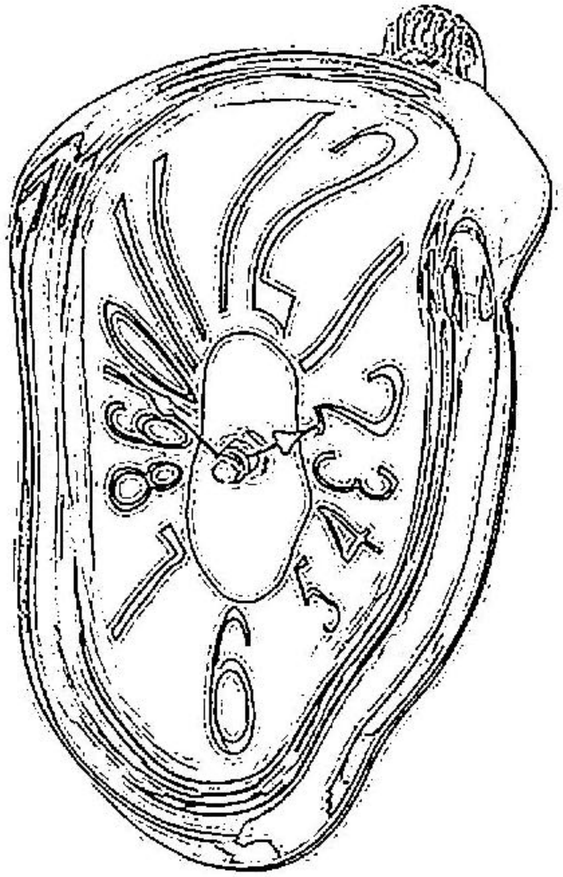
black hat EUROPE 2017



CLKSCREW

Exposing the Perils of Security-Oblivious Energy Management

Adrian Tang, Simha Sethumadhavan, Salvatore Stolfo
Columbia University

COLUMBIA UNIVERSITY IN THE CITY OF NEW YORK

#BHEU / @BLACK HAT EVENTS

$> whoami

Adrian Tang — Ph.D. candidate @ Columbia University

Member of
- Intrusion Detection Systems Labs (IDS)
- Computer Architecture and Security Technologies Lab (CASTL)

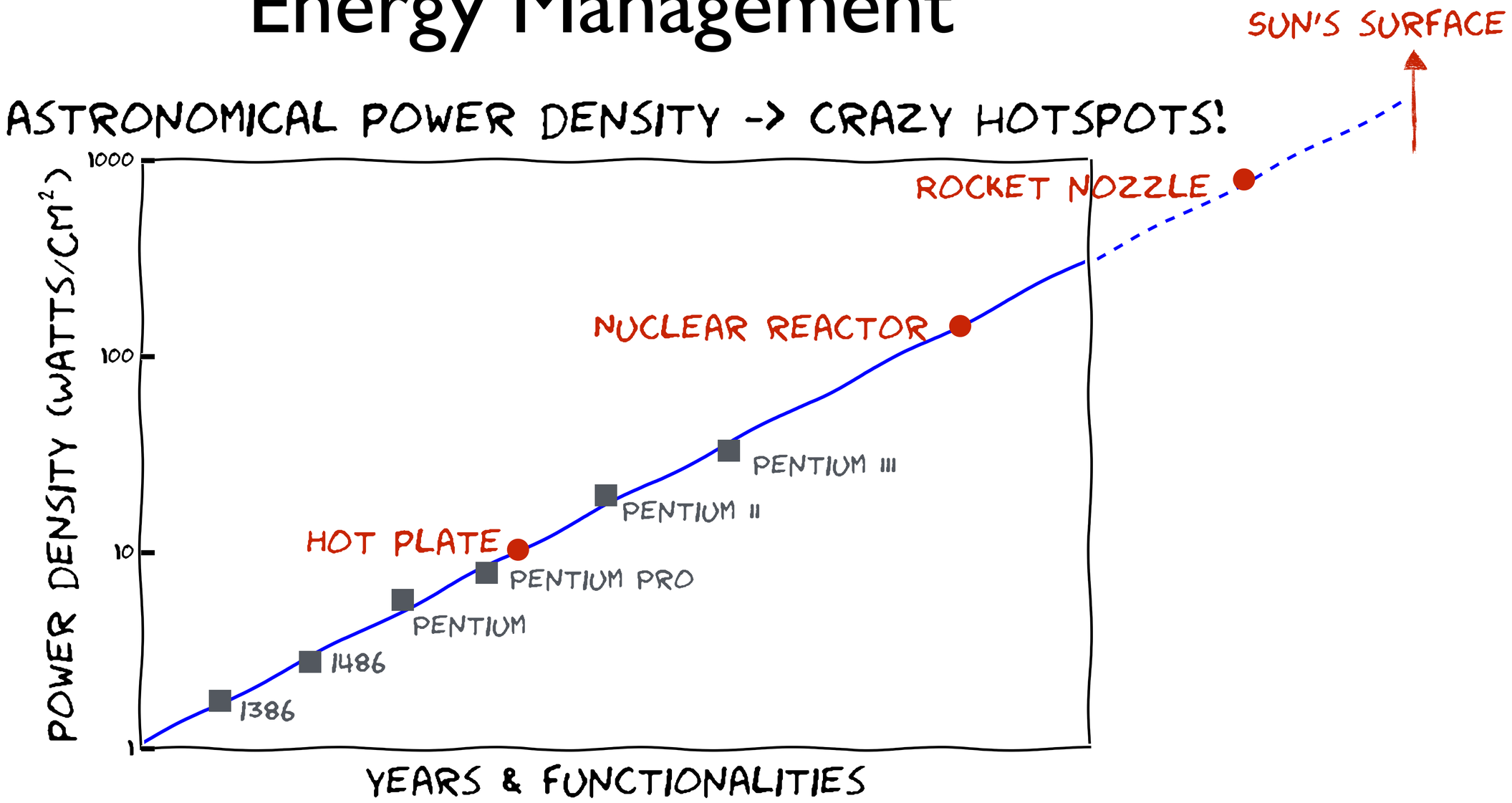
Interests
- Reverse engineering
- Bug hunting
- Malware analysis

Security research mainly focusing on hardware-software interfaces

@0x0atang

https://0x0atang.github.io

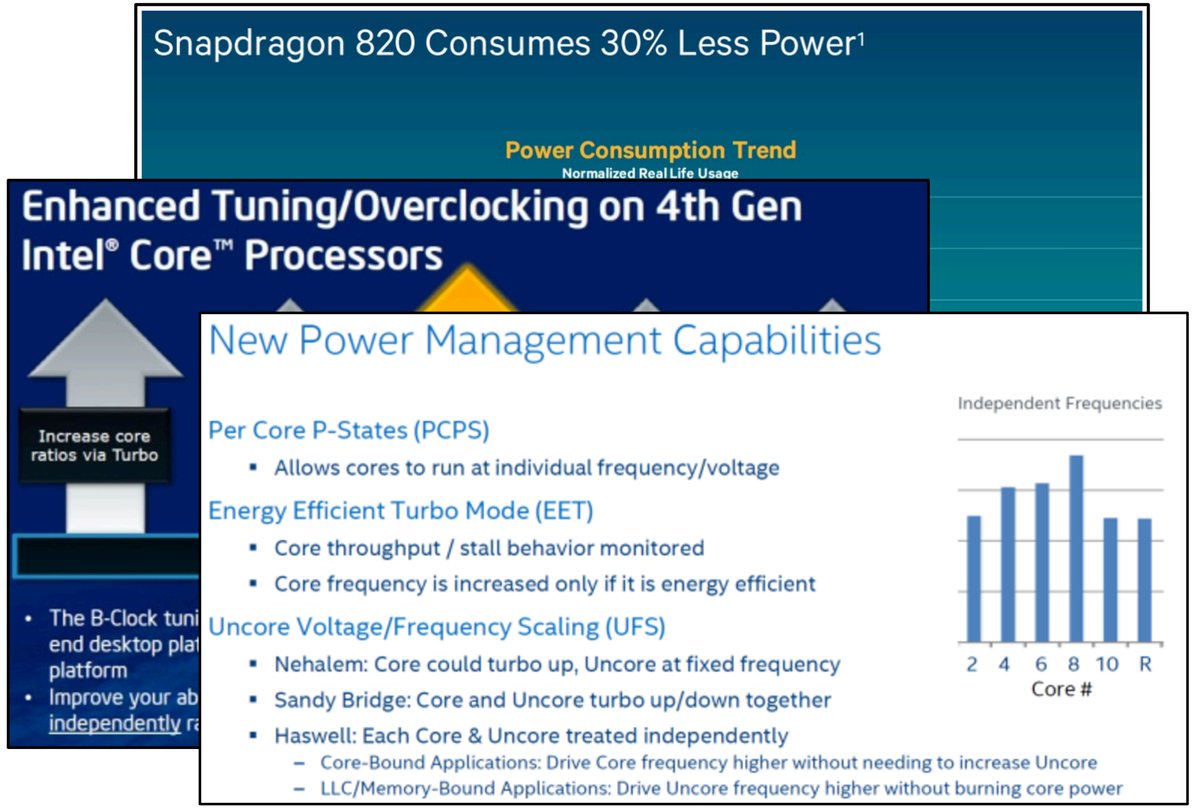
Today's systems cannot exist without **Energy Management**



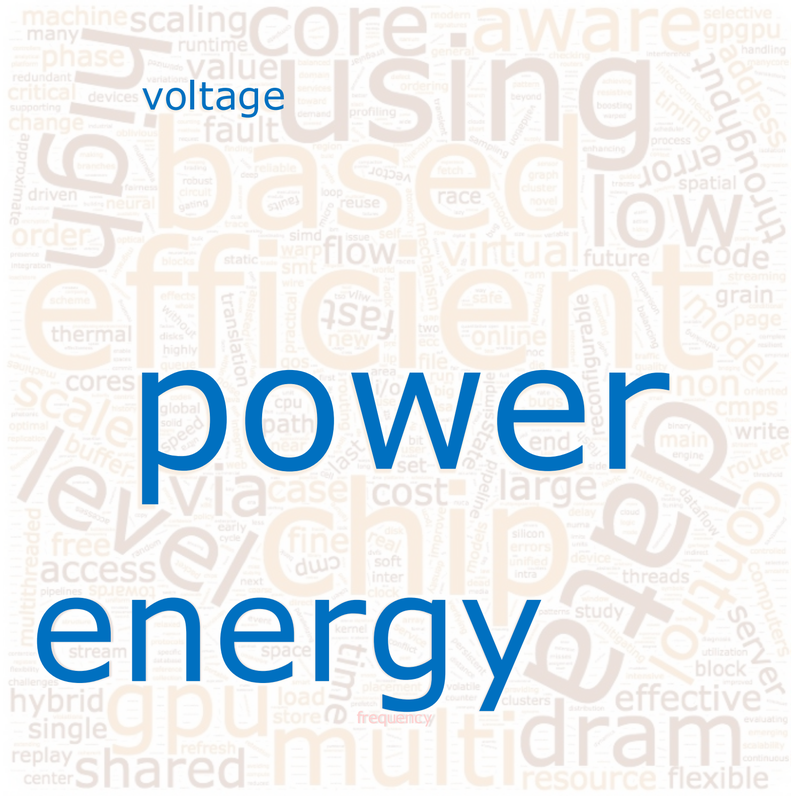
Source: Adapted from S. Borkar (Intel)

Today's systems cannot exist without **Energy Management**

Industry



Academia



Source: Word-cloud from ISCA, ASPLOS, MICRO, HPCA (2000 - 2016)

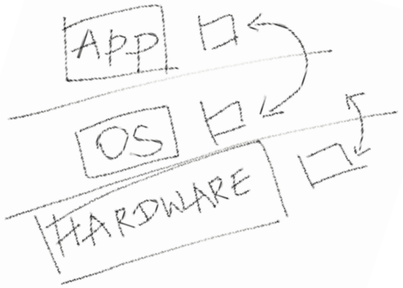
Pervasive

Essential

stay secure with

Today's systems cannot ~~exist without~~

Energy Management

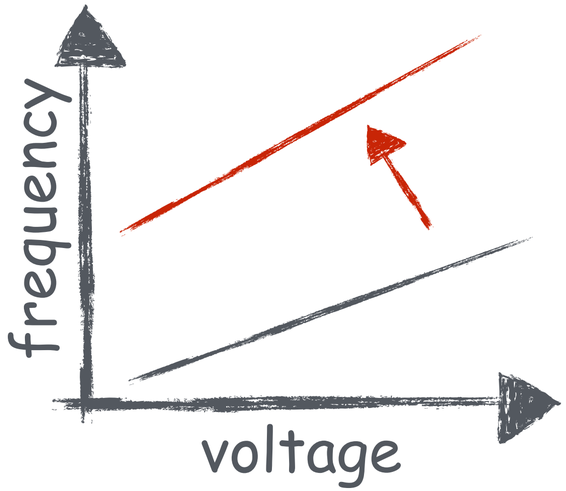


Complicated

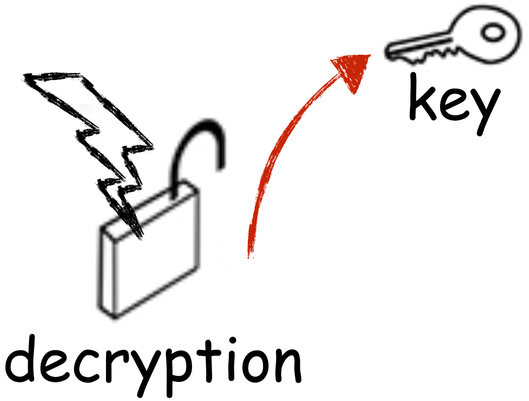
Exploiting software interfaces to **Energy Management**

Software-based attacker

Stretch operational limits



Induce faults



Exploiting software interfaces to **Energy Management**

Software-based attacker

Stretch

Traditional fault attacks

~~Need physical proximity~~

~~Need separate equipment~~

~~Soldering, crocodile clips, wire, etc~~

Induce faults

CLKSCREW: Exposing the perils of security-oblivious **Energy Management**

New attack vector that exploits energy management

Practical attack on trusted computing on ARM devices

Impacts hundreds of millions of deployed devices

Lessons for future energy management designs to be security-conscious

Outline

I. DVFS and Deep Dive into Hardware Regulators

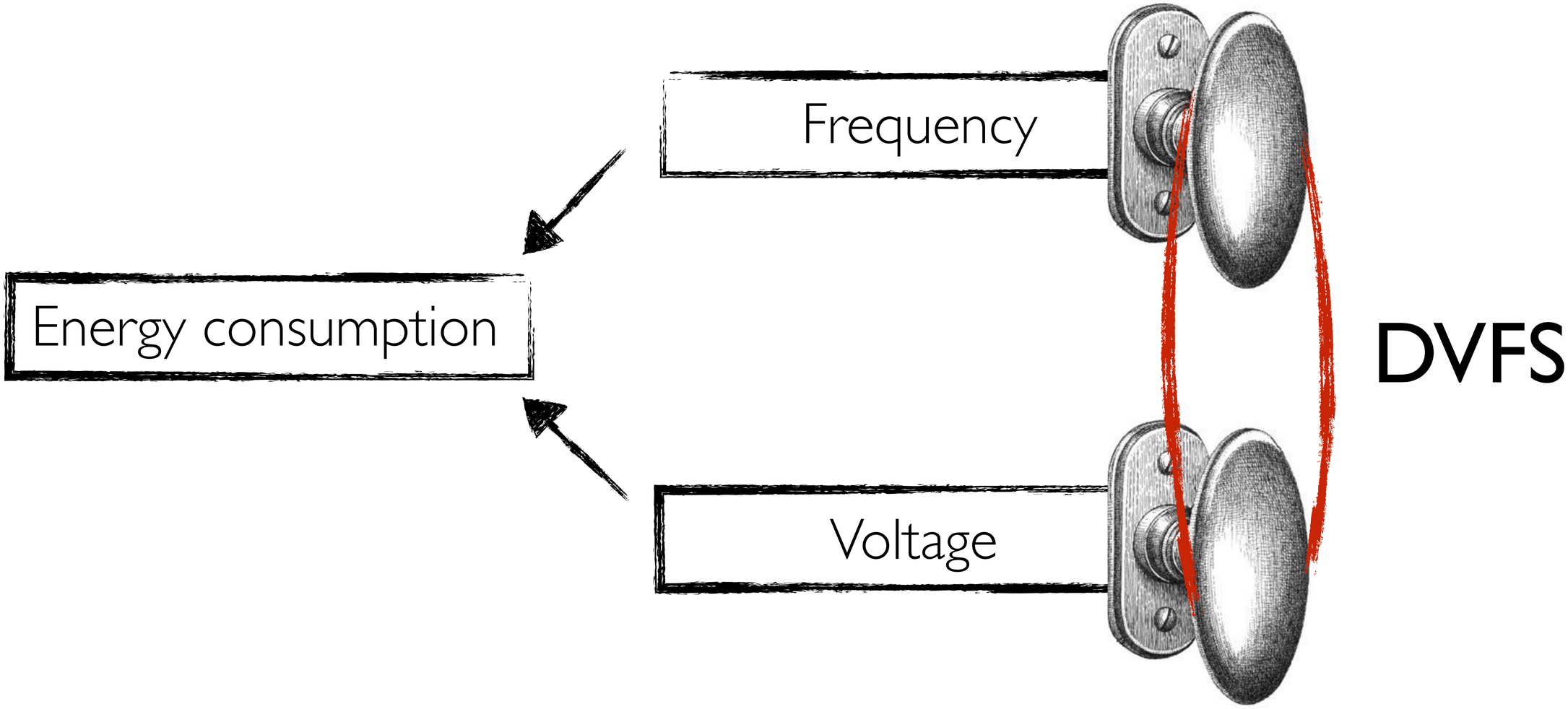
II. The CLKSCREW Attack

III. Trustzone Attack 1: Secret AES Key Inference

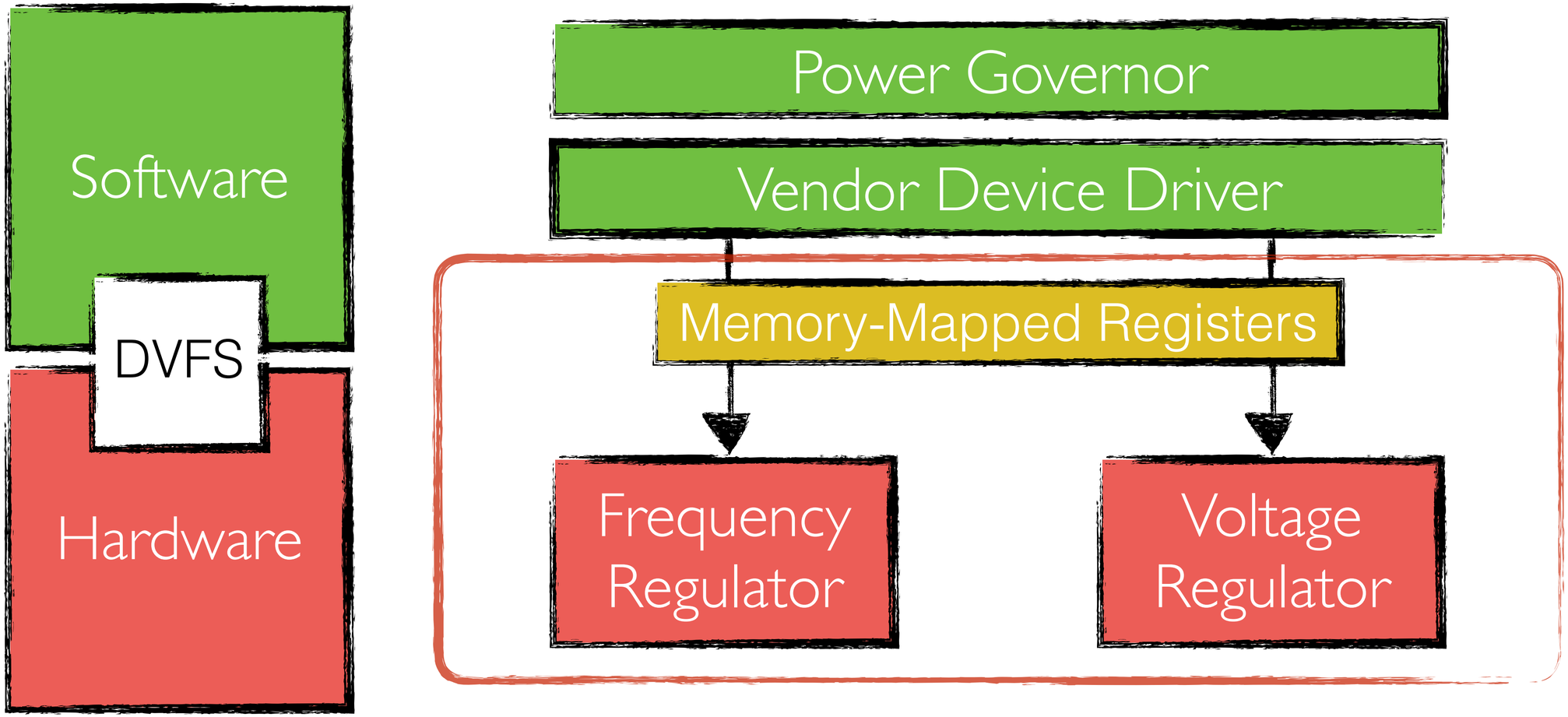
IV. Trustzone Attack 2: Tricking RSA Signature Validation

V. Concluding Remarks

Dynamic Voltage and Frequency Scaling (DVFS)



Hardware & Software Support for DVFS

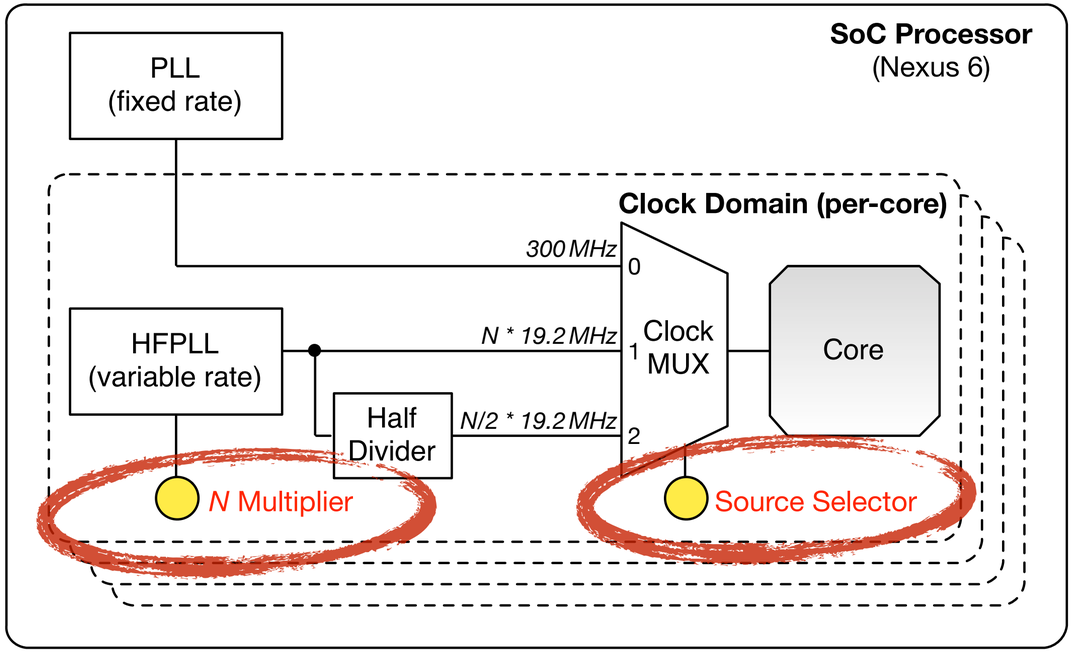


Our Target

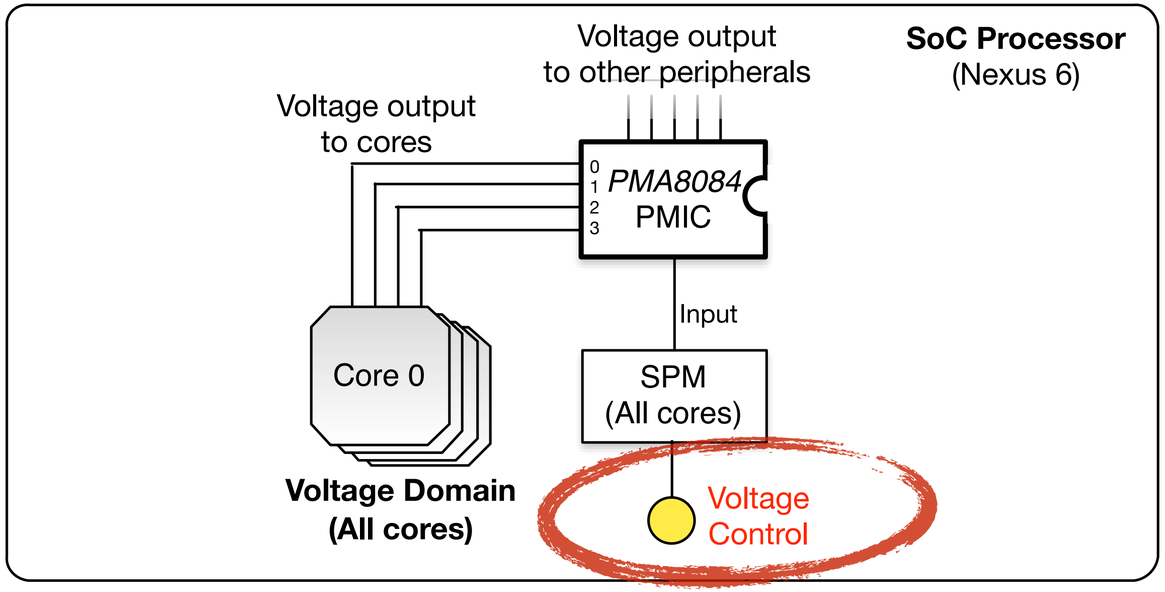
Nexus 6 - ARMv7
Quad-core 2.6GHz
Snapdragon Krait SoC

Nexus 6 - HW Regulators and SW Interfaces

Frequency regulators



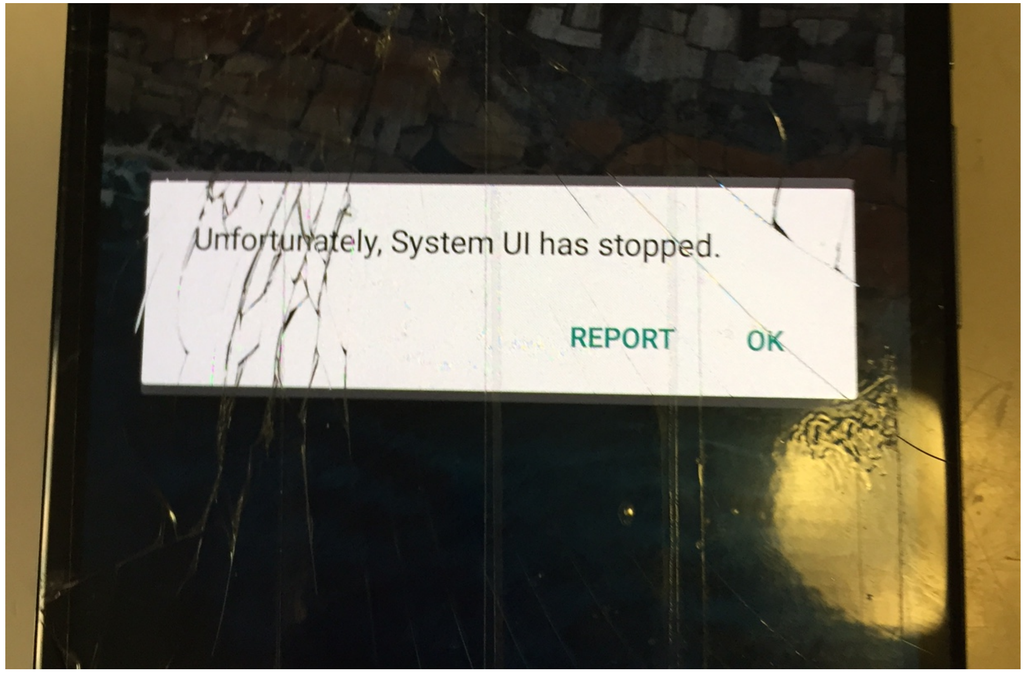
Voltage regulators



Operating frequency and voltage can be configured
via memory-mapped registers from software

https://github.com/0x0atang/clkscrew/blob/master/faultmin_SD805/glitch_sd805.c

First sign of trouble…



Curious crashes

```
root@shamu:/ # logcat | grep "fault"
    Fatal signal 11 (SIGSEGV), code 1, fault addr 0x0 in tid 5077 (uprimebenchmark)
    signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 0x0
    Fatal signal 11 (SIGSEGV), code 2, fault addr 0xa0950e08 in tid 5422 (RenderThread)
    signal 11 (SIGSEGV), code 2 (SEGV_ACCERR), fault addr 0xa0950e08
    Fatal signal 4 (SIGILL), code 1, fault addr 0xa93f5d40 in tid 4953 (RenderThread)
    signal 4 (SIGILL), code 1 (ILL_ILLOPC), fault addr 0xa93f5d40
    signal 6 (SIGABRT), code -6 (SI_TKILL), fault addr --------
```

Thinking out of the box…

Temperature matters

Do hardware regulators impose limits
to frequency/voltage changes?

Freq / Voltage Operating Point Pairs (OPPs)

- ✓ Unintended computing behaviors
- ✓ Software-controlled frequency and voltage settings
- ✓ Verify frequency and voltage settings are indeed properly configured

Frequency: cat /d/clk/krait0_clk/measure

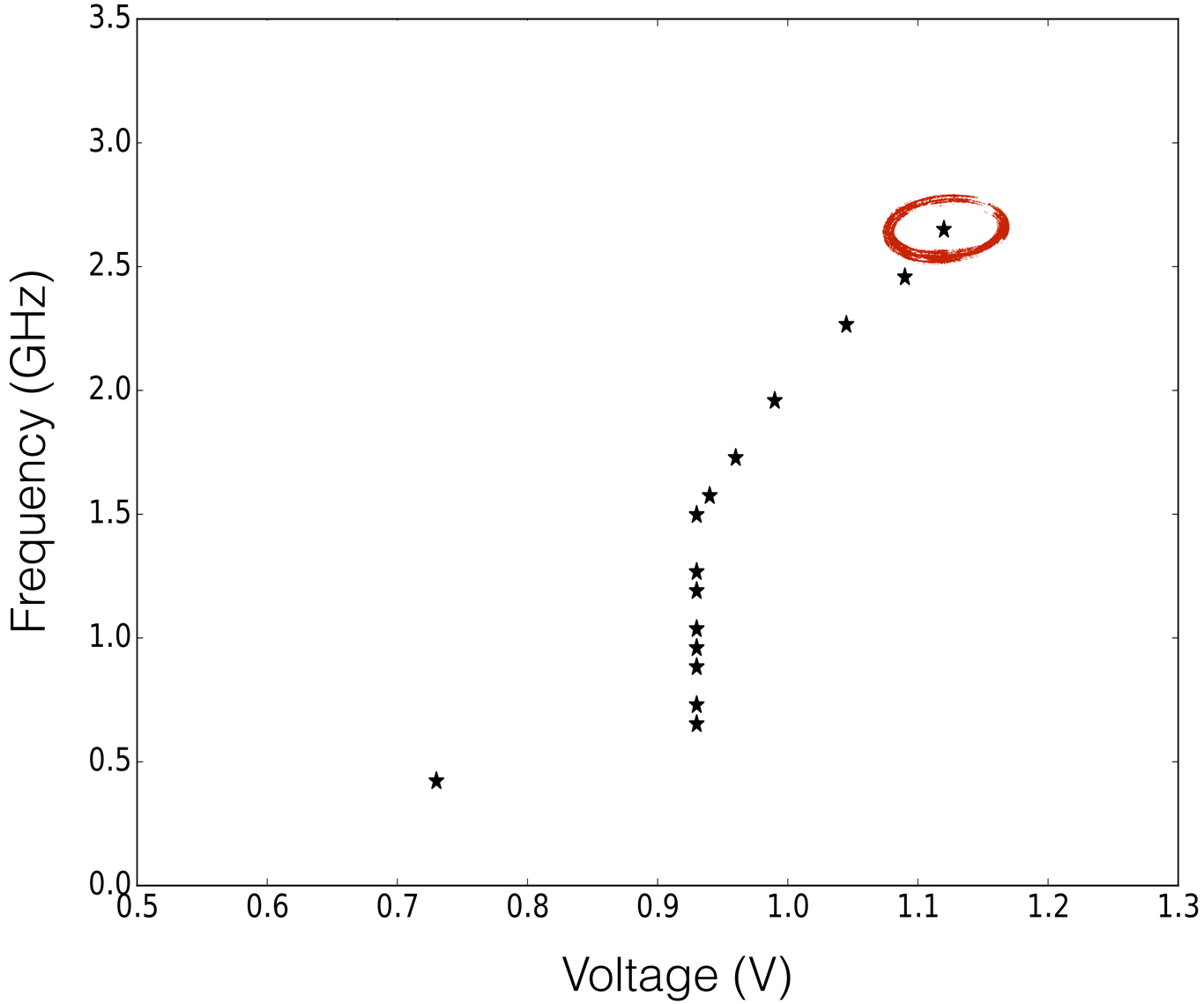
Voltage: cat /d/regulator/krait0/voltage

Freq / Voltage Operating Point Pairs (OPPs)

Nexus 6

★★★★☆ 791 2

- Android v5(Lollip
- Turbo Charging
- 5.94 inch QHD A
- 2.7 GHz Process



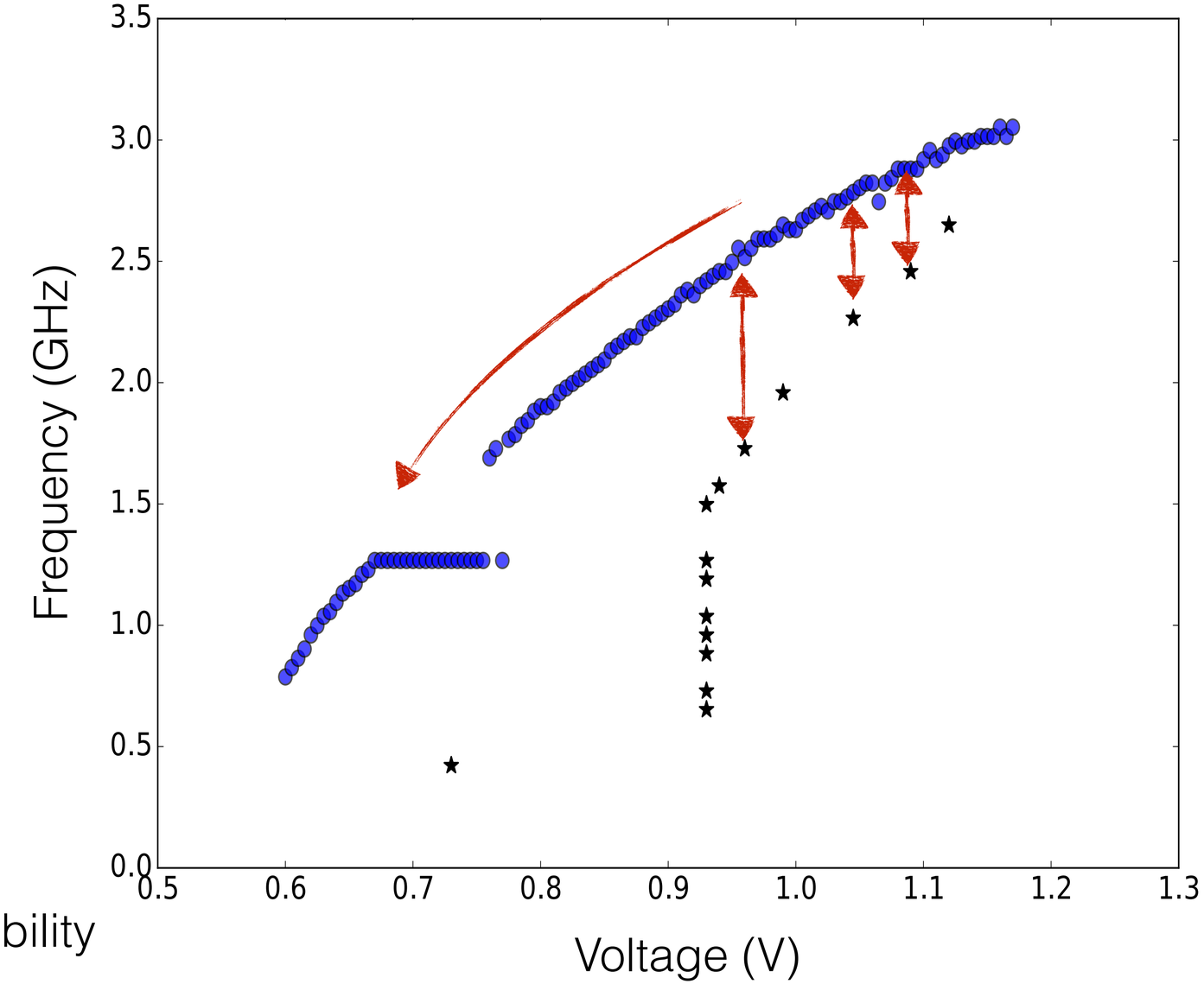
Legend:

★★★ Vendor-recommended

Frequency / Voltage Operating Point Pairs (OPPs)

No safeguard hardware limits

Lower voltage ⟶
Lower minimum required
frequency to induce instability



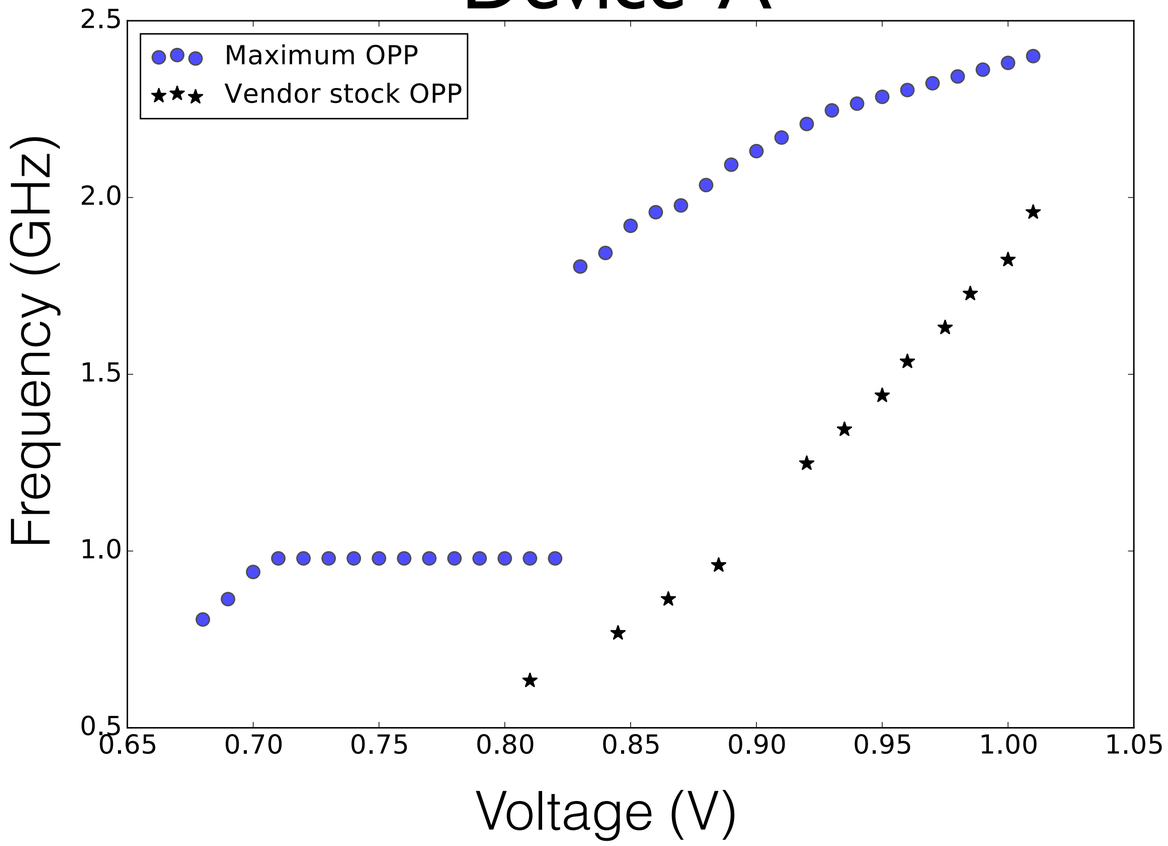
Legend:

★★★ Vendor-recommended

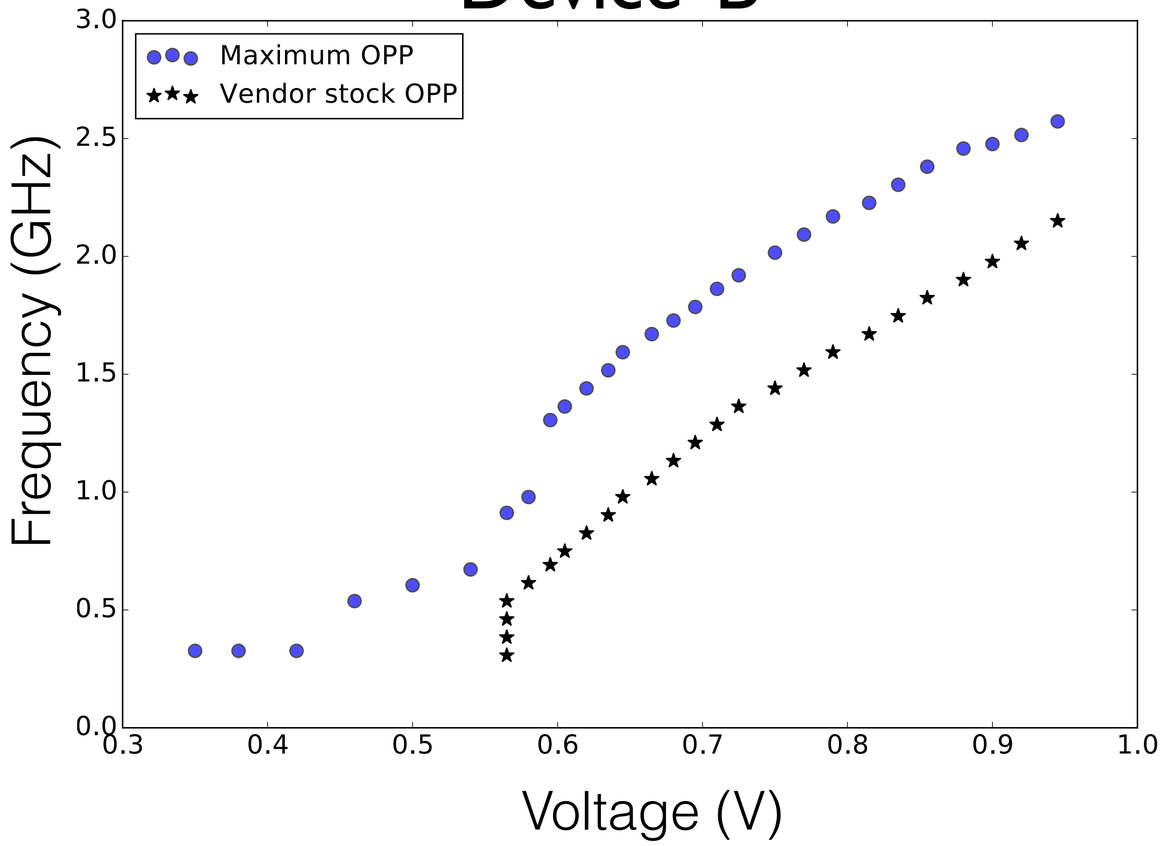
●●● Max OPP reached before instability

Freq / Voltage Operating Point Pairs (OPPs)

Device ‘A’



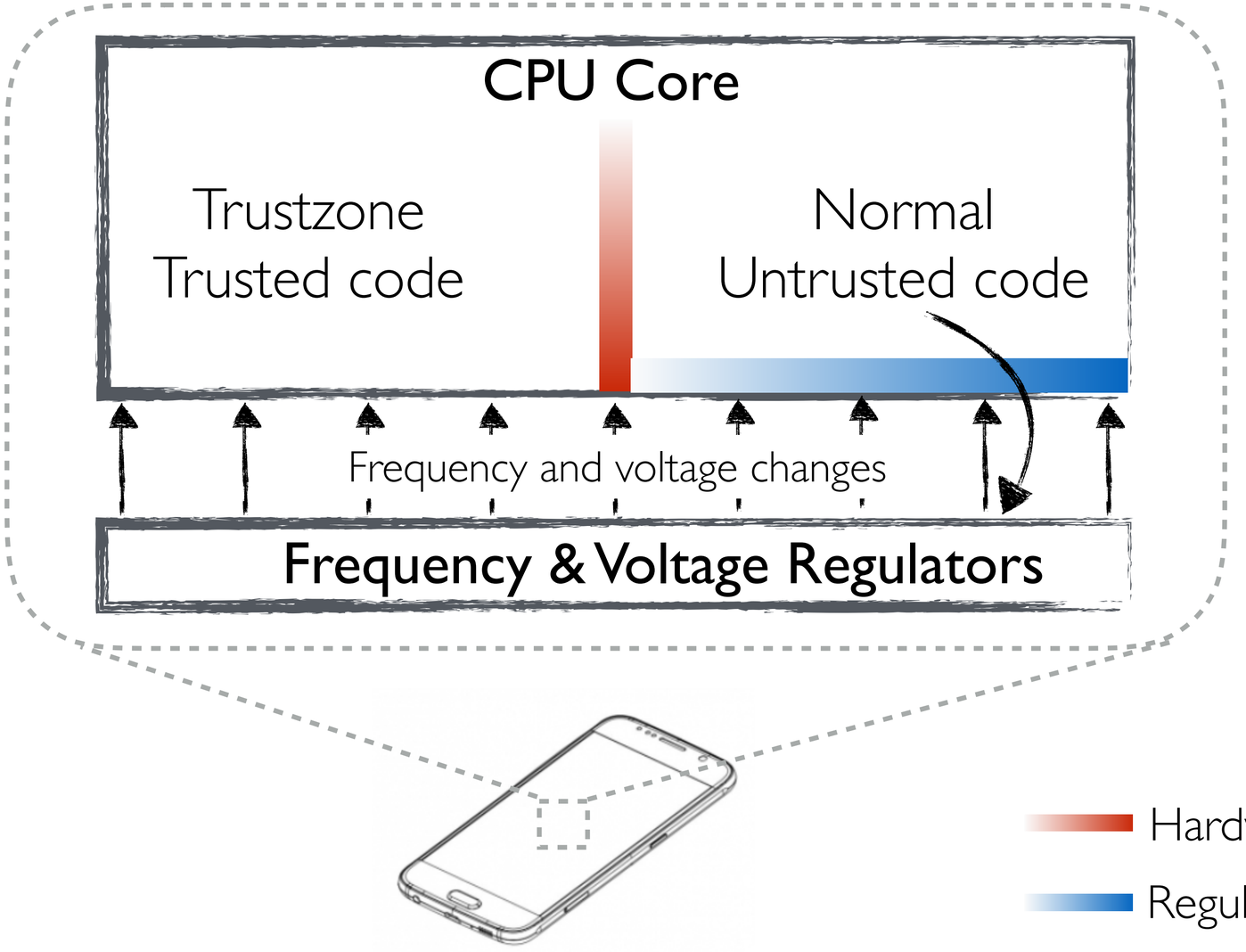
Device ‘B’



Do regulators operate across security boundaries?

Trusted Execution Environments (TEE)

Is DVFS Trustzone-Aware? No!



Outline

I. DVFS and Deep Dive into Hardware Regulators

II. The CLKSCREW Attack

III. Trustzone Attack 1: Secret AES Key Inference

IV. Trustzone Attack 2: Tricking RSA Signature Validation

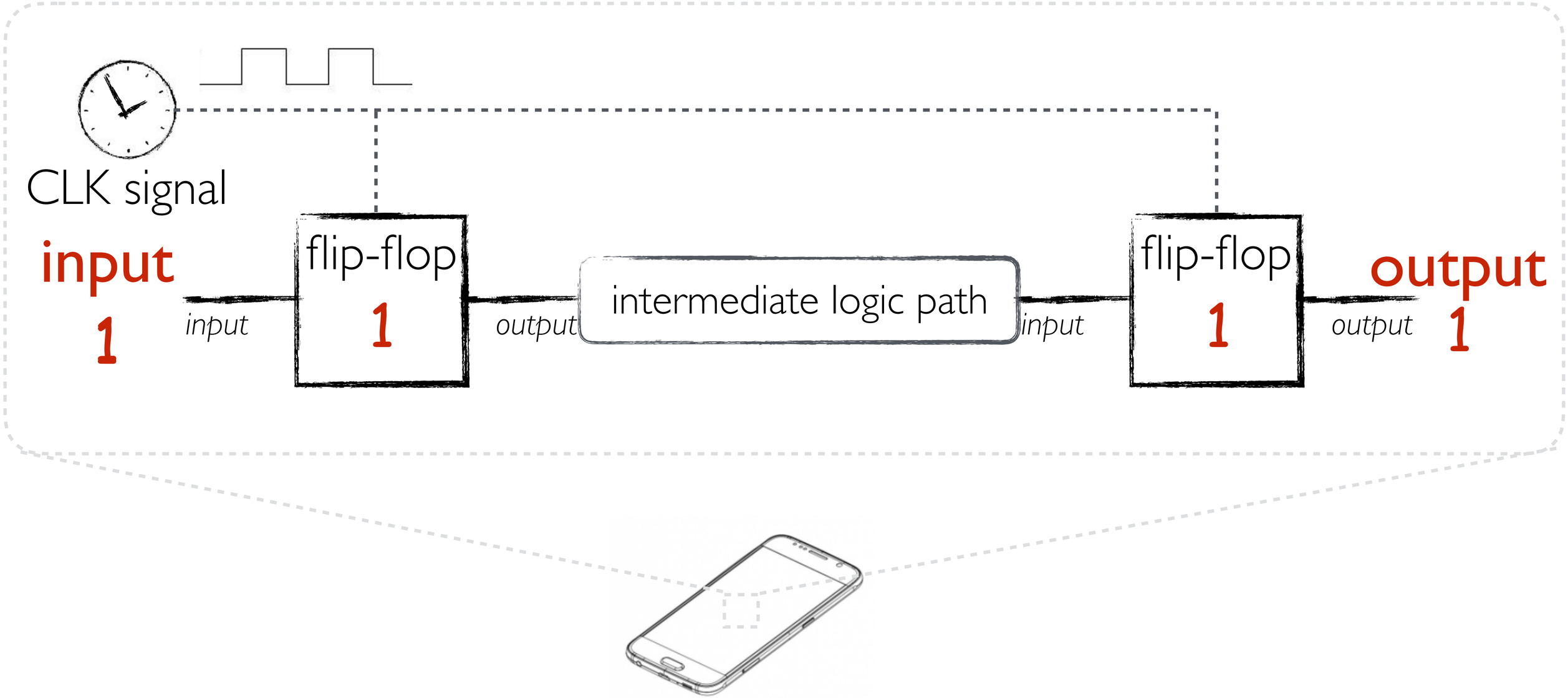
V. Concluding Remarks

Can we attack Trustzone code execution
using software-only control of the regulators?

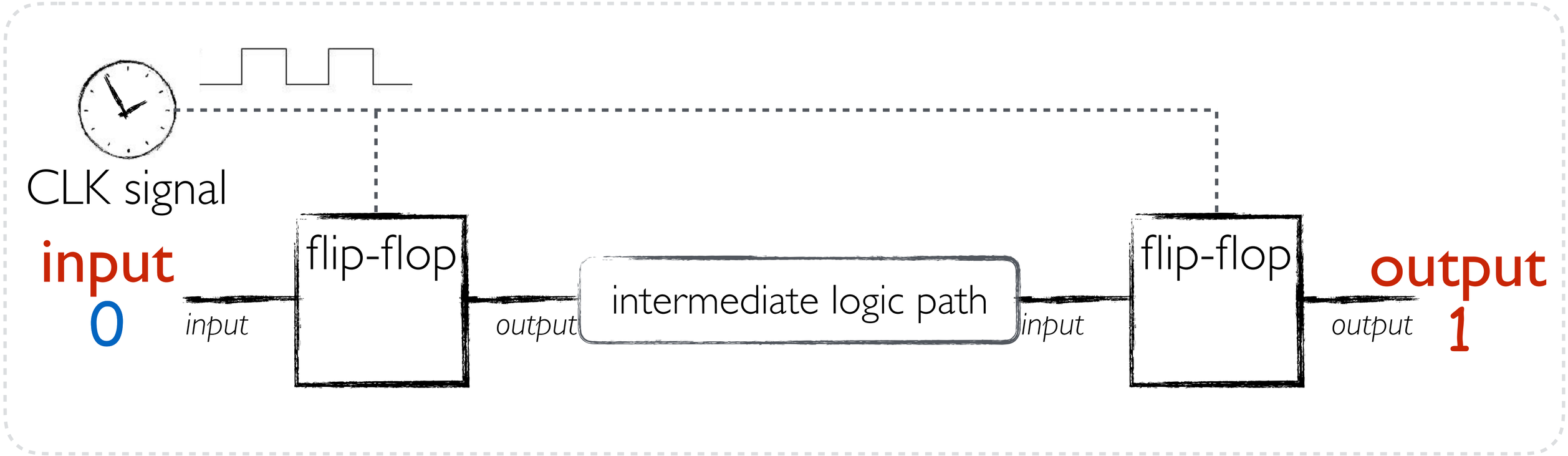
Induce timing faults

confidentiality
integrity
~~availability~~

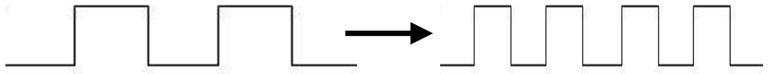
How do faults occur (due to over-raising frequency)?



How do faults occur (due to over-raising frequency)?



higher frequency



$\Longrightarrow$

less time for data to propagate

$\Longrightarrow$

timing violation

'0' → '1'

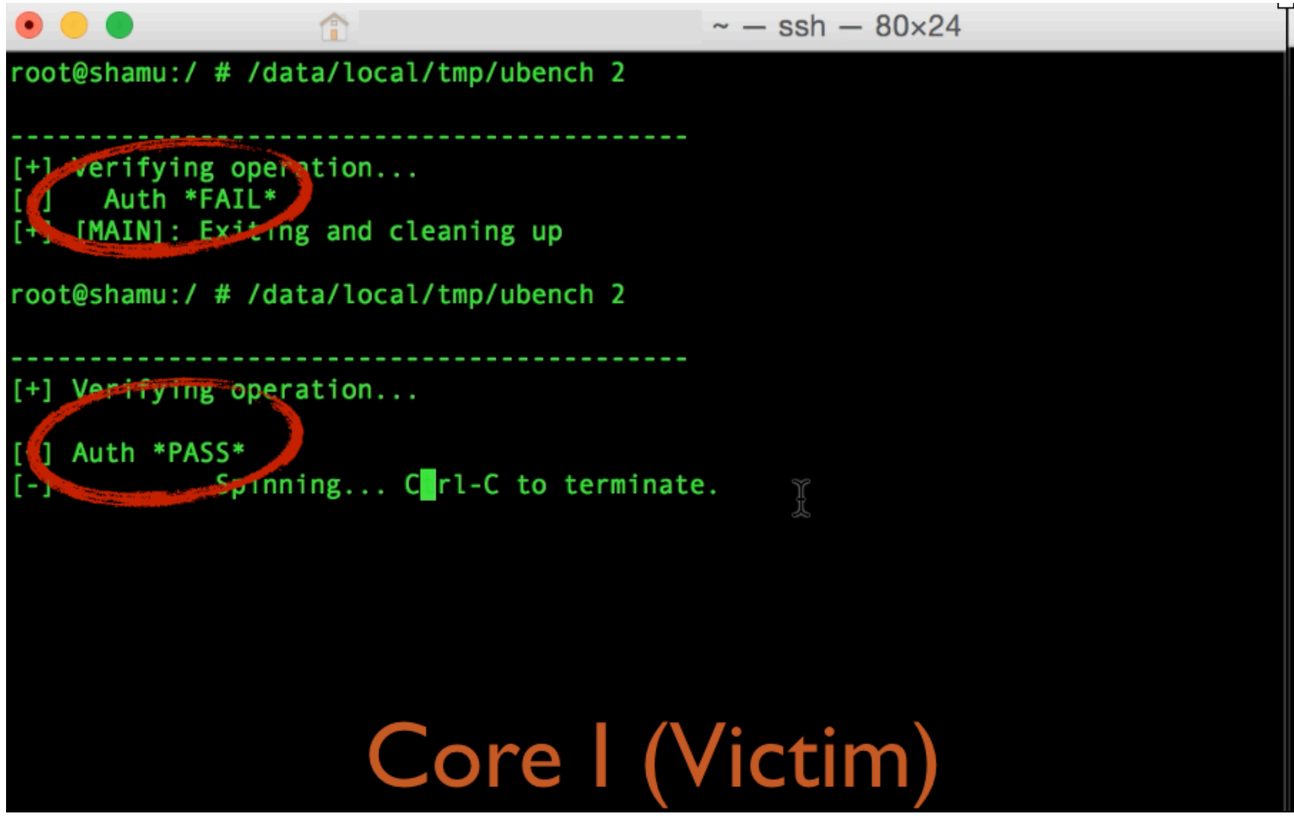
How dangerous are faults
induced by software-based overclocking/undervolting?

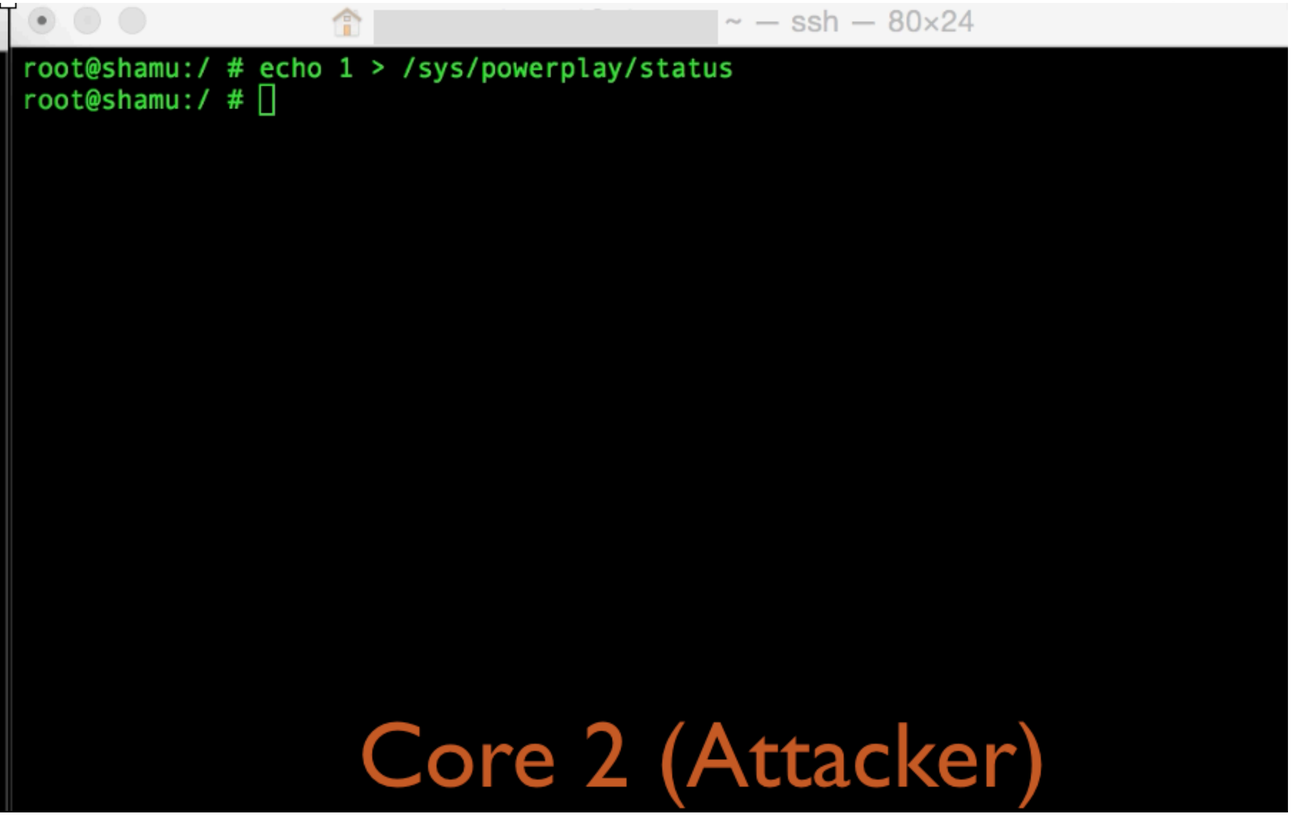
Faults induced by software-based overclocking

Influence **control** flow

Expected: Authentication fails

Runtime Fault Attack: Induce authentication to pass



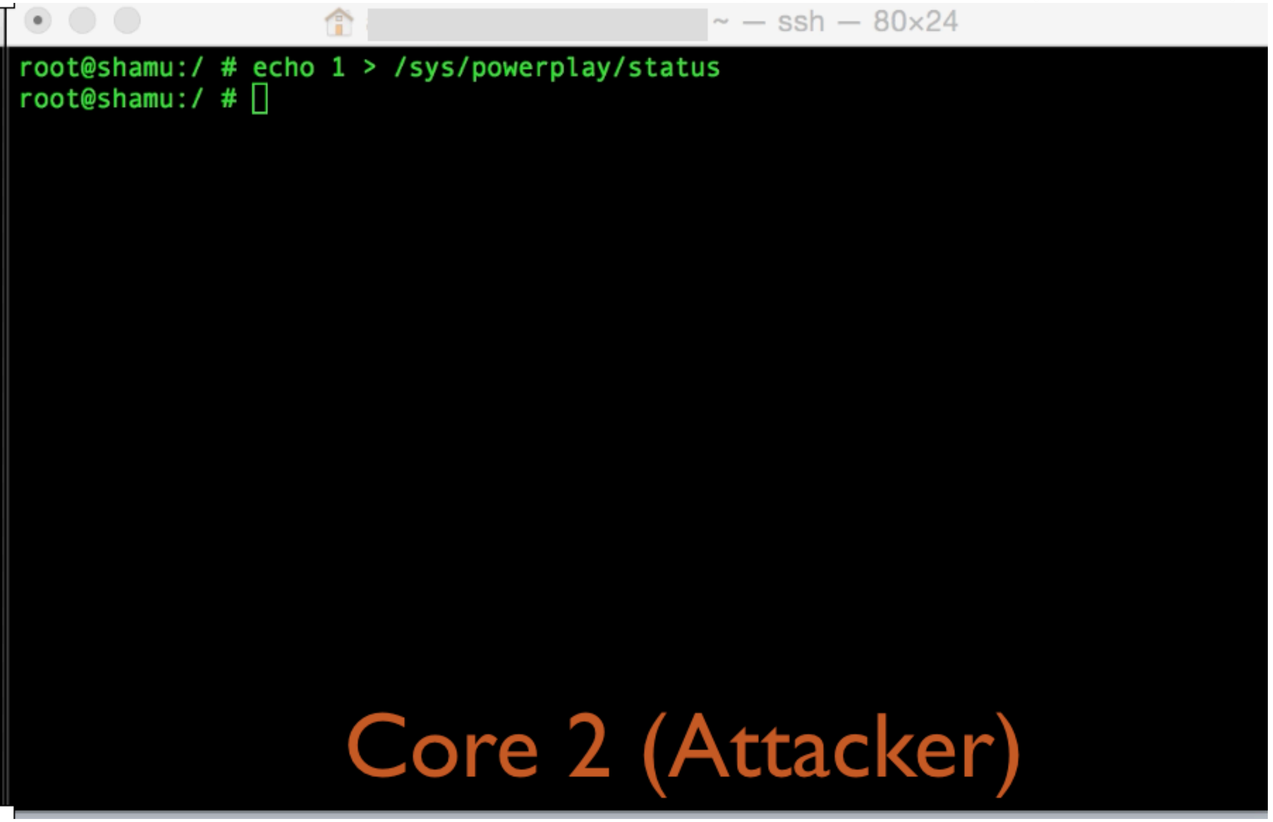


Faults induced by software-based overclocking

Influence **data** flow

Expected: Computation should return (0, 1, 2)

Runtime Fault Attack: Corrupt result to (nan, 1, 2)



CLKSCREW Challenges & Solutions

#1: Regulator operating limits

#2: Self-containment within same device

#3: Noisy complex OS environment

#4: Precise timing

#5: Fine-grained timing resolution

CLKSCREW Challenges & Solutions

#1: Regulator operating limits

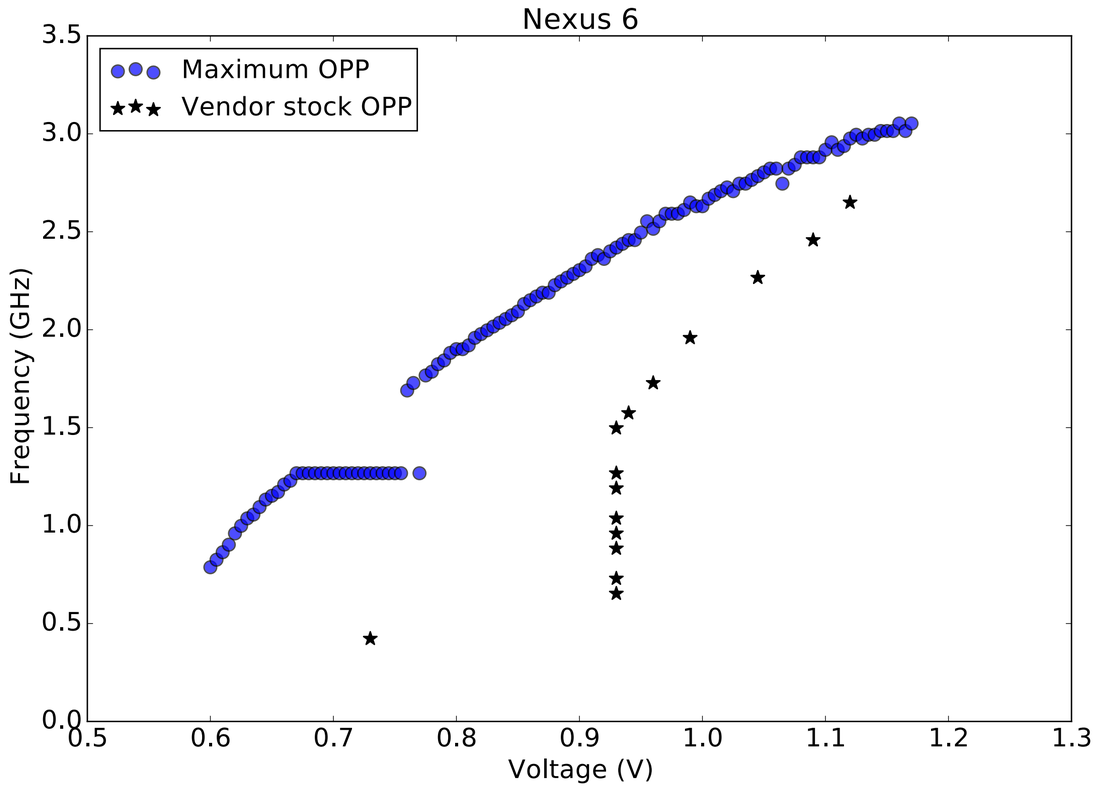
#2: Self-containment within same device

#3: Noisy complex OS environment

#4: Precise timing

#5: Fine-grained timing resolution

Addressed earlier in DVFS regulators



CLKSCREW Challenges & Solutions

#1: Regulator operating limits

#2: Self-containment within same device

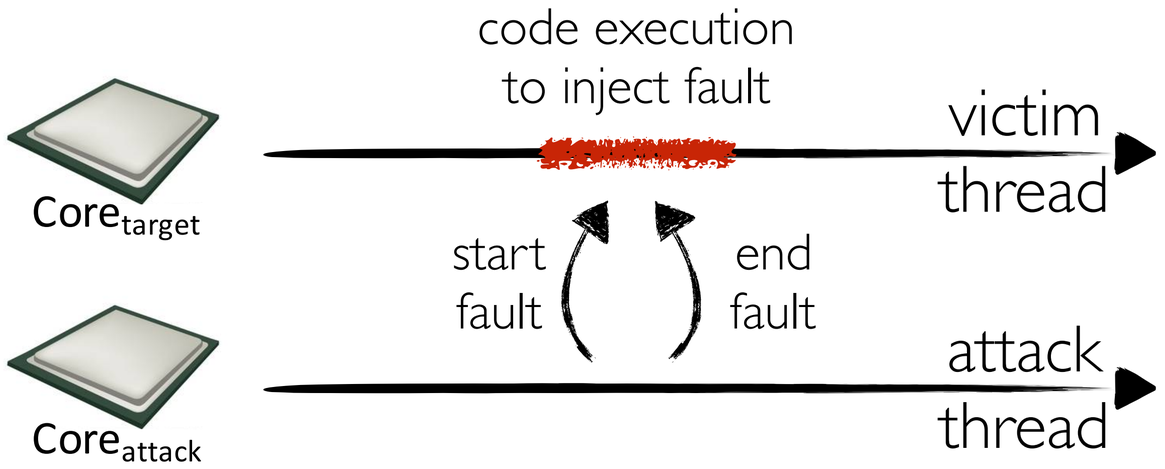
#3: Noisy complex OS environment

#4: Precise timing

#5: Fine-grained timing resolution

Cores have different frequency regulators

Core pinning



CLKSCREW Challenges & Solutions

#1: Regulator operating limits

#2: Self-containment within same device

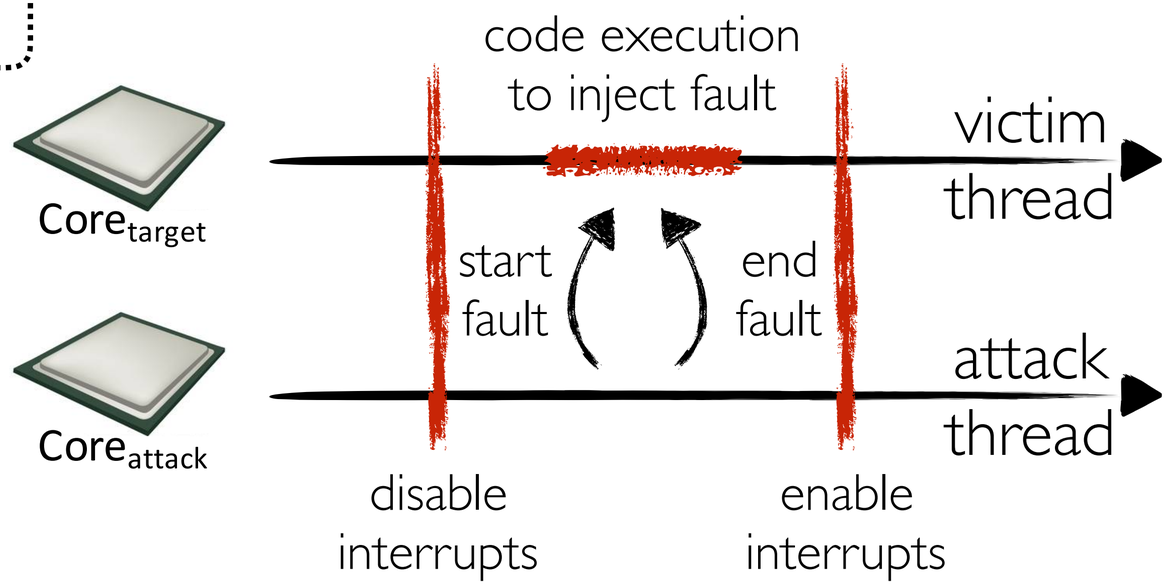
#3: Noisy complex OS environment

#4: Precise timing

#5: Fine-grained timing resolution

Core pinning

Disable interrupts during attack

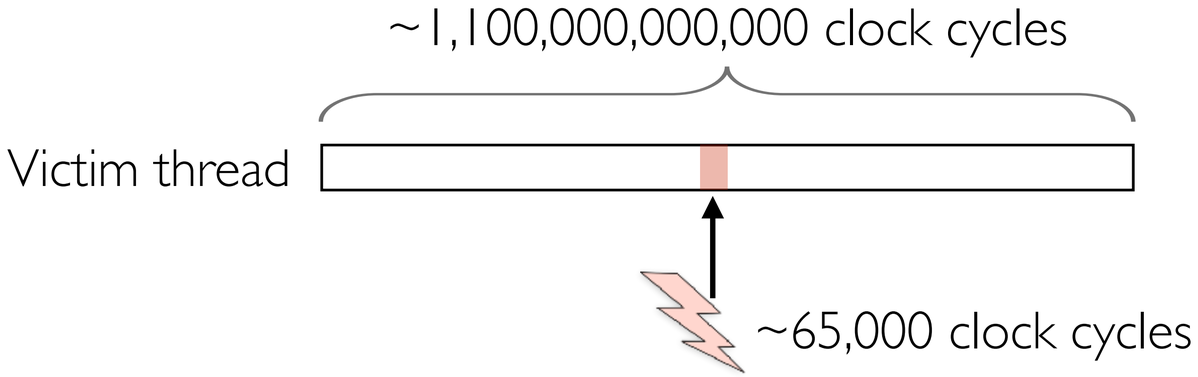


CLKSCREW Challenges & Solutions

#1: Regulator operating limits

#2: Self-containment within same device

#3: Noisy complex OS environment




```
asm volatile("1:  subs %0, %0, #1 \n"
             "    bhi 1b \n"::"r" (loops));
```

#4: Precise timing

#5: Fine-grained timing resolution

High-precision timing loops in attack architecture

Cache-based execution timing profiling

Outline

I. DVFS and Deep Dive into Hardware Regulators

II. The CLKSCREW Attack

III. Trustzone Attack 1: Secret AES Key Inference

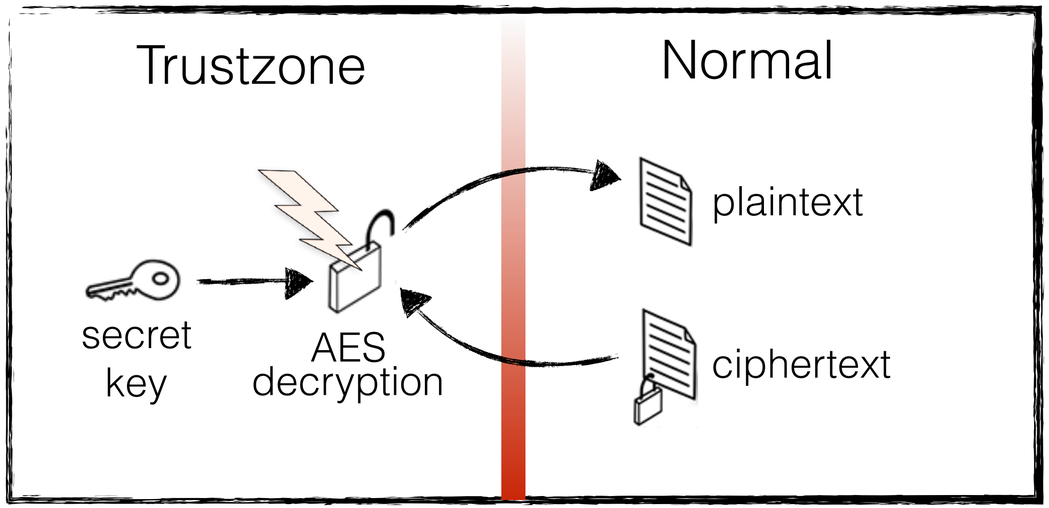
IV. Trustzone Attack 2: Tricking RSA Signature Validation

V. Concluding Remarks

Subverting Trustzone Isolation with CLKSCREW

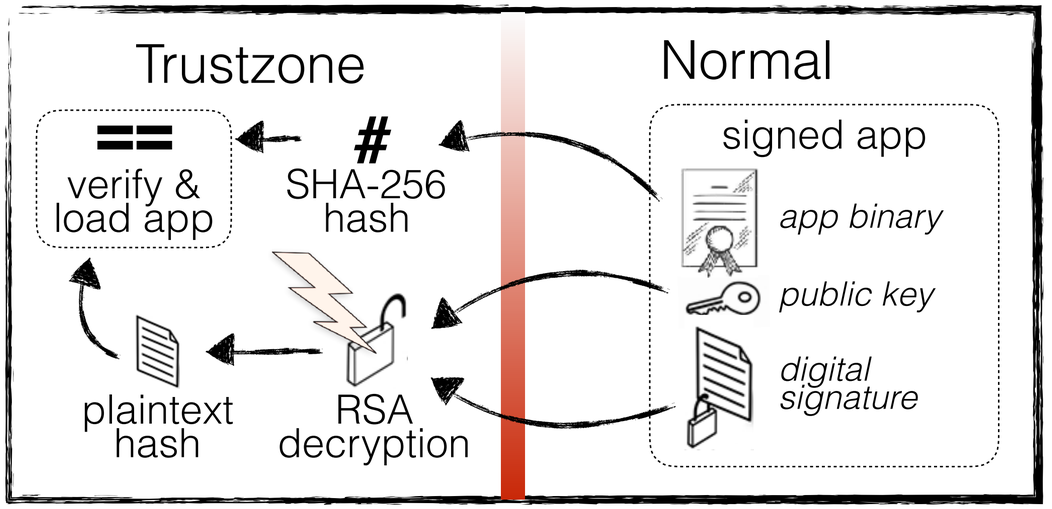
I. Confidentiality Attack

infer secret AES key stored within Trustzone



II. Integrity Attack

load self-signed app into Trustzone



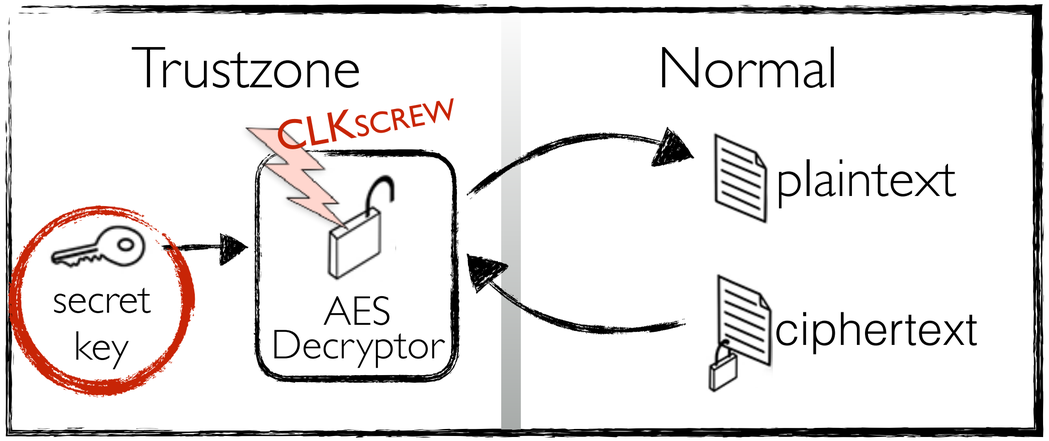
Key Inference Attack: Threat Model

Victim app: AES decryption app executing in Trustzone

Attacker's goal: Get secret AES key from outside Trustzone

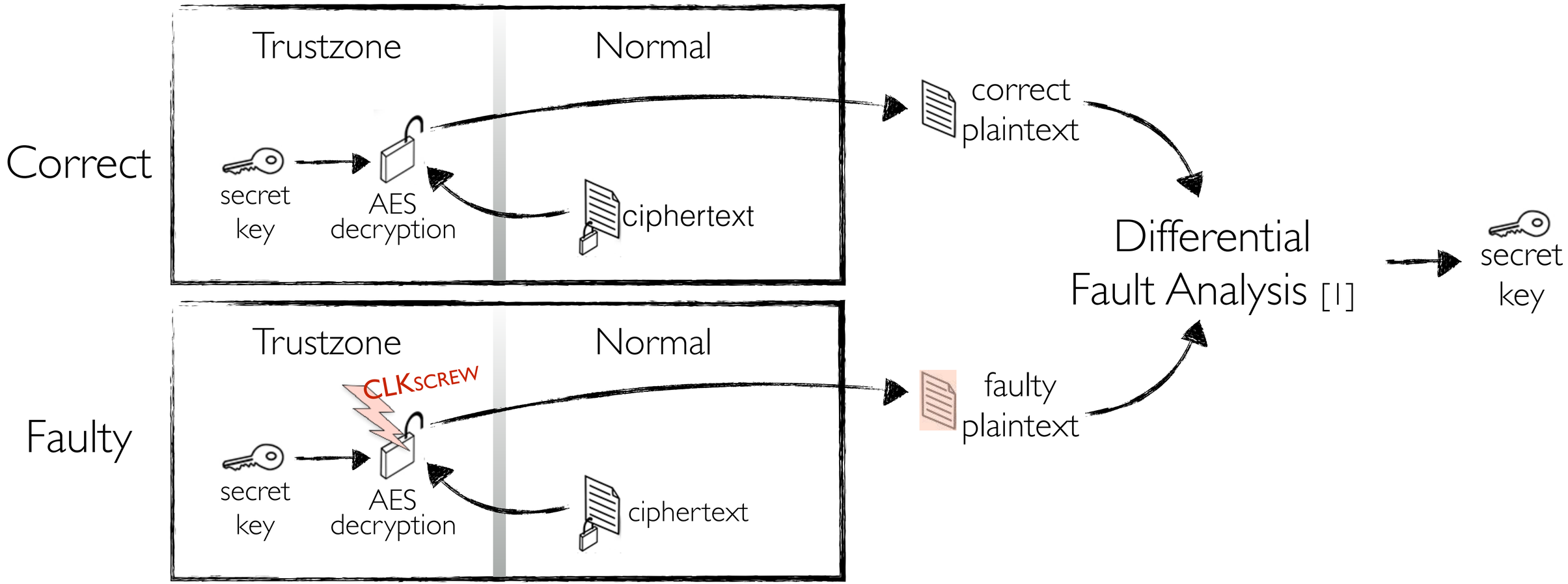
Attacker's capabilities: 1) Can repeatedly invoke the decryption app

2) Has software access to hardware regulators



Key Inference Attack: Summary

Idea: Induce a fault during the AES decryption
Infer key from a pair of correct and faulty plaintext



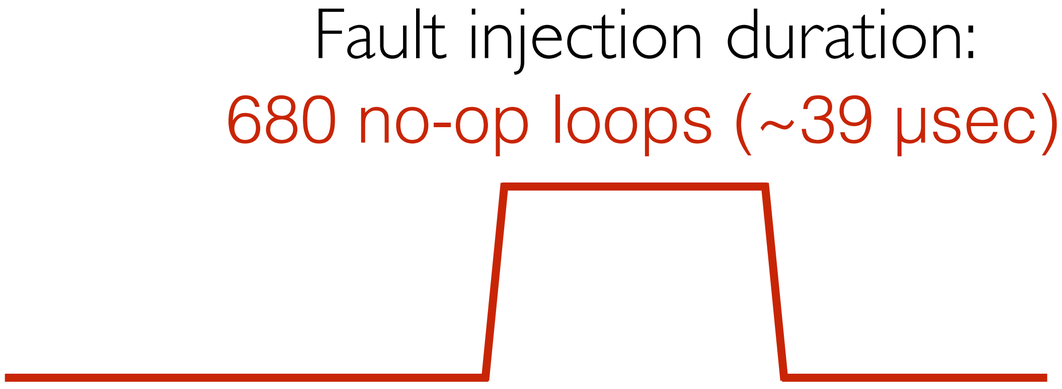
[1] Tunstall et al. Differential Fault Analysis of the Advanced Encryption Standard using a Single Fault. In IFIP International Workshop on Information Security Theory and Practices (2011).

Key Inference Attack: CLKSCREW Parameters

Base voltage: 1.055V

High frequency: 3.69GHz

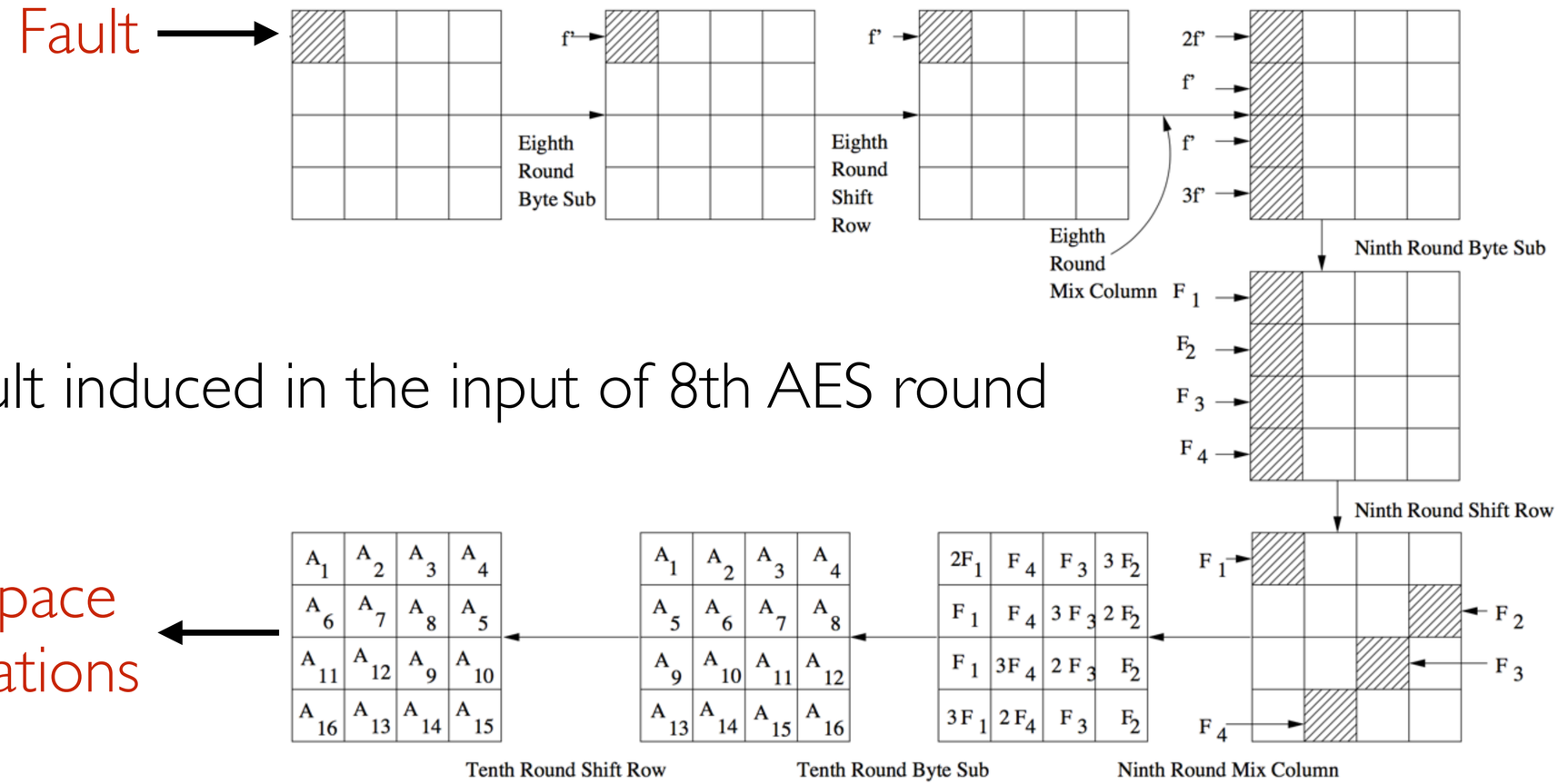
Low frequency: 2.61GHz



Differential Fault Analysis needs CLKSCREW to deliver a
one-byte fault to the 7th AES round

Key Inference Attack: Differential Fault Analysis

Check out code at: https://github.com/Daeinar/dfa-aes



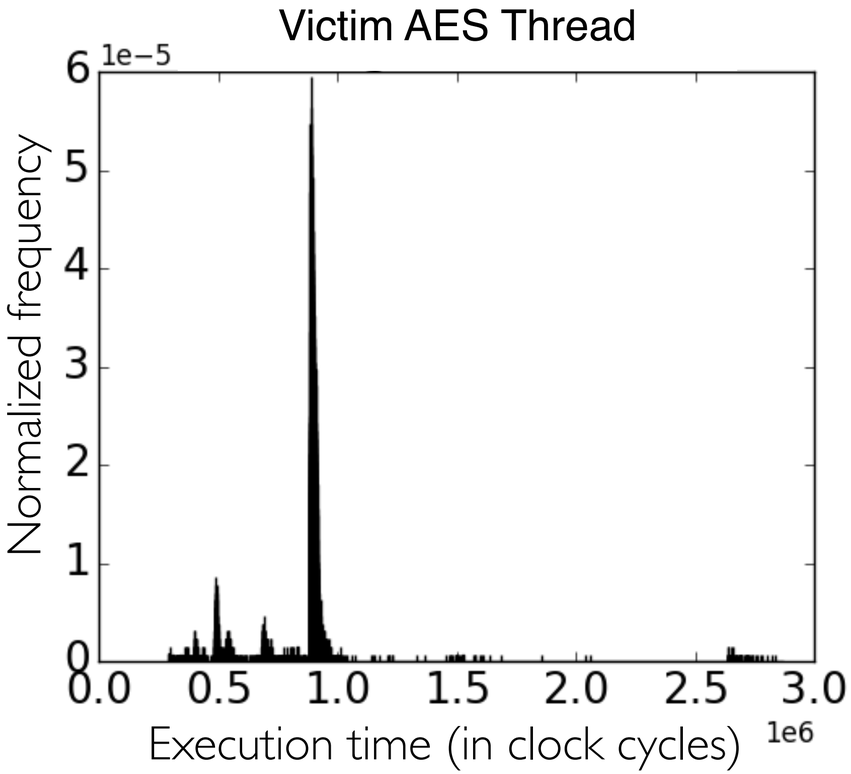
Credits: *Tunstall et al. Differential Fault Analysis of the Advanced Encryption Standard using a Single Fault. In IFIP International Workshop on Information Security Theory and Practices (2011).*

Key Inference Attack: Timing Profiling

Execution timing of Trustzone code can be profiled with hardware cycle counters that are accessible outside of Trustzone

Key Inference Attack: Timing Profiling

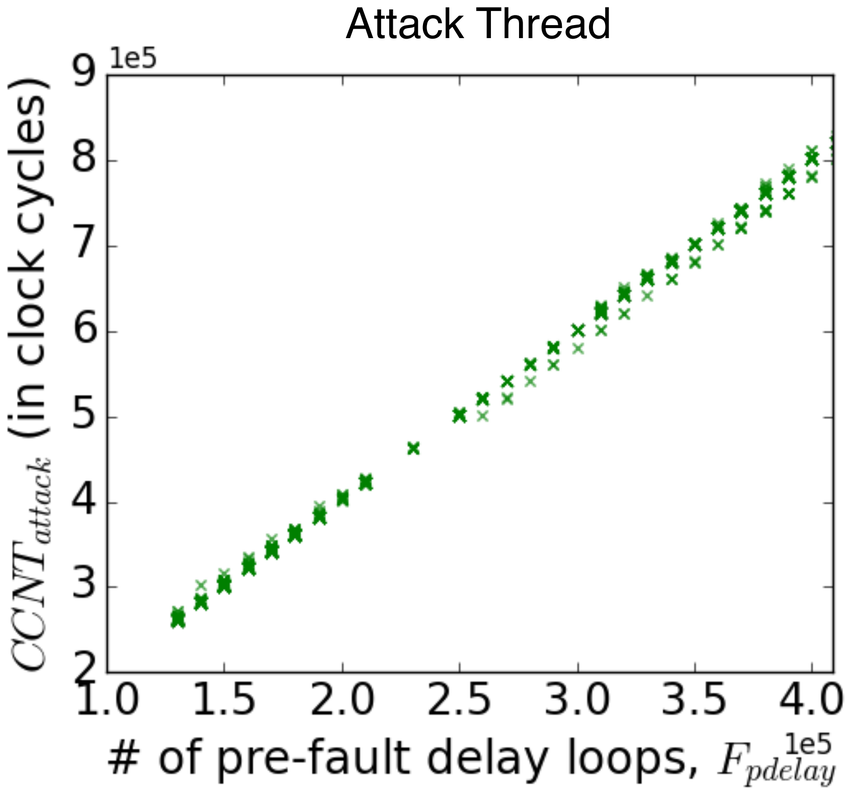
How varied is the execution timing of the victim decryption app?



Not too much variability in terms of execution time

Key Inference Attack: Timing Profiling

Can we effectively control the timing of the fault delivery with no-op loops?



Number of no-op loops is a good proxy to control timing of fault delivery

Key Inference Attack: Fault Model

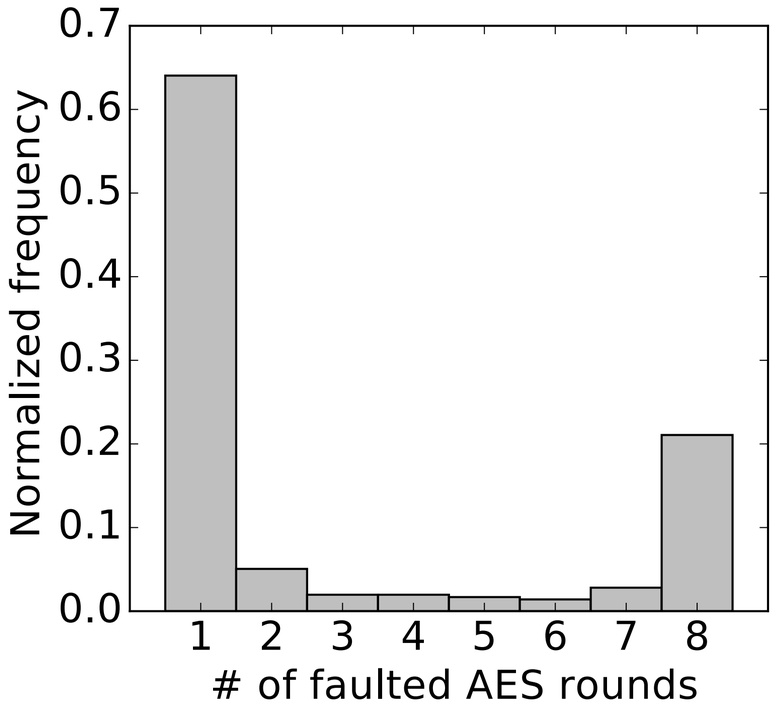
Our fault model requires our attack to inject fault

Exactly one AES round at the 7th round

Corruption of exactly one byte

Key Inference Attack: Fault Model

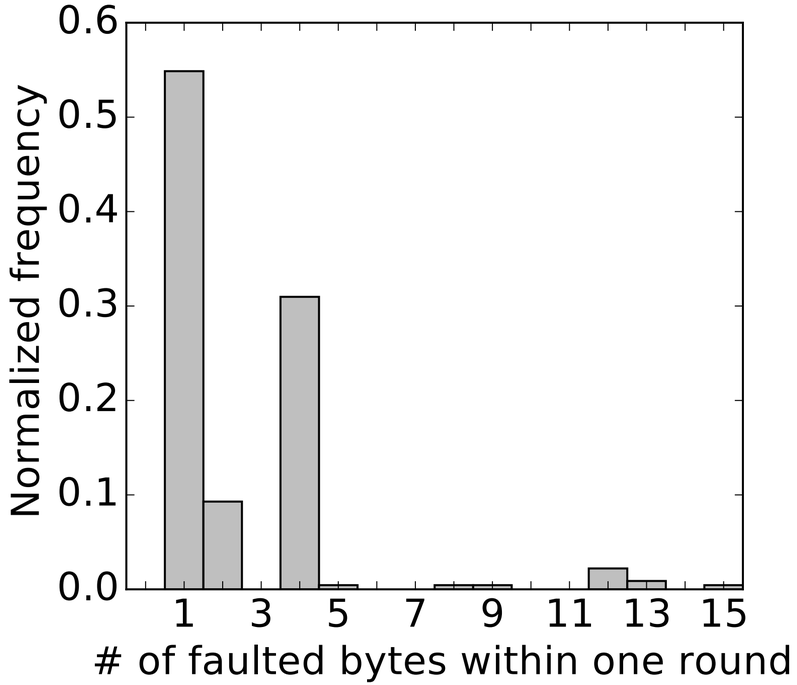
Precision: How likely can we inject fault in exactly one AES round?



More than 60% of the resulting faults are precise enough to corrupt exactly one AES round

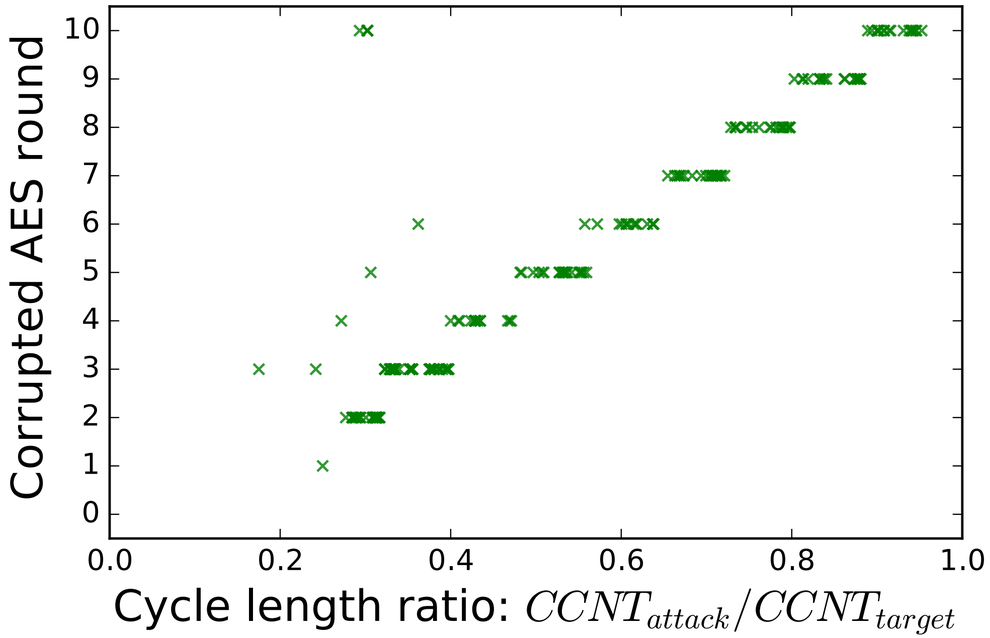
Key Inference Attack: Fault Model

Transience: How likely can we corrupt exactly one byte?



Out of the above faults that affect one AES round, more than half are transient enough to corrupt exactly one byte

Key Inference Attack: Results



Controlling F_{pdelay} allows us to precisely time the delivery of the fault to the targeted AES round

Statistics:

~20 faulting attempts to induce one-byte fault to desired AES round.

~12 min on a 2.7GHz quad core CPU to generate 3650 key hypotheses

Outline

I. DVFS and Deep Dive into Hardware Regulators

II. The CLKSCREW Attack

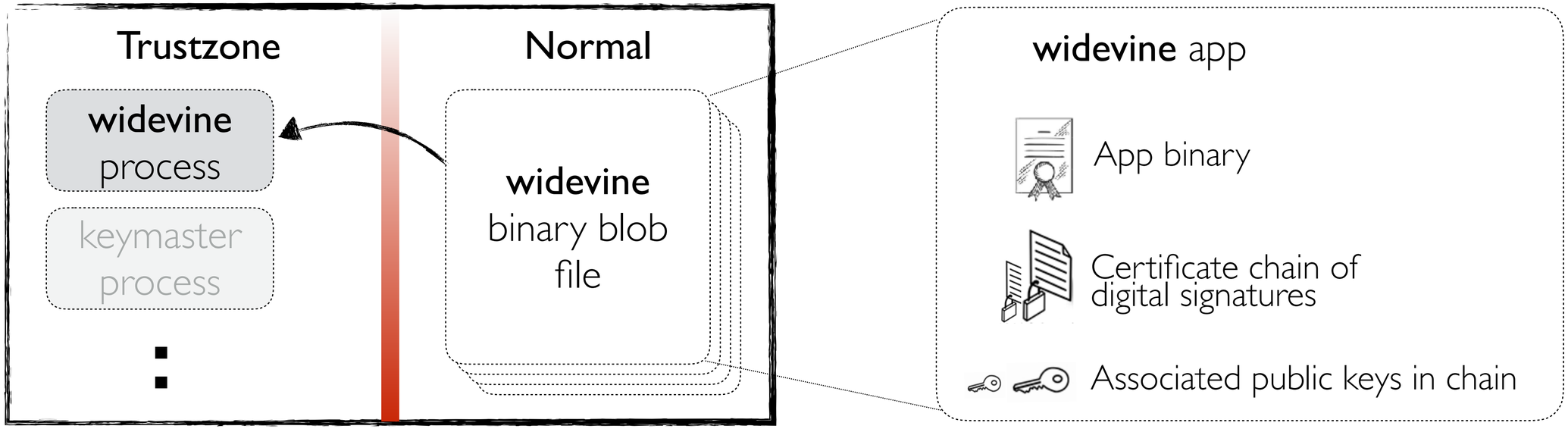
III. Trustzone Attack 1: Secret AES Key Inference

IV. Trustzone Attack 2: Tricking RSA Signature Validation

V. Concluding Remarks

Real-world Apps in Trustzone

- Apps running in Trustzone are building blocks for security
  - Eg: widevine (DRM), keymaster (hardware-backed key storage)
- Trustzone apps loaded from binary blob files at runtime
- Trustzone OS checks for a valid RSA signature before loading app



RSA Signature Attack: Threat Model

To attack: RSA signature chain verification routine in Trustzone

Attacker's goal: Trick routine into accepting a self-signed binary

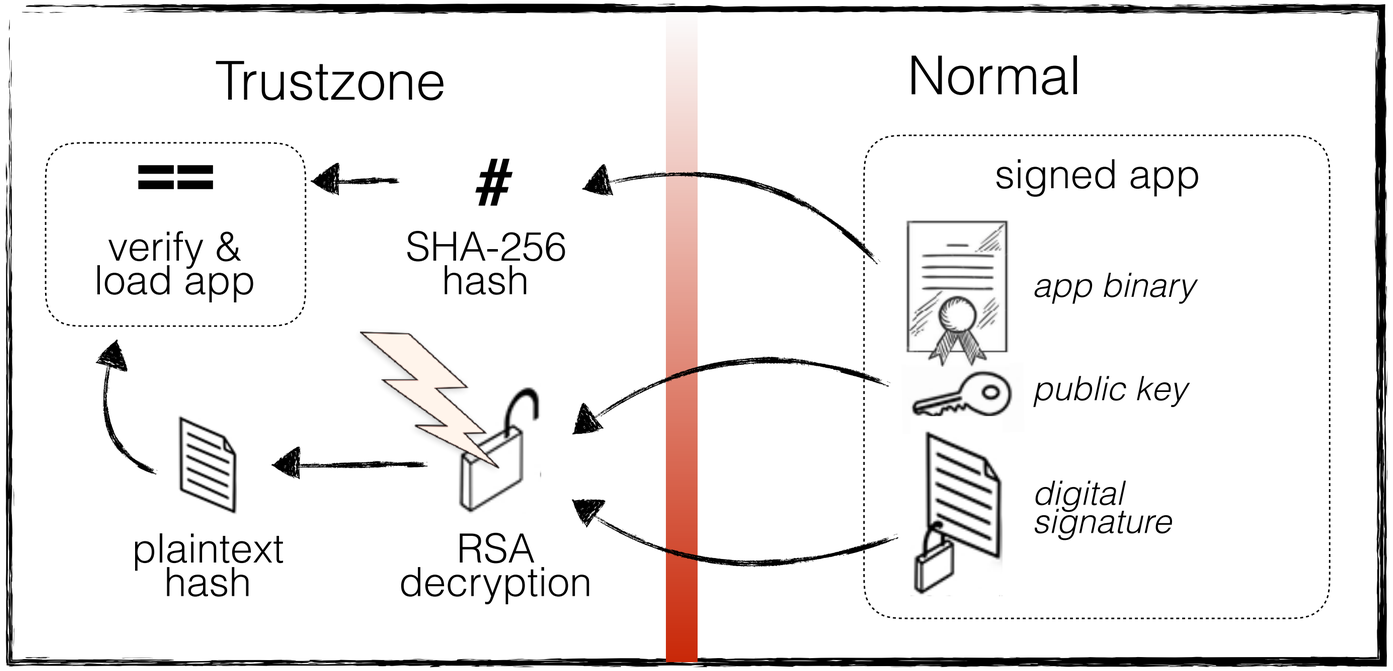
Attacker's capabilities: 1) Can repeatedly invoke Trustzone to load app

2) Has software access to hardware regulators

3) Know when the app has been loaded successfully

RSA Signature Attack: Summary

Idea:
- Self-sign an app binary and invoke the app loading
- Inject fault during signature verification
- Corrupt RSA modulus used at runtime



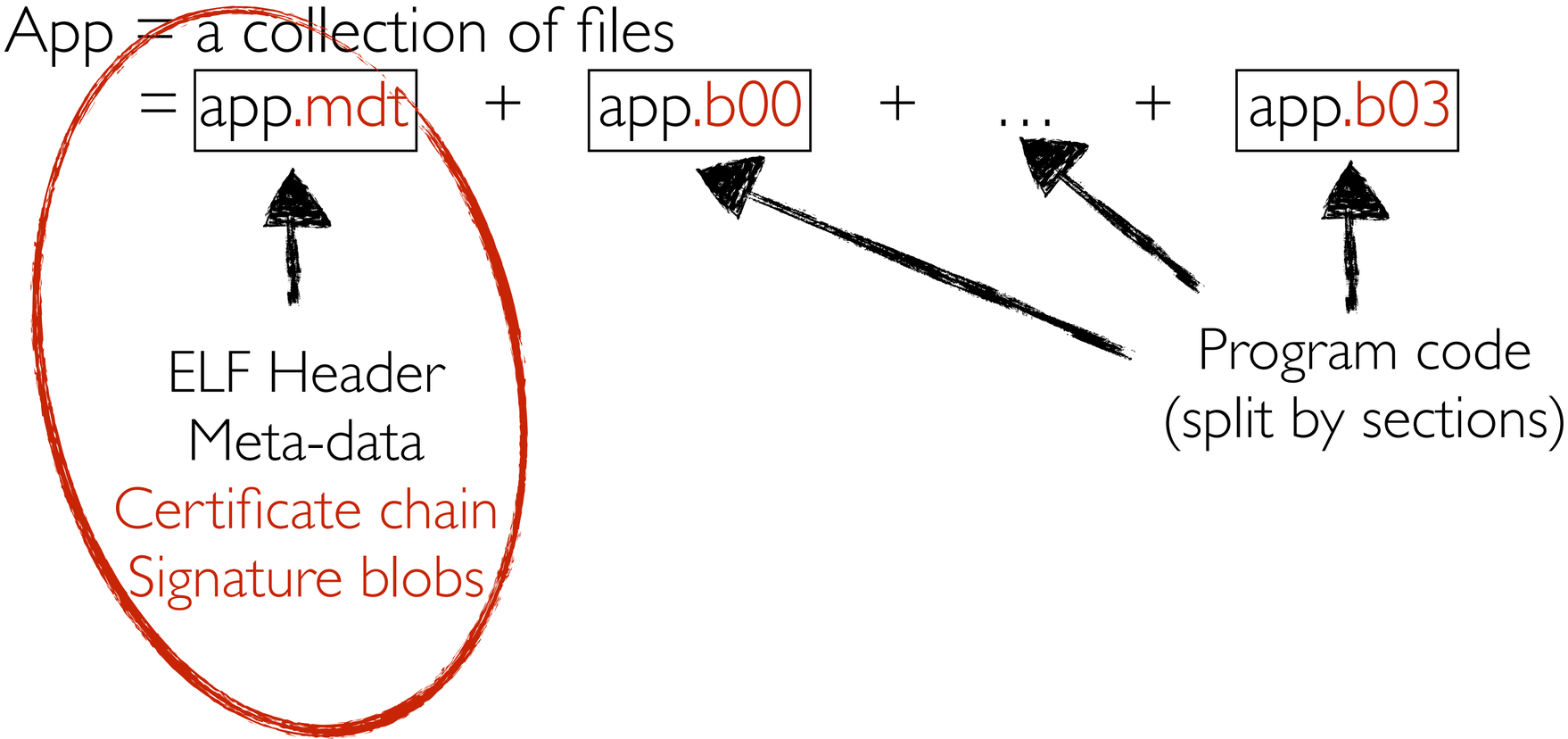
Attack Exploration and Formulation

Trustzone Apps: How to craft self-signed app binary files?

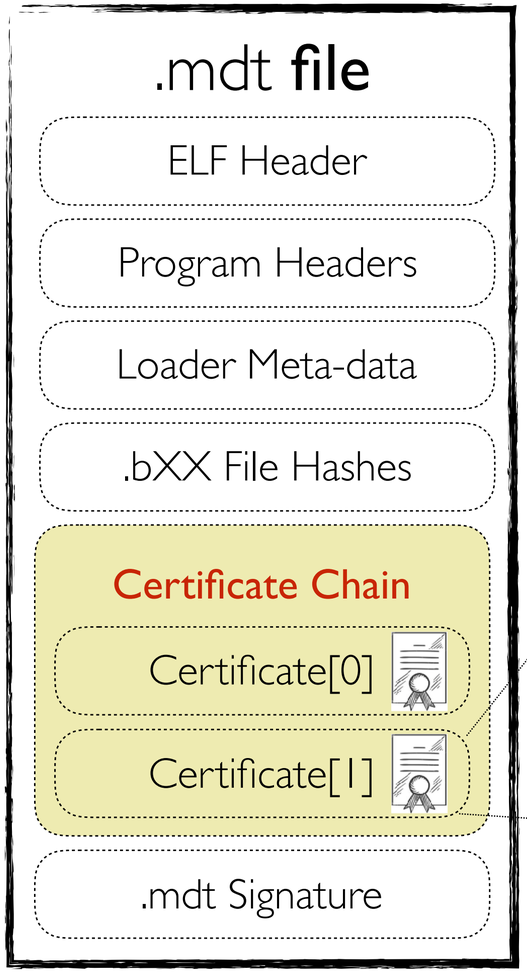
Trustzone OS: How, where and when to inject the CLKSCREW fault?

Trustzone Apps: Format of Binary Files

- Trustzone firmware updates contain the Trustzone app binary files
- App = a collection of files



Trustzone Apps: Format of Binary Files



Certificate stored in proprietary Motorola HAB ("High Assurance Boot") binary format

Actual Certificate[1] in <widevine.mdt>

Subj: O=Motorola Inc, OU=Motorola PKI, CN=CSF CA 637-1
Issuer: O=Motorola Inc, OU=Motorola PKI, CN=APP 637-1-2; …

Public exponent, e: 0x10001

Modulus, N: c44dc735f6682a261a0b8545a62dd13df4c646a5ed…

Signature: 3cc1961f0d833a6197bd5537ee3f7d784dcf5dfb83b0…

Code: https://github.com/0x0atang/clkscrew/blob/master/pycrypto/parse_mdt_certs.py

Credits:

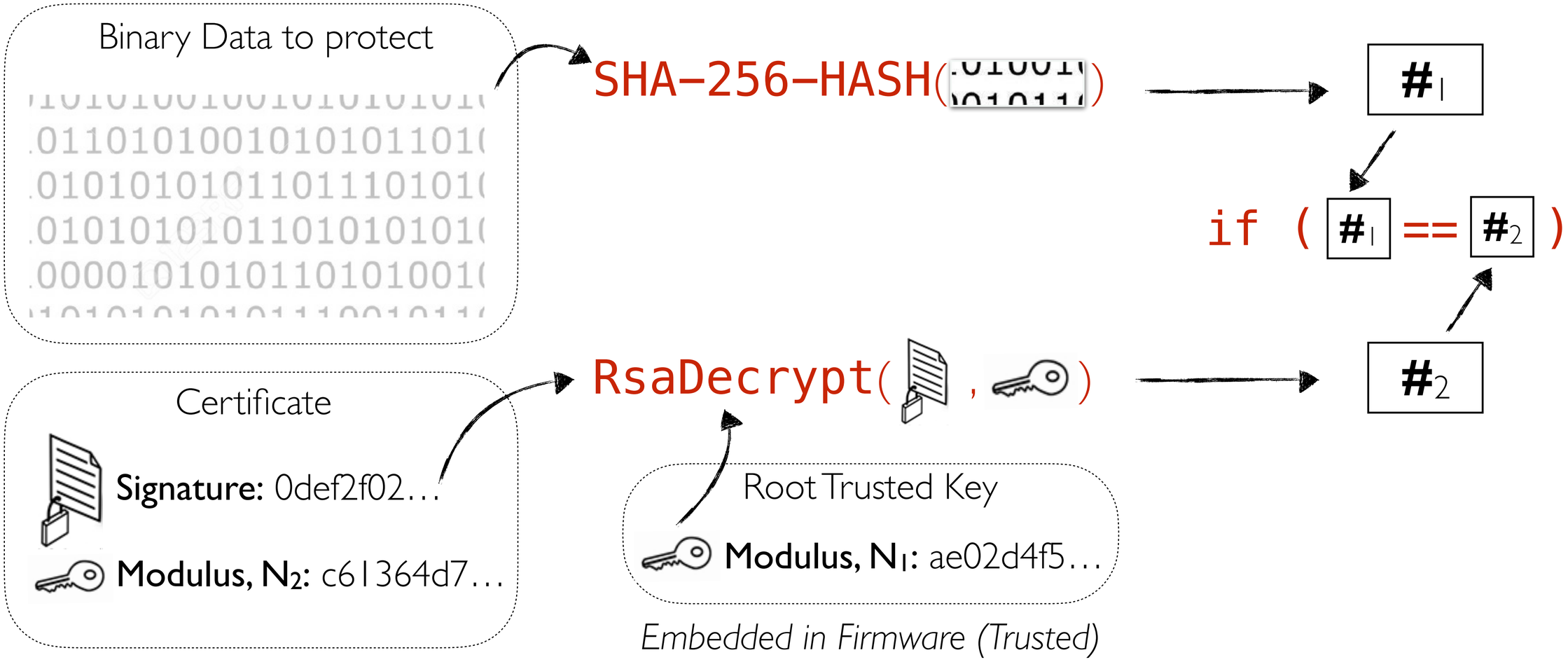
- Gal Beniamini: http://bits-please.blogspot.com/2016/04/exploring-qualcomms-secure-execution.html
- Tal Aloni: http://vm1.duckdns.org/Public/Qualcomm-Secure-Boot/Qualcomm-Secure-Boot.htm

Attack Exploration and Formulation

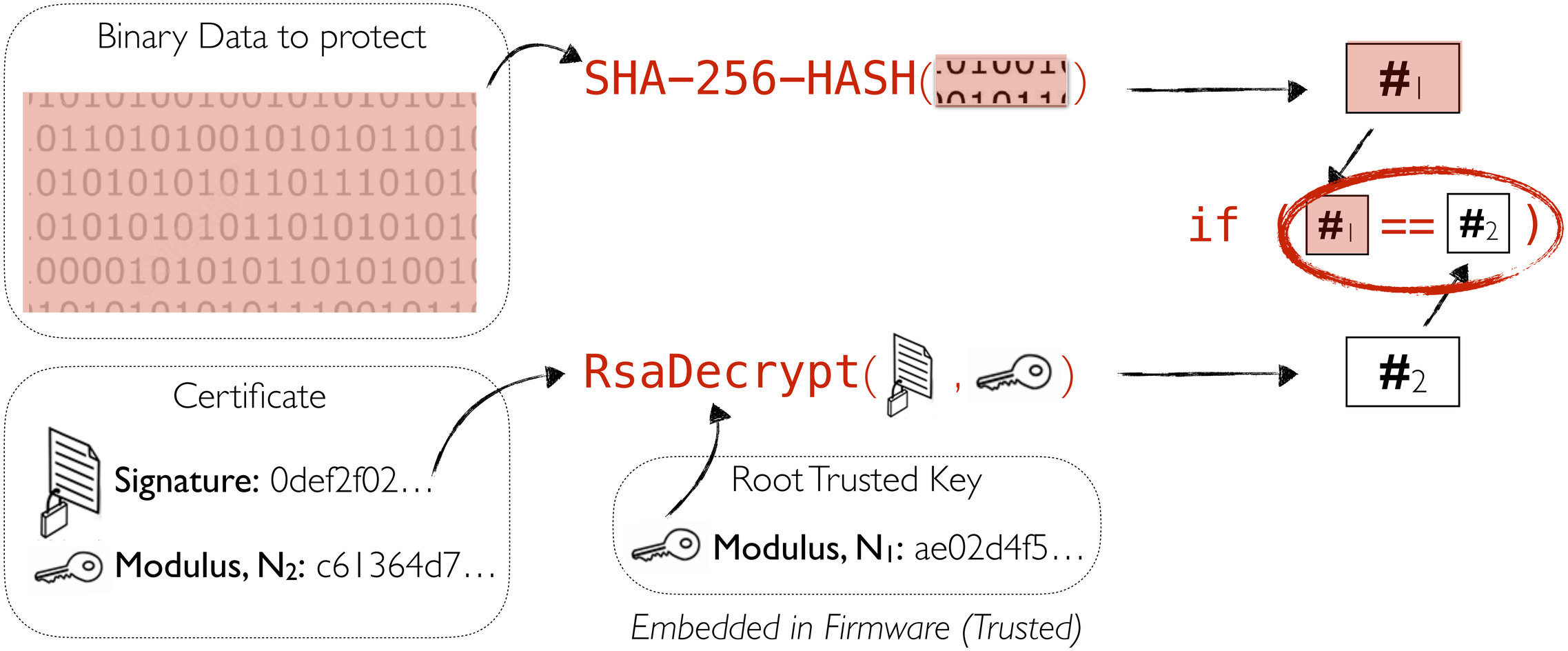
Trustzone Apps: How to craft self-signed app binary files?

Trustzone OS: How, where and when to inject the CLKSCREW fault?

Quick Review: How signatures are verified

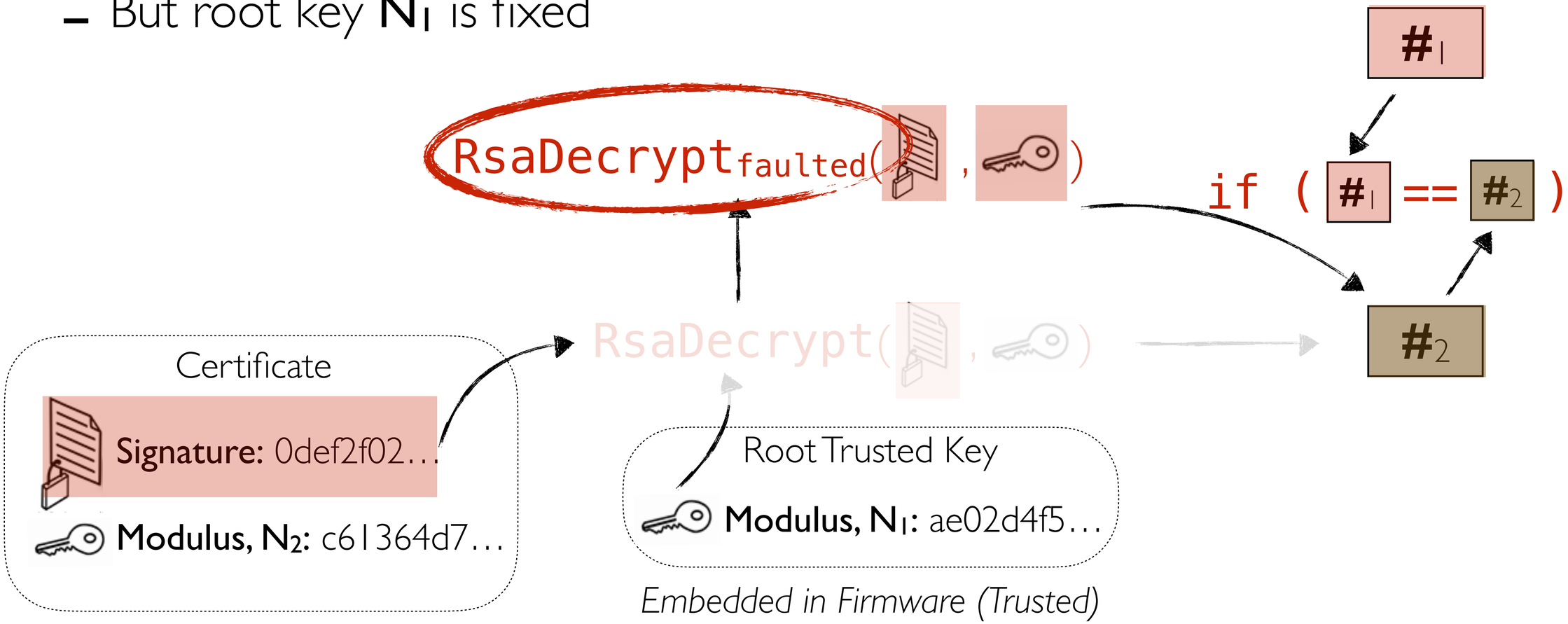


What if we modify the binary data (Our App)?



Why do we need to inject fault at runtime?

- We need **#1 == #2**
- But root key **N1** is fixed



Digging deep into the firmware

```
LOAD:FE824E9C qword_FE824E9C  DCD 0                   ; DATA XREF: sub_FE84C974+1D2↓o
LOAD:FE824E9C
LOAD:FE824EA0 SRK_modulus     DCB 0xAE, 2, 0xD4, 0xF5, 0xC5, 0x6F, 0x3C, 0xE0, 0xE3
LOAD:FE824EA0                 DCB 0xE7, 0x72, 0xE6, 0x7F, 0x52, 0x2E, 0xE8, 0xBF, 0xCE
LOAD:FE824EA0                 DCB 0x3A, 0xC4, 0x24, 0x2B, 0xCD, 0x5F, 0x1A, 0x96, 0x72
LOAD:FE824EA0                 DCB 0x47, 0x62, 0x5D, 0xC6, 0x4B, 0x71, 0xF4, 0xD0, 0xD4
LOAD:FE824EA0                 DCB 0xF, 0x9E, 0x3B, 0x4D, 0x44, 0x21, 0xB9, 0x5F, 0x3E
LOAD:FE824EA0                 DCB 0xA0, 0xE, 0x58, 0xEC, 0xEC, 0x3E, 0xD4, 0xC7, 0x3A
LOAD:FE824EA0                 DCB 0xD5, 0x58, 0xF7, 0x3C, 0x9C, 0x2A, 0xC4, 0xE, 0xD7
LOAD:FE824EA0                 DCB 0x82, 0x23, 0x7A, 0xB6, 2, 0xA, 0xA6, 0xB6, 0xFE, 0xB5
LOAD:FE824EA0                 DCB 0xB7, 0x44, 0x2B, 0x44, 0x8B, 0xD, 0x57, 0x10, 0x42
LOAD:FE824EA0                 DCB 0xC4, 0x2E, 0x4F, 0x72, 0xF6, 0xC9, 0x88, 0x7B, 0xF8
LOAD:FE824EA0                 DCB 0xB2, 0xB5, 0x58, 0x7C, 0x43, 0x7A, 0xA0, 0x4D, 0xC7
LOAD:FE824EA0                 DCB 0xA8, 0x3E, 0x3A, 0xD5, 0xC0, 0x7C, 0xA3, 0xEA, 0x30
LOAD:FE824EA0                 DCB 0xA4, 0x6F, 0x5B, 6, 0x4E, 0xB9, 0x9F, 0xE6, 0xF0
LOAD:FE824EA0                 DCB 0xFD, 0xAF, 0xB9, 0xBC, 0x62, 0x6E, 0x74, 0x97, 0x5B
LOAD:FE824EA0                 DCB 7, 0x13, 0x8E, 0x39, 0xD2, 0x13, 0xBD, 0x7C, 0x6C
LOAD:FE824EA0                 DCB 0xDC, 0xF2, 0xB, 0xE4, 0x50, 0xDD, 0x60, 0x3C, 0x57
LOAD:FE824EA0                 DCB 0x77, 0xD0, 0x5B, 0x99, 0xC, 0xD0, 0xCD, 0xAE, 0x42
LOAD:FE824EA0                 DCB 0xBA, 0x2F, 0x28, 0xA5, 0xCE, 0x1D, 0x13, 0xCE, 0xCB
LOAD:FE824EA0                 DCB 0xB2, 0x68, 0x38, 0xD2, 0xF5, 0xEC, 0xDC, 0xF1, 0xB9
LOAD:FE824EA0                 DCB 0xF9, 0xFF, 0x56, 0x27, 0x47, 0x66, 0xDE, 0x59, 0x4F
LOAD:FE824EA0                 DCB 0xC0, 0xA5, 0xF4, 0x48, 0xEC, 0xB7, 0x61, 0xBA, 0xF1
LOAD:FE824EA0                 DCB 0x82, 0xB3, 0x96, 0x2A, 0xC1, 0x19, 0x12, 0xD3, 0xE4
LOAD:FE824EA0                 DCB 0x2F, 0xFE, 0x91, 0xBF, 0xFE, 0xFD, 0x8C, 0xE, 0x9F
LOAD:FE824EA0                 DCB 0x40, 0x3A, 0xE1, 0xD5, 0xEB, 0xBF, 0x92, 0x6E, 0xCD
LOAD:FE824EA0                 DCB 0x3B, 0xA9, 0x17, 0x3E, 0x41, 0xAF, 0x39, 0x28, 0xAC
LOAD:FE824EA0                 DCB 0x55, 0x5A, 2, 1, 0x81, 0x7C, 0x69, 0x14, 5, 0x3F
LOAD:FE824EA0                 DCB 0x1D, 0x27, 0x2D, 0x32, 0xAD, 0x8F, 0x85, 0xD6, 0x77
LOAD:FE824EA0                 DCB 0x65, 0xC3, 0xF0, 0x7E, 0xB1, 0x92, 0x76, 0x86, 0xC7
LOAD:FE824EA0                 DCB 0x54, 0xA3
LOAD:FE824FA0                 DCB 0xDA ;
LOAD:FE824FA1                 DCB 0x7C ; |
```

Root Trusted Key

Modulus, N_1: ae02d4f5…

Embedded in Firmware (Trusted)

Super Root Key (SRK) modulus
2048-bit

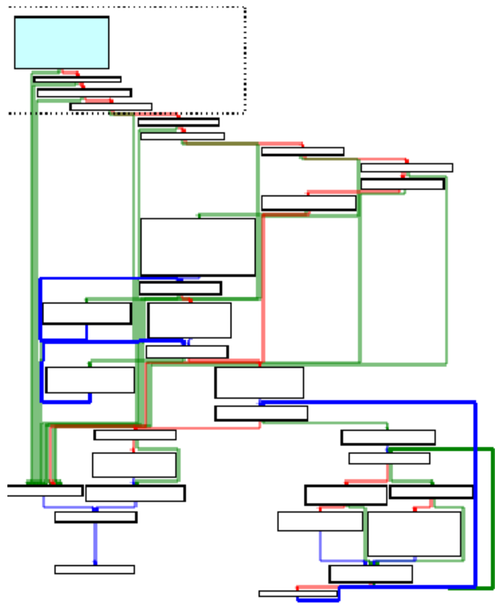
RSA-2048

Nexus 6 Trustzone Firmware (Shamu Build: MOD31S)

Digging deep into the firmware

signature modulus public exponent (0x10001)

$$\texttt{RsaDecrypt}(S, N, e) = S^e \bmod N = \text{decrypted hash}$$



```
FE8643C0
FE8643C0
FE8643C0
FE8643C0                        ; int __cdecl sub_FE8643C0(char *a1, char *a2, int a3, int a4, unsigned int a5, int a6)
FE8643C0                        sub_FE8643C0
FE8643C0
FE8643C0                        var_38= -0x38
FE8643C0                        var_2C= -0x2C
FE8643C0                        a5=  0
FE8643C0                        a6=  4
FE8643C0
FE8643C0 2D E9 FF 5F            PUSH.W          {R0-R12,LR}
FE8643C4 56 08                  LSRS            R6, R2, #1
FE8643C6 DD F8 3C B0            LDR.W           R11, [SP,#0x38+a6]
FE8643CA B7 00                  LSLS            R7, R6, #2
FE8643CC 00 98                  LDR             R0, [SP,#0x38+var_38]
FE8643CE 48 F6 08 65            MOVW            R5, #0x8E08
FE8643D2 0C 46                  MOV             R4, R1
FE8643D4 91 46                  MOV             R9, R2
FE8643D6 C0 F6 C8 75            MOVT.W          R5, #0xFC8
FE8643DA 4F F0 00 08            MOV.W           R8, #0
FE8643DE 4F EA 92 0A            MOV.W           R10, R2,LSR#2
FE8643E2 D0 B1                  CBZ             R0, loc_FE86441A
```

```
FE8643E4 CC B1                  CBZ             R4, loc_FE86441A
```

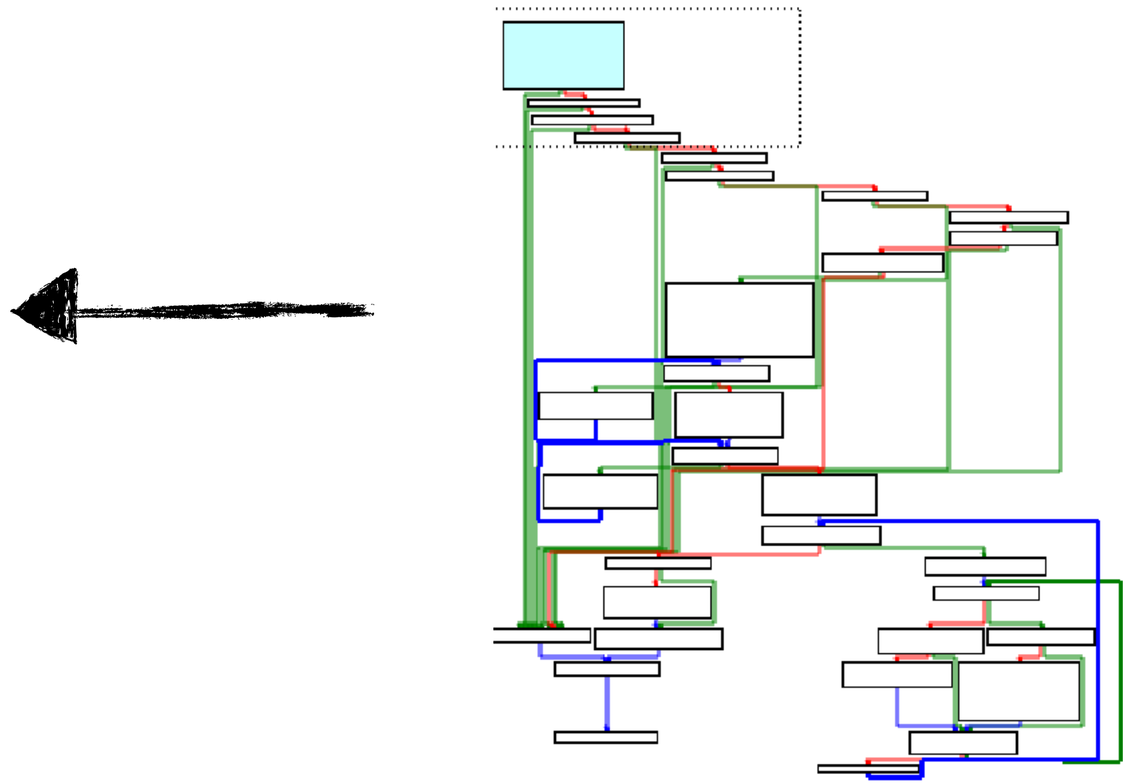
Nexus 6 Trustzone Firmware (Shamu Build: MOD31S)

Digging deep into the firmware

```
1:  procedure DecryptSig(S, e, N)
2:      r ← 2^2048
3:      R ← r^2 mod N
4:      N_rev ← FlipEndianness(N)
5:      r^-1 ← ModInverse(r, N_rev)
6:      found_first_one_bit ← false
7:      for i ∈ {bitlen(e) − 1 .. 0} do
8:          if found_first_one_bit then
9:              x ← MontMult(x, x, N_rev, r^-1)
10:             if e[i] == 1 then
11:                 x ← MontMult(x, a, N_rev, r^-1)
12:             end if
13:         else if e[i] == 1 then
14:             S_rev ← FlipEndianness(S)
15:             x ← MontMult(S_rev, R, N_rev, r^-1)
16:             a ← x
17:             found_first_one_bit ← true
18:         end if
19:     end for
20:     x ← MontMult(x, 1, N_rev, r^-1)
21:     H ← FlipEndianness(x)
22:     return H
23: end procedure
```

signature modulus public exponent (0x10001)

```
RsaDecrypt( S , N , e )
```



Digging deep into the firmware

```
1:  procedure DecryptSig(S, e, N)
2:      r ← 2^2048
3:      R ← r^2 mod N
4:      N_rev ← FlipEndianness(N)
5:      r^-1 ← ModInverse(r, N_rev)
6:      found_first_one_bit ← false
7:      for i ∈ {bitlen(e) − 1 .. 0} do
8:          if found_first_one_bit then
9:              x ← MontMult(x, x, N_rev, r^-1)
10:             if e[i] == 1 then
11:                 x ← MontMult(x, a, N_rev, r^-1)
12:             end if
13:         else if e[i] == 1 then
14:             S_rev ← FlipEndianness(S)
15:             x ← MontMult(S_rev, R, N_rev, r^-1)
16:             a ← x
17:             found_first_one_bit ← true
18:         end if
19:     end for
20:     x ← MontMult(x, 1, N_rev, r^-1)
21:     H ← FlipEndianness(x)
22:     return H
23: end procedure
```

signature modulus public exponent (0x10001)

```
RsaDecrypt( S , N , e )
```

Reverse engineering approaches:

- Statically via IDA
- Emulation using angr - http://angr.io/
- Dynamic code instrumentation on Trustzone code on actual phones (more details in future!)

Digging deep into the firmware

```
1:  procedure DecryptSig(S, e, N)
2:      r ← 2^2048
3:      R ← r^2 mod N
4:      N_rev ← FlipEndianness(N)
5:      r^-1 ← ModInverse(r, N_rev)
6:      found_first_one_bit ← false
7:      for i ∈ {bitlen(e) − 1 .. 0} do
8:          if found_first_one_bit then
9:              x ← MontMult(x, x, N_rev, r^-1)
10:             if e[i] == 1 then
11:                 x ← MontMult(x, a, N_rev, r^-1)
12:             end if
13:         else if e[i] == 1 then
14:             S_rev ← FlipEndianness(S)
15:             x ← MontMult(S_rev, R, N_rev, r^-1)
16:             a ← x
17:             found_first_one_bit ← true
18:         end if
19:     end for
20:     x ← MontMult(x, 1, N_rev, r^-1)
21:     H ← FlipEndianness(x)
22:     return H
23: end procedure
```

signature modulus public exponent (0x10001)

```
RsaDecrypt( S , N , e )
```

- Computes modular exponentiation: $S^e \bmod N$
- Implemented with an efficient form of multiplication called Montgomery Multiplication[1]

$$\text{MontMult}(x, y, N, r^{-1}) \leftarrow x \cdot y \cdot r^{-1} \bmod N$$

- Uses a memory-intensive function that reverses memory buffers

$$S_{rev} \leftarrow \text{FlipEndianness}(S)$$

[1] KOC, C. K. High-speed RSA implementation. Tech. rep., Technical Report, RSA Laboratories, 1994.

Digging deep into the firmware

```
1:  procedure DecryptSig(S, e, N)
2:      r ← 2^2048
3:      R ← r^2 mod N
4:      N_rev ← FlipEndianness(N)
5:      r^-1 ← ModInverse(r, N_rev)
6:      found_first_one_bit ← false
7:      for i ∈ {bitlen(e) − 1 .. 0} do
8:          if found_first_one_bit then
9:              x ← MontMult(x, x, N_rev, r^-1)
10:             if e[i] == 1 then
11:                 x ← MontMult(x, a, N_rev, r^-1)
12:             end if
13:         else if e[i] == 1 then
14:             S_rev ← FlipEndianness(S)
15:             x ← MontMult(S_rev, R, N_rev, r^-1)
16:             a ← x
17:             found_first_one_bit ← true
18:         end if
19:     end for
20:     x ← MontMult(x, 1, N_rev, r^-1)
21:     H ← FlipEndianness(x)
22:     return H
23: end procedure
```

signature modulus public exponent (0x10001)

Where to inject the runtime fault?

$$N_{A,rev} \xleftarrow{fault} \text{FLIPENDIANNESS}(N)$$

- Co
- Implemented with an efficient form of multiplication called Montgomery Multiplication[1]

$$\text{MONTMULT}(x, y, N, r^{-1}) \leftarrow x \cdot y \cdot r^{-1} \bmod N$$

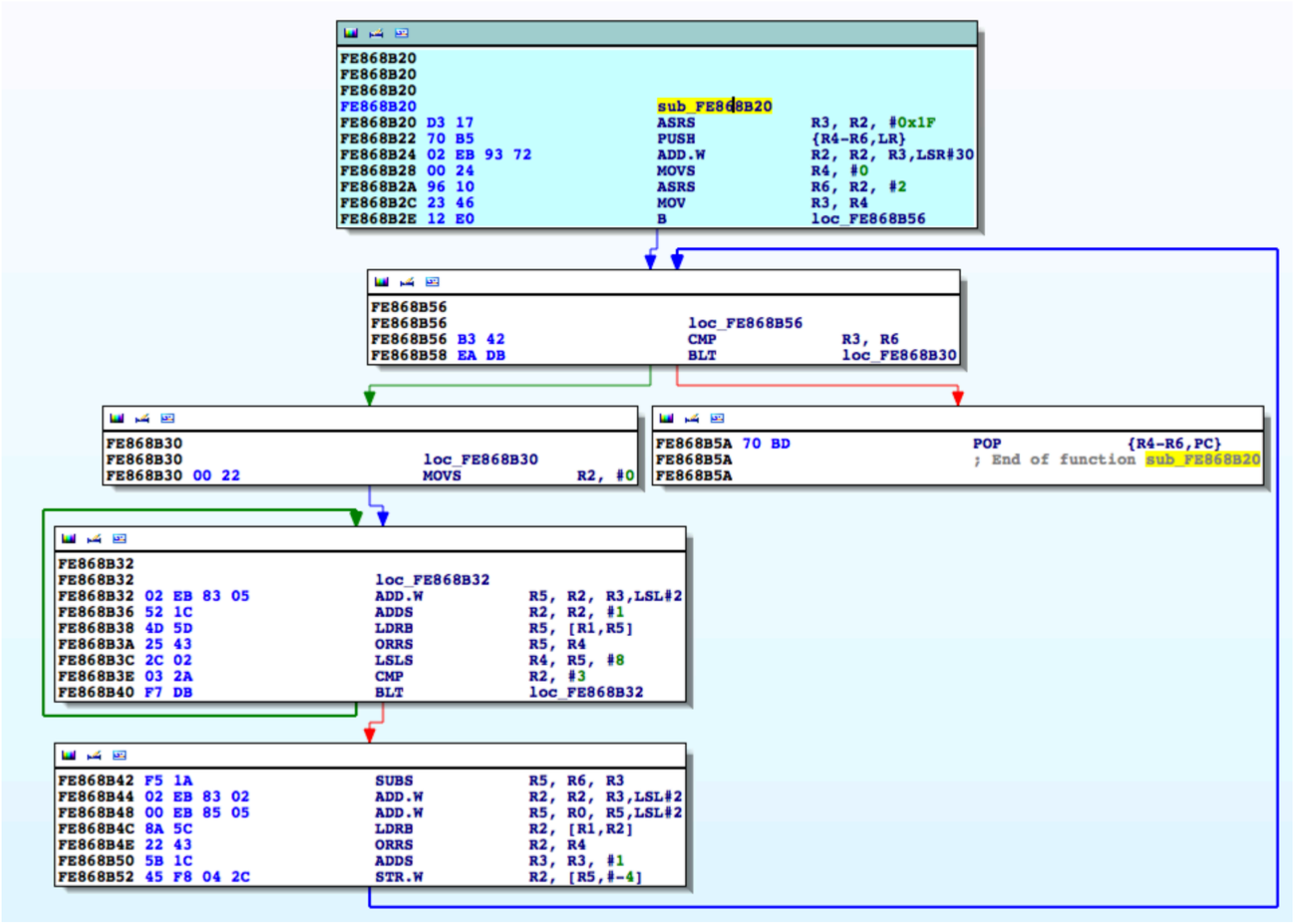
- Uses a memory-intensive function that reverses memory buffers

$$S_{rev} \leftarrow \text{FLIPENDIANNESS}(S)$$

[1] KOC, C. K. High-speed RSA implementation. Tech. rep., Technical Report, RSA Laboratories, 1994.

Corrupting FlipEndianness with Runtime Fault

```
1:  procedure FlipEndianness(src)
2:      d ← 0
3:      dst ← {0}
4:      for i ∈ {0 .. len(src)/4 − 1} do
5:          for j ∈ {0 .. 2} do
6:              d ← (src[i ∗ 4 + j] | d) ≪ 8
7:          end for
8:          d ← src[i ∗ 4 + 3] | d
9:          k ← len(src) − i ∗ 4 − 4
10:         dst[k .. k + 3] ← d
11:     end for
12:     return dst
13: end procedure
```



Nexus 6 Trustzone Firmware (Shamu Build: MOD31S)

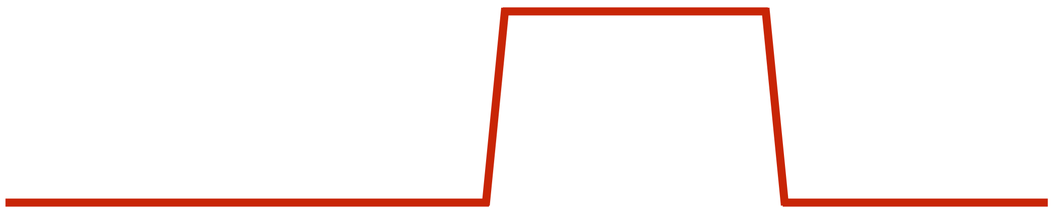
Corrupting FlipEndianness with Runtime Fault

Base voltage: 1.055V

High frequency: 4.10GHz

Low frequency: 2.68GHz

Fault injection duration:
5 no-op loops (~0.287 µsec)



Expected modulus: … `bc099b4a` …

Faulty modulus used: … `bc09`**`4a`**`4a` …

Demo: https://asciinema.org/a/5vvn3s9nzula930xui1z7tg65

Code: https://github.com/0x0atang/clkscrew/blob/master/faultmin_SD805/

Digging deep into the firmware

```
1:  procedure DecryptSig(S, e, N)
2:      r ← 2^2048
3:      R ← r^2 mod N
4:      N_rev ← FlipEndianness(N)
5:      r^-1 ← ModInverse(r, N_rev)
6:      found_first_one_bit ← false
7:      for i ∈ {bitlen(e) − 1 .. 0} do
8:          if found_first_one_bit then
9:              x ← MontMult(x, x, N_rev, r^-1)
10:             if e[i] == 1 then
11:                 x ← MontMult(x, a, N_rev, r^-1)
12:             end if
13:         else if e[i] == 1 then
14:             S_rev ← FlipEndianness(S)
15:             x ← MontMult(S_rev, R, N_rev, r^-1)
16:             a ← x
17:             found_first_one_bit ← true
18:         end if
19:     end for
20:     x ← MontMult(x, 1, N_rev, r^-1)
21:     H ← FlipEndianness(x)
22:     return H
23: end procedure
```

public exponent (0x10001)

signature modulus

Where to inject the runtime fault?

$$N_{A,rev} \xleftarrow{fault} \textsc{FlipEndianness}(N)$$

- Co
- Implemented with an efficient form of multiplication called Montgomery Multiplication[1]
- Us

How to craft attack signature S_A'?

$$\textsc{DecryptSig}(S_A{}', e, N) \xrightarrow{fault} H(C_A)$$

[1] KOC, C. K. High-speed RSA implementation. Tech. rep., Technical Report, RSA Laboratories, 1994.

How to craft attack signature?

```
1:  procedure DECRYPTSIG(S, e, N)
2:      r ← 2^2048
3:      R ← r^2 mod N
4:      N_rev ← FLIPENDIANNESS(N)
5:      r^-1 ← MODINVERSE(r, N_rev)
6:      found_first_one_bit ← false
7:      for i ∈ {bitlen(e) − 1 .. 0} do
8:          if found_first_one_bit then
9:              x ← MONTMULT(x, x, N_rev, r^-1)
10:             if e[i] == 1 then
11:                 x ← MONTMULT(x, a, N_rev, r^-1)
12:             end if
13:         else if e[i] == 1 then
14:             S_rev ← FLIPENDIANNESS(S)
15:             x ← MONTMULT(S_rev, R, N_rev, r^-1)
16:             a ← x
17:             found_first_one_bit ← true
18:         end if
19:     end for
20:     x ← MONTMULT(x, 1, N_rev, r^-1)
21:     H ← FLIPENDIANNESS(x)
22:     return H
23: end procedure
```

Trickier than expected!!

original N

Line 3: $R \leftarrow r^2 \bmod N$

Line 4: $N_{A,rev} \xleftarrow{fault} \textsc{FlipEndianness}(N)$

corrupted/faulted N

Line 15: $\textsc{MontMult}(S_A^{'},\ r^2 \bmod N,\ N_A,\ r_A^{-1})$

$= S_A^{'} \cdot (r^2 \bmod N) \cdot r_A^{-1} \bmod N_A$

different moduli N's used

(More cryptanalysis in white paper…)

https://github.com/0x0atang/clkscrew/blob/master/pycrypto/pycrypto.py

When (during execution) to inject fault?

- `DecryptSig()` is invoked *4 times* when verifying an app
  - 1) SRK.modulus => CERT[0]
  - 2) CERT[0].modulus => cert chain meta-data
  - 3) CERT[0].modulus => CERT[1]
  - 4) CERT[1].modulus => .mdt file hashes

- We need a way to profile when invocation (4) executes within Trustzone

- **Attack Enabler:** Memory accesses from outside Trustzone can evict cache lines used by Trustzone code

Side-channel-based
cache profiling

When (during execution) to inject fault?

- Instruction-cache **Prime+Probe** profiling more reliable than data-cache ones
  - More info on side-channel-based profiling attacks on ARM [1, 2, 3]

- I-Cache profiling not as convenient as D-Cache profiling
  - Instead of using memory read operations
  - Need to execute instructions at memory address congruent to cache sets we are monitoring
  - Create a JIT compiler — Given a list of cache sets to monitor
  - Allocate a large chunk of executable memory
  - Chain relative BR branch instructions in addresses congruent to monitored cache sets

[1] GRUSS, D., SPREITZER, R., AND MANGARD, S. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. USENIX 2015

[2] LIPP, M., GRUSS, D., SPREITZER, R., MAURICE, C., AND MANGARD, S. Armageddon: Cache attacks on mobile devices. USENIX 2016.

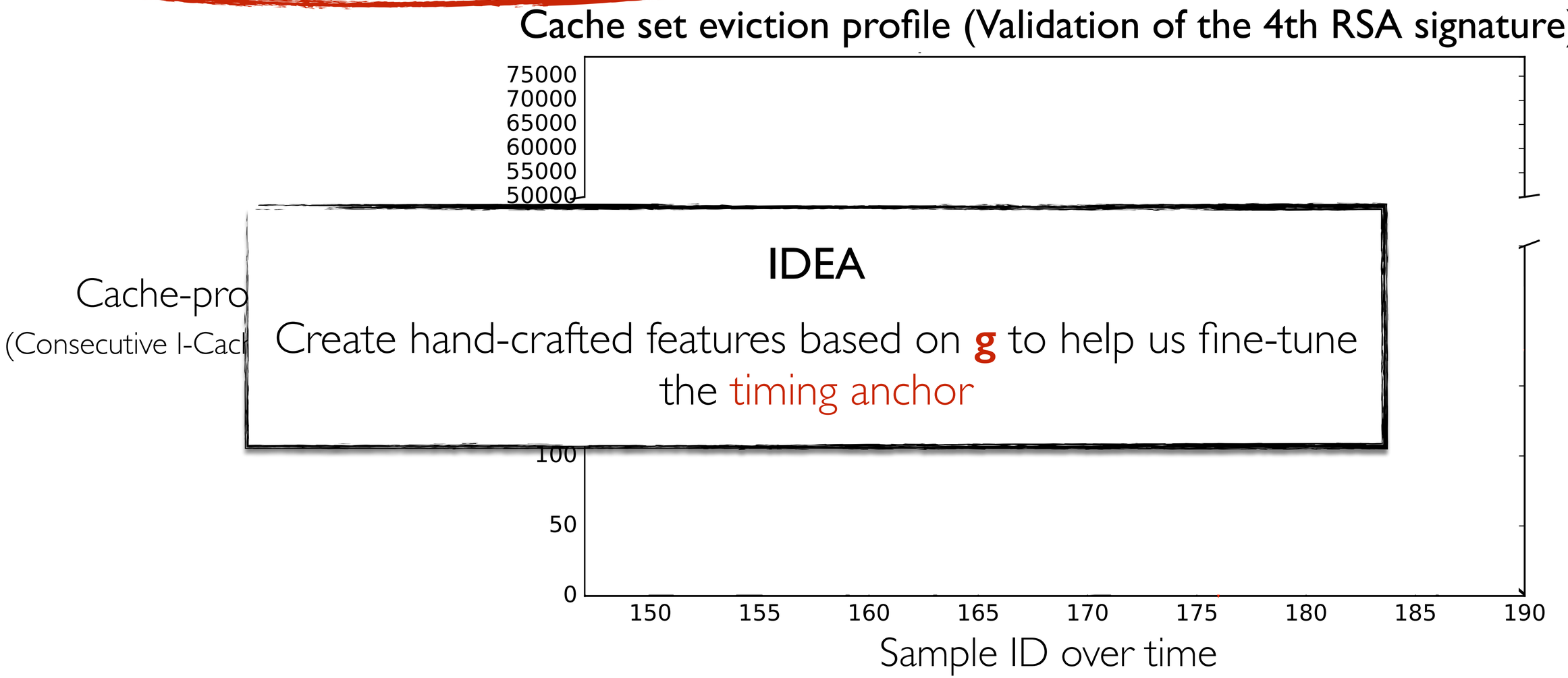
[3] ZHANG, X., XIAO, Y., AND ZHANG, Y. Return-Oriented Flush- Reload Side Channels on ARM and Their Implications for Android Devices. CCS 2016.

When (during execution) to inject fault?

- Sketch of I-Cache profiling
  - Pick a few code areas before the target victim code to monitor
  - Monitor for I-Cache eviction for these cache sets simultaneously (We monitor 4 sets)
  - Say **E** is the event when all these cache sets are found to be evicted
  - Track the next time **E** happens
  - Use an incrementing counter (as a high-precision timer) to track the duration between consecutive **E**'s
  - Call this duration between consecutive E's — **g**
  - Time-series **g** => a fine-grained proxy of Trustcode code execution

Timing-Based Code Execution Profiling

4) CERT[1].modulus => .mdt file hashes



IDEA

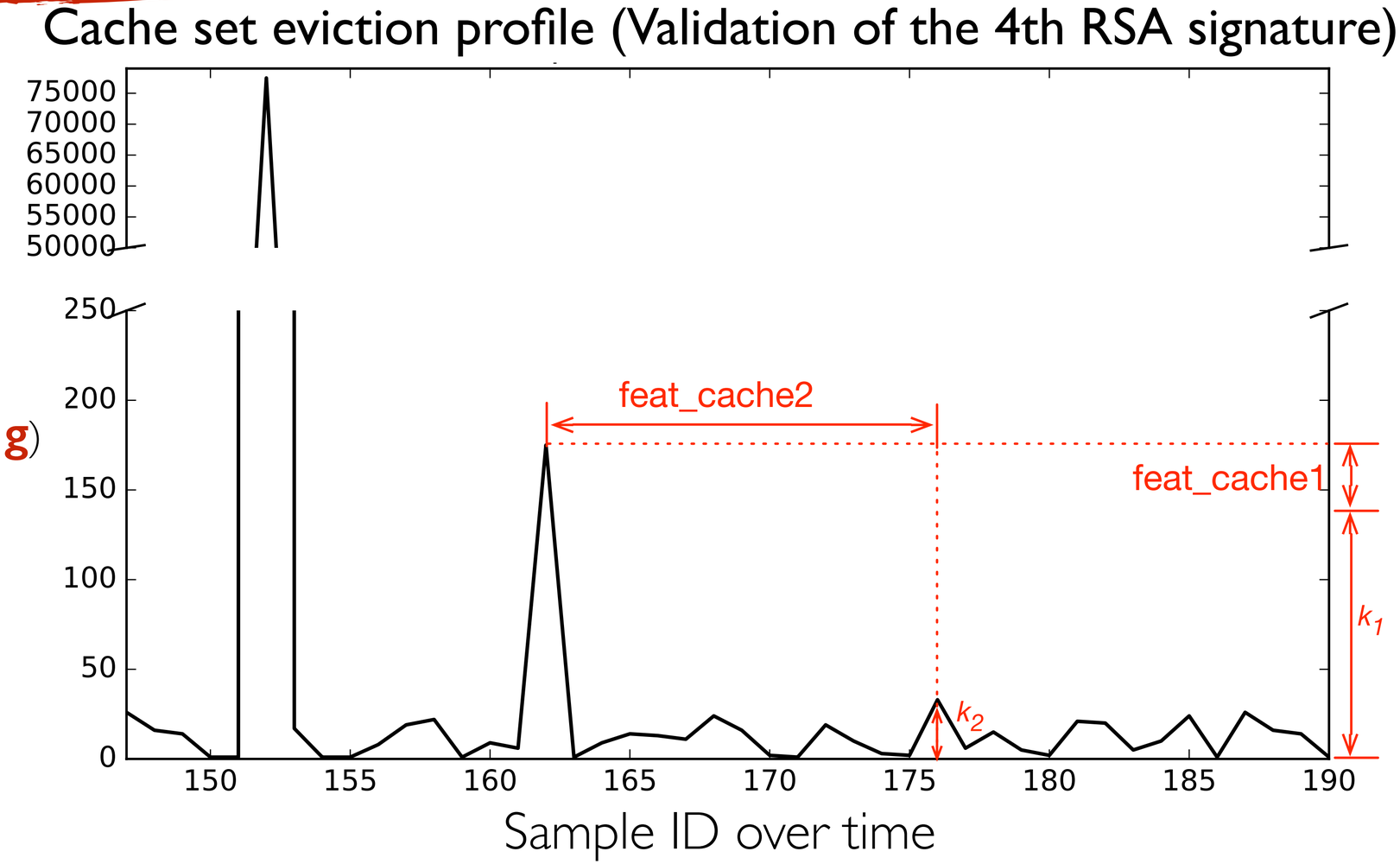
Create hand-crafted features based on **g** to help us fine-tune the timing anchor

Timing-Based Code Execution Profiling

4) CERT[1].modulus => .mdt file hashes

Cache-profiling feature

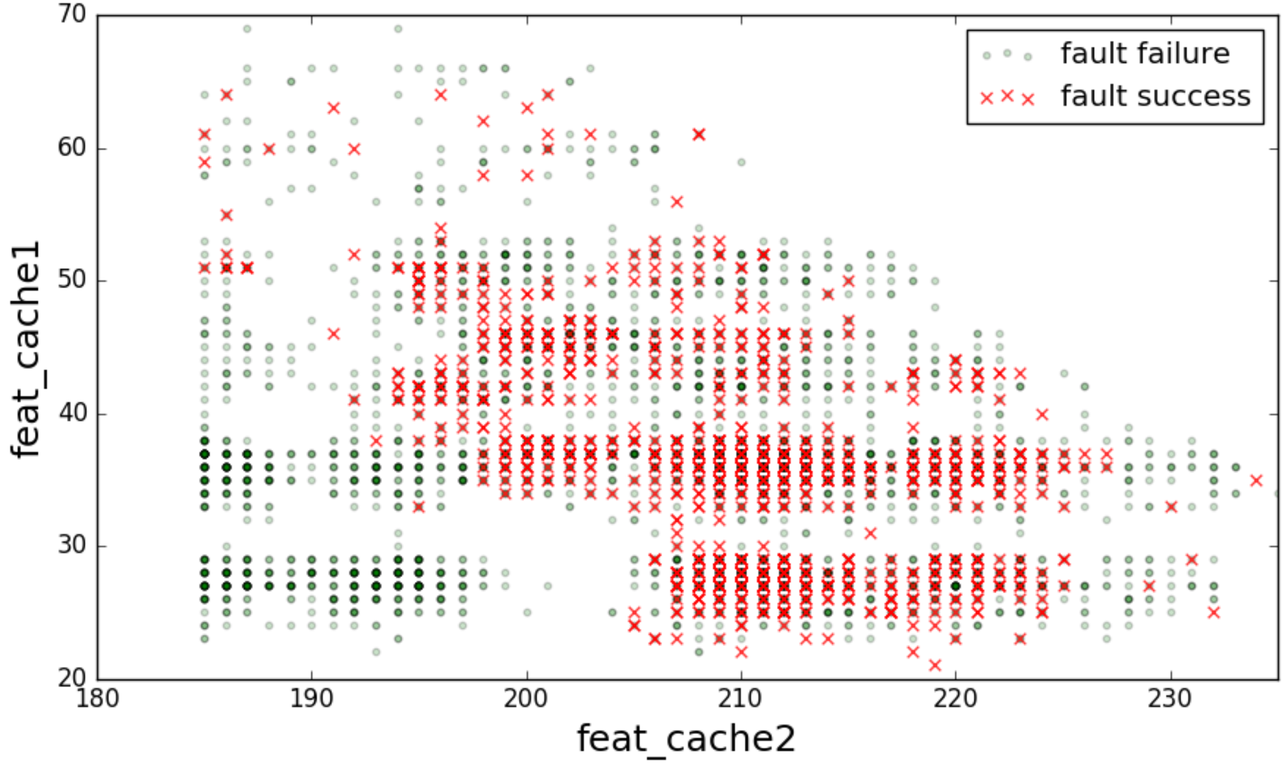
(Consecutive I-Cache eviction interval, **g**)



Timing-Based Code Execution Profiling

Track a "Fault success" as successfully corrupting targeted N modulus

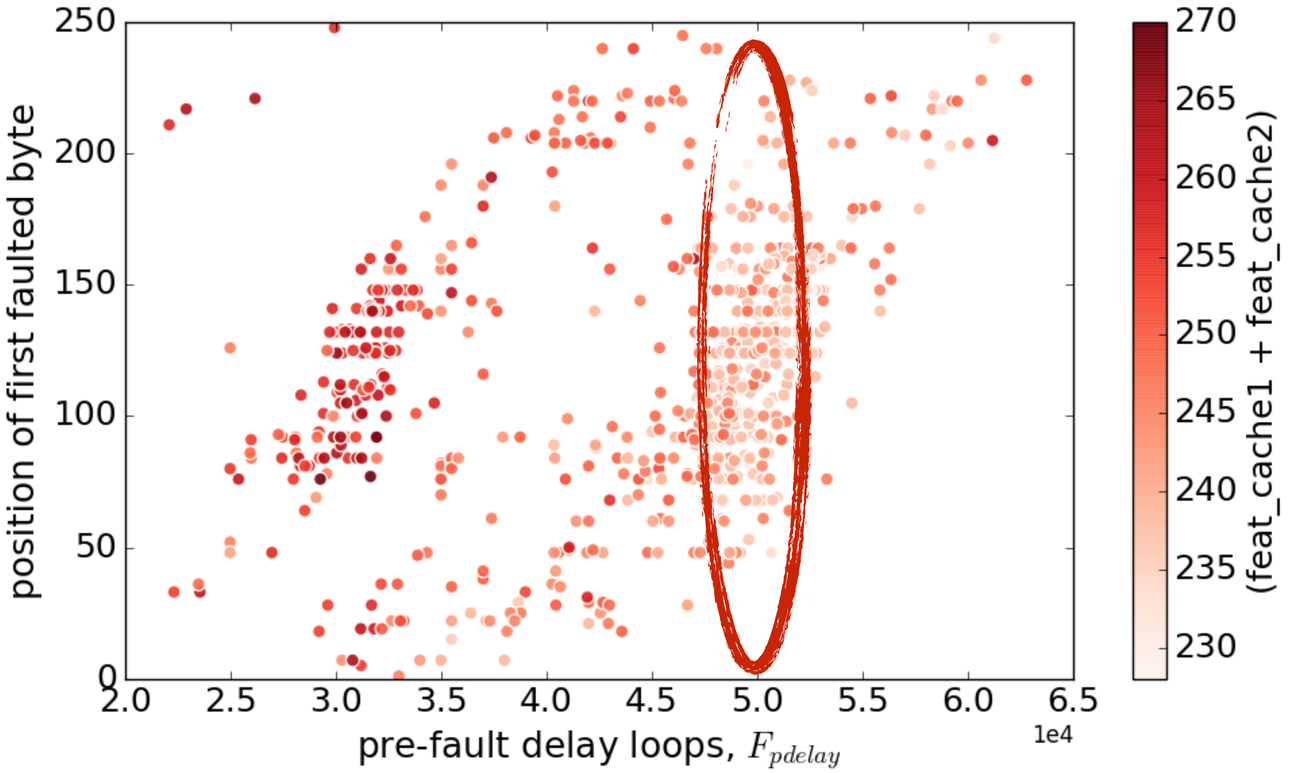
Both feat_cache1 and feat_cache2 can influence success rate of faults



Timing-Based Code Execution Profiling

But these features alone are insufficient!

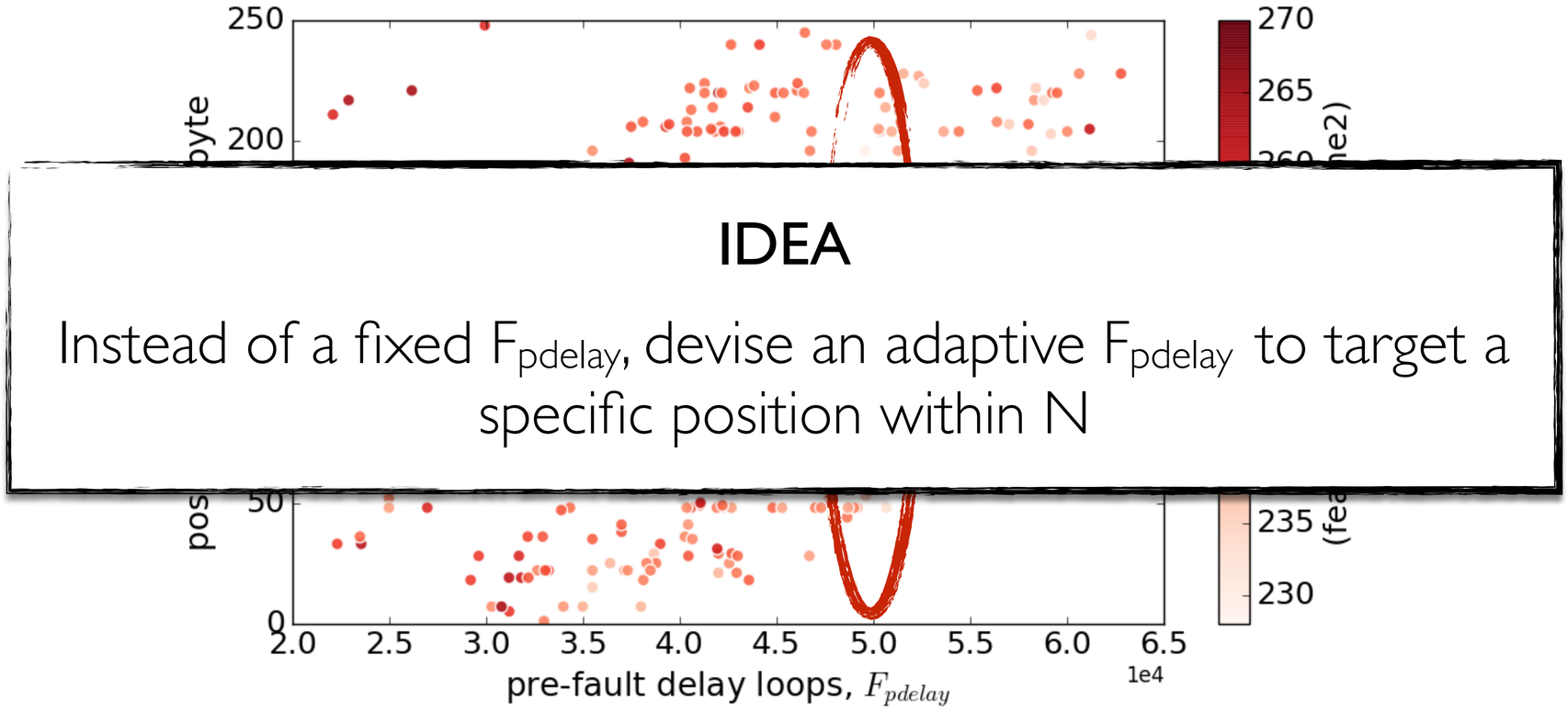
Too much variability given any value of pre-fault delay loops, F_{pdelay}



Timing-Based Code Execution Profiling

But these features alone are insufficient!

Too much variability given any value of pre-fault delay loops, F_{pdelay}



IDEA

Instead of a fixed F_{pdelay}, devise an adaptive F_{pdelay} to target a specific position within N

Timing-Based Code Execution Profiling

Sample lots of faulting parameters and resulting faulted buffer position, F_{pos}

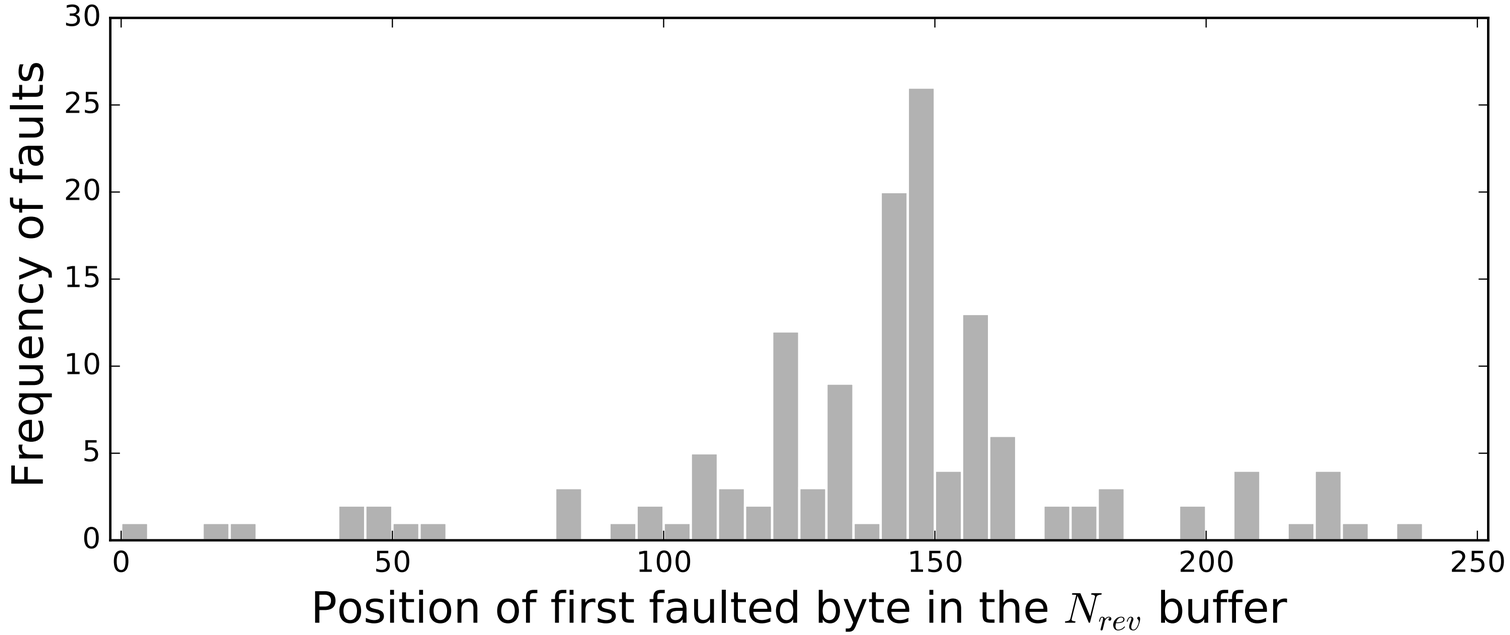
Create a linear regression model based on the empirical observations:

$$F_{pos} \sim feat_{cache1} + feat_{cache2} + F_{pdelay} + temperature + intercept$$

At runtime, we can then adjust our adaptive pre-fault delay loops, F_{pdelay}

$$F_{pdelay} = \mathbf{f}\ (F_{pos}, feat_{cache1}, feat_{cache2}, temperature, intercept)$$

Putting it together



Statistics:

- ~20% of faulting attempts (1153 out of 6000) result in a successful desired fault in the N_{rev} buffer we want
- These faults consist of 805 unique values, of which 38 (4.72%) are factorizable
- One instance of the desired fault in ~65 faulting attempts

Summary of Attack Enablers

I. No hardware safeguard limits in regulators

II. Large range of possible combinations of freq/volt for fault injection

III. Cores deployed in different freq/volt domains

IV. Hardware regulators operate across security boundaries

V. Execution timing of Trustzone code can be profiled with hardware cycle counter from outside Trustzone

VII. Trustzone code execution can be profiled using side-channel-based attacks, like Prime+Probe cache attacks

Outline

I. DVFS and Deep Dive into Hardware Regulators

II. The CLKSCREW Attack

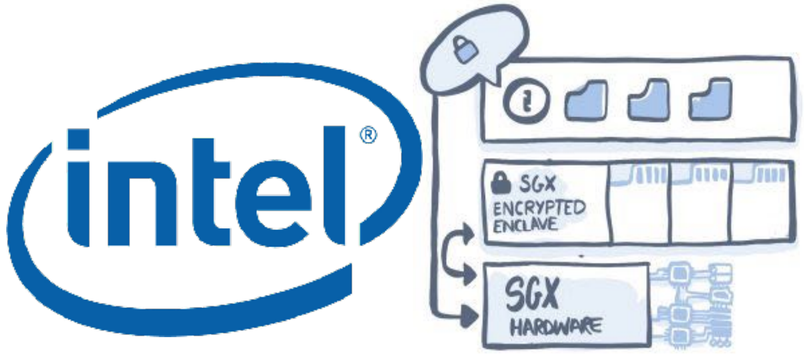
III. Trustzone Attack 1: Secret AES Key Inference

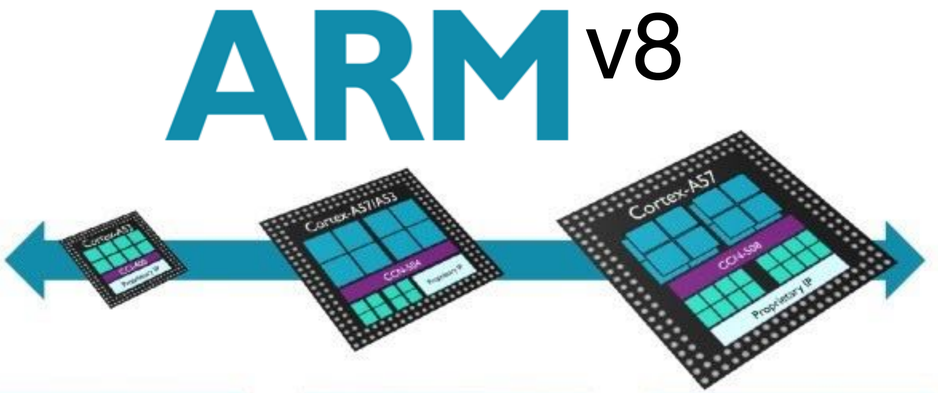
IV. Trustzone Attack 2: Tricking RSA Signature Validation

V. Concluding Remarks

Attack Applicability to Other Platforms

Energy management mechanisms in the industry is trending towards finer-grained and increasingly heterogeneous designs





Cloud computing providers

Possible Defenses

Hardware-Level

- Operating limits in hardware
- Separate cross-boundary regulators
- Microarchitectural Redundancy

Software-Level

- Randomization
- Code execution redundancy

Sound Bytes

New attack surface via energy management software interfaces

Not a hardware or software bug
Fundamental design flaws in energy management mechanisms

Future energy management designs must take security into consideration

black hat EUROPE 2017

CLKSCREW

Exposing the Perils of Security-Oblivious Energy Management

Adrian Tang

Simha Sethumadhavan, Salvatore Stolfo

@0x0atang

https://github.com/0x0atang/clkscrew

https://0x0atang.github.io

COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK