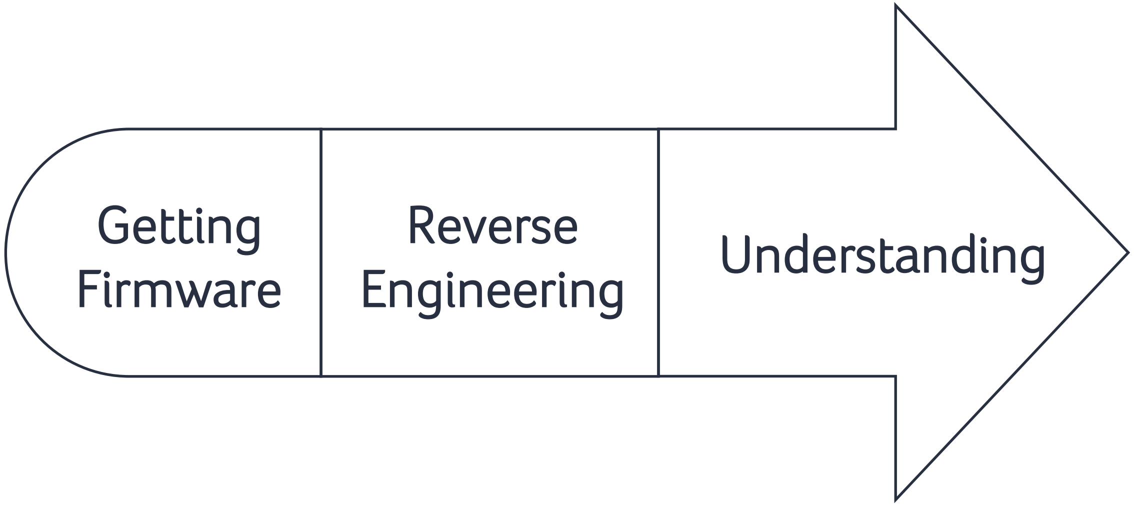
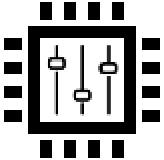
Efficient Reverse Engineering of Automotive Firmware

Alyssa Milburn
Security Analyst, Riscure
milburn@riscure.com / @noopwafel

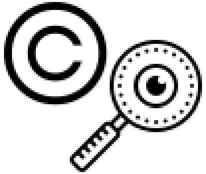
(with Niek Timmers)

Reverse Engineering

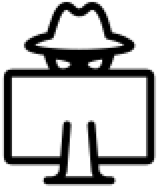




Tuning / manipulation



IP



Hacking

???

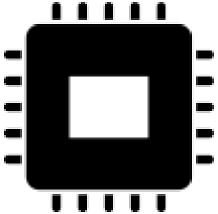
Automotive Firmware?

Instrument Cluster

- Speedometer/gauges
- Display (screen)
- Speaker!
- Blinky lights!

- 32-bit CPU
- CAN bus
- I2C bus
  - EEPROM

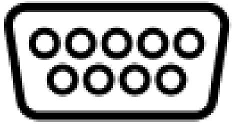
How can we get the firmware?



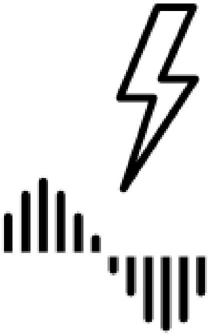
External flash

Software vulnerabilities

Leaks



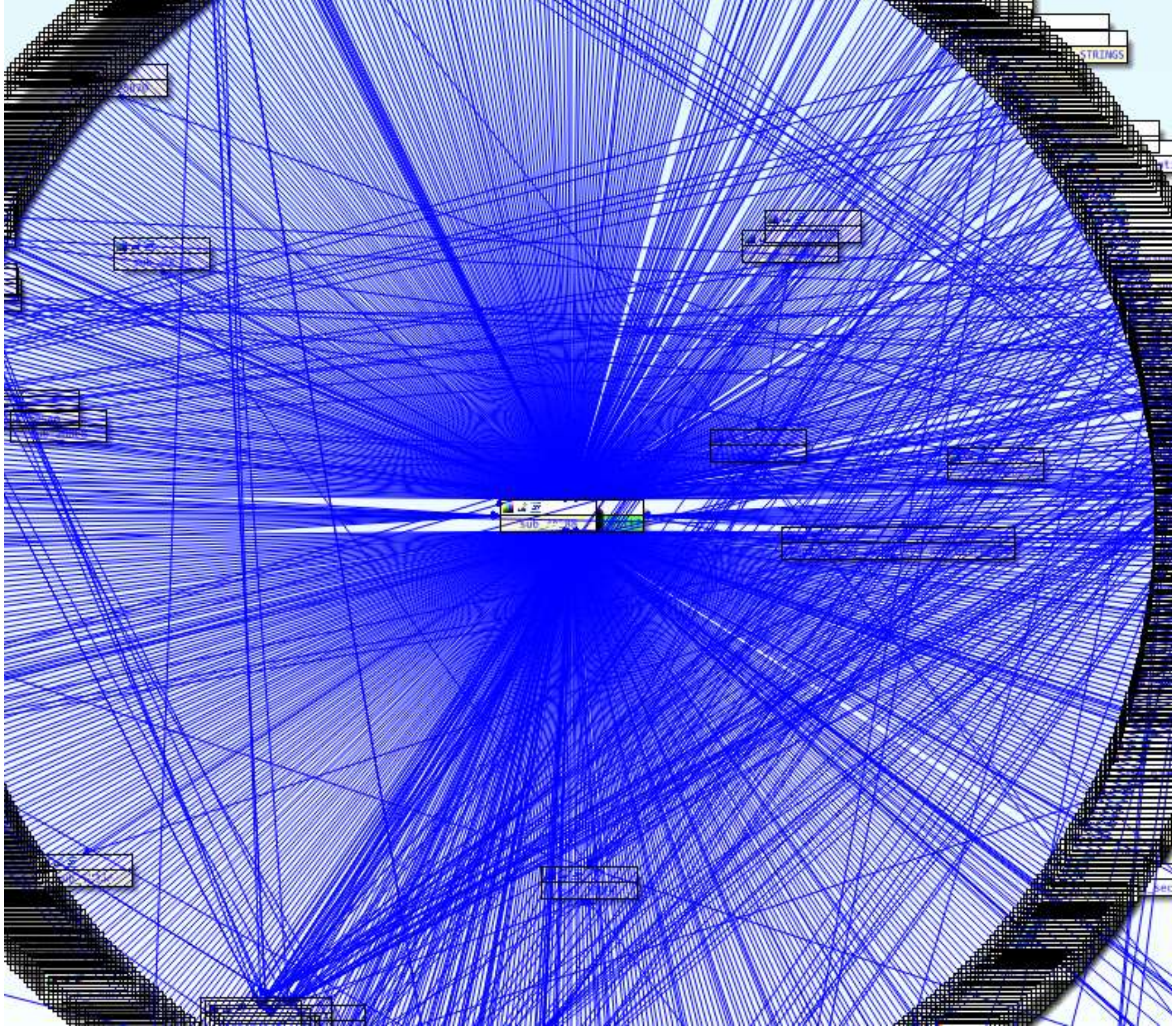
Debug interfaces



Hardware attacks

What makes this challenging?

- “Non-standard” platforms
- New concepts
- Complexity


STRINGS

What makes this challenging?

- **Static analysis** (disassembly): too complicated
- **Dynamic analysis** (emulation / debugging): no tools?

No tools?! Let's make some!

What do we **need**?

- Processor (instruction set) emulator

- Timers, interrupts
- CAN controller
- I2C controller
  - EEPROM
  - Display controller

Emulating the CPU architecture

```
case [redacted]:
    INSTX(or, "r%d, r%d", low, high);
    assert(high != 0);
    if (high != 0) {
        m_registers[high] |= m_registers[low];
        TAINT_REG_OR(high, low);
        ZERO_FLAG(m_registers[high]);
        NEG_FLAG(m_registers[high]);
        updatePSW(false, PSW_OV);
    }
    pc += 2;
    break;
```

“Implementing” peripherals

```
case 0x[redacted]:
    // [redacted]
    // not implemented yet
    break;
case 0x[redacted]:
    // [redacted]
    break;
case 0x[redacted]:
    // [redacted]
    // for now, we just pretend the clock initializes instantly
    printf("** clock init **\n");
    *(uint8_t *)&m_memory[addr] = 0;
    break;
```

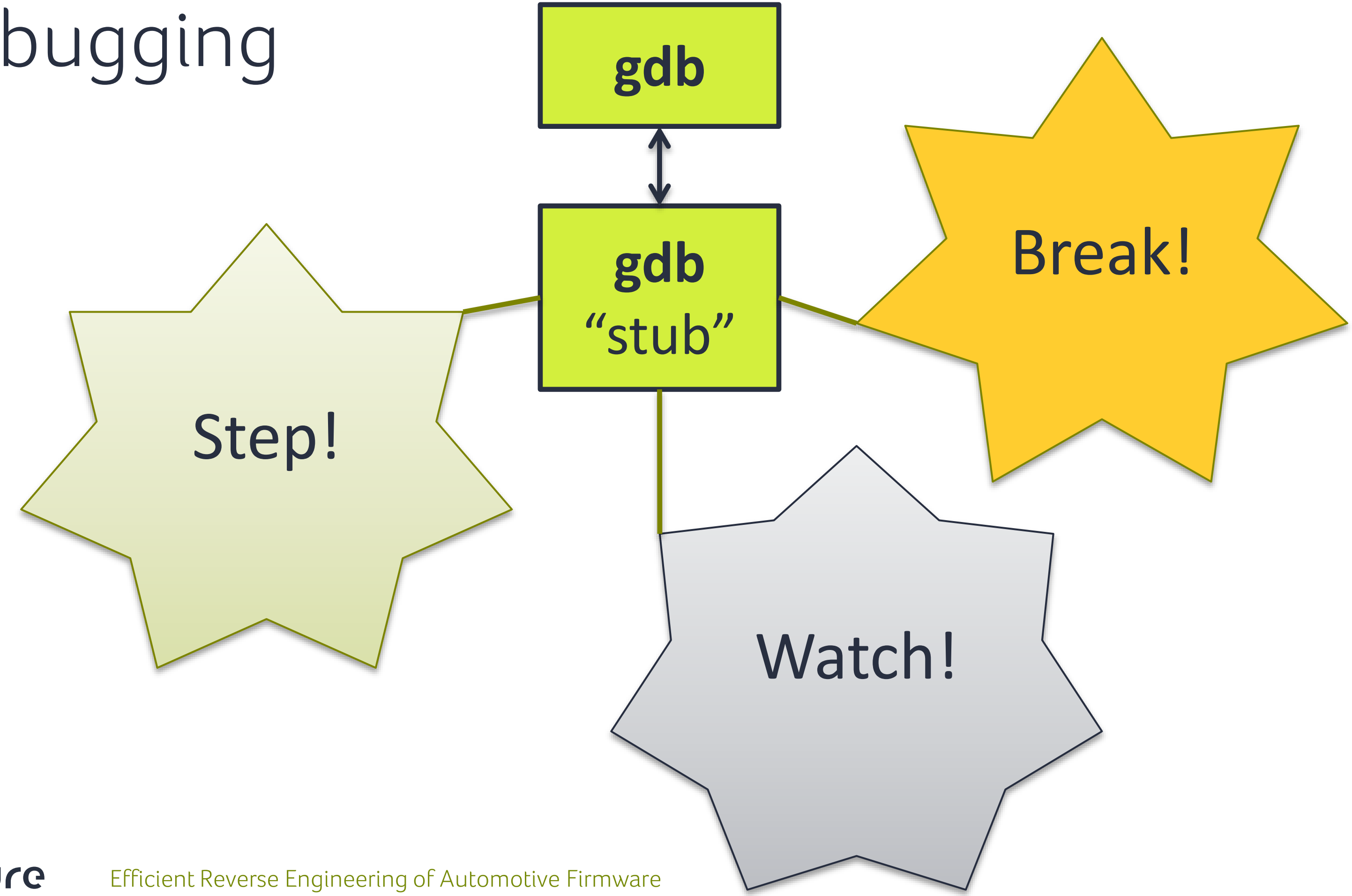
How difficult was it?

~ **1 man-week** of work

~ **3000 lines** of (terrible) code
(excluding support tooling)

Dynamic analysis

Debugging



Debugging

```
(gdb) hbreak *0x11032
Hardware assisted breakpoint 1 at 0x11032

(gdb) c
Continuing.

0x00011032 in ?? ()
(gdb)
```

Execution tracing

```
            call    getChecksumChunkSize, lp
            mov     r10, r7
            mov     r27, r6
            call    calculateChecksum, lp -- r6 is pointer (note: skips first 2 bytes)
                                      -- r7 is size to check (in bytes)
                                      -- returns     checksum in r10
            cmp     r10, r29
            bz      ret
            xor     0xAAAA, r29, r0
            bz      ret
            mov     0xFFFF, r0, r1
            set     3, (g_globalIntegrityState - 0x3FF0000)[r1]
            mov     1, r28            -- checksum was invalid (manipulation)

ret:                                  -- CODE XREF: performChecksumVerification+1C↑j
                                      -- performChecksumVerification+22↑j
            mov     r28, r10
            z       r10
            call    pop_r26tor29_lp
-- End of function performChecksumVerification
```

Execution tracing

0x02920
0x02922 (jump)
0x02926
0x02928
0x0292c
0x02930

Execution tracing

0x02920
0x02922 (jump)
0x02926
0x02928
0x0292c
0x02930

Execution tracing

```
                call    getChecksumChunkSize, lp
                mov     r10, r7
                mov     r27, r6
                call    calculateChecksum, lp -- r6 is pointer (note: skips first 2 bytes)
                                              -- r7 is size to check (in bytes)
                                              -- returns      checksum in r10
                cmp     r10, r29
                bz      ret
                xor     0xAAAA, r29, r0
                bz      ret
                mov     0xFFFF, r0, r1
                set     3, (g_globalIntegrityState - 0x3FF0000)[r1]
                mov     1, r28                -- checksum was invalid (manipulation)

ret:                                          -- CODE XREF: performChecksumVerification+1C↑j
                                              -- performChecksumVerification+22↑j
                mov     r28, r10
                z       r10
                call    pop_r26tor29_lp
-- End of function performChecksumVerification
```

Execution tracing

```
                call    getChecksumChunkSize, lp
                mov     r10, r7
                mov     r27, r6
                call    calculateChecksum, lp -- r6 is pointer (note: skips first 2 bytes)
                                              -- r7 is size to check (in bytes)
                                              -- returns      checksum in r10
                cmp     r10, r29
                bz      ret
                xor     0xAAAA, r29, r0
                bz      ret
                mov     0xFFFF, r0, r1
                set     3, (g_globalIntegrityState - 0x3FF0000)[r1]
                mov     1, r28                -- checksum was invalid (manipulation)

ret:                                          -- CODE XREF: performChecksumVerification+1C↑j
                                              -- performChecksumVerification+22↑j
                mov     r28, r10
                z       r10
                call    pop_r26tor29_lp
-- End of function performChecksumVerification
```

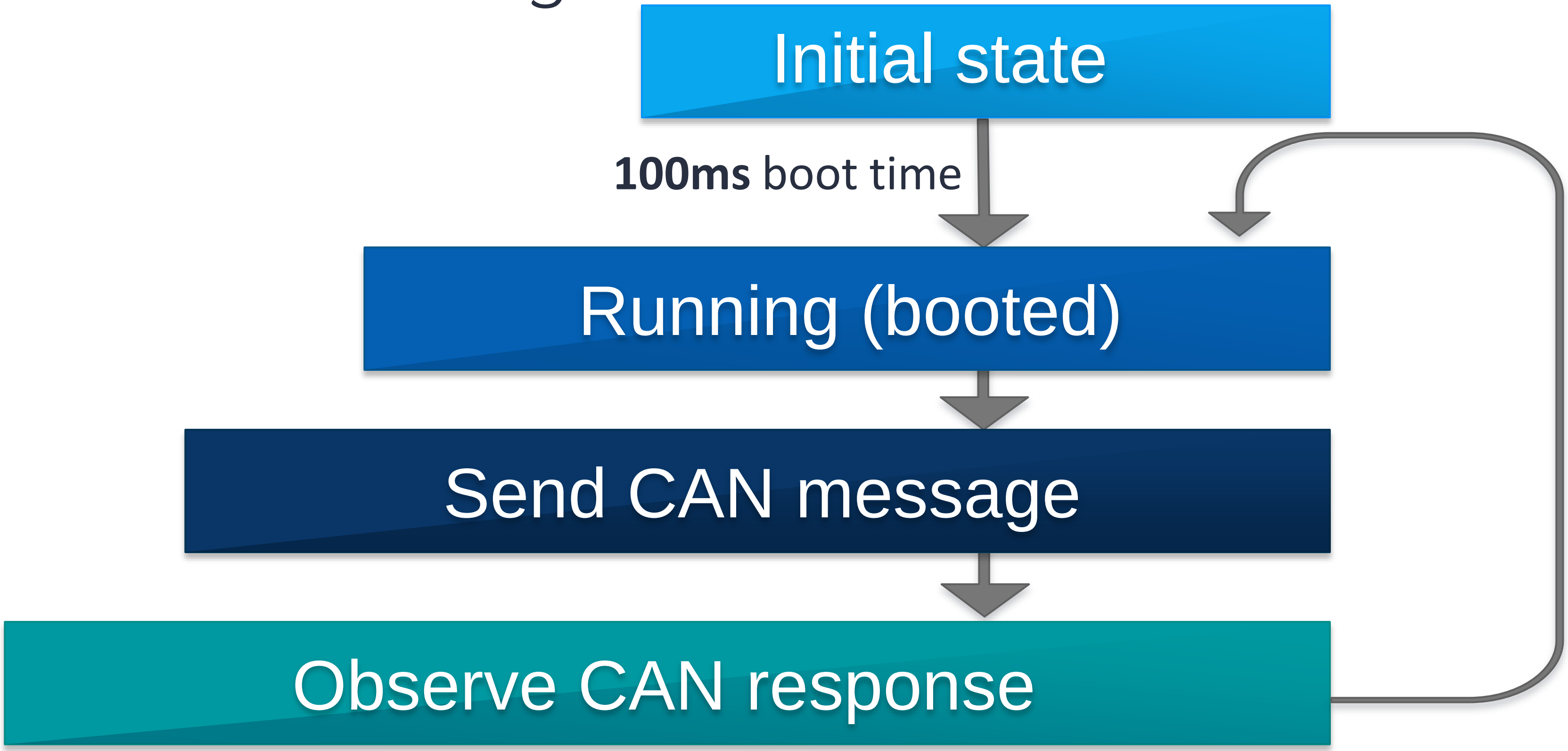
Hacks!

Hacks!

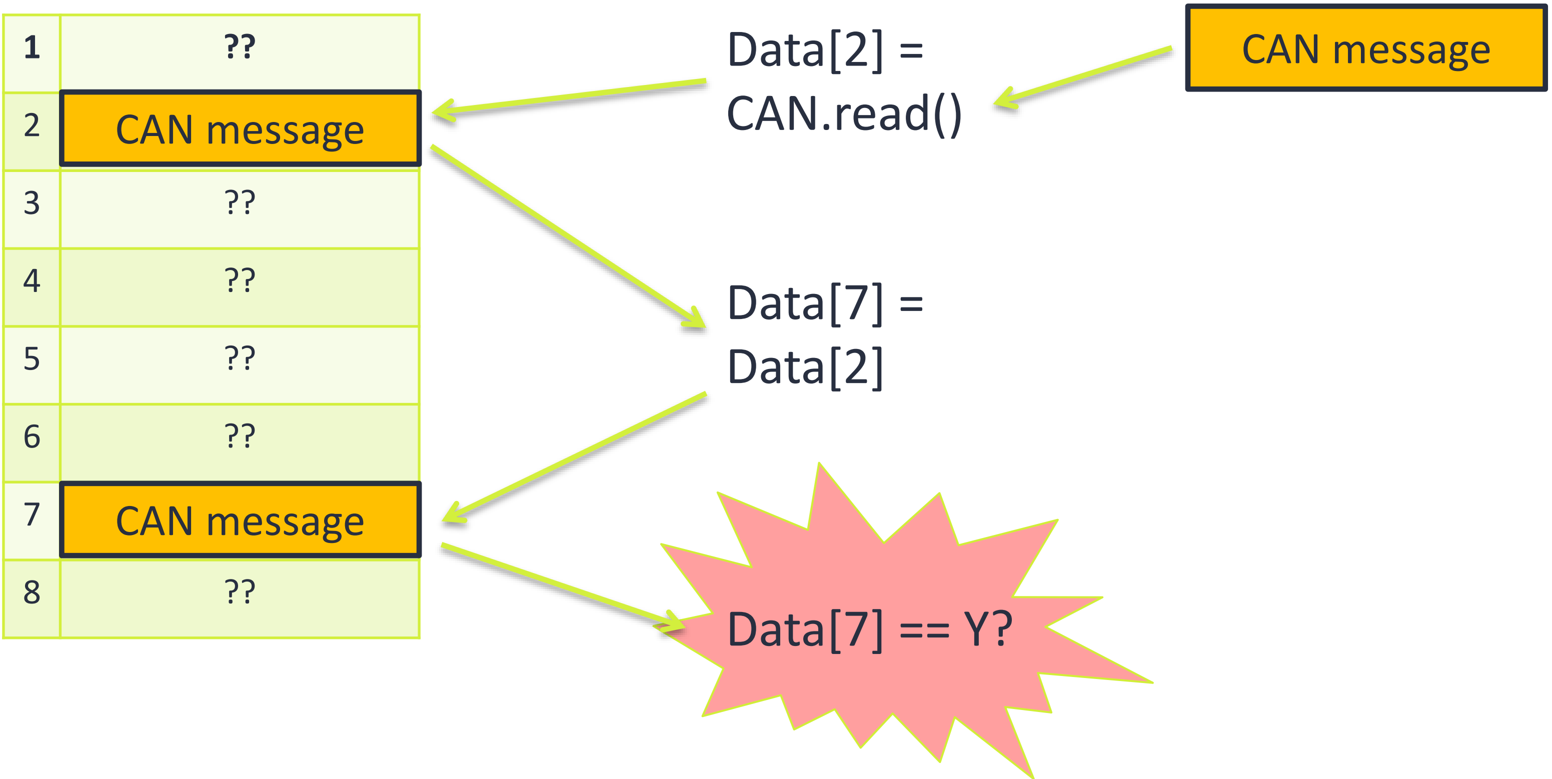
```
if (m_pc == 0x[redacted]) {
    // end of message display: print tmp buffer
    printf("\n");
    hexdump(&m_memory[0x[redacted]], 30);
    printf("\n");
}

if (m_pc == 0x[redacted]) {
    // segments on/off
    if (m_registers[7])
        printf("[on  %02x: %02x] ", m_registers[6] >> 3, m_registers[6] & 0x7);
    else
        printf("[off %02x: %02x] ", m_registers[6] >> 3, m_registers[6] & 0x7);
}
```

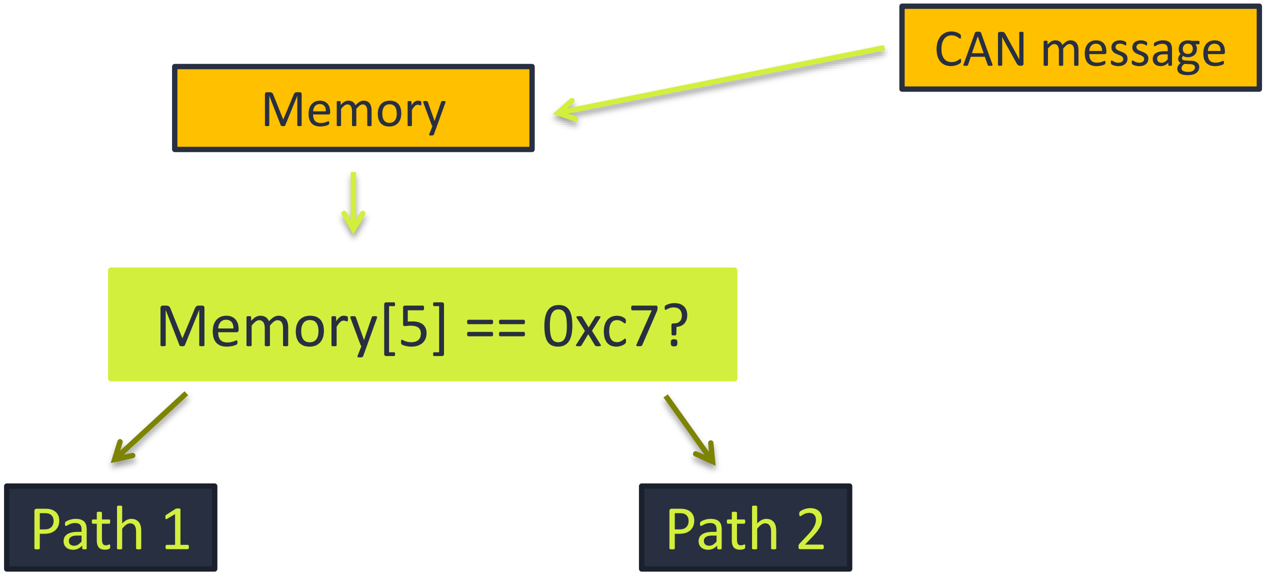
State rewinding



Taint tracking



Fuzzing

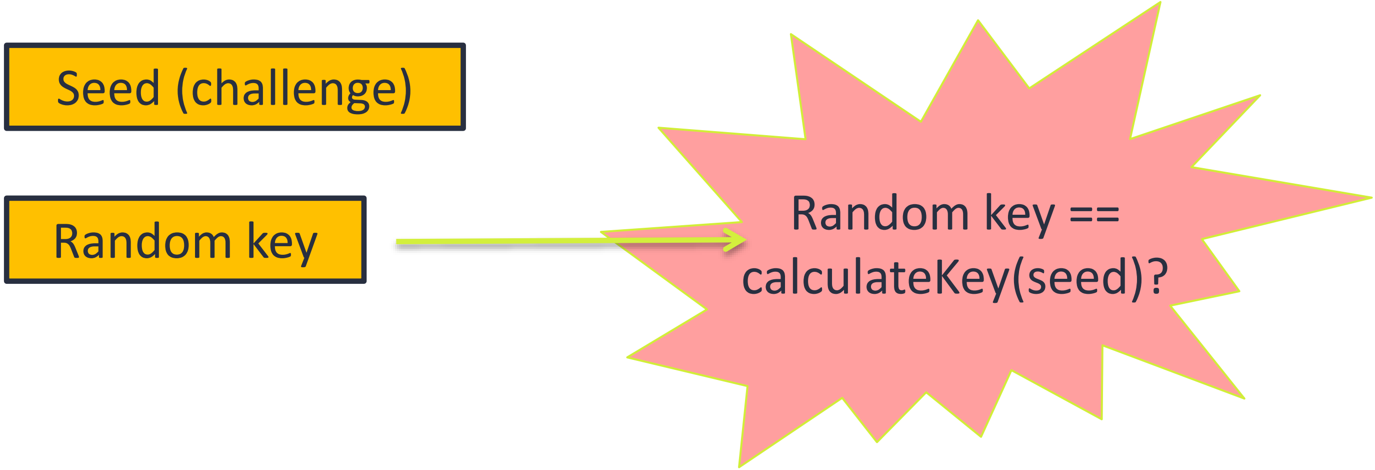


UDS

```
./cc.py dcm discovery

CARING CARIBOU v0.1
-------------------
Starting diagnostics service discovery
Found diagnostics at arbitration ID 0x     ,
reply at 0x
```

UDS: security access



We found *calculateKey*!

UDS: security access

```
sending requestSeed (0x3)
CAN0: RCV [id [illegible]] 02 27 03 aa aa aa aa aa
CAN0: TRQ [id [illegible]] 06 67 03 47 2e 8e 70 aa
sending sendKey
CAN0: RCV [id [illegible]] 06 27 04 41 9b 35 42 aa

comparison at 0002f390 (419b3542 vs
419b3542) is tainted with 000000c0

CAN0: TRQ [id [illegible]] 02 67 04 aa aa aa aa aa
```

EEPROM contents

Identification (VIN)

Features/ configuration

(UDS) security state

Odometer 😐

Reverse engineering is hard work!

updateEEPROM(id, value)

Takeaways

- Reverse engineering is **not so hard**!
- Lots of other “tricks” to try:
  - Symbolic execution
  - Deobfuscation (if necessary)
  - Smarter fuzzing
- You can’t hide secrets in firmware:
  - Use **asymmetric cryptography** (i.e. public keys)
  - Use the **secure hardware** inside modern processors

Thanks to...

Eloi Sanfelix

Santiago Cordoba

Ramiro Pareja

Efficient Reverse Engineering of Automotive Firmware

Alyssa Milburn
Security Analyst, Riscure
milburn@riscure.com / @noopwafel

riscure

Challenge your security

- Training
- Tools
- Services

icons8.com