SONY

GHIDRA FAULT EMULATION

BRUSSELS LAB

Roman Korkikian, 15th of November 2022

About me and the team

- Independent contractor mostly working for Sony
- Specializing in side-channel analysis, fault attacks, crypto…
- >10 year of industrial experience, PhD

- Sony security team has +10 people to perform internal products analysis
- We work with devices and services (applications)
- The team is spread over the globe

Agenda

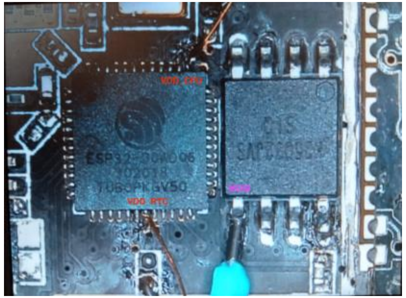
- Introduction and reasoning
- ARCv2 Ghidra support
- AES-128 ARCv2 Emulation
- AES-128 Fault Injection
- Conclusions

Introduction and reasoning

Start with Faults

- Faults are physical stresses (EM pulse or power glitch) that skip instruction, modify data, or cause another effect that can be used by an attacker

- ESP32 was intensively attacked by different people
  - Fault to bypass secure boot
  - Fault to bypass encryption process
  - Fault to read an encryption key from the OTP
- NRF52
- STM32
- and many other devices were reported to be vulnerable to faults



Project Objectives

- Often, we need to execute parts of the code when a device is not accessible (still in production or for other reasons)
- We needed a tool:
  - To emulate **fault injection**
  - To perform **code fuzzing**
  - To perform other analyses (RE, API calls, countermeasures check, etc.)
- Tool requirements:
  - Adding a new instruction set must be feasible (ARCv2)
  - A tool shall be used by people of different expertise

Tools Selection

	Ghidra	Radare2	QEMU
Support emulation	+	+	+
Complexity to add new CPU	+	+/-	+/-
User-friendliness	+	+/-	+/-
Performance	+/-	+/-	+
Other functionality (RE, disassembly,…)	+	+	+/-

The selection was made by implementing some instructions for ARCv2

Ghidra

Ghidra is a reverse-engineering tool developed by NSA and released to public (and open source) usage:

- Ghidra has similar functionality as IDA Pro
- Open source with various features
- Integrates disassembly, decompiler and emulation facilities

Ghidra competes with **IDA Pro**, **radare2** and other reverse-engineering tools.

Adding ARCv2

ARCv2 Where to Find

- In my practice, I encountered twice with ARCv2 devices that were difficult to analyse due to a lack of tools:
  - This year IDA released ARCv2 decompilation (available as a separate module)

- ARCv2 CPUs are mainly used in special-purpose devices
  - STAR1000P NVMe solid state drive (SSD) controller
  - Arbe Phoenix High-Resolution Imaging Radar chipset

- … but also can be found in IoT/general microcontrollers:
  - EM9304 - for Bluetooth 5.0 low energy enabled products
  - PLS10 ultra-low power integrated general-purpose MCU

Current State-of-the Art

- Nicolas IOOSS implemented ARCompact support for Ghidra

- Unfortunately, ARCv2 is extremely different from ARCompact

ARCompact is different from ARCv2

ARCompact

- Different instructions

ABSS

Absolute with Saturation

ADDSDW

Signed Add with Saturation Dual Word

- Different Auxiliary registers

ARCv2

- Different instructions

AEX

Function

Swap contents of an auxiliary register with a core register.

ENTER_S

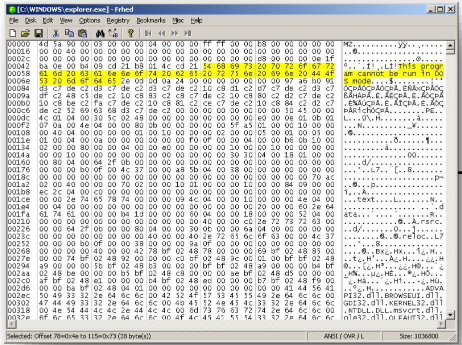
Function

Function Prolog Sequence

- Different Auxiliary registers

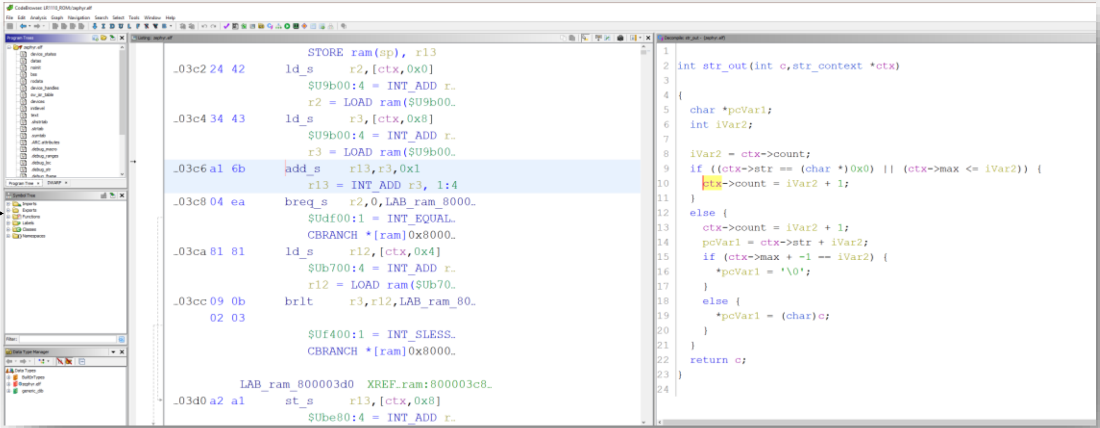
0x38	Saved Normal Kernel Stack Pointer, AUX_KERNEL_SP	normal kernel stack swap register
0x39	Saved Secure User Stack Pointer, AUX_SEC_U_SP	Secure user stack swap register
0x3A	Saved Secure Kernel Stack Pointer, AUX_SEC_K_SP	Secure kernel stack swap register
0x3B	Saved Shadow Normal Stack Pointer, AUX_NSEC_SP	Saved shadow stack pointer register

How Ghidra Works



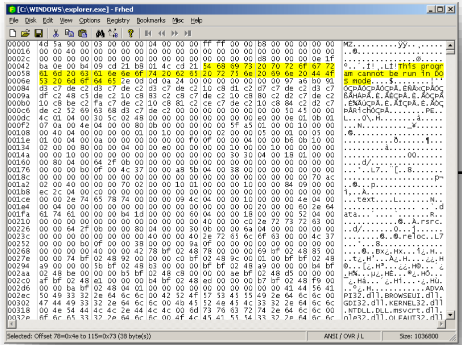
Raw binary file: .elf, .exe, …

Load a file in Ghidra



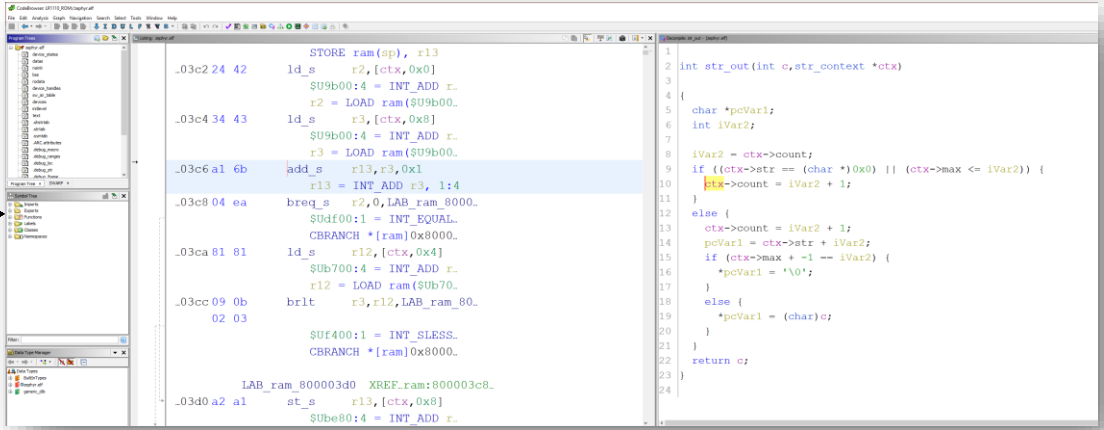
Ghidra interprets the bytes into instructions
Instructions into a decompiled C-code

How Ghidra Works



Raw binary file: .elf, .exe, …

Load a file
in Ghidra



Ghidra interprets the bytes into instructions
Instructions into a decompiled C-code

SLEIGH is used to describe instructions

```
define token instr16 (16)
  F16_MAJOR_OPCODE = (11, 15)
  f16_b            = (8, 10)
  f16_c            = (5,7)
  f16_i            = (3,4)
  f16_u            = (0,2);

attach variables [f16_b f16_c] [r0 r1 r2 r3 r12 r13 r14 r15];

with : F16_MAJOR_OPCODE = 0b01101 {
  :add_s f16_c, f16_b, f16_u is f16_i = 0b00 & f16_b & f16_c
& f16_u{
    f16_c = f16_b + f16_u;
  }
}
```

Ghidra ARCv2 Example

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
I[4:0]					b[2:0]			c[2:0]			i[1:0]		u[2:0]		
0	1	1	0	1	b	b	b	c	c	c	i	i	u	u	u

← The way instructions are encoded

Syntax:

ADD_S c, b, u3

SUB_S c, b, u3

ASL_S c, b, u3

ASR_S c, b, u3

← Possible instructions to encode

ARCv2 Documentation

Table 9-3 16-Bit, ADD/SUB Register-Immediate

Sub-opcode i field (2 bits)	Instruction	Operation	Description
0x00	ADD_S	c ← b + u3	Add
0x01	SUB_S	c ← b - u3	Subtract
0x02	ASL_S	c ← b asl u3	Multiple arithmetic shift left
0x03	ASR_S	c ← b asr u3	Multiple arithmetic shift right

← Sub-opcode that define instructions

Ghidra ARCv2 Example

```
define token instr16 (16)
  F16_MAJOR_OPCODE = (11, 15)
  f16_b            = (8, 10)
  f16_c            = (5,7)
  f16_i            = (3,4)
  f16_u            = (0,2);
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
I[4:0]					b[2:0]			c[2:0]			i[1:0]		u[2:0]		
0	1	1	0	1	b	b	b	c	c	c	i	i	u	u	u

Ghidra ARCv2 Example

```
define token instr16 (16)
  F16_MAJOR_OPCODE = (11, 15)
  f16_b            = (8, 10)
  f16_c            = (5,7)
  f16_i            = (3,4)
  f16_u            = (0,2);

attach variables [f16_b f16_c] [r0 r1 r2 r3 r12 r13 r14 r15];
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
I[4:0]					b[2:0]			c[2:0]			i[1:0]		u[2:0]		
0	1	1	0	1	b	b	b	c	c	c	i	i	u	u	u

Ghidra ARCv2 Example

```
define token instr16 (16)
  F16_MAJOR_OPCODE = (11, 15)
  f16_b            = (8, 10)
  f16_c            = (5,7)
  f16_i            = (3,4)
  f16_u            = (0,2);

attach variables [f16_b f16_c] [r0 r1 r2 r3 r12 r13 r14 r15];

with : F16_MAJOR_OPCODE = 0b01101 {
  :add_s f16_c, f16_b, f16_u is f16_i = 0b00 & f16_b & f16_c & f16_u{
    f16_c = f16_b + f16_u;
  }

  :asl_s f16_c, f16_b, f16_u is f16_i = 0b10 & f16_b & f16_c & f16_u{
    f16_c = f16_b << f16_u;
  }
}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
I[4:0]					b[2:0]			c[2:0]			i[1:0]		u[2:0]		
0	1	1	0	1	b	b	b	c	c	c	i	i	u	u	u

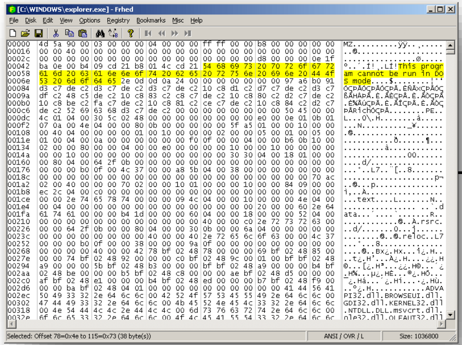
Ghidra ARCv2 Example

```
define token instr16 (16)
  F16_MAJOR_OPCODE = (11, 15)
  f16_b            = (8, 10)
  f16_c            = (5,7)
  f16_i            = (3,4)
  f16_u            = (0,2);

attach variables [f16_b f16_c] [r0 r1 r2 r3 r12 r13 r14 r15];

with : F16_MAJOR_OPCODE = 0b01101 {
  :add_s f16_c, f16_b, f16_u is f16_i = 0b00 & f16_b & f16_c & f16_u{…}
  :asl_s f16_c, f16_b, f16_u is f16_i = 0b10 & f16_b & f16_c & f16_u{…}
}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
I[4:0]					b[2:0]			c[2:0]			i[1:0]		u[2:0]		
0	1	1	0	1	b	b	b	c	c	c	i	i	u	u	u

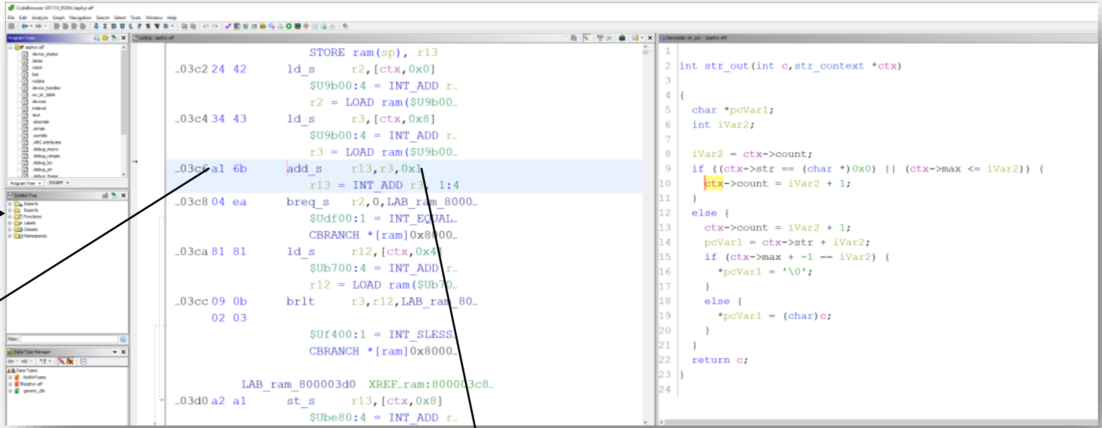
```
0x6BA1 = 0b0110101110100001 shall disassemble in add_s r13, r3, #0x1
F16_MAJOR_OPCODE = (11, 15)   -> 01101  Major opcode
f16_b            = (8, 10)    -> 011    Register #3 – r3
f16_c            = (5,7)      -> 101    Register #5 – r13
f16_i            = (3,4)      -> 00     Major opcode + this value gives add_s
f16_u            = (0,2)      -> 001    Immediate value 1
```

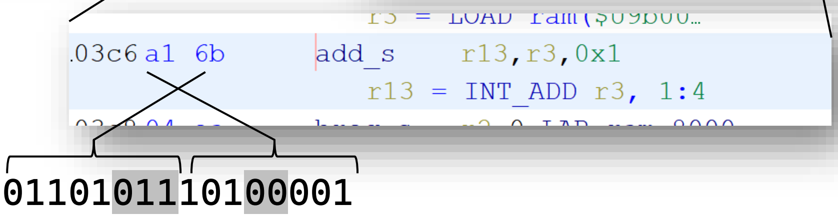
How Ghidra Works



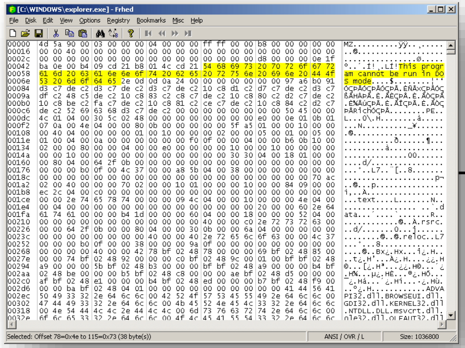
Raw binary file: .elf, .exe, …

Load a file in Ghidra



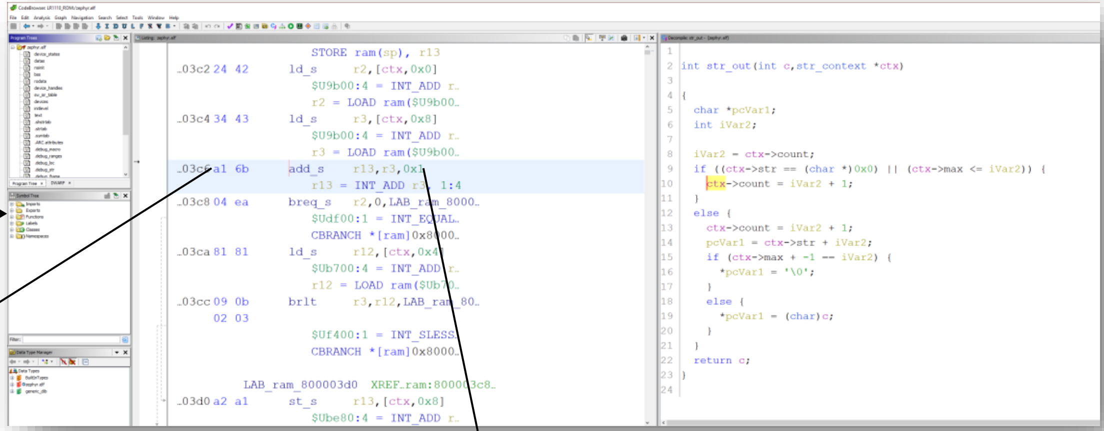


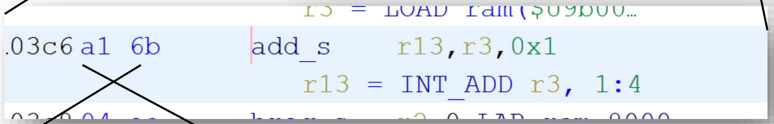
How Ghidra Works



Raw binary file: .elf, .exe, …

Load a file in Ghidra

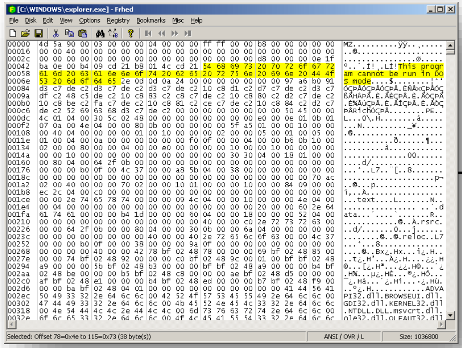





```
0110101110100001

ADD_S   c,b,u3     01101bbbccc00uuu
```

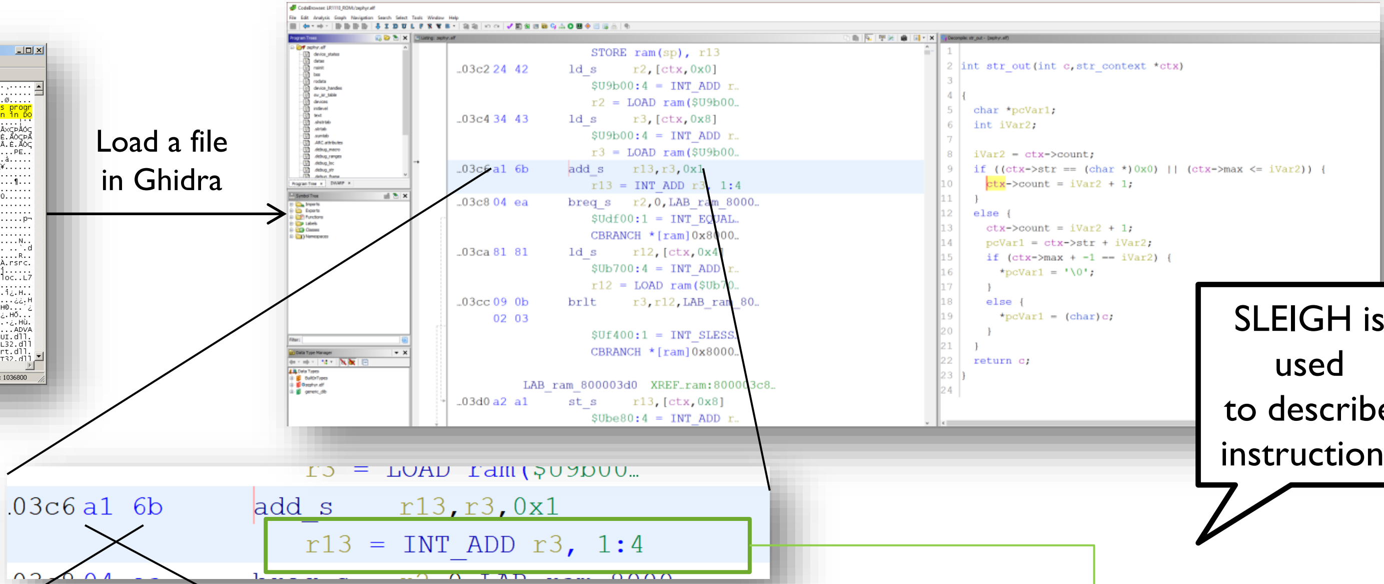
```
:add_s f16_c,f16_b,f16_u is …{
    f16_c = f16_b + f16_u;
}
```

How Ghidra Works



Raw binary file: .elf, .exe, …

Load a file in Ghidra



SLEIGH is used to describe instructions

```
0110101110100001

ADD_S   c,b,u3     01101bbbccc00uuu
```

```
:add_s f16_c,f16_b,f16_u is …{
      f16_c = f16_b + f16_u;
}
```

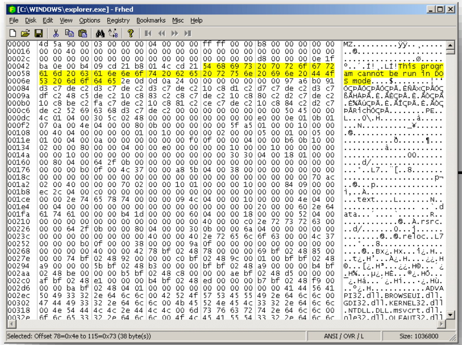

SONY

Ghidra ARCv2 Example

- Ghidra ARCv2 ISA support includes more than 5500 lines of code:
  - 380 Sleigh-described instructions (to improve emulation speed, one instruction can have more than one description)
- The ARCv2 support can be found here:
  `https://github.com/korkikian/ARCv2`
- Please, keep in mind that this is a work in progress, perhaps some instructions or corner cases are not correctly supported:
  - F32_EXT5 class is not supported (DSP mostly)
  - F32_APEX class is only disassembled (those instructions can be customized)

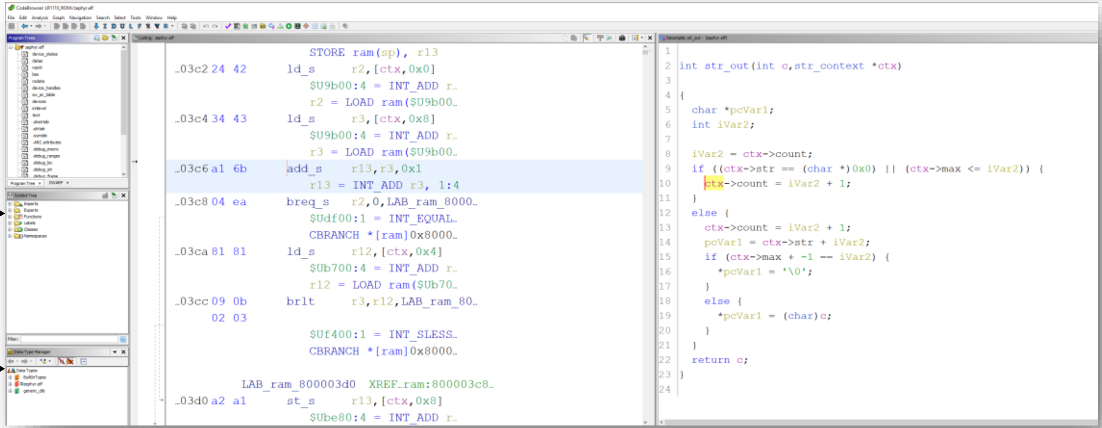
Emulating ARCv2 Binary

How Ghidra Emulation Works

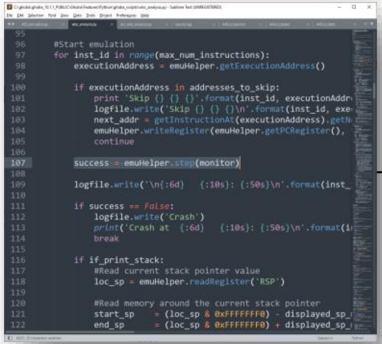


Raw binary file: .elf, .exe, …

Load a file
in Ghidra

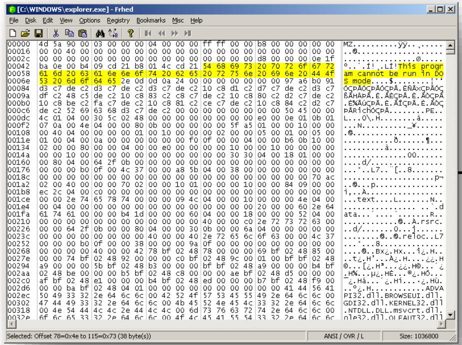


Use Ghidra API



A script file that steps through the
instructions to emulate the code

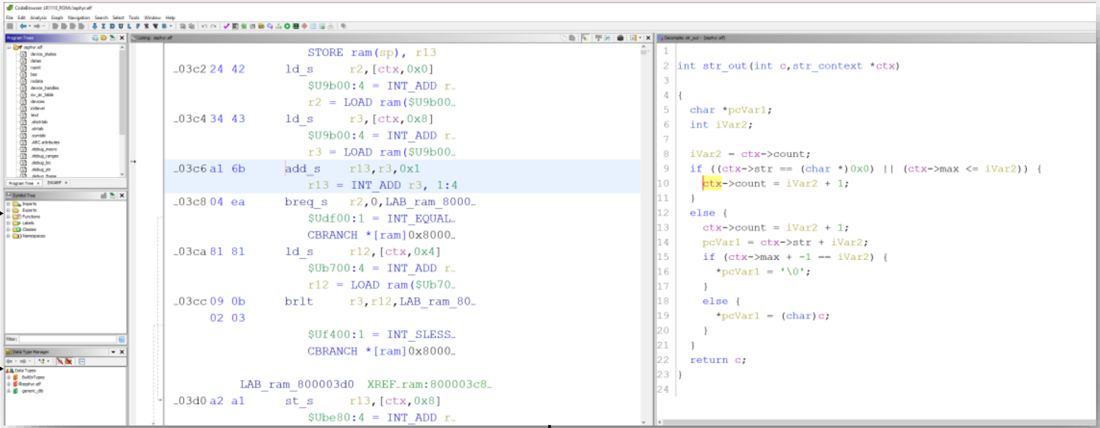
How Ghidra Emulation Works



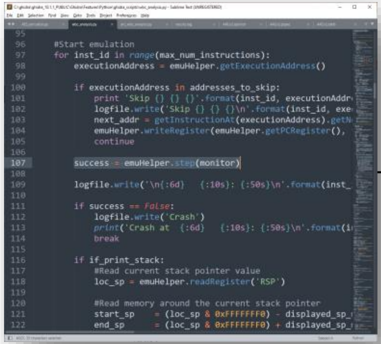
Raw binary file: .elf, .exe, …

Load a file in Ghidra

Use Ghidra API



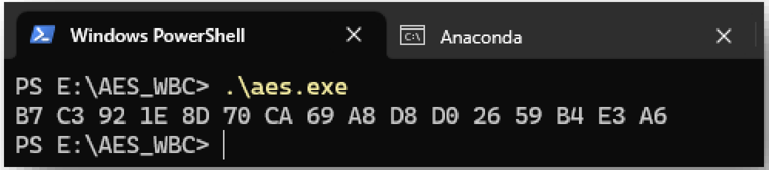

```
ram:000003c6: add_s r13,r3,0x1

STACK
00806B20:   00000000   00000000   c0010000   d8010000   |  ................
00806B30:   20580000  |30000000|  0c010000   babaedfe   |  .X..0...........
GP
00804000:   02000001   03000000   00000000   00000000   |  ................

CPU context
r0  = 00000033 r1  = 00005850                        r2  = 00000000 r3  = 00804030
r4  = 00000000 r5  = 00000000                        r6  = 00000000 r7  = 00000000
r8  = 00000000 r9  = 00000000                        r10 = 00000000 r11 = 00000000
r12 = 00000000 r13 = 00804031 (new) - 00804000 (old) r14 = 00005820 r15 = 00005818
r16 = 00000030 sp  = 00806B34                        gp  = 00804000 fp  = 00806B38
```



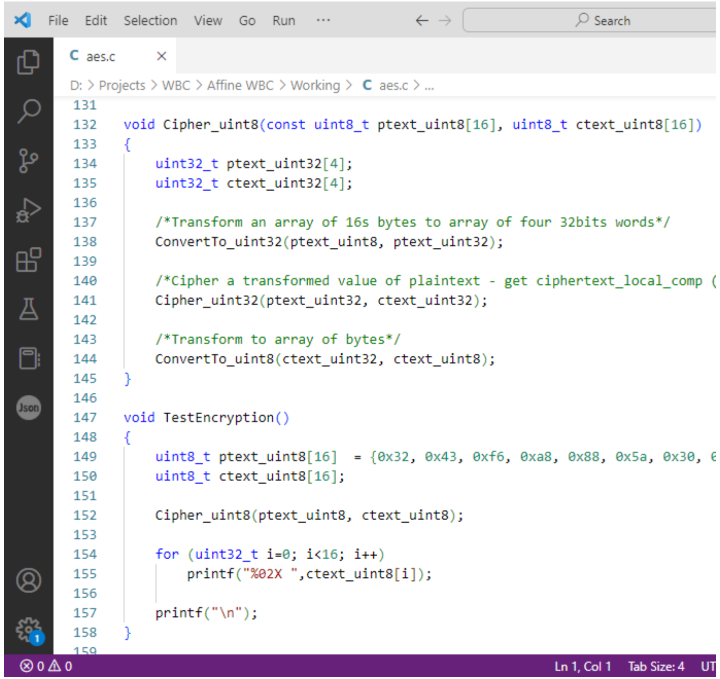
A script file that steps through the instructions to emulate the code

Emulating ARCv2 Binary

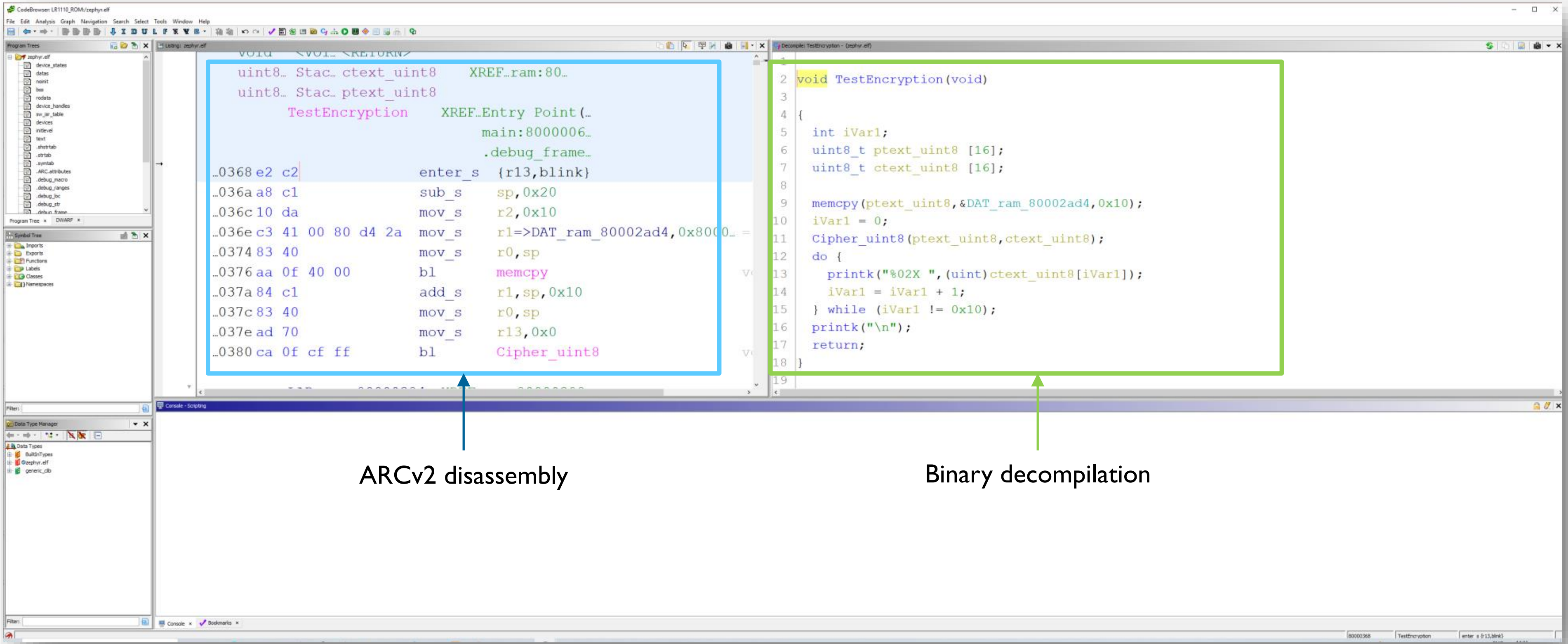
- AES-128 cryptographic algorithm with an embedded key (white-box protection) and a constant plaintext was used for this presentation
- The AES-128 code can be compiled for any CPU (x64 example below)



- The same code was compiled for ARCv2



AES-128 ARCv2 Disassembly



SONY

AES-128 ARCv2 Disassembly

```c
void TestEncryption()
{
    uint8_t ptext_uint8[16]  = {0x32, 0x43, 0xf6, 0xa8, 0x88, 0x5a, 0x30, 0
    uint8_t ctext_uint8[16];

    Cipher_uint8(ptext_uint8, ctext_uint8);

    for (uint32_t i=0; i<16; i++)
        printf("%02X ",ctext_uint8[i]);

    printf("\n");
}
```

⊗ 0 ⚠ 0   Ln 1, Col 1   Tab Size: 4   UT

```c
void TestEncryption(void)

{
  int iVar1;
  uint8_t ptext_uint8 [16];
  uint8_t ctext_uint8 [16];

  memcpy(ptext_uint8,&DAT_ram_80002ad4,0x10);
  iVar1 = 0;
  Cipher_uint8(ptext_uint8,ctext_uint8);
  do {
    printk("%02X ",(uint)ctext_uint8[iVar1]);
    iVar1 = iVar1 + 1;
  } while (iVar1 != 0x10);
  printk("\n");
  return;
}
```

- Ghidra decompilation looks alike with the initial C code
- Decompilation is achieved when the instructions are correctly described with Sleigh
- Complex instructions are more difficult to decompile, so optimise instructions as much as possible

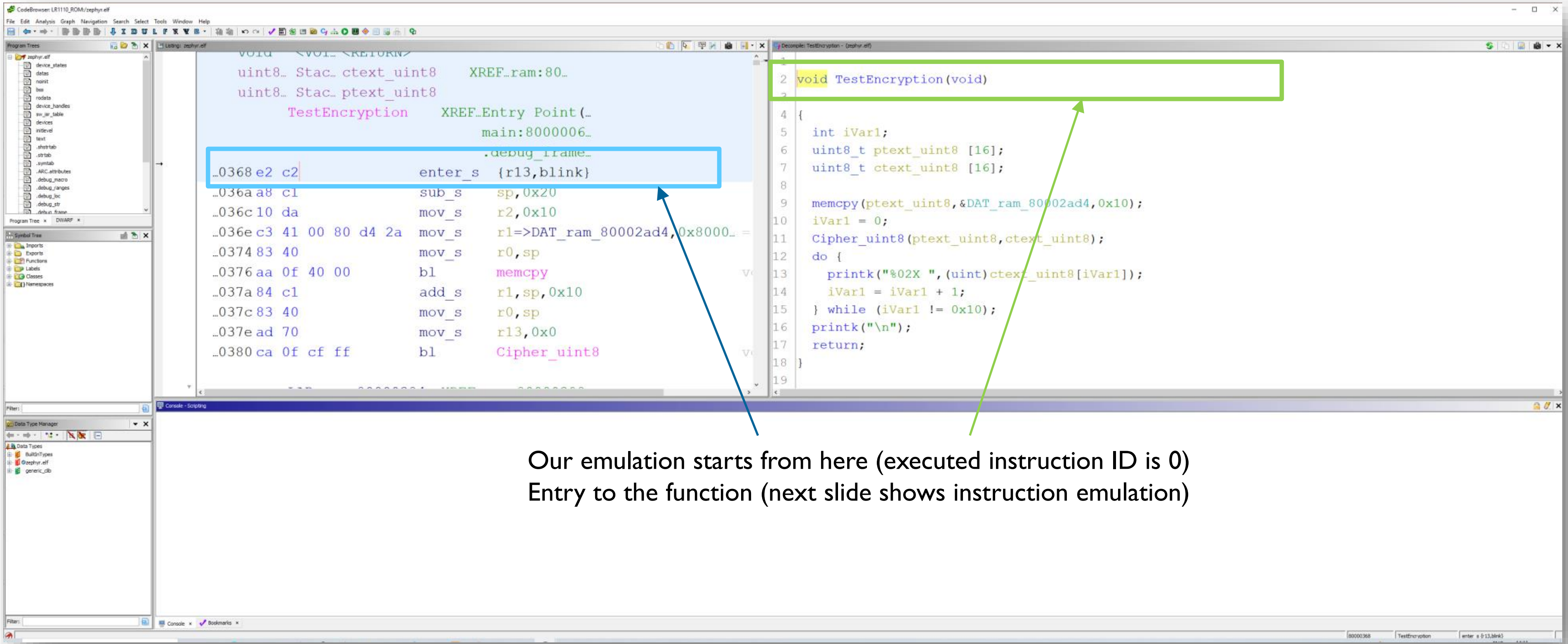
AES-128 ARCv2 Emulation

- A python script that sets the initial CPU state:
  - Set stack pointer, program counter and registers
  - Initialize memory if needed
  - Define success conditions
- Then step by step execute disassembled instructions

```
for inst_id in range(max_num_instructions):
    executionAddress = emuHelper.getExecutionAddress()
    success = emuHelper.step(monitor)
    # Read CPU registers, memory, and perform other operations
```

- A script controls CPU registers and memory content at any emulation step

AES-128 ARCv2 Emulation



Our emulation starts from here (executed instruction ID is 0)
Entry to the function (next slide shows instruction emulation)

AES-128 ARCv2 Emulation

Instruction at the current address

```
0   ram:80000368: enter_s {r13,blink}
Stack
1FFFFFA0:   00000000   00000000   00000000   00000000    |  ................
1FFFFFB0:   00000000   00000000   00000000   00000000    |  ................
1FFFFFC0:   00000000   00000000   00000000   00000000    |  ................
1FFFFFD0:   00000000   00000000   00000000   00000000    |  ................
1FFFFFE0:   00000000   00000000   00000000   00000000    |  ................
1FFFFFF0:   00000000   00000000  |efbeadde| *babaedfe*  |  ................

r0  = 00000000 r1  = 00000000                          r2  = 00000000 r3  = 00000000
r4  = 00000000 r5  = 00000000                          r6  = 00000000 r7  = 00000000
r8  = 00000000 r9  = 00000000                          r10 = 00000000 r11 = 00000000
r12 = 00000000 r13 = FEEDBABA                          r14 = 00000000 r15 = 00000000
r16 = 00000000 sp  = 1FFFFFF8 (new) - 20000000 (old)
blink = DEADBEEF
C   = 00000000 V   = 00000000 N   = 00000000 Z   = 00000000
...
```

AES-128 ARCv2 Emulation

CPU stack

```
0   ram:80000368: enter_s {r13,blink}
Stack
1FFFFFA0:   00000000   00000000   00000000   00000000    |  ................
1FFFFFB0:   00000000   00000000   00000000   00000000    |  ................
1FFFFFC0:   00000000   00000000   00000000   00000000    |  ................
1FFFFFD0:   00000000   00000000   00000000   00000000    |  ................
1FFFFFE0:   00000000   00000000   00000000   00000000    |  ................
1FFFFFF0:   00000000   00000000  |efbeadde| *babaedfe*   |  ................

r0  = 00000000 r1  = 00000000                      r2  = 00000000 r3  = 00000000
r4  = 00000000 r5  = 00000000                      r6  = 00000000 r7  = 00000000
r8  = 00000000 r9  = 00000000                      r10 = 00000000 r11 = 00000000
r12 = 00000000 r13 = FEEDBABA                      r14 = 00000000 r15 = 00000000
r16 = 00000000 sp  = 1FFFFFF8 (new) - 20000000 (old)
blink = DEADBEEF
C   = 00000000 V   = 00000000 N   = 00000000 Z   = 00000000
...
```

AES-128 ARCv2 Emulation

```
0   ram:80000368: enter_s {r13,blink}
Stack
1FFFFFA0:   00000000   00000000   00000000   00000000     |  ................
1FFFFFB0:   00000000   00000000   00000000   00000000     |  ................
1FFFFFC0:   00000000   00000000   00000000   00000000     |  ................
1FFFFFD0:   00000000   00000000   00000000   00000000     |  ................
1FFFFFE0:   00000000   00000000   00000000   00000000     |  ................
1FFFFFF0:   00000000   00000000  |efbeadde| *babaedfe*    |  ................

r0  = 00000000 r1  = 00000000                          r2  = 00000000 r3  = 00000000
r4  = 00000000 r5  = 00000000                          r6  = 00000000 r7  = 00000000
r8  = 00000000 r9  = 00000000                          r10 = 00000000 r11 = 00000000
r12 = 00000000 r13 = FEEDBABA                          r14 = 00000000 r15 = 00000000
r16 = 00000000 sp  = 1FFFFFF8 (new) - 20000000 (old)
blink = DEADBEEF
C   = 00000000 V   = 00000000 N   = 00000000 Z   = 00000000
...
```

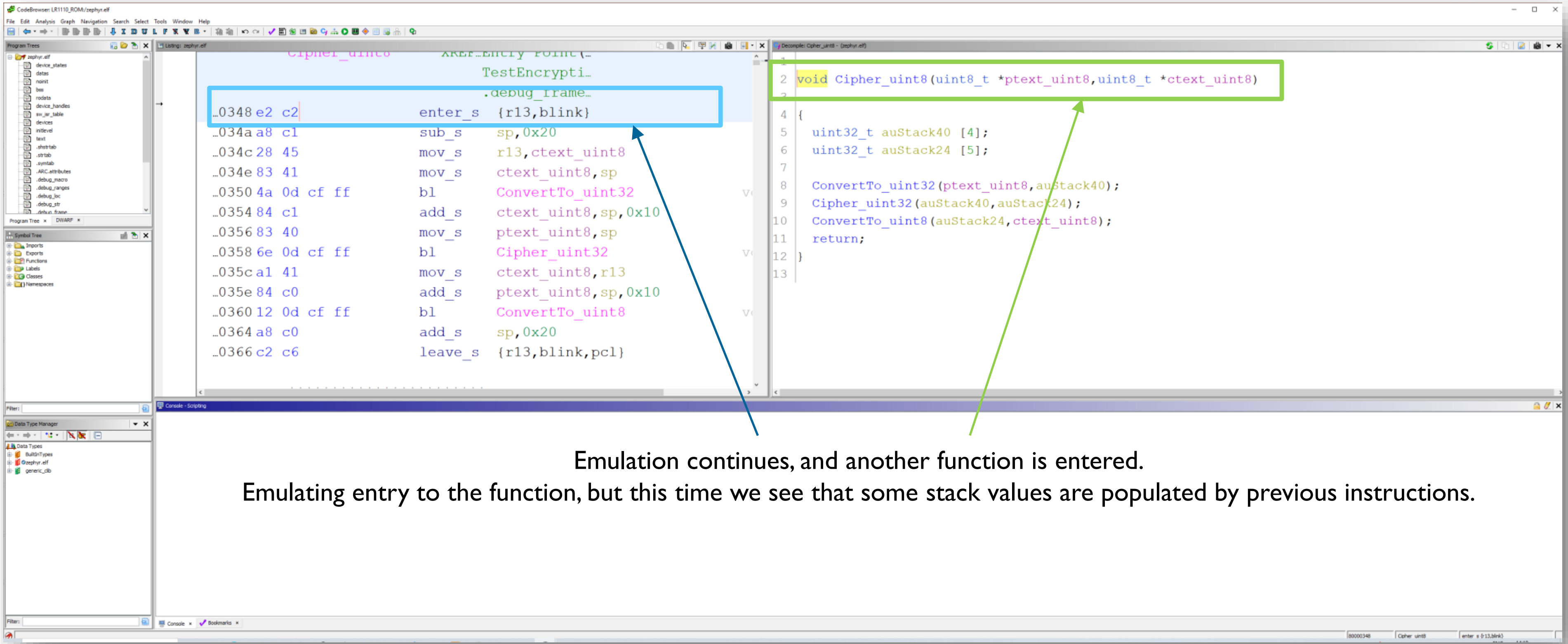
CPU registers

AES-128 ARCv2 Emulation

```
0   ram:80000368: enter_s {r13,blink}
Stack
1FFFFFA0:   00000000   00000000   00000000   00000000    |  ................
1FFFFFB0:   00000000   00000000   00000000   00000000    |  ................
1FFFFFC0:   00000000   00000000   00000000   00000000    |  ................
1FFFFFD0:   00000000   00000000   00000000   00000000    |  ................
1FFFFFE0:   00000000   00000000   00000000   00000000    |  ................
1FFFFFF0:   00000000   00000000  |efbeadde| *babaedfe*   |  ................

r0  = 00000000 r1  = 00000000                          r2  = 00000000 r3  = 00000000
r4  = 00000000 r5  = 00000000                          r6  = 00000000 r7  = 00000000
r8  = 00000000 r9  = 00000000                          r10 = 00000000 r11 = 00000000
r12 = 00000000 r13 = FEEDBABA                          r14 = 00000000 r15 = 00000000
r16 = 00000000 sp  = 1FFFFFF8 (new) - 20000000 (old)
blink = DEADBEEF
C   = 00000000 V   = 00000000 N   = 00000000 Z   = 00000000
...
```

Registers used by the current instruction

AES-128 ARCv2 Emulation



Emulation continues, and another function is entered.
Emulating entry to the function, but this time we see that some stack values are populated by previous instructions.

SONY

AES-128 ARCv2 Emulation

Plaintext to encrypt

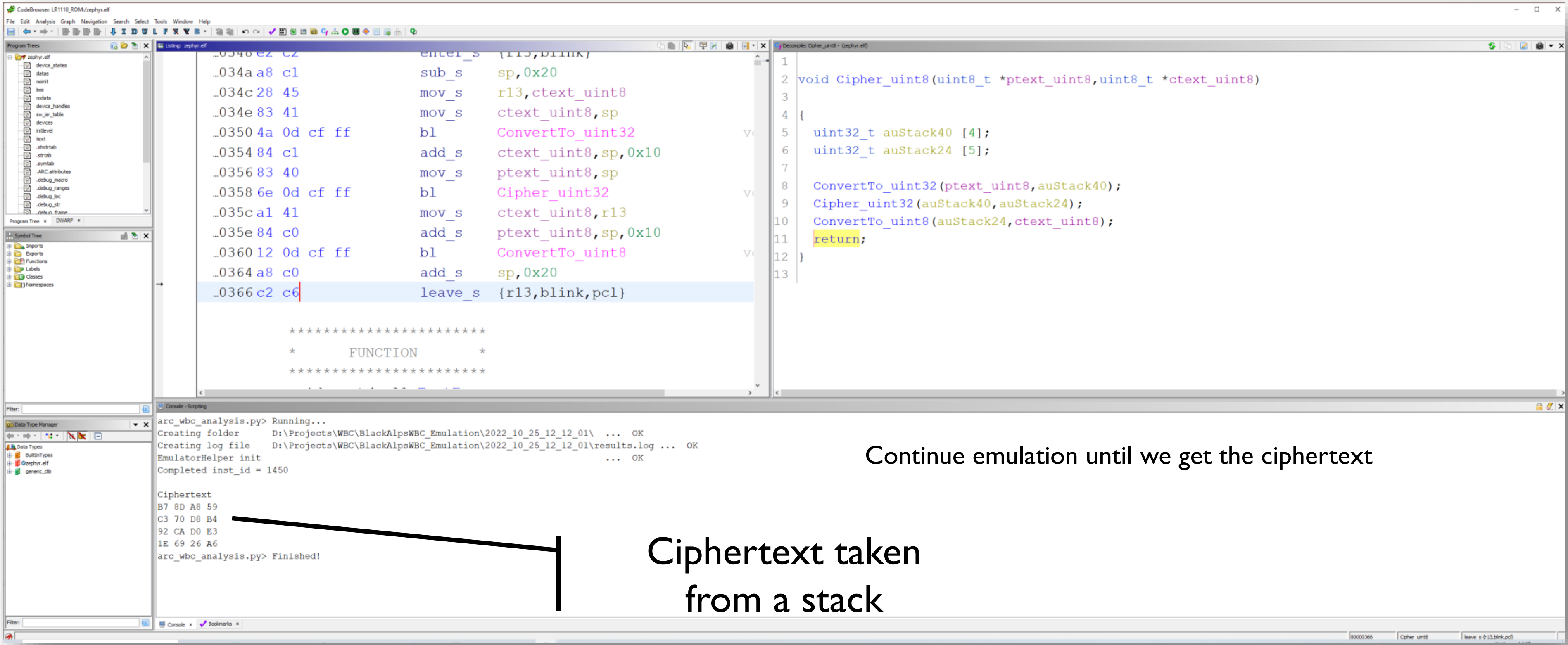
```
109   ram:80000348: enter_s {r13,blink}
Stack
1FFFFFA0:   00000000   00000000   00000000   00000000    |  ................
1FFFFFB0:   00000000   00000000   00000000   00000000    |  ................
1FFFFFC0:   00000000   00000000   00000000   00000000    |  ................
1FFFFFD0:  |84030080|  00000000   3143f6a8   885a308d    |  ........1C...Z0.
1FFFFFE0:   313198a2   e0370734   00000000   00000000    |  11...7.4........
1FFFFFF0:   00000000   00000000   efbeadde   babaedfe    |  ................
CPU context
r0  = 1FFFFFD8 r1  = 1FFFFFE8                          r2  = 00000000 r3  = 1FFFFFE8
r4  = 00000000 r5  = 00000000                          r6  = 00000000 r7  = 00000000
r8  = 00000000 r9  = 00000000                          r10 = 00000000 r11 = 00000000
r12 = 00000034 r13 = 00000000                          r14 = 00000000 r15 = 00000000
r16 = 00000000 sp  = 1FFFFFD0 (new) - 1FFFFFD8 (old)
blink = 80000384
C   = 00000000 V   = 00000000 N   = 00000000 Z   = 00000001

...
```

AES-128 ARCv2 Emulation

Plaintext to encrypt

```
109   ram:80000348: enter_s {r13,blink}
Stack
1FFFFFA0:   00000000   00000000   00000000   00000000    |  ................
1FFFFFB0:   00000000   00000000   00000000   00000000    |  ................
1FFFFFC0:   00000000   00000000   00000000   00000000    |  ................
1FFFFFD0:  |84030080|  00000000   3143f6a8   885a308d    |  ........1C...Z0.
1FFFFFE0:   313198a2   e0370734   00000000   00000000    |  11...7.4........
1FFFFFF0:   00000000   00000000   efbeadde   babaedfe    |  ................
CPU context
r0   = 1FFFFFD8 r1   = 1FFFFFE8                          r2   = 00000000 r3   = 1FFFFFE8
r4   = 00000000 r5   = 00000000                          r6   = 00000000 r7   = 00000000
r8   = 00000000 r9   = 00000000                          r10  = 00000000 r11  = 00000000
r12  = 00000034 r13  = 00000000                          r14  = 00000000 r15  = 00000000
r16  = 00000000 sp   = 1FFFFFD0 (new) - 1FFFFFD8 (old)
blink = 80000384
C    = 00000000 V    = 00000000 N    = 00000000 Z    = 00000001

...
```

AES-128 ARCv2 Emulation



SONY

AES-128 ARCv2 Emulation

Plaintext

Ciphertext

```
1446   ram:80000366: leave_s {r13,blink,pcl}
Stack
1FFFFFA0:   5c030080   e8ffff1f   00000000   e8ffff1f    |  \...
1FFFFFB0:   a8f64331   8d305a88   a2983131   340737e0    |  ..C1.0Z...114.7.
1FFFFFC0:   1e92c3b7   69ca708d   26d0d8a8   a6e3b459    |  ....i.p........Y
1FFFFFD0:   84030080   00000000  |3143f6a8|  885a308d    |  ........1C...Z0.
1FFFFFE0:   313198a2   e0370734   b7c3921e   8d70ca69    |  11...7.4.....p.i
1FFFFFF0:   a8d8d026   59b4e3a6   efbeadde   babaedfe    |  ....Y...........

r0  = 1FFFFFC0 r1  = 1FFFFFE8                          r2  = 00000004 r3  = 59B4E3A6
r4  = 2F2F715E r5  = C65197C6                          r6  = 00000001 r7  = 00000096
r8  = 00000000 r9  = 00000000                          r10 = 00000000 r11 = 00000000
r12 = 1FFFFFF4 r13 = 00000000 (new) - 1FFFFFE8 (old) r14 = 00000000 r15 = 00000000
r16 = 00000000 sp  = 1FFFFFD8 (new) - 1FFFFFD0 (old)
blink = 80000384 (new) - 80000364 (old)
C   = 00000000 V   = 00000000 N   = 00000000 Z   = 00000001
```

Windows PowerShell | Anaconda

```
PS E:\AES_WBC> .\aes.exe
B7 C3 92 1E 8D 70 CA 69 A8 D8 D0 26 59 B4 E3 A6
PS E:\AES_WBC>
```

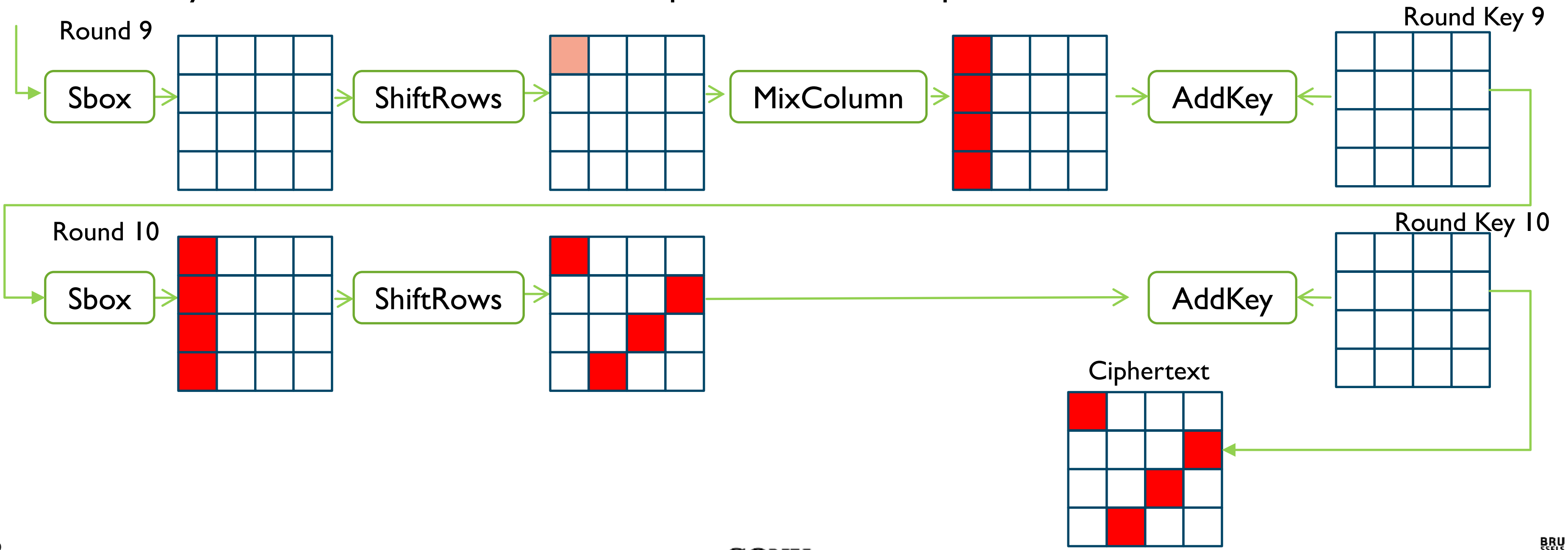
Fault Injection Into AES-128
ARCv2

Attacks on White-Box Crypto

- White-Box Cryptography mathematically hides the master key into the operations/tables
- Three types of attacks are possible:
  - Differential Fault Attacks (emulation or instrumentation)
  - Reverse engineering + mathematical attacks (collisions and others)
  - Differential Computation Analysis – side-channels for WBC (emulation or instrumentation)

Attacks on White-Box Crypto

- WBC can not change AES structure: Sbox, ShiftRows and MixColumn are present in the code
- One byte before the last MixColumn operation is the simplest fault attack



State-of-the-Art Tools for WBC

Side-Channel Marvels

SCA-related projects

- Dynamic binary instrumentation (Intel PIN, Valgrind)
- One of the most popular tools (Philippe Teuwen)

Riscure / **FiSim** Public

Unboxing the White-Box

Practical attacks against Obfuscated Ciphers

Eloi Sanfelix eloi@riscure.com
Cristofaro Mune cristofaro@riscure.com
Job de Haas job@riscure.com

- Qemu based (Unicorn) emulation

kudelskisecurity / **radare2-fault-simulator** Public

- Radare2 based emulation

korkikian / **ARCv2** Public

- Ghidra based emulation

SONY

BRUSSELS LAB

Fault Models

- Practical fault injection is somewhat unpredictable (we don’t know in advance which effects are achievable)
- Most common faults observed in various evaluations:
  - Instruction skipping
  - Register modification
- Those fault models can be emulated with Ghidra

Instruction Skipping Faults in AES-128

- An instruction skipping fault

```
for inst_id in range(max_num_instructions):
  executionAddress = emuHelper.getExecutionAddress()
  inst             = getInstructionAt(executionAddress)

  if inst_id in fault_instr_index and fault_type == 'skip':
    next_inst = inst.getNext()
    next_addr = getInstructionContext().getAddress().getAddressableWordOffset()
    emuHelper.writeRegister(emuHelper.getPCRegister(), next_addr)
    continue

  success = emuHelper.step(monitor)
```

Get the next instruction before emulating current

Update PC value with a next instruction address

Skip current instruction emulation

Instruction Skipping Faults in AES-128

```
Instruction skipping fault at 250 ram:800000b6 brne r3,0x4,0x800000a4
Completed inst_id = 1450

Correct             Current            XOR
B7 8D A8 59         B7 8D A8 59         00 00 00 00
C3 70 D8 B4         C3 70 D8 B4         00 00 00 00
92 CA D0 E3         92 CA D0 E3         00 00 00 00
1E 69 26 A6         1E 69 26 A6         00 00 00 00
```

A fault at a not-taken branch does not corrupt ciphertext

```
Instruction skipping fault at 700 ram:800001b4 asl_s r15,r15,0x2
Completed inst_id = 1450

Correct             Current            XOR
B7 8D A8 59         9A FA 59 E3         2D 77 F1 BA
C3 70 D8 B4         77 9A B8 F6         B4 EA 60 42
92 CA D0 E3         46 C7 0B E8         D4 0D DB 0B
1E 69 26 A6         F4 7C 32 C9         EA 15 14 6F
```

A fault at early rounds totally modifies a ciphertext

```
Instruction skipping fault at 1250 ram:800001c4 bmsk r5,r5,0x7
Completed inst_id = 1450

Correct             Current            XOR
B7 8D A8 59         B7 8D 2C 59         00 00 84 00
C3 70 D8 B4         C3 7A D8 B4         00 0A 00 00
92 CA D0 E3         2D CA D0 E3         BF 00 00 00
1E 69 26 A6         1E 69 26 89         00 00 00 2F
```

A fault at latest steps results in a required error pattern

Register Modification Fault in AES-128

- A register modification fault

```
inst                 = getInstructionAt(executionAddress)

if inst_id in fault_instr_index and fault_type == '1bit':
  num_operands = inst.getNumOperands()
  reg0         = inst.getRegister(0)

  if inst.getOperandRefType(0) == RefType.READ_WRITE and reg0:
    prev_value = emuHelper.readRegister(reg0)
    next_value = prev_value ^ 0x01
    emuHelper.writeRegister(reg0, next_value)
    up_value = emuHelper.readRegister(reg0)
```

Read a register, update its value and write back

Register Modification Fault in AES-128

```
Injecting 1-bit fault into instruction at 200 ram:800000ae add_s r3,r3,0x1
r3   = 00000004 (new) <- 00000005 (old)
Completed inst_id = 4000

Correct           Current           XOR
B7 8D A8 59       B7 8D A8 59        00 00 00 00
C3 70 D8 B4       C3 70 D8 B4        00 00 00 00
92 CA D0 E3       92 CA D0 E3        00 00 00 00
1E 69 26 A6       1E 69 26 A6        00 00 00 00
```

A fault at certain instructions does not corrupt ciphertext but changes the number of emulated instructions

```
Injecting 1-bit fault into instruction at 700 ram:800001b4 asl_s r15,r15,0x2
r15  = 0000E2F8 (new) <- 0000E2F9 (old)
Completed inst_id = 1450

Correct           Current           XOR
B7 8D A8 59       3C C2 41 DA        8B 4F E9 83
C3 70 D8 B4       A2 A4 E3 C8        61 D4 3B 7C
92 CA D0 E3       80 84 F5 4E        12 4E 25 AD
1E 69 26 A6       16 0C 80 E6        08 65 A6 40
```

A fault at early rounds totally modifies the ciphertext (an instruction skipping at this address modifies a ciphertext as well)

```
Injecting 1-bit fault into instruction at 1250 ram:800001c4 bmsk r5,r5,0x7
r5   = 00000049 (new) <- 00000048 (old)
Completed inst_id = 1450

Correct           Current           XOR
B7 8D A8 59       B7 8D 42 59        00 00 EA 00
C3 70 D8 B4       C3 09 D8 B4        00 79 00 00
92 CA D0 E3       F1 CA D0 E3        63 00 00 00
1E 69 26 A6       1E 69 26 3D        00 00 00 9B
```

A fault at latest steps results in a required error pattern

Application of Emulation

- Security testing
  - Fuzzing:

    www.protect.airbus.com/blog/fuzzing-exotic-arch-with-afl-using-ghidra-emulator/
  - Attacks on White-Box Cryptography:

    www.blackhat.com/docs/eu-15/materials/eu-15-Sanfelix-Unboxing-The-White-Box-Practical-Attacks-Against-Obfuscated-Ciphers-wp.pdf
- Functionality testing
  - Algorithm optimisation
  - Verification

Conclusions

- Ghidra emulation added to the list of fault injection tools
  - Faults: instruction skipping, register modification and others

- Ghidra ARCv2 support is released (and being worked on)

- Reversing and emulating a rare CPU architecture is feasible

Useful Links

- Current work:
  https://github.com/korkikian/ARCv2
- ARCompact:
  https://github.com/niooss-ledger/ghidra
- SLEIGH description:
  https://fossies.org/linux/ghidra/GhidraDocs/languages/html/sleigh.html
- Ghidra:
  https://github.com/NationalSecurityAgency/ghidra
- Side-channel Marvels:
  https://github.com/SideChannelMarvels
- Radare2 fault emulation:
  https://github.com/kudelskisecurity/radare2-fault-simulator
- Riscure Fisim
  https://github.com/Riscure/FiSim

THANK YOU!

My contacts:

roman.korkikian@gmail.com
+41799062793