CLKScrew
Exposing the Perils of Security-Oblivious Energy Management

Adrian Tang, Simha Sethumadhavan, Salvatore Stolfo
Columbia University
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
Today’s systems cannot exist without Energy Management

ASTRONOMICAL POWER DENSITY -> CRAZY HOTSPOTS!

Source: Adapted from S. Borkar (Intel)
Today’s systems cannot exist without Energy Management

Industry

Academia

Today’s systems cannot exist without Energy Management to stay secure with.

Essential

Pervasive

Complicated
Exploiting software interfaces to **Energy Management**

Software-based attacker → Stretch operational limits → Induce faults

- Frequency
- Voltage
- Decryption key
Exploiting software interfaces to Energy Management

Traditional fault attacks
- Need physical proximity
- Need separate equipment
- Soldering, crocodile clips, wire, etc.
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)

Energy consumption

Frequency

Voltage

DVFS
Hardware & Software Support for DVFS

Software

DVFS

Hardware

Power Governor

Vendor Device Driver

Memory-Mapped Registers

Frequency Regulator

Voltage Regulator
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)

Legend:

🌟🌟🌟🌟🌟 791 2
- Android v5(Lollipop)
- Turbo Charging
- 5.94 inch OHD A
- 2.7 GHz Process

Nexus 6
Lower voltage → Lower minimum required frequency to induce instability

Legend:
- ★★★ Vendor-recommended
- ●●● Max OPP reached before instability

No safeguard hardware limits
Freq / Voltage Operating Point Pairs (OPPs)

Device ‘A’

Device ‘B’

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!

CPU Core

- Trustzone
  - Trusted code
- Normal
  - Untrusted code

Frequency & Voltage Regulators

- Frequency and voltage changes

- Hardware-enforced isolation
- Regulator HW-SW interface
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
How do faults occur (due to over-raising frequency)?
How do faults occur (due to over-raising frequency)?

CLK signal

input

0

flip-flop

input

output

intermediate logic path

input

flip-flop

output

1

higher frequency

less time for data to propagate

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-contained 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
#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

Start fault

End fault

Code execution to inject fault

Disable interrupts

Enable interrupts

Victim thread

Attack thread
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

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

Confidentiality Attack
infer secret AES key stored within Trustzone

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

Key Inference Attack: **CLKSCREW Parameters**

- **Base voltage:** 1.055V
- **High frequency:** 3.69GHz
- **Low frequency:** 2.61GHz
- **Fault injection duration:** 680 no-op loops (~39 μsec)

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](https://github.com/Daeinar/dfa-aes)

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
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 a 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 $F_{p\text{delay}}$ 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.
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
  \[ \text{app.mdt} + \text{app.b00} + \ldots + \text{app.b03} \]

  **ELF Header**
  **Meta-data**
  **Certificate chain**
  **Signature blobs**

  Program code (split by sections)
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](https://github.com/0x0atang/clkscrew/blob/master/pycrypto/parse_mdt_certs.py)

Credits:
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

- **Signature:** 0def2f02...
  - **Certificate**
    - **Modulus, N₂:** c61364d7...
  - **Binary Data to protect**
    - 1010100100101010101010...

- **Root Trusted Key**
  - **Modulus, N₁:** ae02d4f5...
  - **Embedded in Firmware (Trusted)**

- **SHA-256-HASH**
  - $\text{SHA-256-HASH}(0\text{def2f02})$

- **RsaDecrypt**
  - $\text{RsaDecrypt}(\text{Certificate}, \text{Root Trusted Key})$

- **if**
  - if ($\#₁ == \#₂$)
What if we modify the binary data (Our App)?

Binary Data to protect

Certificate

Signature: 0def2f02...

Modulus, N2: c61364d7...

Root Trusted Key

Modulus, N1: ae02d4f5...

Embedded in Firmware (Trusted)

SHA-256-HASH(010011)

RsaDecrypt(#1, #2)

if #1 == #2

#2
Why do we need to inject fault at runtime?

- We need \#1 == \#2
- But root key \( N_1 \) is fixed

\[
\text{Certificate} \quad \text{Signature: 0def2f02...}
\]

\[
\text{Modulus, } N_2: \text{ c61364d7...}
\]

\[
\text{RsaDecrypt}_{\text{faulted}}(\ldots, \ldots)
\]

\[
\text{Root Trusted Key} \quad \text{Modulus, } N_1: \text{ ae02d4f5...}
\]

\[
\text{Embedded in Firmware (Trusted)}
\]
Digging deep into the firmware

Nexus 6 Trustzone Firmware (Shamu Build: MOD31S)

Super Root Key (SRK) modulus
2048-bit

RSA-2048
Digging deep into the firmware

\[ \text{RsaDecrypt}(S, N, e) = S^e \mod N = \text{decrypted hash} \]

Nexus 6 Trustzone Firmware (Shamu Build: MOD31S)
Digging deep into the firmware

1: procedure DECYPTRSIG(S, e, N)
2:   \( r \leftarrow 2^{2048} \)
3:   \( R \leftarrow r^2 \mod N \)
4:   \( N_{rev} \leftarrow \text{FIPENDIANNESS}(N) \)
5:   \( r^{-1} \leftarrow \text{MODINVERSE}(r, N_{rev}) \)
6:   \text{found_first_one_bit} \leftarrow \text{false}
7:   \text{for } i \in \text{bitlen}(e) - 1 \ldots 0 \text{ do}
8:     \text{if } \text{found_first_one_bit} \text{ then}
9:       x \leftarrow \text{MONTMULT}(x, x, N_{rev}, r^{-1})
10:      \text{if } e[i] == 1 \text{ then}
11:         x \leftarrow \text{MONTMULT}(x, a, N_{rev}, r^{-1})
12:      \text{end if}
13:     \text{else if } e[i] == 1 \text{ then}
14:       S_{rev} \leftarrow \text{FIPENDIANNESS}(S)
15:       x \leftarrow \text{MONTMULT}(S_{rev}, R, N_{rev}, r^{-1})
16:       a \leftarrow x
17:       \text{found_first_one_bit} \leftarrow \text{true}
18:     \text{end if}
19:   \text{end for}
20:   x \leftarrow \text{MONTMULT}(x, 1, N_{rev}, r^{-1})
21:   H \leftarrow \text{FIPENDIANNESS}(x)
22: return H
23: end procedure
Digging deep into the firmware

```
1: procedure DecompSig(S, e, N)
2:     r ← 2^2048
3:     R ← r^2 mod N
4:     N_\text{rev} ← \text{flipEndianness}(N)
5:     r^{-1} ← \text{modInverse}(r, N_\text{rev})
6:     found_first_one_bit ← false
7:     for i ∈ \{bitlen(e) - 1 .. 0\} do
8:         if found_first_one_bit then
9:             x ← \text{montMult}(x, x, N_\text{rev}, r^{-1})
10:                if e[i] == 1 then
11:                    x ← \text{montMult}(x, a, N_\text{rev}, r^{-1})
12:                end if
13:         else if e[i] == 1 then
14:             S_\text{rev} ← \text{flipEndianness}(S)
15:             x ← \text{montMult}(S_\text{rev}, R, N_\text{rev}, r^{-1})
16:             a ← x
17:                found_first_one_bit ← true
18:         end if
19:     end for
20:     x ← \text{montMult}(x, 1, N_\text{rev}, r^{-1})
21:     H ← \text{flipEndianness}(x)
22:     return H
23: end procedure
```

RsaDecrypt( S , N , e )

Reverse engineering approaches:
- Statically via IDA
- Dynamic code instrumentation on Trustzone code on actual phones (more details in future!)
Digging deep into the firmware

$$\text{RsaDecrypt}(S, N, e)$$

- Computes modular exponentiation: $$S^e \mod N$$
- Implemented with an efficient form of multiplication called Montgomery Multiplication\[1\]
  $$\text{MONTMULT}(x, y, N, r^{-1}) \leftarrow x \cdot y \cdot r^{-1} \mod N$$
- Uses a memory-intensive function that reverses memory buffers
  $$S_{rev} \leftarrow \text{FLIPENDIANNESS}(S)$$

Digging deep into the firmware

1: procedure DecryptSig(S, e, N)
2:     r ← 2**2648
3:     R ← r^2 mod N
4:     N_{rev} ← FlipEndianness(N)
5:     found_first_one_bit ← false
6:     for i ∈ \{bitlen(e) − 1 .. 0\} do
7:         if found_first_one_bit then
8:             x ← MontMult(x, x, N_{rev}, r^{-1})
9:             if e[i] == 1 then
10:                x ← MontMult(x, a, N_{rev}, r^{-1})
11:         end if
12:     else if e[i] == 1 then
13:         S_{rev} ← FlipEndianness(S)
14:         x ← MontMult(S_{rev}, R, N_{rev}, r^{-1})
15:         a ← x
16:         found_first_one_bit ← true
17:     end if
18:     end for
19:     x ← MontMult(x, 1, N_{rev}, r^{-1})
20:     H ← FlipEndianness(x)
21:     return H
22: end procedure

Where to inject the runtime fault?

- Computes modular exponentiation:
  \( S^e \mod N \)

- Implemented with an efficient form of multiplication called Montgomery Multiplication\[1\]
  \( \text{MontMult}(x, y, N, r^{-1}) \leftarrow x \cdot y \cdot r^{-1} \mod N \)

- Uses a memory-intensive function that reverses memory buffers

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

---

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
```
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: … bc094a4a …

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

Code: https://github.com/0x0atang/clkscrew/blob/master/faultmin_SD805/
Digging deep into the firmware

\[ \text{RSADecrypt}(S, \text{N}, e) \]

signature
modulus
public exponent

\( \text{Computes modular exponentiation:} \quad S^e \mod N \)


Where to inject the runtime fault?

How to craft attack signature \( S_A' \)?

\[ \text{DecryptSig}(S_A', e, N) \xrightarrow{\text{fault}} H(C_A) \]

How to craft attack signature?

```
1: procedure DECRYPTSIG(S, e, N)
2:     s ← 2^2648
3:     R ← r^2 mod N
4:     N_{rev} ← FLIPENDIANNESS(N)
5:     r ← 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 ← FLIPENDIANNESS(S)
15:        end if
16:     x ← MONTMULT(S_{rev}, R, N_{rev}, r^{-1})
17:     a ← x
18:     found_first_one_bit ← true
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!!

Line 3: \( R \leftarrow r^2 \mod N \)

Line 4: \( N_{A,rev} \leftarrow \text{FLIPENDIANNESS}(N) \)

Line 15: \( \text{MONTMULT}(S'_A, r^2 \mod N, N_A, r_A^{-1}) = S_A' \cdot (r^2 \mod N) \cdot r_A^{-1} \mod N_A \)

(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
\[^{3}\]\ 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

Cache set eviction profile (Validation of the 4th RSA signature)

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 set eviction profile (Validation of the 4th RSA signature)

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:

$$F_{pos} \sim \text{feat}_{cache1} + \text{feat}_{cache2} + F_{pdelay} + \text{temperature} + \text{intercept}$$

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

$$F_{pdelay} = f \left( F_{pos}, \text{feat}_{cache1}, \text{feat}_{cache2}, \text{temperature}, \text{intercept} \right)$$
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.
Possible Defenses

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

Software-Level
- Randomization
- Code execution redundancy
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
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