipher: ([a-f,0-9]{32})?", output).group(1)

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

Code

Python 3

We trace the program scanning only the PC register.

[13]:
```
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.

[14]:
```
tracer.registers      = ["pc"]
tracer.filt_pre.in_pc.ranges =  [[0x400458, 0x4034DC]]
```

We indicate where the traces will be generated.

[15]:
```
tracer.directory_out = "traces"
```

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]:
```
plain_input = "061FC9F588B2F9D2001986822C121179"
```

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

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

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

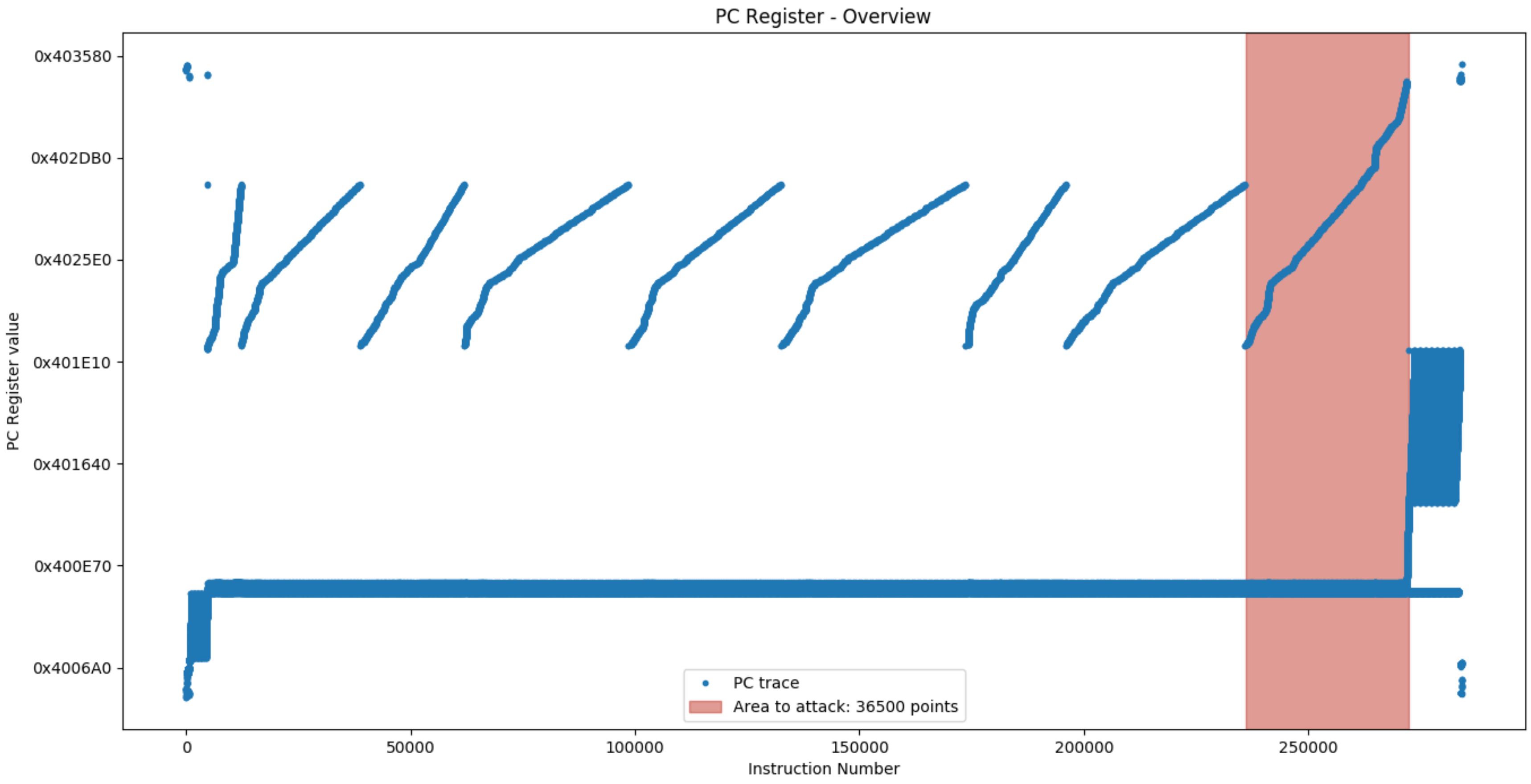
eShard

File Edit View Run Kernel Tabs Settings Help

faulter_presentation.ipynb

Markdown

Python 3



You can explore the PC trace by zooming in.

We see 9 repeated patterns, it might correspond to the AES round (the two last rounds are likely to be merged together). We decide to attack the last pattern area.

eShard

File Edit View Run Kernel Tabs Settings Help

Markdown Python 3

Fault injection campaign

We will inject a simple fault attack to perform a DFA of Piret, it means we have to obtain outputs with 4 faulty bytes.

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

[21]:
```
from essva.faulter import Faulter
import essva.sva_constants as const

faulter = Faulter(bin_hand)
```

We indicate where to generate the log.

[22]:
```
faulter.directory_out = "faulter_outputs_one_fault"
```

We use two trig&act to focus on the last pattern area:

- when the PC 0x400cd2 was called 5297 times, we start to inject faults
- when the PC 0x400c5c was calld 4938 times, we stop the fault injection

See the presentation "Optimize your binary tracing: an example with an ECDSA implementation" for more details about the trig&act.

[23]:
```
nb_inst_start, nb_inst_end = 236000, 272500
pc_start, pc_end = int(pc_np[nb_inst_start]), int(pc_np[nb_inst_end])
pc_start_count, pc_end_count = len(np.where(pc_np[:nb_inst_start+1] == pc_start)[0]), len(np.where(pc_np[:nb_inst_end+1] == pc_end
```

[24]:
```
trig_and_act_start_fault_injection = [pc_start, pc_start_count, const.TAA_F_START_FAULT]
trig_and_act_stop_fault_injection = [pc_end, pc_end_count, const.TAA_F_STOP_FAULT]
```

eShard

File Edit View Run Kernel Tabs Settings Help

faulter_presentation.ipynb

Markdown

We are going to fault all the registers, excepting the PC.

[ ]:
```
reg_lst = ["rax", "rcx", "rdx", "rbx", "rsp", "rbp",
           "rsi", "rdi", "r8", "r9", "r10", "r11", "r12",
           "r13", "r14", "r15"]
```

To configure a fault, we use the FaultCfgMono object:

- we indicate the register to fault
- we indicate which fault model to use, in our case we set the register to zero
- we indicate when the fault must be injected with the two trig&act created previously

Then, it is linked to esFaulter with `faulter.faults_opt` that is a list of FaultCfgMono object:

- in that way, you can easily perform simple or multiple fault injection
- esFaulter analyzes the configuration of each fault to make **automatic** and **exhaustive** fault injection

[ ]:
```
from essva.faulter import FaultCfgMono

for register in reg_lst:
    fault_1 = FaultCfgMono(bin_hand)

    fault_1.trig_and_acts = [trig_and_act_start_fault_injection,
                             trig_and_act_stop_fault_injection]

    fault_1.register = register
    fault_1.model = const.FM_SET_ZERO

    faulter.faults_opt = [fault_1]
    faulter.directory_out = "fault_1_%s" % register

    faulter.generate_faults(plain_input)
```

eShard

File Edit View Run Kernel Tabs Settings Help

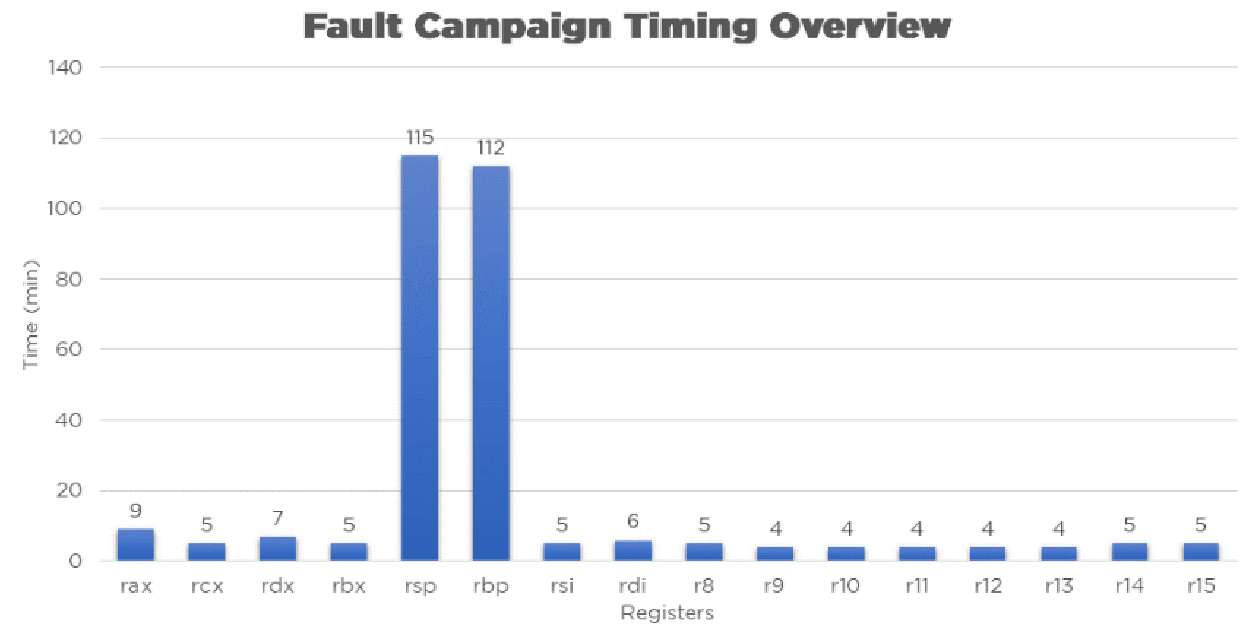
Markdown Python 3

[28]:
```python
slideshow('Simple Fault Injection',
          './img/presentation_faulter',
          ['Diapositive%d.PNG' % hit for hit in range(36,39)],
          slide_width)
```

Simple Fault Injection

Slide 0 | Slide 1 | Slide 2

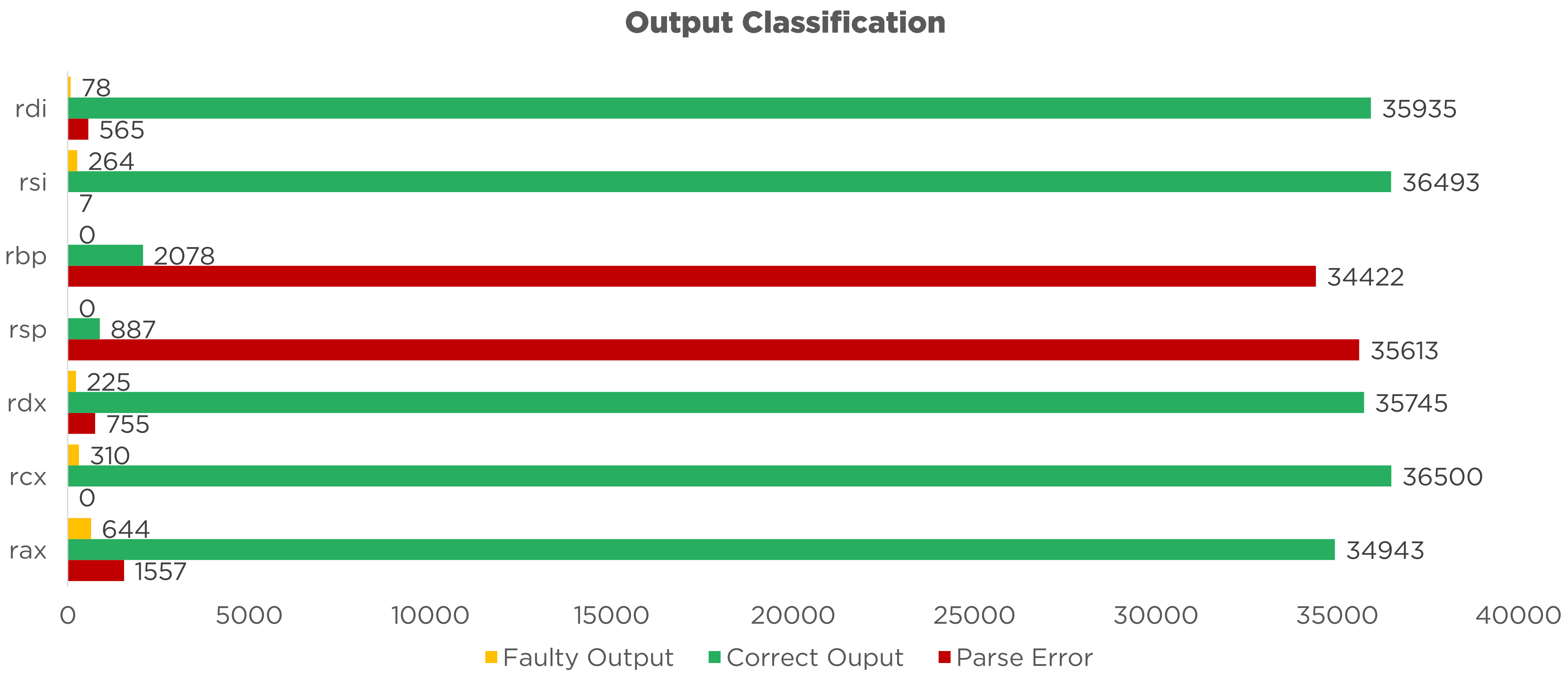
SIMPLE FAULT INJECTION



36

eShard

SIMPLE FAULT INJECTION



No effect for the other registers (rbx, r8, r9, r10, r11, r12, r13, r14, r15)

eShard

SIMPLE FAULT INJECTION

- 14 campaigns faulting one register (rsp/rbp not included)
- 511,000 injected faults
- ~76 min (multi-thread not used)
- ~112 injected faults by second

File Edit View Run Kernel Tabs Settings Help

faulter_presentation.ipynb

Markdown Python 3

RSP Register Fault analysis

We analyze the log for the rsp fault injection. Instead of looking at the log file, we can use a log parser to have a pretty view.

[ ]:
```
from essva.esdynamic.faulter_output import FaulterLogParser
import glob
import re

logfile = glob.glob("./fault_1_rsp/*.bin")[0]
parser = FaulterLogParser(logfile, output_parser = bin_hand.process_output)
parser.parse_file()
# Print faulty output
parser.get_faulty_output()
```

Most of the fault leads to the crash of the application. It is an expected behavior since we set the stack pointer to 0. The same behavior happens for RBP, which is the base pointer.

RAX Register Fault analysis

We analyze the log for the rax fault injection.

[ ]:
```
from essva.esdynamic.faulter_output import FaulterLogParser
import glob
import re

logfile = glob.glob("./fault_1_rax/*.bin")[0]
parser = FaulterLogParser(logfile, output_parser = bin_hand.process_output)
parser.parse_file()
# Print faulty output
parser.get_faulty_output()
```

The program never crashed. But, all the faults have the same effects: we obtain a block of 16 bytes set to 0.

File Edit View Run Kernel Tabs Settings Help

Code

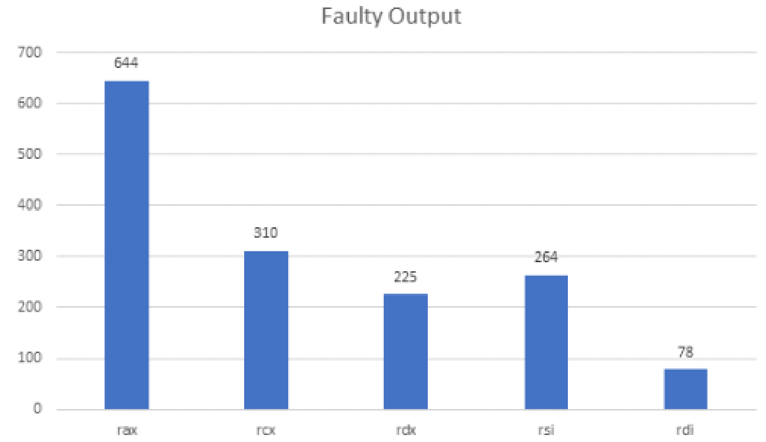
Python 3

[31]:
```
slideshow('Introduction',
          './img/presentation_faulter',
          ['Diapositive%d.PNG' % hit for hit in range(39,43)],
          slide_width)
```

Introduction

Slide 0 | Slide 1 | Slide 2 | Slide 3

SIMPLE FAULT INJECTION



All faulty ouputs return a ciphertext all bytes set to zero!

How can we interpret this behaviour?

eShard

HOW TO INTERPRET THE FAULTS?

Correct/
Faulty output
analysis

→

Execute algorithm:
- To recover the key
- Or to detect in which round the fault was injected
- A lot of public algorithm available, but if they fail it gives no information

HOW TO INTERPRET THE FAULTS?

Correct/
Faulty output
analysis

Execute algorithm:
- To recover the key
- Or to detect in which round the fault was injected
- A lot of public algorithm available, but if they fail it gives no information

Reverse
engineering

- Understand the effect of the fault on the program execution
- Give a way to understand very accurately the fault but it requires reverse engineering skills

HOW TO INTERPRET THE FAULTS?

Correct/ Faulty output analysis → Execute algorithm:
- To recover the key
- Or to detect in which round the fault was injected
- A lot of public algorithm available, but if they fail it gives no information

Reverse engineering →
- Understand the effect of the fault on the program execution
- Give a way to understand very accurately the fault but it requires reverse engineering skills

Fault & Trace →
- Fault and trace at the same time (memory access, Program Counter registers …)
- Give a visual way to understand accurately the fault effect without reverse engineering skills

LET TRY THIS WAY

eShard

File Edit View Run Kernel Tabs Settings Help

Markdown Python 3

This first try is a failure: we did not manage to have faulty outputs with 4 modified bytes. However, a lot of time, we have a block of 16 bytes set to 0. It is very strange to obtain these results.

Which analysis can we do to understand these results?

- We could reverse engineering the binary to understand the effect of each injected fault. It could take a long time.
- Otherwise, we could compare a correct execution and a faulted execution.
  - Indeed, with esFaulter and esTracer, it is very easy to fault and trace.
  - So, we can trace one or several registers during a correct execution without fault
  - And then, we trace in the same conditions excepting we inject the fault.
  - Finally, we can display the traces and compare them.

The comparison of the PC trace is a good start since it represents the address of the executed instructions.

eShard

File Edit View Run Kernel Tabs Settings Help

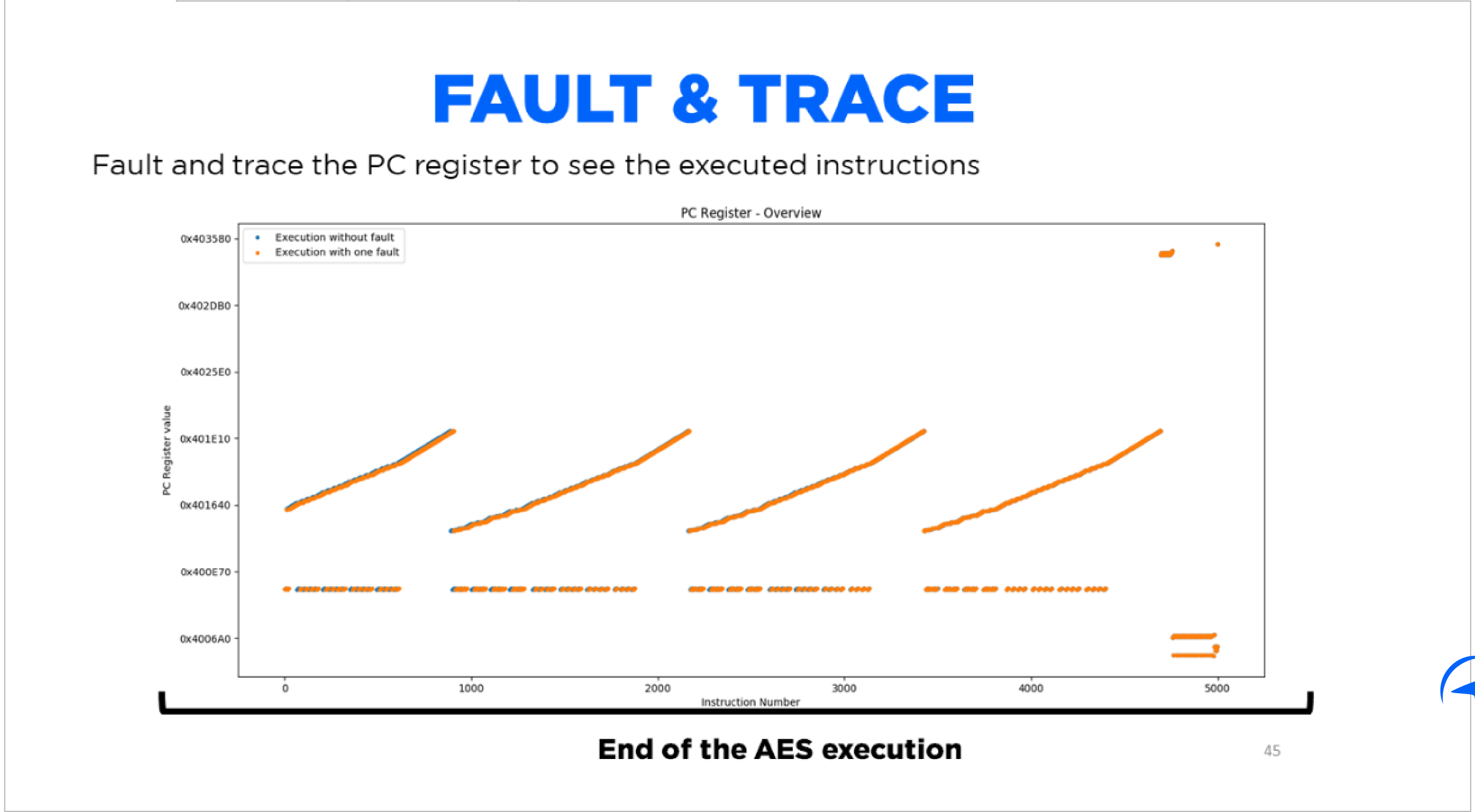
Markdown Python 3

Fault & trace

[58]:
```
slideshow('Introduction',
          './img/presentation_faulter',
          ['Diapositive%d.PNG' % hit for hit in [45, 47, 48]],
           slide_width)
```

Introduction

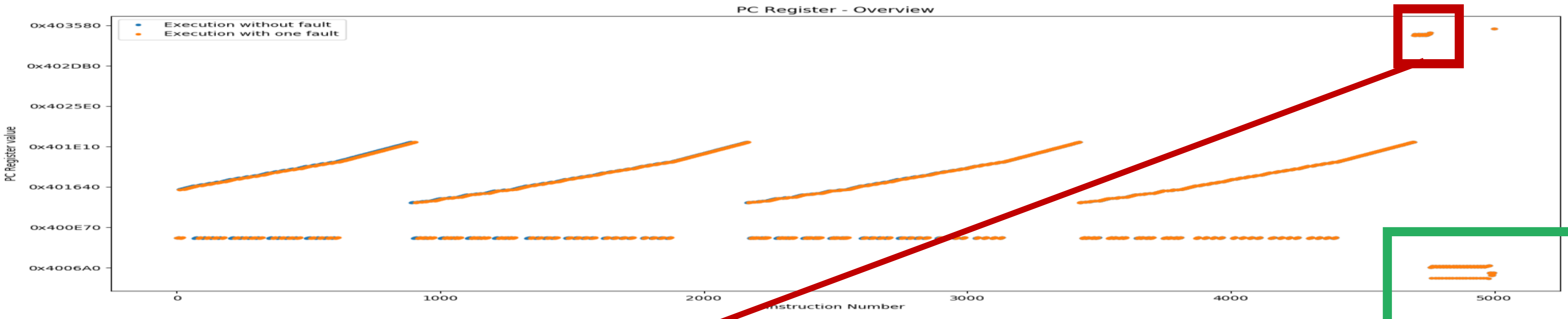
Slide 0 | Slide 1 | Slide 2



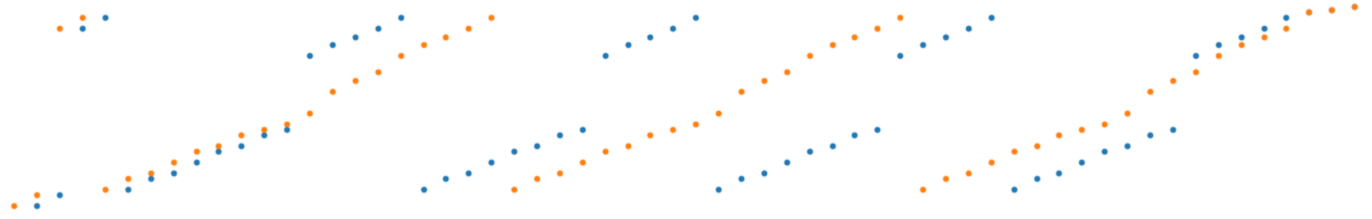
eShard

FAULT & TRACE

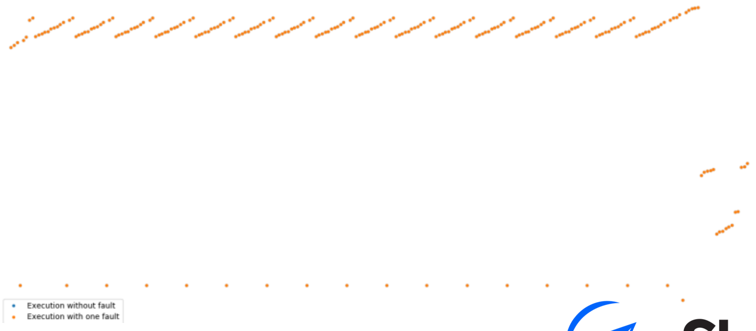
Fault and trace the PC register to see the executed instructions



Traces are different

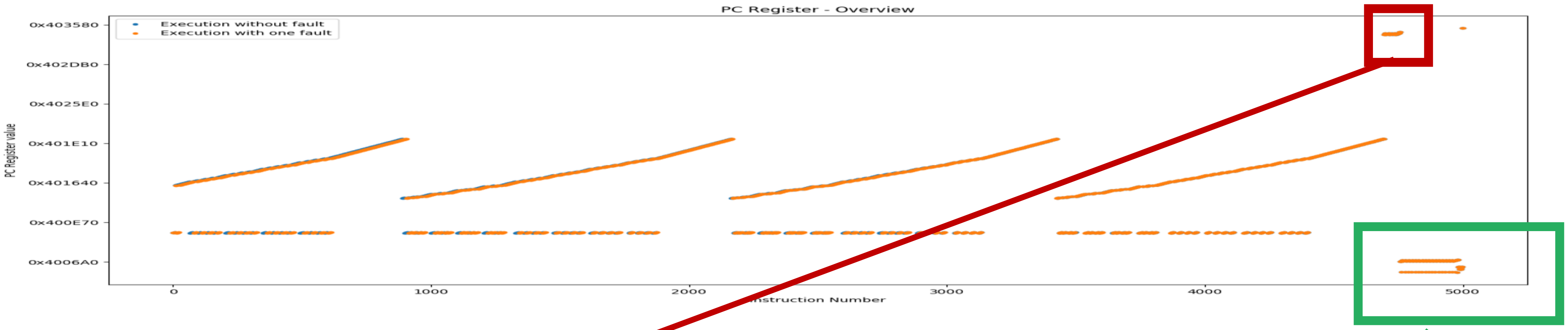


Traces are identical



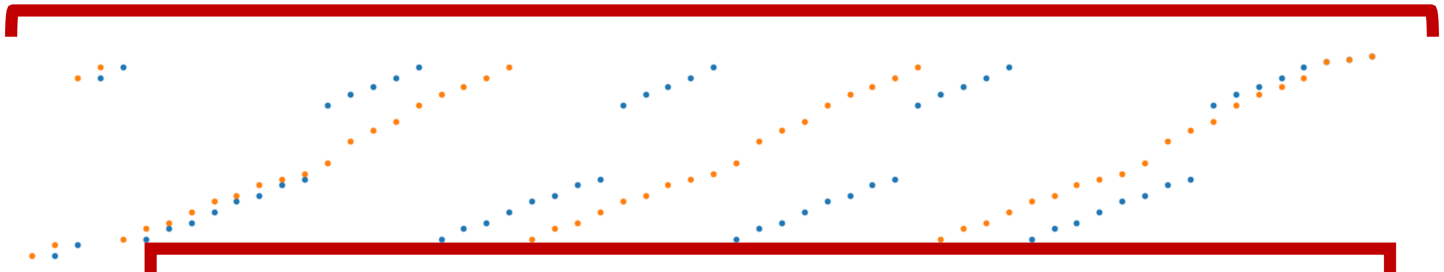
FAULT & TRACE

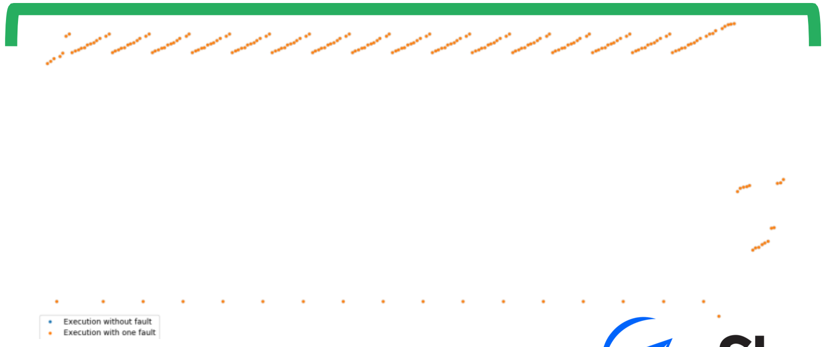
Fault and trace the PC register to see the executed instructions



Traces are different

Traces are identical





LET SEE WHAT IS THE CODE IN THIS AREA

eShard

File Edit View Run Kernel Tabs Settings Help

Markdown Python 3

Fault & trace

It is very easy to fault and trace:

- you create a tracer object
- you configure it
- and you link it to the faulter
- automatically, for each injected fault, a trace will be generated.

We have already a tracer recording the PC, so we will use it:

[32]:
```
faulter.tracer = tracer
```

We analyze the trace when we fault rax register.

- We will not once again inject 36500 faults, and create 36500 traces.
- We only need to inject one fault leading to a zero output.

So, we analyze a log to get a PC value triggering a zero output.

[34]:
```
for hit in parser.get_fault_dico()["OUTPUT"]["00000000000000000000000000000000"]["faults"]:
    pc_to_attack = int(hit[0]["pc"], 16)
hex(pc_to_attack)
```

[34]: '0x40138c'

This PC value might be called several time in the area we attack, so we use a trig&act to fault it just once.

[35]:
```
nb_inst_start_f1 = np.where(pc_np == 0x40138c)[0][0]
nb_inst_end_f1 = nb_inst_start_f1 + 1
pc_f1_start, pc_f1_end = int(pc_np[nb_inst_start_f1]), int(pc_np[nb_inst_end_f1])
pc_f1_start_count, pc_f1_end_count = len(np.where(pc_np[:nb_inst_start_f1+1] == pc_f1_start)[0]), len(np.where(pc_np[:nb_inst_end_
```

eShard

File Edit View Run Kernel Tabs Settings Help

Markdown Python 3

We modify the fault options:

- we set the PC we want to fault
- we set the trig&act

[36]:
```
fault_1.filt_pre.in_pc.pts = [pc_to_attack]
fault_1.trig_and_acts = [[pc_start, pc_start_count, const.TAA_F_START_FAULT],
                         [pc_end, pc_end_count, const.TAA_F_STOP_FAULT]]
fault_1.register = "rax"
```

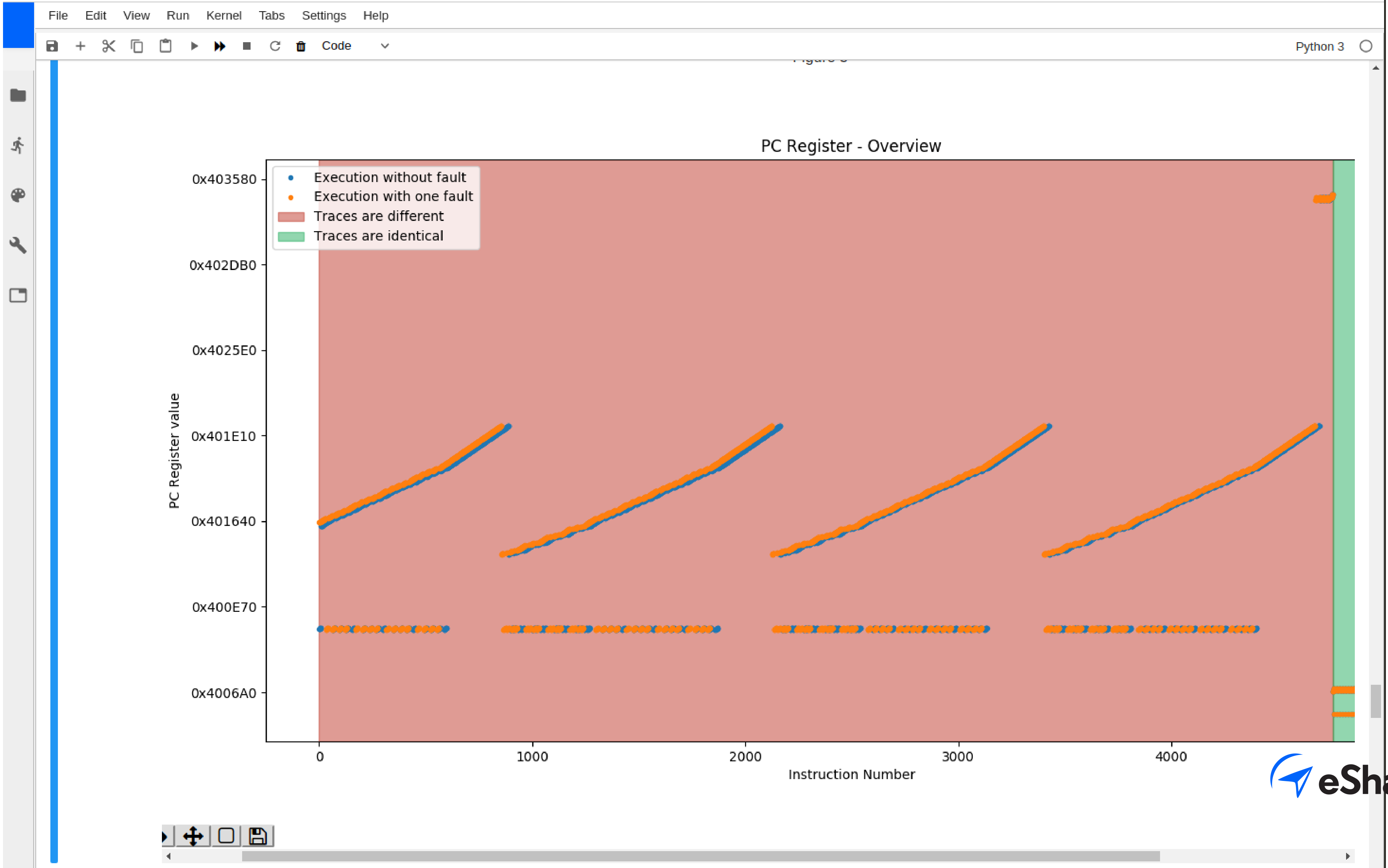
We inject the fault.

[37]:
```
faulter.directory_out = "fault_1_rax_with_trace"
faulter.generate_faults(plain_input)
```

```
Injecting 1 faults (estimation). Please wait...
```

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

eShard

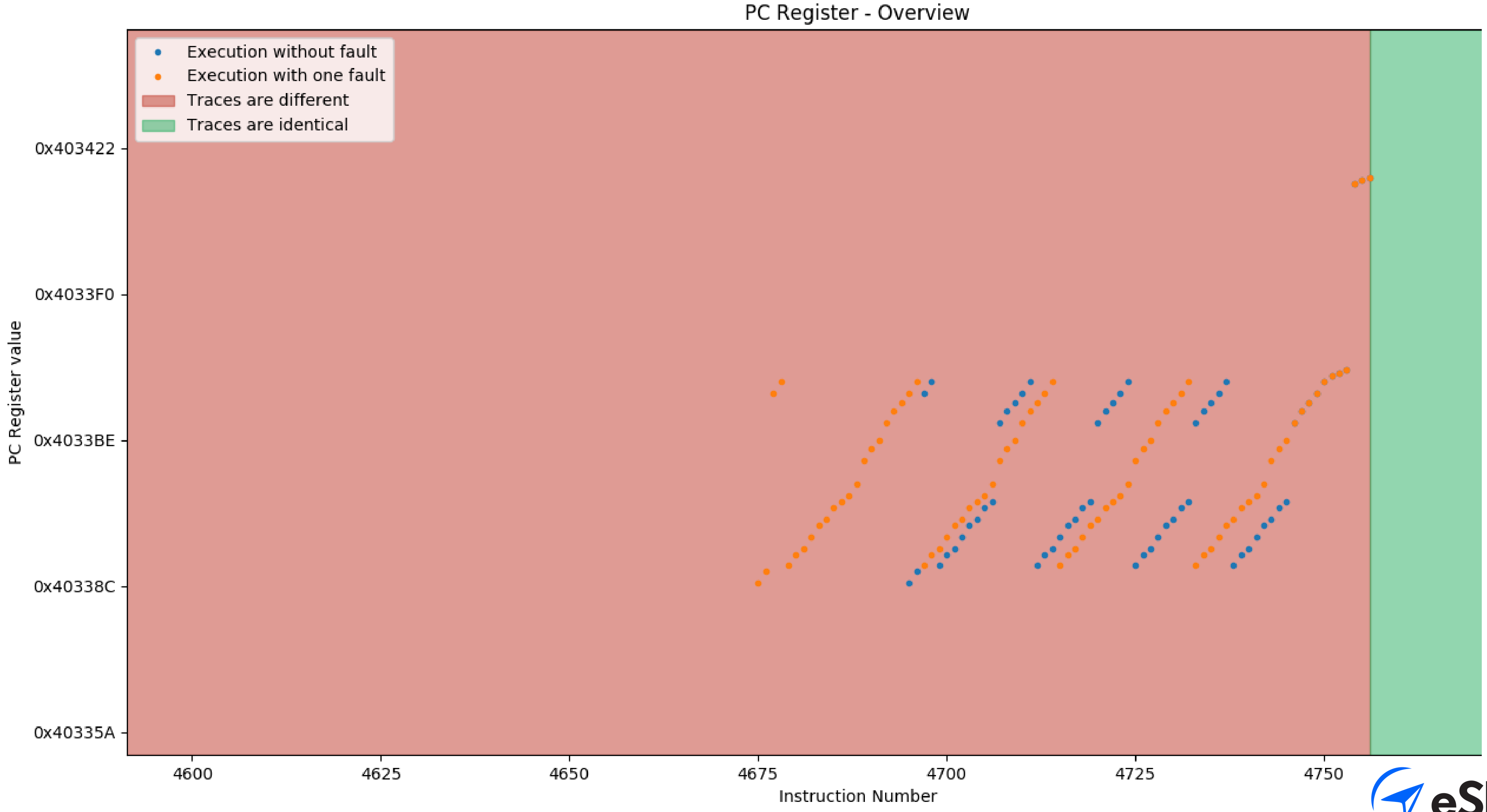


File Edit View Run Kernel Tabs Settings Help
Code
Python 3
PC Register - Overview
Execution without fault
Execution with one fault
Traces are different
Traces are identical
PC Register value
0x403580
0x402DB0
0x4025E0
0x401E10
0x401640
0x400E70
0x4006A0
0
1000
2000
3000
4000
Instruction Number

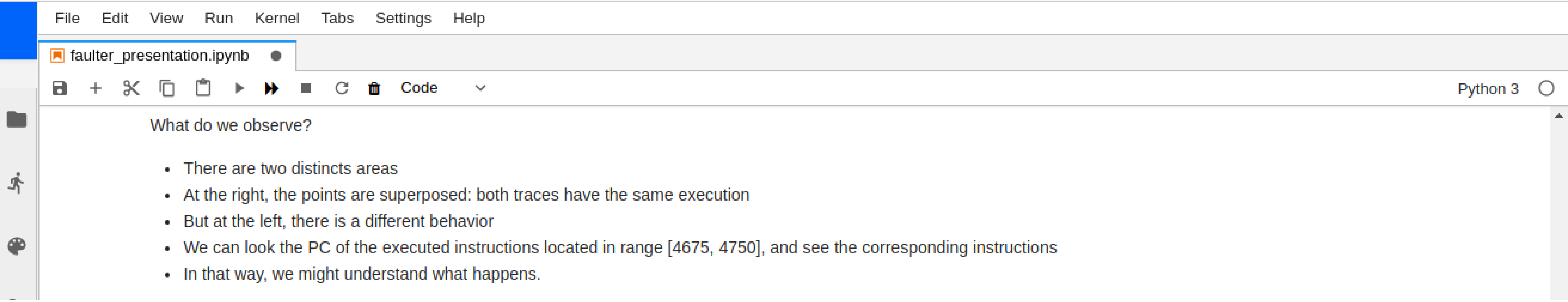
File Edit View Run Kernel Tabs Settings Help

Code

Python 3



eShard



What do we observe?

- There are two distincts areas
- At the right, the points are superposed: both traces have the same execution
- But at the left, there is a different behavior
- We can look the PC of the executed instructions located in range [4675, 4750], and see the corresponding instructions
- In that way, we might understand what happens.

File Edit View Run Kernel Tabs Settings Help

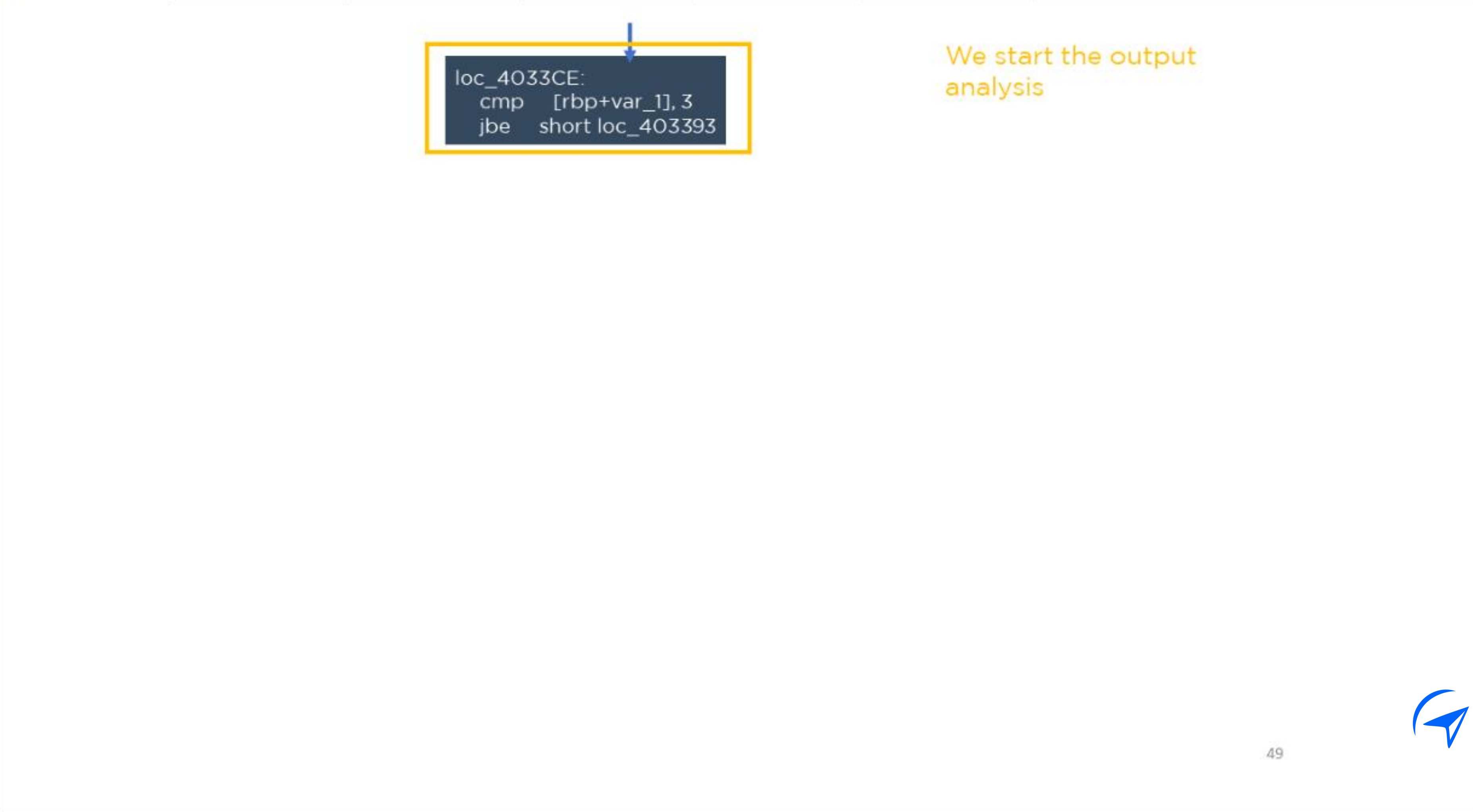
Code

Python 3

[59]:
```
slideshow('Introduction',
          './img/presentation_faulter',
          ['Diapositive%d.PNG' % hit for hit in [49, 50, 51, 52, 54, 55]],
           slide_width)
```

Introduction

Slide 0	Slide 1	Slide 2	Slide 3	Slide 4	Slide 5



eShard

```
loc_4033CE:
    cmp     [rbp+var_1], 3
    jbe     short loc_403393
```

```
loc_403393:
    movzx   eax, [rbp+var_1]
    cdqe
    mov     edx, [rbp+rax*4+var_20]
    movzx   eax, [rbp+var_1]
    cdqe
    mov     eax, [rbp+rax*4+var_30]
    cmp     edx, eax
    jz      short loc_4033C4
```

```
loc_4033D4
    nop
    leave
    retn
```

We start the output analysis

The output was computed twice. Its consistency is checked by block of four bytes.

eShard

```
loc_4033CE:
    cmp     [rbp+var_1], 3
    jbe     short loc_403393
```

```
loc_403393:
    movzx   eax, [rbp+var_1]
    cdqe
    mov     edx, [rbp+rax*4+var_20]
    movzx   eax, [rbp+var_1]
    cdqe
    mov     eax, [rbp+rax*4+var_30]
    cmp     edx, eax
    jz      short loc_4033C4
```

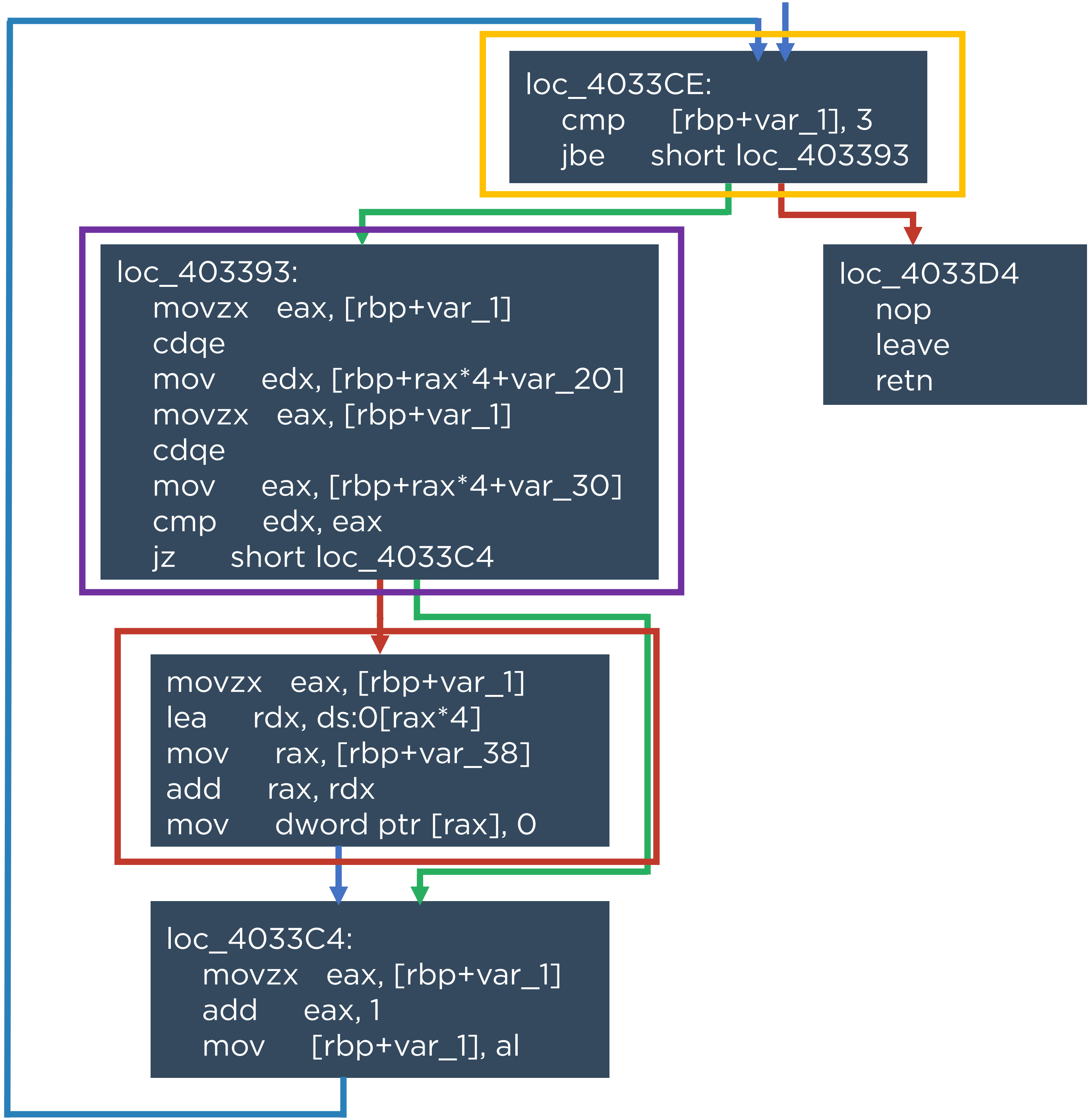
```
loc_4033D4
    nop
    leave
    retn
```

```
movzx   eax, [rbp+var_1]
lea     rdx, ds:0[rax*4]
mov     rax, [rbp+var_38]
add     rax, rdx
mov     dword ptr [rax], 0
```

We start the output analysis

The output was computed twice. Its consistency is checked by block of our bytes.

In case of a failure, the four-byte block is set to zero.



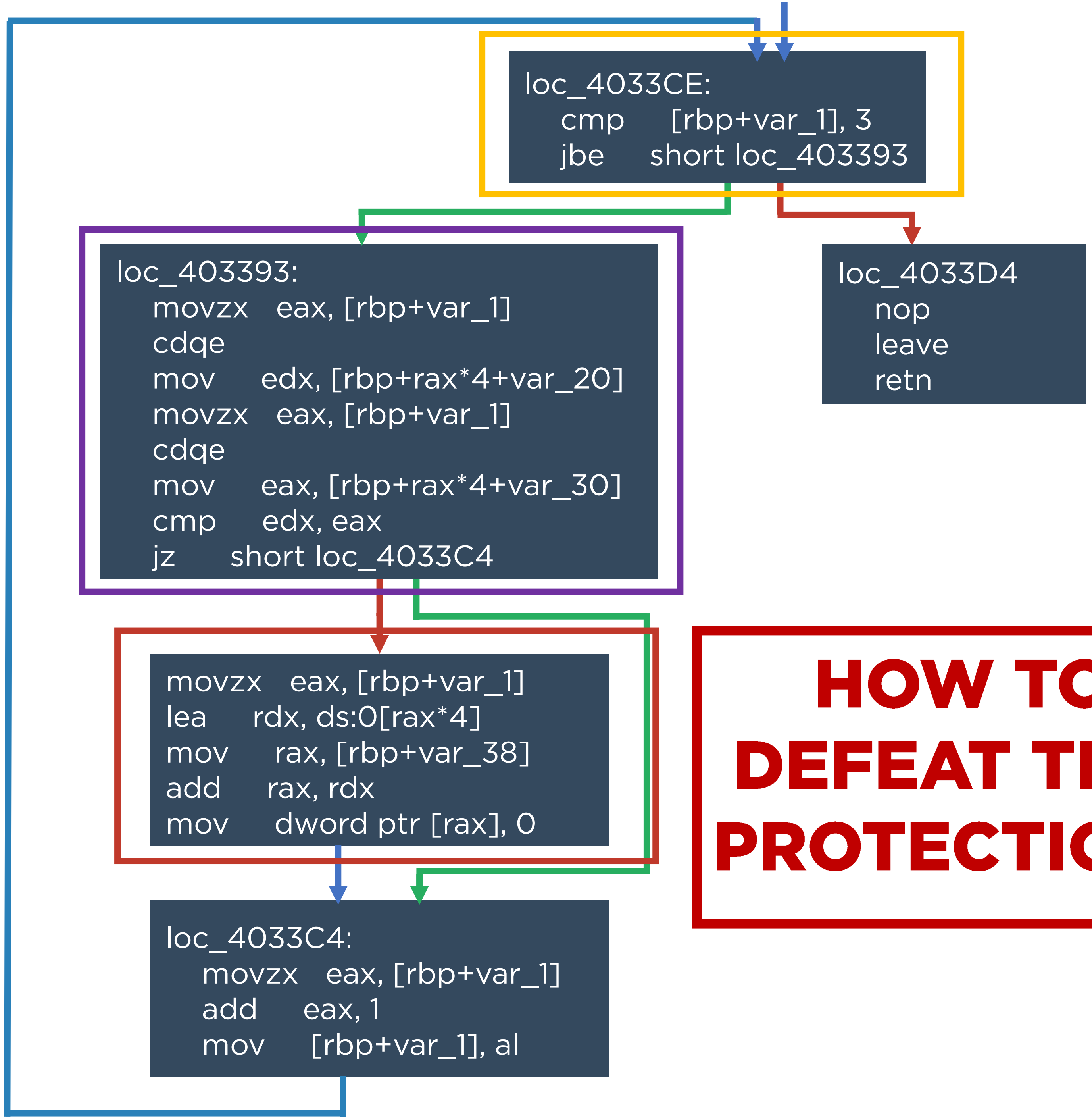
We start the output analysis

The output was computed twice. Its consistency is checked.

In case of a failure, the four-byte block is set to zero.

These operation are done 4 times to analyze all the output.

eShard



HOW TO DEFEAT THIS PROTECTION?

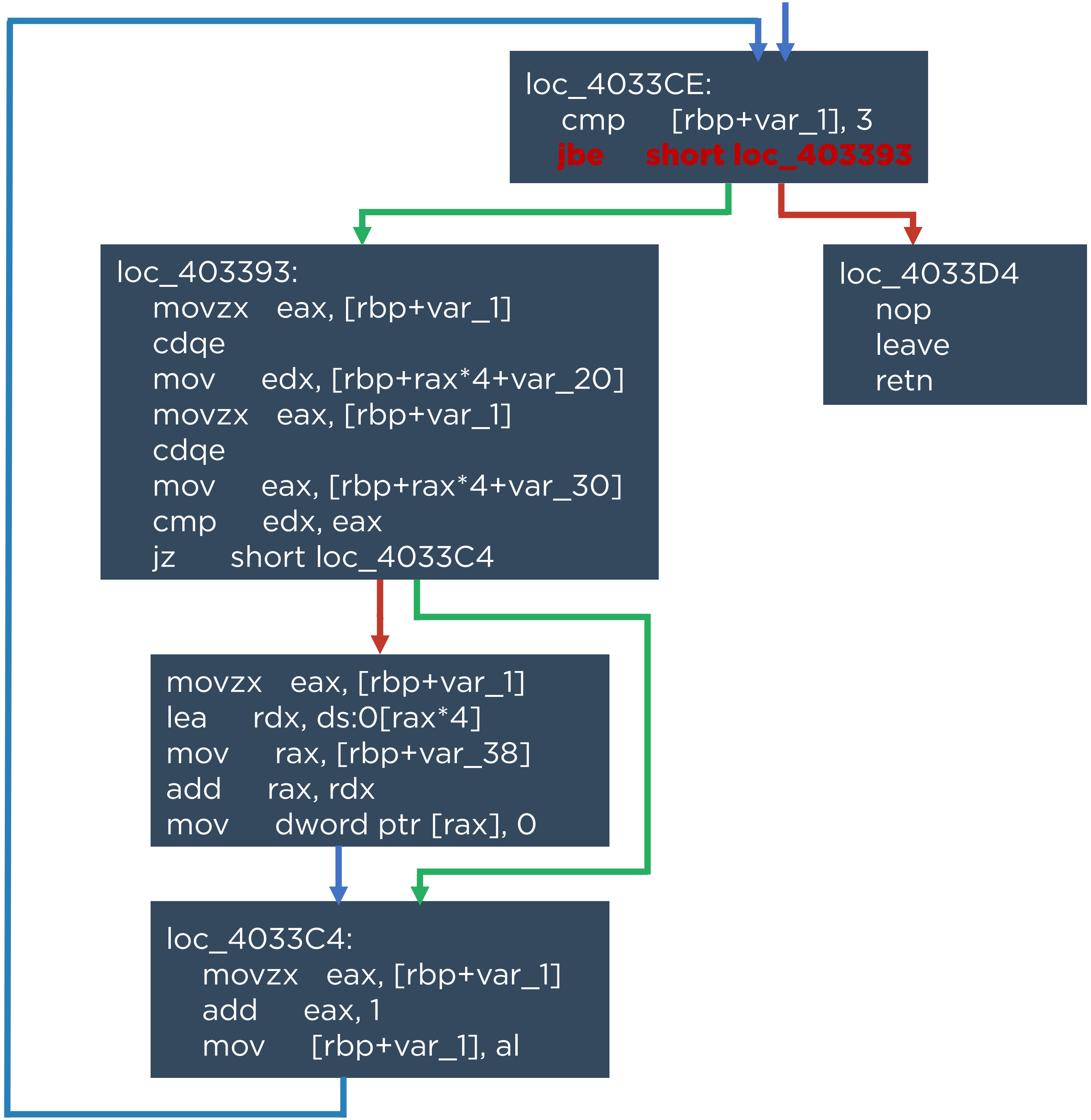
We start the output analysis

The output was computed twice. Its consistency is checked.

In case of a failure, the four-byte block is set to zero.

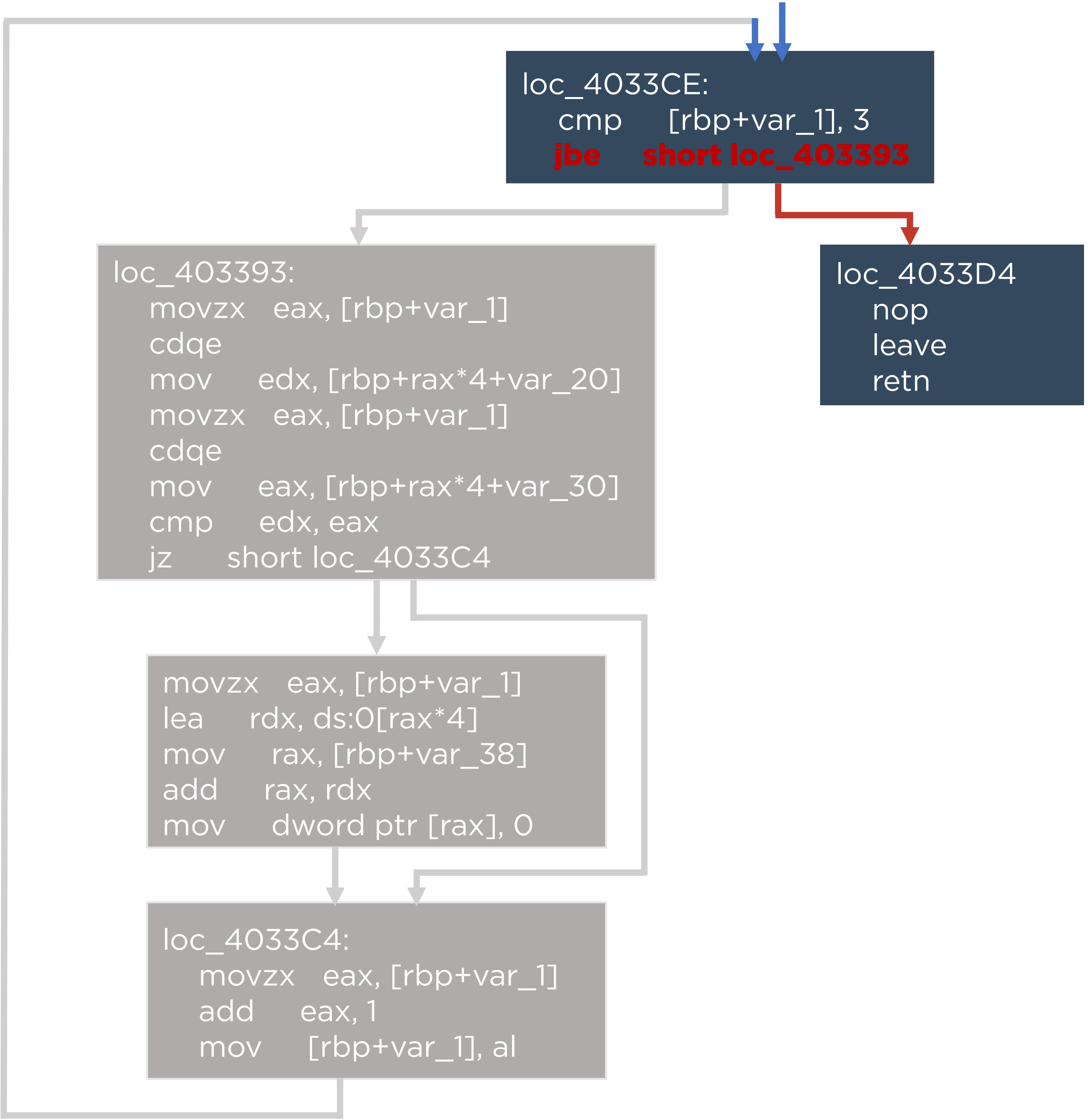
These operation are done 4 times to analyze all the output.

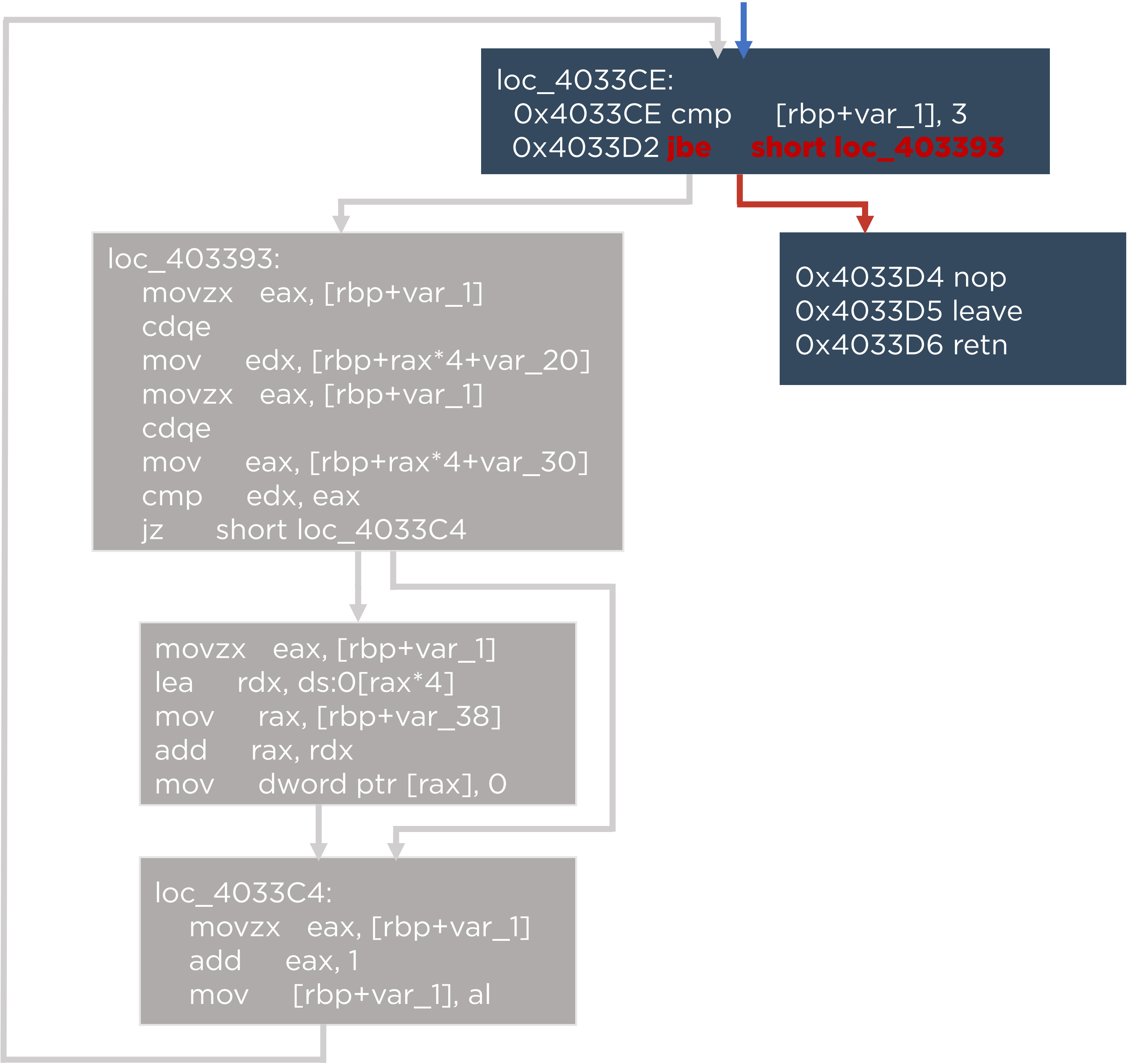
eShard



We can disturb the control flow:

- **We jump from jbe instruction directly to 0x4033D4**
- **We exit the function**
- **No check is done**
- **No byte is set to zero**


loc_4033CE:
cmp [rbp+var_1], 3
jbe short loc_403393
loc_403393:
movzx eax, [rbp+var_1]
cdqe
mov edx, [rbp+rax*4+var_20]
movzx eax, [rbp+var_1]
cdqe
mov eax, [rbp+rax*4+var_30]
cmp edx, eax
jz short loc_4033C4
loc_4033D4
nop
leave
retn
movzx eax, [rbp+var_1]
lea rdx, ds:0[rax*4]
mov rax, [rbp+var_38]
add rax, rdx
mov dword ptr [rax], 0
loc_4033C4:
movzx eax, [rbp+var_1]
add eax, 1
mov [rbp+var_1], al



2nd fault paramer:

- **Register: Program Counter**
- **When: instruction located at 0x4033D2 is reached**
- **Fault model: Add user value + 2 to jump directly to 0x4033D4**

File Edit View Run Kernel Tabs Settings Help

faulter_presentation.ipynb

Markdown Python 3

2nd FAULT INJECTION

We are going to inject two faults:

- the first fault aims to set the output to zero
- the second fault targets the consistency verification.

We remove the fault&trace option.

```
faulter.tracer = None
```

We will attack only the rax register, and target only the PC that enabled to have an output set to zero.

```
pc_to_attack = []
for hit in parser.get_fault_dico()["OUTPUT"]["00000000000000000000000000000000"]["faults"]:
    pc_to_attack.append(int(hit[0]["pc"], 16))
```

We configure the first fault.

```
fault_1.register = "rax"
fault_1.filt_pre.in_pc.pts = pc_to_attack[0:32]
fault_1.trig_and_acts = [trig_and_act_start_fault_injection,
                         trig_and_act_stop_fault_injection]
```

We change the directory.

```
faulter.directory_out = "faulter_outputs_two_fault"
```

We configure the second fault:

- we want to modify the PC register
- when its value is 0x4033D2, we add 2 to it to skip an instruction and jump directly to 0x4033D4

```
fault_2 = FaultCfgMono(bin_hand)
```

```
fault_2.filt_pre.in_pc.pts = [0x4033D2]
fault_2.register = "pc"
fault_2.param = 2
fault_2.model = const.FM_ADD_USER_VAL
```

eShard

File Edit View Run Kernel Tabs Settings Help

Code Python 3

We set the two faults

[50]:
```python
faulter.faults_opt = [fault_1, fault_2]
```

[51]:
```python
faulter.generate_faults("061FC9F588B2F9D2001986822C121179")
```

```
Injecting 6 faults (estimation). Please wait...
```

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

File Edit View Run Kernel Tabs Settings Help

Launcher | faulter_presentation.ipynb

Markdown Python 3

We analyse the new log.

[77]:

```python
logfile = glob.glob("./%s/*.bin" % faulter.directory_out)[0]
parser = FaulterLogParser(logfile, output_parser = bin_hand.process_output)
parser.parse_file()
# Print faulty outputs with 4 modified bytes
parser.get_faulty_output([4])
```

[77]:

```
[['14ed01ea7ce2a551c9791ae85c7cecf4', '6AED01EA7CE2A570C979D3E85CD2ECF4'],
 ['14ed01ea7ce2a551c9791ae85c7cecf4', '4EED01EA7CE2A586C979FAE85C90ECF4'],
 ['14ed01ea7ce2a551c9791ae85c7cecf4', '27ED01EA7CE2A5AEC97950E85C40ECF4'],
 ['14ed01ea7ce2a551c9791ae85c7cecf4', '142301EA11E2A551C9791AC25C7C67F4'],
 ['14ed01ea7ce2a551c9791ae85c7cecf4', '142B01EA7FE2A551C9791A575C7CD9F4'],
 ['14ed01ea7ce2a551c9791ae85c7cecf4', '147101EAC2E2A551C9791AAA5C7C90F4'],
 ['14ed01ea7ce2a551c9791ae85c7cecf4', '145E01EA6CE2A551C9791A5C5C7C53F4'],
 ['14ed01ea7ce2a551c9791ae85c7cecf4', '14ED09EA7C97A551CE791AE85C7CEC98'],
 ['14ed01ea7ce2a551c9791ae85c7cecf4', '14ED2CEA7CA2A55163791AE85C7CECAC'],
 ['14ed01ea7ce2a551c9791ae85c7cecf4', '14ED4AEA7C78A5514E791AE85C7CEC30'],
 ['14ed01ea7ce2a551c9791ae85c7cecf4', '14ED2BEA7C35A55146791AE85C7CEC59'],
 ['14ed01ea7ce2a551c9791ae85c7cecf4', '14ED01057CE2D551C9F61AE8F77CECF4'],
 ['14ed01ea7ce2a551c9791ae85c7cecf4', '14ED01E17CE2EF51C91B1AE80C7CECF4'],
 ['14ed01ea7ce2a551c9791ae85c7cecf4', '14ED01E47CE22451C99B1AE8447CECF4'],
 ['14ed01ea7ce2a551c9791ae85c7cecf4', '14ED01E87CE24B51C9E51AE8E57CECF4'],
 ['14ed01ea7ce2a551c9791ae85c7cecf4', '14ED01847CE22551C94D1AE80C7CECF4']]
```

This time, we have faulty outputs with 4 modified bytes.

eShard

File Edit View Run Kernel Tabs Settings Help

Markdown Python 3

Key recovery

[53]:
```python
from estoolkit2.fault_attack.aes import DfaAesPiret

# We perform the DFA of Piret
# We need output with 4 faulty bytes
lst = parser.get_faulty_output([4])
dfa = DfaAesPiret(lst)
dfa.run()

found_key = dfa.get_found_key()
dfa.summary()
```

DFA summary:

Words	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Found bytes	0x97	0x78	0x13	0xD5	0x5A	0x7B	0x8A	0xE1	0x55	0x8C	0x31	0x3C	0xA0	0xBA	0x7E	0x6D

We found the round key 0x97 0x78 0x13 0xD5 0x5A 0x7B 0x8A 0xE1 0x55 0x8C 0x31 0x3C 0xA0 0xBA 0x7E 0x6D. However, we still need to recover the master key, and then check it.

To recover the master key, we need:

- the extracted round key,
- its round number,

[54]:
```python
from essva.tools.tools import inv_key_schedule_aes
master_key = inv_key_schedule_aes(round_key=found_key, round_nb=10, verbose = True)
```

Computed AES Master Key:

Words	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Round key 10	0x97	0x78	0x13	0xD5	0x5A	0x7B	0x8A	0xE1	0x55	0x8C	0x31	0x3C	0xA0	0xBA	0x7E	0x6D
Computed Master Key	0x 1	0x12	0x23	0x34	0x45	0x56	0x67	0x78	0x89	0x9A	0xAB	0xBC	0xCD	0xDE	0xEF	0xFF

File Edit View Run Kernel Tabs Settings Help

Markdown Python 3

We found a key! Let's test it.

[55]:
```
message = "096F90870768821A84BD665C2F7DDD4A"
res = bin_hand.check_key(key=master_key, message=message)
```

Key matching: SUCCESS

Words	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Key bytes	0x01	0x12	0x23	0x34	0x45	0x56	0x67	0x78	0x89	0x9A	0xAB	0xBC	0xCD	0xDE	0xEF	0xFF

Words	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Plaintext	0x09	0x6F	0x90	0x87	0x07	0x68	0x82	0x1A	0x84	0xBD	0x66	0x5C	0x2F	0x7D	0xDD	0x4A
Binary ciphertext	0x39	0x5A	0xF9	0x24	0xAC	0xB5	0x2E	0xB5	0xA2	0xDC	0x8F	0xAF	0xFC	0x41	0xDC	0x58
Expected ciphertext	0x39	0x5A	0xF9	0x24	0xAC	0xB5	0x2E	0xB5	0xA2	0xDC	0x8F	0xAF	0xFC	0x41	0xDC	0x58

It matches! The AES key is 0x01 0x12 0x23 0x34 0x45 0x56 0x67 0x78 0x89 0x9A 0xAB 0xBC 0xCD 0xDE 0xEF 0xFF.

File Edit View Run Kernel Tabs Settings Help

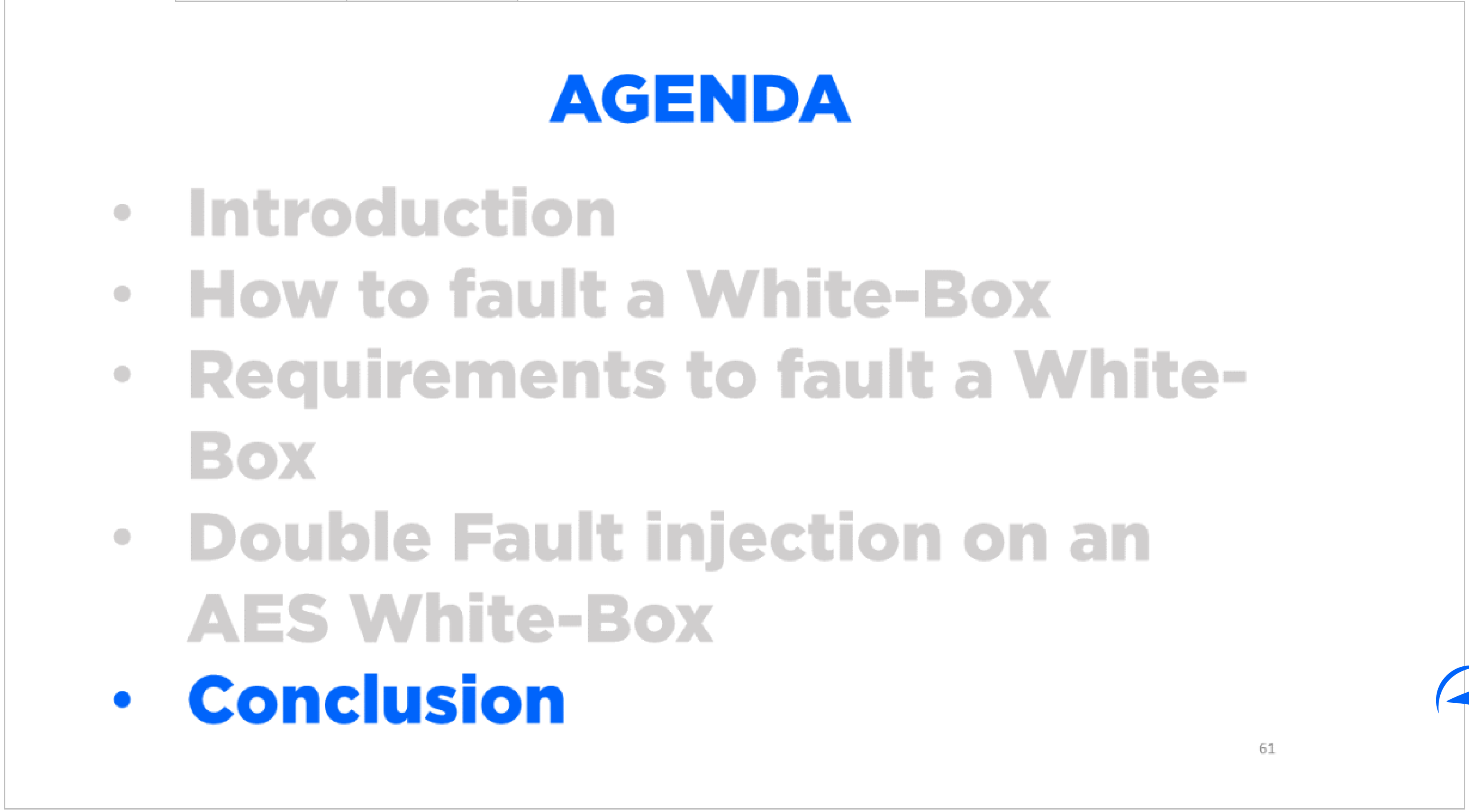
Code Python 3

Conclusion

[56]:
```
slideshow('Conclusion',
          './img/presentation_faulter',
          ['Diapositive%d.PNG' % hit for hit in [61,62,63 ]],
          slide_width)
```

Conclusion

Slide 0 | Slide 1 | Slide 2



eShard

CONCLUSION

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

- Dynamic fault injection is a prerequisite
- Accurate multiple faults can be injected
- Security mechanisms can be defeated

But

Combinatorial complexity ***nb_ins x nb_fault_model x nb_target x nb_input x nb_area***

CONCLUSION

Strategies to defeat these issues:

- Pattern detector to fault only interesting area
- Focus faulting on specific Register / Program Counter or instructions
- Fault & trace to understand the effect of a fault or downgrade security

In that way, it is possible to execute successful multi-fault attacks, in reasonable amount of time.