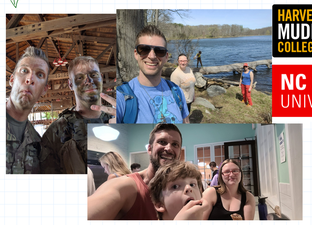
Introduction to Hardware (In)Security with the ChipWhisperer-Nano

Nathan Jones, Managing Member

Mindstorms Engineering, LLC
Embedded Systems Design & Education



Learning Objectives

- Hardware attack sequence
- Side-channel attacks
  - Simple power analysis
  - Differential power analysis ← Recover a full AES-128 key
- Fault injection attacks ← Bypass a password check

Embedded Online Conference (Ref10)
- "Exception Handling" (workshop)
- "The Power of a Look-up Table"
- "OOP in C"
- "Building a Simple Command-line Interface"

EmbeddedRelated.com (Ref11)
- "You Don't Need an RTOS"
- "Simulate Your Embedded Project"

Hackaday Supercon: "Inside the Vojax" (Ref12)

Maker.io (DigiKey) (Ref13)
- printf series
- "How Hardware Gets Hacked"

Level 1: Researching

- Product manuals
- Schematics, FW
- Repair tutorials
- Patent filings
- FCC filings
- Datasheets
- (Un)Labelled ports



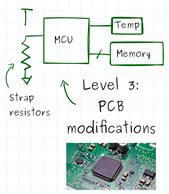
Level 2: Snooping


(R01)

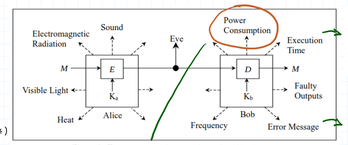
- JTAG
- UART
- SPI
- Peripherals

← Unlocked? Susceptible to fault injection?
← Does execution time leak the correct pw?
← Can we spoof or fault these signals for an atk?

Level 3: PCB modifications



Level 4: Side-channel attacks



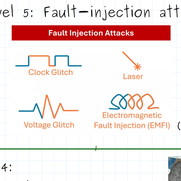
(R04)

Figure 2: The cryptographic model including side-channel



"Incorrect password for this user name"

Level 5: Fault-injection attacks



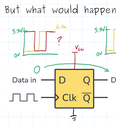
(R02)

Every MCU has required operating conditions...

Symbol	Parameter	Min	Max	Unit
f_{HCLK}	Internal AHB clock frequency	0	48	MHz
f_{PCLK}	Internal APB clock frequency	0	48	
V_{DD}	Standard operating voltage	2.0	3.6	V

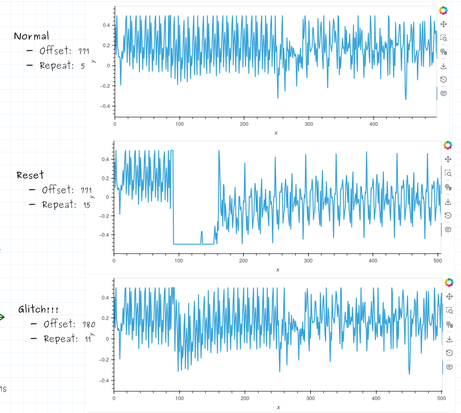
(R04)

But what would happen if those conditions weren't met?



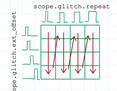
EX

We don't know! But *possibly*:
- Skipped instructions
- Altered/corrupted instructions
- Altered/corrupted memory



Conducting a **Fault Injection Attack**

1) Find a target (like one of those instructions)
   - Ex: beq.n 800066a
2) Determine a triggering event
   - Reset line going high
   - UART start/stop bit
3) Do a parameter sweep
4) Pray!




```
In:  [x2b x9a x5e xd1 xb4 xff x92 x26 xff xf6 x35 x1e x14 xe3 x08 x88]
Out: [xc7 x03 x2c xae xc8 x10 xc9 x5b x23 xa3 x19 x8c x74 x38 xb1 x81]
```

Should be x33, not x16!! (sbox(xff^x44) + x33)

Closing thoughts

- What are three things you want to remember from today's workshop?
- Fear not, countermeasures *do* exist!
- For more fun, check out "The Hardware Hacking Handbook"
- Please consider giving me 1 more minute of your time to help me learn what went well today (and what didn't go well!)





https://forms.gle/xAzbAmS3rR5q4q3E7

A lot of well-written software has **huge** vulnerabilities if the MCU is forced to randomly skip an instruction or if a register value is suddenly and inexplicably changed.

```
bool access = false;
char * secret = "SECRET THING";
char * denied = "ACCESS DENIED";

uint8_t get_secret(uint8_t* in, uint8_t len)
{
    if(access) simpleserial_put('r', strlen(secret), (uint8_t*)secret);
    else simpleserial_put('r', strlen(denied), (uint8_t*)denied);
    return 0x00;
}
```

Compiles into

```
uint8_t get_secret(uint8_t* in, uint8_t len)
{
Preamble
 8000638:   b580            push    {r7, lr}
 800063a:   b082            sub     sp, #8
 800063c:   af00            add     r7, sp, #0
 800063e:   6078            str     r0, [r7, #4]
 8000640:   000a            movs    r2, r1
 8000642:   1cfb            adds    r3, r7, #3
 8000644:   701a            strb    r2, [r3, #0]
Test access variable
    if(access) simpleserial_put('r', strlen(secret), (uint8_t*)secret);
 8000646:   4b12            ldr     r3, [pc, #72]   ; (8000690 <get_secret+0x58>)
 8000648:   781b            ldrb    r3, [r3, #0]
 800064a:   2b00            cmp     r3, #0
 800064c:   d00d            beq.n   800066a <get_secret+0x32>
Call simpleserial_put with secret msg
 800064e:   4b11            ldr     r3, [pc, #68]   ; (8000694 <get_secret+0x5c>)
 8000650:   681b            ldr     r3, [r3, #0]
 8000652:   0018            movs    r0, r3
 8000654:   f7ff fe9c       bl      8000390 <strlen>
 8000658:   0003            movs    r3, r0
 800065a:   b2d9            uxtb    r1, r3
 800065c:   4b0d            ldr     r3, [pc, #52]   ; (8000694 <get_secret+0x5c>)
 800065e:   681b            ldr     r3, [r3, #0]
 8000660:   001a            movs    r2, r3
 8000662:   2072            movs    r0, #114        ; 0x72
 8000664:   f000 f98e       bl      8000984 <simpleserial_put>
 8000668:   e00c            b.n     8000684 <get_secret+0x4c>
    else simpleserial_put('r', strlen(denied), (uint8_t*)denied);
 800066a:   4b0b            ldr     r3, [pc, #44]   ; (8000698 <get_secret+0x60>)
Call simpleserial_put with denied msg
 800066c:   681b            ldr     r3, [r3, #0]
 800066e:   0018            movs    r0, r3
 8000670:   f7ff fe8e       bl      8000390 <strlen>
 8000674:   0003            movs    r3, r0
 8000676:   b2d9            uxtb    r1, r3
 8000678:   4b07            ldr     r3, [pc, #28]   ; (8000698 <get_secret+0x60>)
 800067a:   681b            ldr     r3, [r3, #0]
 800067c:   001a            movs    r2, r3
 800067e:   2072            movs    r0, #114        ; 0x72
 8000680:   f000 f980       bl      8000984 <simpleserial_put>
    return 0x00;
 8000684:   2300            movs    r3, #0
}
Postamble
 8000686:   0018            movs    r0, r3
 8000688:   46bd            mov     sp, r7
 800068a:   b002            add     sp, #8
 800068c:   bd80            pop     {r7, pc}
 800068e:   46c0            nop                     ; (mov r8, r8)
 8000690:   20000028        .word   0x20000028
 8000694:   20000004        .word   0x20000004      Addr of secret msg
 8000698:   20000000        .word   0x20000000      Addr of denied msg
```

Normally:

After:	r3
ldr	Addr of access
ldrb	Value of access
cmp	Value of access

If ldrb is skipped:

After:	r3
ldr	Addr of access
ldrb	Value of secret
cmp	Addr of access

If addr is miscalculated:

After:	r3
ldr	Addr of access
ldrb	Value of $#x^1
cmp	Value of $#x^1

```
 800064c:   d00d            beq.n   800066a <get_secret+0x32>
 800064e:   4b11            ldr     r3, [pc, #68]   ; (8000694 <get_secret+0x5c>)
 8000650:   681b            ldr     r3, [r3, #0]
```

Execution goes straight into the "if" branch!!

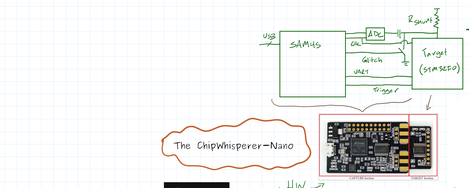
As above:
- Skip this instruction (r3 still has "value of access")
- Malform the addr (strlen operates on addr of $#x^1; not addr of denied)

```
 800066a:   4b0b            ldr     r3, [pc, #44]   ; (8000698 <get_secret+0x60>)
 800066c:   681b            ldr     r3, [r3, #0]
 800066e:   0018            movs    r0, r3
 8000670:   f7ff fe8e       bl      8000390 <strlen>
 8000674:   0003            movs    r3, r0
```

r0 holds addr of denied, not strlen(denied)!

If we can force "strlen(denied)" to be longer than "denied", we can read "secret"!!

But wait! There's more!
- Start-up code where access is set to 0/false
- Code that updates access if correct pw is entered



ADC data

"mul2_loop"

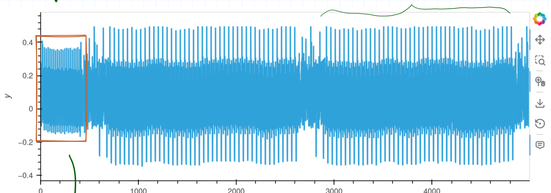
```
void get_pt(uint8_t* pt, uint8_t len)
{
    trigger_high();

    volatile long int A = 0xE6A9;
    for(volatile int i = 0; i < 20; i++) A *= 2;

    trigger_low();

    simpleserial_put('r', 16, pt);
}
```

One byte of the return message




```
API
  Scope API
  Target API
  Capture API
  Analyzer API
```

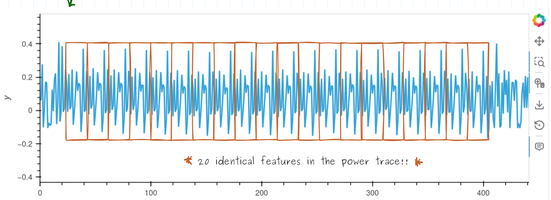
SW & HW

```
Example usage:

import chipwhisperer as cw

# Initialize scope and target
scope = cw.scope()
target = cw.target(scope, cw.targets.SimpleSerial)
scope.default_setup()

# Get a power trace (not shown: flashing the target)
ktp = cw.ktp.Basic()
trace = cw.capture_trace(scope, target, ktp.next_text())
# Sends 'p' + text to target; ADC records when trigger is set
# trace is a namedtuple with
#  - wave (ADC data)
#  - textin (the text that was sent)
#  - textout (the reply from the target)
#  - key (if used)
```



2.0 identical features in the power trace!!

Two important notes:
- MCUs are deterministic
- ADC data = Dynamic (switching) power = Hamming distance

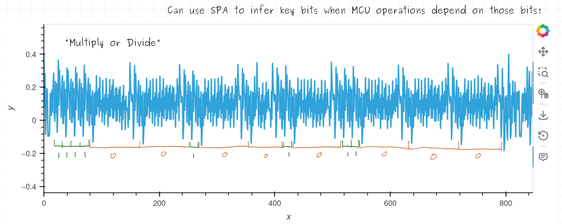


⇒ 6 transitions
Hamming distance = 6

Hamming weight = 5 & HD(0 → x) = HW(x)

This is **Simple Power Analysis**

Can use SPA to infer key bits when MCU operations depend on those bits:



RSA keys can be extracted in exactly this way.

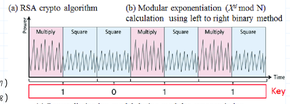
$C = P^e \bmod N$
$P = C^d \bmod N$

C: Cipher Text
P: Plain Text
e: Public Key
d: Private Key
N: modulo

```
input : X, N, d = (d_{t-1}, d_{t-2}, ..., d_0)
output: Z = X^d mod N
Z←1;
For i = k − 1 down to 0 do
    Z←Z^2 mod N;   //Square
    if (d_i = 1) then
        Z←Z × X mod N;  //Multiply
    end
end
return Z;
```

(a) RSA crypto algorithm

(b) Modular exponentiation ($X^d \bmod N$) calculation using left to right binary method

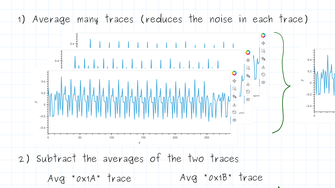


(c) Power dissipation model during modular exponentiation

(Ref07)
(Ref08)

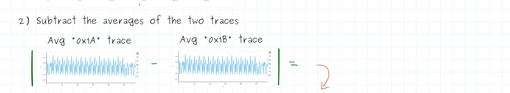
But what do we do if the traces look identical!?

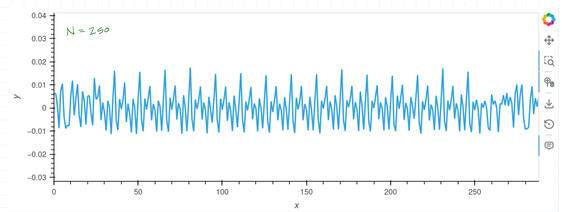
Input → MCU, running "table look-up" → Output (ADC data)

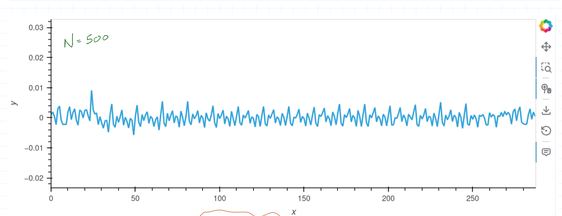


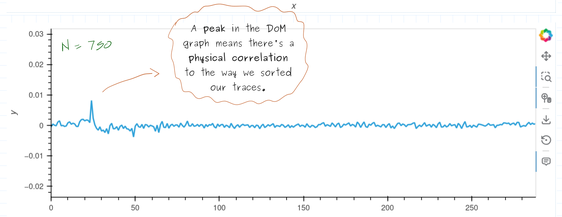
1) Average many traces (reduces the noise in each trace)

2) Subtract the averages of the two traces





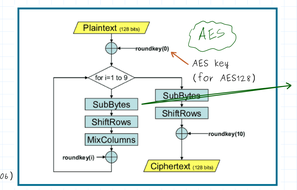




Requirements/assumptions:
- Traces are aligned in time
- Corollary: Exact same operation is occurring
- ADC captures relevant data
- Sufficient number of traces to reduce noise

Difference of Means (DoM)

We can use this to attack AES!!



(R06)

AES S-box

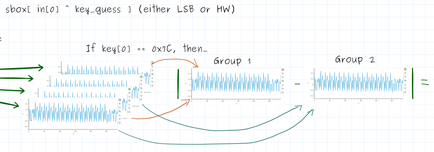
	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
30	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
40	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
50	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
60	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
70	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
80	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
90	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a0	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b0	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c0	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d0	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e0	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f0	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

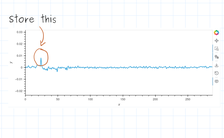
1) Collect many (1000s) of traces of device performing AES encryption

Input → MCU, running AES encryption → Output (ADC data)

```
[0x34, 0xA0, ... ]
[0x7D, 0x06, ... ]
[0x0C, 0xF1, ... ]
[0xB5, 0x82, ... ]
```

2) 256 times (for each possible value of key[0]):
- Sort traces based on sbox[in[0] ^ key_guess] (either LSB or HW)
- Calculate DoM
- Store peak DoM value





3) The key guess with the highest peak is the likeliest value!
4) Repeat steps 2 and 3 for each other byte of the 16-byte AES key

This is a **Differential Power Analysis (DPA)** attack on AES