CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management

Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo, ***Columbia University***

https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang

This paper is included in the Proceedings of the 26th USENIX Security Symposium

August 16–18, 2017 • Vancouver, BC, Canada

ISBN 978-1-931971-40-9

Open access to the Proceedings of the 26th USENIX Security Symposium is sponsored by USENIX

CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management

Adrian Tang
Columbia University

Simha Sethumadhavan
Columbia University

Salvatore Stolfo
Columbia University

Abstract

The need for power- and energy-efficient computing has resulted in aggressive cooperative hardware-software energy management mechanisms on modern commodity devices. Most systems today, for example, allow software to control the frequency and voltage of the underlying hardware at a very fine granularity to extend battery life. Despite their benefits, these software-exposed energy management mechanisms pose grave security implications that have not been studied before.

In this work, we present the CLKSCREW attack, a new class of fault attacks that exploit the security-obliviousness of energy management mechanisms to break security. A novel benefit for the attackers is that these fault attacks become more accessible since they can now be conducted without the need for physical access to the devices or fault injection equipment. We demonstrate CLKSCREW on commodity ARM/Android devices. We show that a malicious kernel driver (1) can extract secret cryptographic keys from Trustzone, and (2) can escalate its privileges by loading self-signed code into Trustzone. As the first work to show the security ramifications of energy management mechanisms, we urge the community to re-examine these security-oblivious designs.

1 Introduction

The growing cost of powering and cooling systems has made energy management an essential feature of most commodity devices today. Energy management is crucial for reducing cost, increasing battery life, and improving portability for systems, especially mobile devices. Designing effective energy management solutions, however, is a complex task that demands cross-stack design and optimizations: Hardware designers, system architects, and kernel and application developers have to coordinate their efforts across the entire hardware/software system stack to minimize energy consumption and maximize performance. Take as an example, Dynamic Voltage and Frequency Scaling (DVFS) [47], a ubiquitous energy management technique that saves energy by regulating the frequency and voltage of the processor cores according to runtime computing demands. To support DVFS, at the hardware level, vendors have to design the underlying frequency and voltage regulators to be portable across a wide range of devices while ensuring cost efficiency. At the software level, kernel developers need to track and match program demands to operating frequency and voltage settings to minimize energy consumption for those demands. Thus, to maximize the utility of DVFS, hardware and software function cooperatively and at very fine granularities.

Despite the ubiquity of energy management mechanisms on commodity systems, security is rarely a consideration in the design of these mechanisms. In the absence of known attacks, given the complexity of hardware-software interoperability needs and the pressure of cost and time-to-market concerns, the designers of these mechanisms have not given much attention to the security aspects of these mechanisms; they have been focused on optimizing the functional aspects of energy management. These combination of factors along with the pervasiveness of these mechanisms makes energy management mechanisms a potential source of security vulnerabilities and an attractive target for attackers.

In this work, we present the first security review of a widely-deployed energy management technique, DVFS. Based on careful examination of the interfaces between hardware regulators and software drivers, we uncover a new class of exploitation vector, which we term as CLKSCREW. In essence, a CLKSCREW attack exploits unfettered software access to energy management hardware to push the operating limits of processors to the point of inducing faulty computations. This is dangerous when these faults can be induced from lower privileged software across hardware-enforced boundaries, where security sensitive computations are hosted.

We demonstrate that CLKSCREW can be conducted using no more than the software control of energy management hardware regulators in the target devices. CLKSCREW is more powerful than traditional physical fault attacks [19] for several reasons. Firstly, unlike physical fault attacks, CLKSCREW enables fault attacks to be conducted purely from software. Remote exploitation with CLKSCREW becomes possible without the need for physical access to target devices. Secondly, many equipment-related barriers, such as the need for soldering and complex equipment, to achieve physical fault attacks are removed. Lastly, since physical attacks have been known for some time, several defenses, such as special hardened epoxy and circuit chips that are hard to access, have been designed to thwart such attacks. Extensive hardware reverse engineering may be needed to determine physical pins on the devices to connect the fault injection circuits [45]. CLKSCREW sidesteps all these risks of destroying the target devices permanently.

To highlight the practical security impact of our attack, we implement the CLKSCREW attack on a commodity ARMv7[1] phone, Nexus 6. With only publicly available knowledge of the Nexus 6 device, we identify the operating limits of the frequency and voltage hardware mechanisms. We then devise software to enable the hardware to operate beyond the vendor-recommended limits. Our attack requires no further access beyond a malicious kernel driver. We show how the CLKSCREW attack can subvert the hardware-enforced isolation in ARM Trustzone in two attack scenarios: (1) extracting secret AES keys embedded within Trustzone and (2) loading self-signed code into Trustzone. We note that the root cause for CLKSCREW is neither a hardware nor a software bug: CLKSCREW is achievable due to the fundamental design of energy management mechanisms.

We have responsibly disclosed the vulnerabilities identified in this work to the relevant SoC and device vendors. They have been very receptive to the disclosure. Besides acknowledging the highlighted issues, they were able to reproduce the reported fault on their internal test device within three weeks of the disclosure. They are working towards mitigations.

In summary, we make the following contributions in this work:

1. We expose the dangers of designing energy management mechanisms without security in mind by introducing the concept of the CLKSCREW attack. Aggressive energy-aware computing mechanisms can be exploited to influence isolated computing.
2. We present the CLKSCREW attack to demonstrate a new class of energy management-based exploitation vector that exploits software-exposed frequency and voltage hardware regulators to subvert trusted computation.
3. We introduce a methodology for examining and demonstrating the feasibility of the CLKSCREW attack against commodity ARM devices running a full complex OS such as Android.
4. We demonstrate that the CLKSCREW attack can be used to break the ARM Trustzone by extracting secret cryptographic keys and loading self-signed applications on a commodity phone.

The remainder of the paper is organized as follows. We provide background on DVFS and its associated hardware and software support in § 2. In § 3, we detail challenges and steps we take to achieving the first CLKSCREW fault. Next, we present two attack case studies in § 4 and § 5. Finally, we discuss countermeasures and related work in § 6, and conclude in § 7.

2 Background

In this section, we provide the required background in energy management to understand CLKSCREW. We first describe DVFS and how it relates to saving energy. We then detail key classes of supporting hardware regulators and their software-exposed interfaces.

2.1 Dynamic Voltage & Frequency Scaling

DVFS is an energy management technique that trades off processing speed for energy savings. Since its debut in 1994 [60], DVFS has become ubiquitous in almost all commodity devices. DVFS works by regulating two important runtime knobs that govern the amount of energy consumed in a system – frequency and voltage.

To see how managing frequency and voltage can save energy, it is useful to understand how energy consumption is affected by these two knobs. The amount of energy[2] consumed in a system is the product of power and time, since it refers to the total amount of resources utilized by a system to complete a task over time. Power[3], an important determinant of energy consumption, is directly proportional to the product of operating frequency and voltage. Consequently, to save energy, many energy management techniques focus on efficiently optimizing both frequency and voltage.

[1] As of Sep 2016, ARMv7 devices capture over 86% of the worldwide market share of mobile phones [7].

[2] Formally, the total amount of energy consumed, E_T, is the integral of instantaneous dynamic power, P_t over time T: $E_T = \int_0^T P_t \, dt$.

[3] In a system with a fixed capacitative load, at any time t, the instantaneous dynamic power is proportional to both the voltage, V_t and the frequency F_t as follows: $P_t \propto V_t^2 \times F_t$.

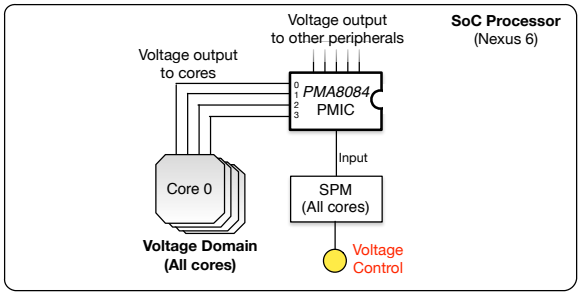

Figure 1: Shared voltage regulator for all *Krait* cores.

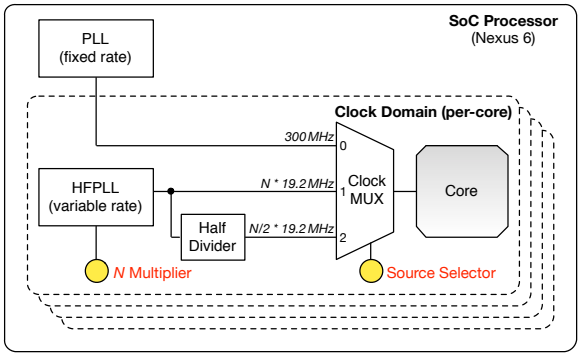

Figure 2: Separate clock sources for each *Krait* core.

DVFS regulates frequency and voltage according to runtime task demands. As these demands can vary drastically and quickly, DVFS needs to be able to track these demands and effect the frequency and voltage adjustments in a timely manner. To achieve this, DVFS requires components across layers in the system stack. The three primary components are (1) the voltage/frequency hardware regulators, (2) vendor-specific regulator driver, and (3) OS-level *CPUfreq* power governor [46]. The combined need for accurate layer-specific feedback and low voltage/frequency scaling latencies drives the prevalence of unfettered and software-level access to the frequency and voltage hardware regulators.

2.2 Hardware Support for DVFS

Voltage Regulators. Voltage regulators supply power to various components on devices, by reducing the voltage from either the battery or external power supply to a range of smaller voltages for both the cores and the peripherals within the device. To support features, such as camera and sensors that are sourced from different vendors and hence operating at different voltages, numerous voltage regulators are needed on devices. These regulators are integrated within a specialized circuit called Power Management Integrated Circuit (PMIC) [53].

Power to the application cores is typically supplied by the step-down regulators within the PMIC on the System-on-Chip (SoC) processor. As an example, Figure 1 shows the PMIC that regulates the shared voltage supply to all the application cores (*a.k.a. Krait* cores) on the Nexus 6 device. The PMIC does not directly expose software interfaces for controlling the voltage supply to the cores. Instead, the core voltages are indirectly managed by a power management subsystem, called the Subsystem Power Manager (SPM) [2]. The SPM is a hardware block that maintains a set of control registers which, when configured, interfaces with the PMIC to effect voltage changes. Privileged software like a kernel driver can use these memory-mapped control registers to direct voltage changes. We highlight these software-exposed controls as yellow-shaded circles in Figure 1.

Frequency PLL-based Regulators. The operating frequency of application cores is derived from the frequency of the clock signal driving the underlying digital logic circuits. The frequency regulator contains a Phase Lock Loop (PLL) circuit, a frequency synthesizer built into modern processors to generate a synchronous clock signal for digital components. The PLL circuit generates an output clock signal of adjustable frequency, by receiving a fixed-rate reference clock (typically from a crystal oscillator) and raising it based on an adjustable multiplier ratio. The output clock frequency can then be controlled by changing this PLL multiplier.

For example, each core on the Nexus 6 has a dedicated clock domain. As such, the operating frequency of each core can be individually controlled. Each core can operate on three possible clock sources. In Figure 2, we illustrate the clock sources as well as the controls (shaded in yellow) exposed to the software from the hardware regulators. A multiplexer (*MUX*) is used to select amongst the three clock sources, namely (1) a PLL supplying a fixed-rate 300-MHz clock signal, (2) a High-Frequency PLL (HFPLL) supplying a clock signal of variable frequency based on a *N multiplier*, and (3) the same HFPLL supplying half the clock signal via a frequency divider for finer-grained control over the output frequency.

As shown in Figure 2, the variable output frequency of the HFPLL is derived from a base frequency of 19.2MHz and can be controlled by configuring the *N multiplier*. For instance, to achieve the highest core operating frequency of 2.65GHz advertised by the vendor, one needs to configure the *N multiplier* to 138 and the *Source Selector* to 1 to select the use of the full HFPLL. Similar to changing voltage, privileged software can initiate per-core frequency changes by writing to software-exposed memory-mapped PLL registers, shown in Figure 2.

2.3 Software Support for DVFS

On top of the hardware regulators, additional software support is needed to facilitate DVFS. Studying these supporting software components for DVFS enables us to better understand the interfaces provided by the hardware regulators. Software support for DVFS comprises two key components, namely vendor-specific regulator drivers and OS-level power management services.

Besides being responsible for controlling the hardware regulators, the vendor-provided PMIC drivers [5, 6] also provide a convenient means for mechanisms in the upper layers of the stack, such as the Linux *CPUfreq* power governor [46] to dynamically direct the voltage and frequency scaling. DVFS requires real-time feedback on the system workload profile to guide the optimization of performance with respect to power dissipation. This feedback may rely on layer-specific information that may only be efficiently accessible from certain system layers. For example, instantaneous system utilization levels are readily available to the OS kernel layer. As such, the Linux *CPUfreq* power governor is well-positioned at that layer to initiate runtime changes to the operating voltage and frequency based on these whole-system measures. This also provides some intuition as to why DVFS cannot be implemented entirely in hardware.

3 Achieving the First CLKSCREW Fault

In this section, we first briefly describe why erroneous computation occurs when frequency and voltage are stretched beyond the operating limits of digital circuits. Next, we outline challenges in conducting a non-physical probabilistic fault injection attack induced from software. Finally, we characterize the operating limits of regulators and detail the steps to achieving the first CLKSCREW fault on a real device.

3.1 How Timing Faults Occur

To appreciate why unfettered access to hardware regulators is dangerous, it is necessary to understand in general why over-extending frequency (*a.k.a.* overclocking) or under-supplying voltage (*a.k.a.* undervolting) can cause unintended behavior in digital circuits.

Synchronous digital circuits are made up of memory elements called flip-flops (FF). These flip-flops store stateful data for digital computation. A typical flip-flop has an input D, and an output Q, and only changes the output to the value of the input upon the receipt of the rising edge of the clock (CLK) signal. In Figure 3, we show two flip-flops, FF_{src} and FF_{dst} sharing a common clock signal and some intermediate combinatorial

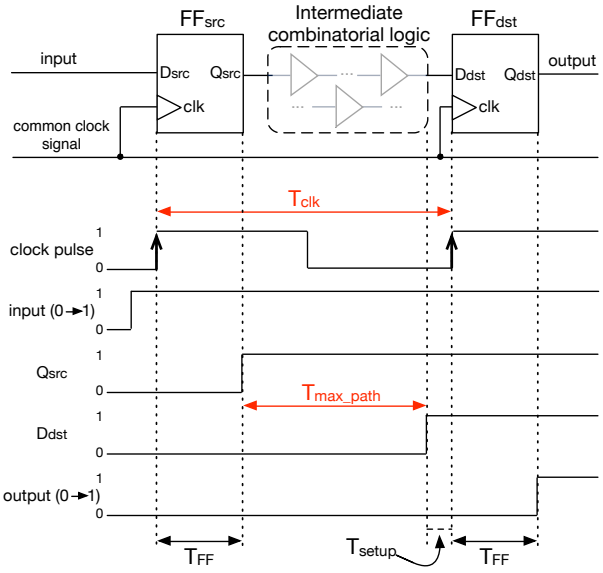

Figure 3: Timing constraint for error-free data propagation from input Q_{src} to output D_{dst} for entire circuit.

logic elements. These back-to-back flip-flops are building blocks for pipelines, which are pervasive throughout digital chips and are used to achieve higher performance.

Circuit timing constraint. For a single flip-flop to properly propagate the input to the output locally, there are three key timing sub-constraints. (1) The incoming data signal has to be held stable for T_{setup} during the receipt of the clock signal, and (2) the input signal has to be held stable for T_{FF} within the flip-flop after the clock signal arrives. (3) It also takes a minimum of $T_{\text{max\_path}}$ for the output Q_{src} of FF_{src} to propagate to the input D_{dst} of FF_{dst}. For the overall circuit to propagate input $D_{\text{src}} \rightarrow$ output Q_{dst}, the minimum required clock cycle period[4], T_{clk}, is bounded by the following timing constraint (1) for some microarchitectural constant K:

$$T_{\text{clk}} \geq T_{\text{FF}} + T_{\text{max\_path}} + T_{\text{setup}} + K \qquad (1)$$

Violation of timing constraint. When the timing constraint is violated during two consecutive rising edges of the clock signal, the output from the source flip-flop FF_{src} fails to latch properly in time as the input at the destination flip-flop FF_{dst}. As such, the FF_{dst} continues to operate with stale data. There are two situations where this timing constraint can be violated, namely (a) overclocking to reduce T_{clk} and (b) undervolting to increase the overall circuit propagation time, thereby increasing $T_{\text{max\_path}}$. Figure 4 illustrates how the output results in an unintended erroneous value of 0 due to overclocking. For comparison, we show an example of a bit-level fault due to undervolting in Figure 15 in Appendix A.1.

[4] T_{clk} is simply the reciprocal of the clock frequency.

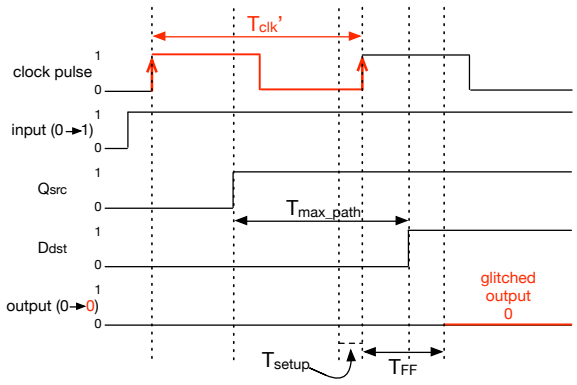

Figure 4: Bit-level fault due to overclocking: Reducing clock period $T_{\text{clk}} \rightarrow T_{\text{clk}}'$ results in a bit-flip in output $1 \rightarrow 0$.

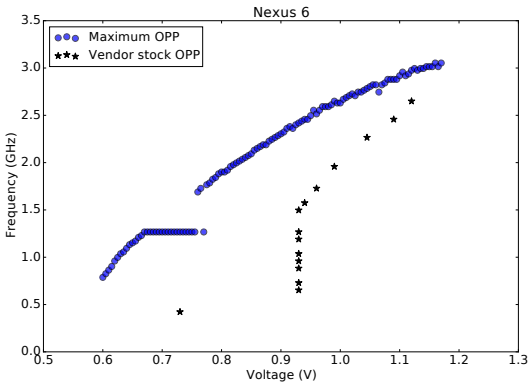

Figure 5: Vendor-stipulated voltage/frequency Operating Performance Points (OPPs) *vs.* maximum OPPs achieved before computation fails.

3.2 Challenges of CLKSCREW Attacks

Mounting a fault attack purely from software on a real-world commodity device using its internal voltage/frequency hardware regulators has numerous difficulties. These challenges are non-existent or vastly different from those in traditional physical fault attacks (that commonly use laser, heat and radiation).

Regulator operating limits. Overclocking or undervolting attacks require the hardware to be configured *far beyond* its vendor-suggested operating range. Do the operating limits of the regulators enable us to effect such attacks in the first place? We show that this is feasible in § 3.3.

Self-containment within same device. Since the attack code performing the fault injection and the victim code to be faulted both reside on the same device, the fault attack must be conducted in a manner that does not affect the execution of the attacking code. We present techniques to overcome this in § 3.4.

Noisy complex OS environment. On a full-fledged OS with interrupts, we need to inject a fault into the target code without causing too much perturbation to non-targeted code. We address this in § 3.4.

Precise timing. To attack the victim code, we need to be relatively precise in *when* the fault is induced. Using two attack scenarios that require vastly different degrees of timing precision in § 4 and § 5, we demonstrate how the timing of the fault can be fine-tuned using a range of execution profiling techniques.

Fine-grained timing resolution. The fault needs to be *transient* enough to occur during the intended region of victim code execution. We may need the ability to target a specific range of code execution that takes orders of magnitude fewer clock cycles within an entire operation. For example, in the attack scenario described in Section § 5.3, we seek to inject a fault into a memory-specific operation that takes roughly 65,000 clock cycles within an entire RSA certificate chain verification operation spanning over 1.1 billion cycles.

3.3 Characterization of Regulator Limits

In this section, we study the capabilities and limits of the built-in hardware regulators, focusing on the Nexus 6 phone. According to documentation from the vendor, Nexus 6 features a 2.7GHz quad-core SoC processor. On this device, DVFS is configured to operate only in one of 15 possible discrete[5] Operating Performance Points (OPPs) at any one time, typically by a DVFS OS-level service. Each OPP represents a state that the device can be in with a voltage and frequency pair. These OPPs are readily available from the vendor-specific definition file, `apq8084.dtsi`, from the kernel source code [3].

To verify that the OPPs are as advertised, we need measurement readings of the operating voltage and frequency. By enabling the *debugfs* feature for the regulators, we can get per-core voltage[6] and frequency[7] measurements. We verify that the *debugfs* measurement readings indeed match the voltage and frequency pairs stipulated by each OPP. We plot these vendor-provided OPP measurements as black-star symbols in Figure 5.

No safeguard limits in hardware. Using the software-exposed controls described in § 2.2, while maintaining a low base frequency of 300MHz, we configure the voltage regulator to probe for the range during which the de-

[5] A limited number of discrete OPPs, instead of a range of continuous voltage/frequency values, is used so that the time taken to validate the configured OPPs at runtime is minimized.

[6] `/d/regulator/krait`*X*`/voltage`

[7] `/d/clk/krait`*X*`_clk/measure`

vice remains functional. We find that when the device is set to any voltage outside the range 0.6V to 1.17V, it either reboots or freezes. We refer to the phone as being *unstable* when these behaviors are observed. Then, stepping through 5mV within the voltage range, for each operating voltage, we increase the clock frequency until the phone becomes *unstable*. We plot each of these maximum frequency and voltage pair (as shaded circles) together with the vendor-stipulated OPPs (as shaded stars) in Figure 5. It is evident that the hardware regulators can be configured past the vendor-recommended limits. This unfettered access to the regulators offers a powerful primitive to induce a software-based fault.

> **ATTACK ENABLER (GENERAL) #1**: There are no safeguard limits in the hardware regulators to restrict the range of frequencies and voltages that can be configured.

Large degree of freedom for attacker. Figure 5 illustrates the degree of freedom an attacker has in choosing the OPPs that have the potential to induce faults. The maximum frequency and voltage pairs (*i.e.* shaded circles in Figure 5) form an almost continuous upward-sloping curve. It is noteworthy that all frequency and voltage OPPs *above* this curve represent potential candidate values of frequency and voltage that an attacker can use to induce a fault.

This "shaded circles" curve is instructive in two ways. First, from the attacker's perspective, the upward-sloping nature of the curve means that reducing the operating voltage simultaneously lowers the minimum required frequency needed to induce a fault in an attack. For example, suppose an attacker wants to perform an overclocking attack, but the frequency value she needs to achieve the fault is beyond the physical limit of the frequency regulator. With the help of this frequency/voltage characteristic, she can then possibly reduce the operating voltage to the extent where the overclocking frequency required is within the physical limit of the regulator.

> **ATTACK ENABLER (GENERAL) #2**: Reducing the operating voltage lowers the minimum required frequency needed to induce faults.

Secondly, from the defender's perspective, the large range of instability-inducing OPPs above the curve suggests that limits of both frequency and voltage, if any, must be enforced *in tandem* to be effective. Combination of frequency and voltage values, while individually valid, may still cause unstable conditions when used together.

Prevalence of Regulators. The lack of safeguard limits within the regulators is not specific to Nexus 6. We observe similar behaviors in devices from other vendors. For example, the frequency/voltage regulators in the Nexus 6P and Pixel phones can also be configured beyond their vendor-stipulated limits to the extent of seeing instability on the devices. We show the comparison of the vendor-recommended and the actual observed OPPs of these devices in Figures 16 and 17 in Appendix A.3.

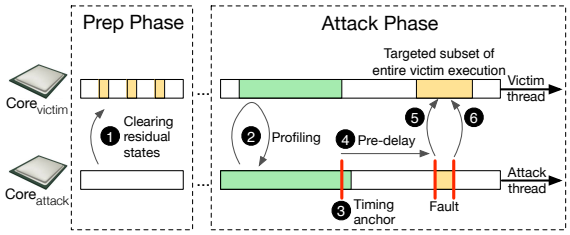

Figure 6: Overview of CLKSCREW fault injection setup.

3.4 Containing the Fault within a Core

The goal of our fault injection attack is to induce errors to specific victim code execution. The challenge is doing so without self-faulting the attack code and accidentally attacking other non-targeted code.

We create a custom kernel driver to launch separate threads for the attack and victim code and to pin each of them to separate cores. Pinning the attack and victim code in separate cores automatically allows each of them to execute in different frequency domains. This core pinning strategy is possible due to the deployment of increasingly heterogeneous processors like the ARM big.LITTLE [12] architecture, and emerging technologies such as Intel PCPS [35] and Qualcomm aSMP [48]. The prevailing industry trend of designing finer-grained energy management favors the use of separate frequency and voltage domains across different cores. In particular, the Nexus 6 SoC that we use in our attack is based on a variant of the aSMP architecture. With core pinning, the attack code can thus manipulate the frequency of the core that the victim code executes on, without affecting that of the core the attack code is running on. In addition to core pinning, we also disable interrupts during the entire victim code execution to ensure that no context switch occurs for that core. These two measures ensure that our fault injection effects are contained within the core that the target victim code is running on.

> **ATTACK ENABLER (GENERAL) #3**: The deployment of cores in different voltage/frequency domains isolates the effects of cross-core fault attack.

3.5 CLKSCREW Attack Steps

The CLKSCREW attack is implemented with a kernel driver to attack code that is executing at a higher priv-

Parameter	Description
F_{volt}	Base operating voltage
F_{pdelay}	Number of loops to delay/wait before the fault
F_{freq_hi}	Target value to raise the frequency *to* for the fault
F_{freq_lo}	Base value to raise the frequency *from* for the fault
F_{dur}	Duration of the fault in terms of number of loops

Table 1: CLKSCREW fault injection parameters.

ilege than the kernel. Examples of such victim code are applications running within isolation technologies such as ARM Trustzone [11] and Intel SGX [9]. In Figure 6, we illustrate the key attack steps within the thread execution of the attack and victim code. The goal of the CLKSCREW attack is to induce a fault in a subset of an entire victim thread execution.

❶ **Clearing residual states.** Before we attack the victim code, we want to ensure that there are no microarchitectural residual states remaining from prior executions. Since we are using a cache-based profiling technique in the next step, we want to make sure that the caches do not have any residual data from non-victim code before each fault injection attempt. To do so, we invoke both the victim and attack threads in the two cores multiple times in quick succession. From experimentation, 5-10 invocations suffice in this preparation phase.

❷ / ❸ **Profiling for an anchor.** Since the victim code execution is typically a subset of the entire victim thread execution, we need to profile the execution of the victim thread to identify a consistent point of execution *just* before the target code to be faulted. We refer to this point of execution as a timing anchor, T_{anchor} to guide when to deliver the fault injection. Several software profiling techniques can be used to identify this timing anchor. In our case, we rely on instruction or data cache profiling techniques in recent work [40].

❹ **Pre-fault delaying.** Even with the timing anchor, in some attack scenarios, there may still be a need to fine-tune the exact delivery timing of the fault. In such cases, we can configure the attack thread to spin-loop with a predetermined number of loops before inducing the actual fault. The use of these loops consisting of *no-op* operations is essentially a technique to induce timing delays with high precision. For this stage of the attack, we term this delay before inducing the fault as F_{pdelay}.

❺ / ❻ **Delivering the fault.** Given a base operating voltage F_{volt}, the attack thread will raise the frequency of the victim core (denoted as F_{freq_hi}), keep that frequency for F_{dur} loops, and then restore the frequency to F_{freq_lo}.

To summarize, for a successful CLKSCREW attack, we can characterize the attacker's goal as the following sub-tasks. Given a victim code and a fault injection target point determined by T_{anchor}, the attacker has to find optimal values for the following parameters to maximize the odds of inducing the desired fault. We summarize the fault injection parameters required in Table 1.

$$F_{\theta|T_{anchor}} = \{F_{volt}, F_{pdelay}, F_{freq\_hi}, F_{dur}, F_{freq\_lo}\}$$

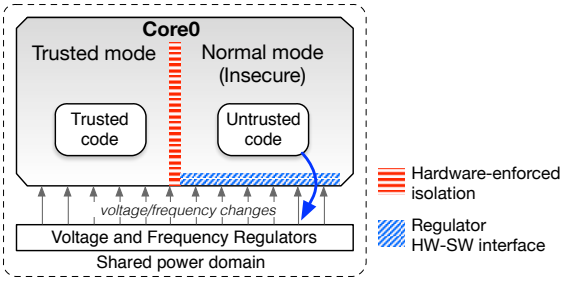

Figure 7: Regulators operate across security boundaries.

3.6 Isolation-Agnostic DVFS

To support execution of trusted code isolated from untrusted one, two leading industry technologies, ARM Trustzone [11] and Intel SGX [9], are widely deployed. They share a common characteristic in that they can execute both trusted and untrusted code on the *same* physical core, while relying on architectural features such as specialized instructions to support isolated execution. It is noteworthy that on such architectures, the voltage and frequency regulators typically operate on domains that apply to cores as a whole (regardless of the security-sensitive processor execution modes), as depicted in Figure 7. With this design, any frequency or voltage change initiated by untrusted code inadvertently affects the trusted code execution, despite the hardware-enforced isolation. This, as we show in subsequent sections, poses a critical security risk.

> **ATTACK ENABLER (GENERAL) #4**: Hardware regulators operate across security boundaries with no physical isolation.

4 TZ Attack #1: Inferring AES Keys

In this section, we show how AES [43] keys stored within Trustzone (TZ) can be inferred by lower-privileged code from outside Trustzone, based on the faulty ciphertexts derived from the erroneous AES encryption operations. Specifically, it shows how lower-privileged code can subvert the isolation guarantee by ARM Trustzone, by influencing the computation of higher-privileged code using the energy management

mechanisms. The attack shows that the confidentiality of the AES keys that should have been kept secure in Trustzone can be broken.

Threat model. In our victim setup, we assume that there is a Trustzone app that provisions AES keys and stores these keys within Trustzone, inaccessible from the non-Trustzone (non-secure) environment. The attacker can repeatedly invoke the Trustzone app from the non-secure environment to decrypt any given ciphertext, but is restricted from reading the AES keys directly from Trustzone memory due to hardware-enforced isolation. The attacker's goal is to infer the AES keys stored.

4.1 Trustzone AES Decryption App

For this case study, since we do not have access to a real-world AES app within Trustzone, we rely on a textbook implementation of AES as the victim app. We implement a AES decryption app that can be loaded within Trustzone. Without loss of generality, we restrict the decryption to 128-bit keys, operating on 16-bit plaintext and ciphertext. A single 128-bit encryption/decryption operation comprises 10 AES rounds, each of which is a composition of the four canonical sub-operations, named `SubBytes`, `ShiftRows`, `MixColumns` and `AddRoundKey` [43].

To load this app into Trustzone as our victim program, we use a publicly known Trustzone vulnerability [17] to overwrite an existing Trustzone syscall handler, `tzbsp_es_is_activated`, on our Nexus 6 device running an old firmware[8]. A non-secure app can then execute this syscall via an ARM Secure Monitor Call [26] instruction to invoke our decryption Trustzone app. This vulnerability serves the sole purpose of allowing us to load the victim app within Trustzone to simulate a AES decryption app in Trustzone. It plays no part in the attacker's task of interest – extracting the cryptographic keys stored within Trustzone. Having the victim app execute within Trustzone on a commodity device allows us to evaluate CLKSCREW across Trustzone-enforced security boundaries in a practical and realistic manner.

4.2 Timing Profiling

As described in § 3.5, one of the crucial attack steps to ensure reliable delivery of the fault to a victim code execution is finding ideal values of F_{pdelay}. To guide this parameter discovery process, we need the timing profile of the Trustzone app performing a single AES encryption/decryption operation. ARM allows the use of hardware cycle counter (CCNT) to track the execution duration (in clock cycles) of Trustzone applications [10]. We enable this cycle counting feature within our custom kernel driver. With this feature, we can now measure how long it takes for our Trustzone app to decrypt a single ciphertext, even from the non-secure world.

[8] Firmware version is *shamu* `MMB29Q` (Feb, 2016)

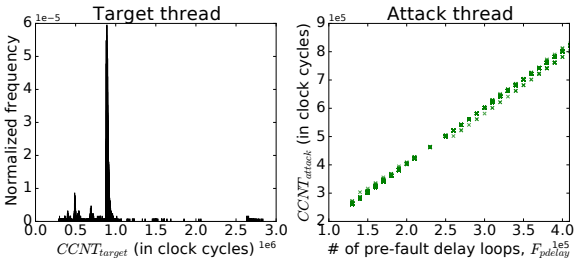

Figure 8: Execution duration (in clock cycles) of the victim and attack threads.

> **ATTACK ENABLER (TZ-SPECIFIC) #5**: Execution timing of code running in Trustzone can be profiled with hardware counters that are accessible outside Trustzone.

Using the hardware cycle counter, we track the duration of each AES decryption operation over about 13k invocations in total. Figure 8 (left) shows the distribution of the execution length of an AES operation. Each operation takes an average of 840k clock cycles with more than 80% of the invocations taking between 812k to 920k cycles. This shows that the victim thread does not exhibit too much variability in terms of its execution time.

Recall that we want to deliver a fault to specific region of the victim code execution and that the faulting parameter F_{pdelay} allows us to fine-tune this timing. Here, we evaluate the degree to which the use of *no-op* loops is useful in controlling the timing of the fault delivery. Using a fixed duration for the fault F_{dur}, we measure how long the attack thread takes in clock cycles for different values of the pre-fault delays F_{pdelay}. Figure 8 (right) illustrates a distinct linear relationship between F_{pdelay} and the length of the attack thread. This demonstrates that number of loops used in F_{pdelay} is a reasonably good proxy for controlling the execution timing of threads, and thus the timing of our fault delivery.

4.3 Fault Model

To detect if a fault is induced in the AES decryption, we add a check after the app invocation to verify that the decrypted plaintext is as expected. Moreover, to know exactly which AES round got corrupted, we add minimal code to track the intermediate states of the AES round and return this as a buffer back to the non-secure environment. A comparison of the intermediate states and their expected values will indicate the specific AES round that

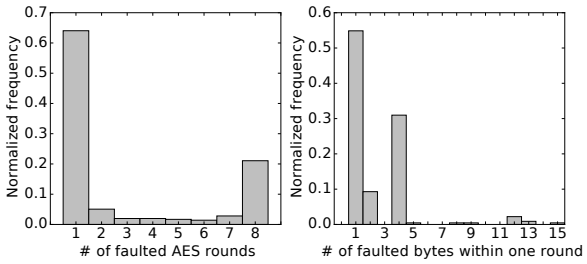

Figure 9: Fault model: Characteristics of observed faults induced by CLKSCREW on AES operation.

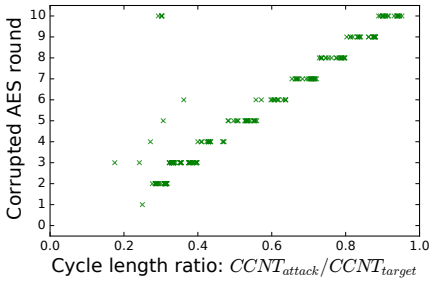

Figure 10: Controlling pre-fault delay, F_{pdelay}, allows us to control which AES round the fault affects.

is faulted and the corrupted value. With these validation checks in place, we perform a grid search for the parameters for the faulting frequency, $F_{\text{freq\_hi}}$ and the duration of the fault, F_{dur} that can induce erroneous AES decryption results. From our empirical trials, we found that the parameters $F_{\text{freq\_hi}} = 3.69GHz$ and $F_{\text{dur}} = 680$ can most reliably induce faults to the AES operation.

For the rest of this attack, we assume the use of these two parameter values. By varying F_{pdelay}, we investigate the characteristics of the observed faults. A total of about 360 faults is observed. More than 60% of the faults are precise enough to affect exactly one AES round, as depicted in Figure 9 (left). Furthermore, out of these faults that induce corruption in one AES round, more than half are sufficiently transient to cause random corruptions of exactly one byte, shown in Figure 9 (right). Being able to induce a one-byte random corruption to the intermediate state of an AES round is often used as a fault model in several physical fault injection works [18, 56].

4.4 Putting it together

Removing use of time anchor. Recall from § 3.5 that CLKSCREW may require profiling for a time anchor to improve faulting precision. In this attack, we choose not to do so, because (1) the algorithm of the AES operation is fairly straightforward (one `KeyExpansion` round, followed by 10 AES rounds [43]) to estimate F_{pdelay}, and (2) the execution duration of the victim thread does not exhibit too much variability. The small degree of variability in the execution timing of both the attack and victim threads allows us to reasonably target specific AES rounds with a maximum error margin of one round.

Differential fault attack. Tunstall *et al.* present a differential fault attack (DFA) that infers AES keys based on pairs of correct and faulty ciphertext [56]. Since AES encryption is symmetric, we leverage their attack to infer AES keys based on pairs of correct and faulty plaintext. Assuming a fault can be injected during the seventh AES round to cause a single-byte random corruption to the intermediate state in that round, with a corrupted input to the eighth AES round, this DFA can reduce the number of AES-128 key hypotheses from the original 2^{128} to 2^{12}, in which case the key can be brute-forced in a trivial exhaustive search. We refer readers to Tunstall *et al.*'s work [56] for a full cryptanalysis for this fault model.

Degree of control of attack. To evaluate the degree of control we have over the specific round we seek to inject the fault in, we induce the faults using a range of F_{pdelay} and track which AES rounds the faults occur in. In Figure 10, each point represents a fault occurring in a specific AES round and when that fault occurs during the entire execution of the victim thread. We use the ratio of $CCNT_{\text{attack}}/CCNT_{\text{target}}$ as an approximation of latter. There are ten distinct clusters of faults corresponding to each AES round. Since $CCNT_{\text{target}}$ can be profiled beforehand and $CCNT_{\text{attack}}$ is controllable via the use of F_{pdelay}, an attacker is able to control which AES round to deliver the fault to for this attack.

Actual attack. Given the faulting parameters, $F_{\theta,\text{ AES-128}} = \{F_{\text{volt}} = 1.055V,\ F_{\text{pdelay}} = 200k,\ F_{\text{freq\_hi}} = 3.69GHz,\ F_{\text{dur}} = 680,\ F_{\text{freq\_lo}} = 2.61GHz\}$, it took, on average, 20 faulting attempts to induce a one-byte fault to the input to the eighth AES round. Given the pair of this faulty plaintext and the expected one, it took Tunstall *et al.*'s DFA algorithm about 12 minutes on a $2.7GHz$ quad-core CPU to generate 3650 key hypotheses, one out of which is the AES key stored within Trustzone.

5 TZ Attack #2: Loading Self-Signed Apps

In this case study, we show how CLKSCREW can subvert the RSA signature chain verification – the primary public-key cryptographic method used for authenticating the loading of firmware images into Trustzone. ARM-based SoC processors use the ARM Trustzone to provide a secure and isolated environment to execute security-critical applications like DRM *widevine* [28] trustlet[9] and

[9] Apps within Trustzone are sometimes referred to as *trustlets*.

Algorithm 1 Given public key modulus N and exponent e, decrypt a RSA signature S. Return plaintext hash, H.

```
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
```

key management *keymaster* [27] trustlet. These vendor-specific firmware are subject to regular updates. These firmware update files consist of the updated code, a signature protecting the hash of the code, and a certificate chain. Before loading these signed code updates into Trustzone, the Trusted Execution Environment (TEE) authenticates the certificate chain and verifies the integrity of the code updates [49].

RSA Signature Validation. In the RSA cryptosystem [51], let N denote the modulus, d denote the private exponent and e denote the public exponent. In addition, we also denote the SHA-256 hash of code C as $H(C)$ for the rest of the section. To ensure the integrity and authenticity of a given code blob C, the code originator creates a signature Sig with its RSA private key: $Sig \leftarrow (H(C))^d \bmod N$. The code blob is then distributed together with the signature and a certificate containing the signing modulus N. Subsequently, the code blob C can be authenticated by verifying that the hash of the code blob matches the plaintext decrypted from the signature using the public modulus N: $Sig^e \bmod N == H(C)$. The public exponent is typically hard-coded to `0x10001`; only the modulus N is of interest here.

Threat model. The goal of the attacker is to provide an arbitrary attack app with a self-signed signature and have the TEE successfully authenticate and load this self-signed app within Trustzone. To load apps into Trustzone, the attackers can invoke the TEE to authenticate and load a given app into Trustzone using the `QSEOS_APP_START_COMMAND` [4] Secure Channel Manager[10] command. The attacker can repeatedly invoke this operation, but only from the non-secure environment.

5.1 Trustzone Signature Authentication

To formulate a CLKSCREW attack strategy, we first examine how the verification of RSA signatures is implemented within the TEE. This verification mechanism is implemented within the bootloader firmware. For the Nexus 6 in particular, we use the *shamu*-specific firmware image (`MOB31S`, dated Jan 2017 [1]), downloaded from the Google firmware update repository.

The RSA decryption function used in the signature verification is the function, DECRYPTSIG[11], summarized in Algorithm 1. At a high level, DECRYPTSIG takes, as input, a 2048-bit signature and the public key modulus, and returns the decrypted hash for verification. For efficient modular exponentiation, DECRYPTSIG uses the function MONTMULT to perform Montgomery multiplication operations [38, 44]. MONTMULT performs Montgomery multiplication of two inputs x and y with respect to the Montgomery *radix*, r [38] and modulus N as follows: $\text{MONTMULT}(x, y, N, r^{-1}) \leftarrow x \cdot y \cdot r^{-1} \bmod N$.

In addition to the use of MONTMULT, DECRYPTSIG also invokes the function, FLIPENDIANNESS[12], multiple times at lines 4, 14 and 21 of Algorithm 1 to reverse the contents of memory buffers. FLIPENDIANNESS is required in this implementation of DECRYPTSIG because the inputs to DECRYPTSIG are big-endian while MONTMULT operates on little-endian inputs. For reference, we outline the implementation of FLIPENDIANNESS in Algorithm 2 in Appendix A.2.

5.2 Attack Strategy and Cryptanalysis

Attack overview. The overall goal of the attack is to deliver a fault during the execution of DECRYPTSIG such that the output of DECRYPTSIG results in the desired hash $H(C_A)$ of our attack code C_A. This operation can be described by Equation 2, where the attacker has to supply an attack signature S_A', and fault the execution of DECRYPTSIG at runtime so that DECRYPTSIG outputs the intended hash $H(C_A)$. For comparison, we also describe the typical decryption operation of the *original* signature S to the hash of the original code blob, C in Equation 3.

$$\textbf{Attack}: \quad \text{DECRYPTSIG}(S_A', e, N) \xrightarrow{fault} H(C_A) \quad (2)$$

$$\textbf{Original}: \quad \text{DECRYPTSIG}(S, e, N) \longrightarrow H(C) \quad (3)$$

[10] This is a vendor-specific interface that allows the non-secure world to communicate with the Trustzone secure world.

[11] DECRYPTSIG loads at memory address `0xFE8643C0`.

[12] FLIPENDIANNESS loads at memory address `0xFE868B20`

For a successful attack, we need to address two questions: (a) At which portion of the runtime execution of DECRYPTSIG($S_A^{'}$, e, N) do we inject the fault? (b) How do we craft $S_A^{'}$ to be used as an input to DECRYPTSIG?

5.2.1 Where to inject the runtime fault?

Target code of interest. The fault should target operations that manipulate the input modulus N, and ideally before the beginning of the modular exponentiation operation. A good candidate is the use of the function FLIPENDIANNESS at Line 4 of Algorithm 1. From experimentation, we find that FLIPENDIANNESS is especially susceptible to CLKSCREW faults. We observe that N can be corrupted to a predictable N_A as follows:

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

Since $N_{A,rev}$ is N_A in reverse byte order, for brevity, we refer to $N_{A,rev}$ as N_A for the rest of the section.

Factorizable N_A. Besides being able to fault N to N_A, another requirement is that N_A must be factorizable. Recall that the security of the RSA cryptosystem depends on the computational infeasibility of factorizing the modulus N into its two prime factors, p and q [21]. This means that with the factors of N_A, we can derive the corresponding keypair $\{N_A, d_A, e\}$ using the Carmichael function in the procedure that is described in Razavi *et al.*'s work [50]. With this keypair $\{N_A, d_A, e\}$, the hash of the attack code C_A can then be signed to obtain the signature of the attack code, $S_A \leftarrow (H(C_A))^{d_A} \bmod N_A$.

We expect the faulted N_A to be likely factorizable due to two reasons: (a) N_A is likely a composite number of more than two prime factors, and (b) some of these factors are small. With sufficiently small factors of up to 60 bits, we use Pollard's ρ algorithm to factorize N_A and find them [42]. For bigger factors, we leverage the Lenstra's Elliptic Curve factorization Method (ECM) that has been observed to factor up to 270 bits [39]. Note that all we need for the attack is to find a *single* N_A that is factorizable and reliably reproducible by the fault.

5.2.2 How to craft the attack signature $S_A^{'}$?

Before we begin the cryptanalysis, we note that the attack signature $S_A^{'}$ (an input to DECRYPTSIG) is *not* the signed hash of the attack code, S_A (private-key encryption of the $H(C_A)$). We use $S_A^{'}$ instead of S_A primarily due to the pecularities of our implementation. Specifically, this is because the operations that follow the injection of the fault also use the parameter values derived before the point of injected fault. Next, we sketch the cryptanalysis of delivering a fault to DECRYPTSIG to show how the desired $S_A^{'}$ is derived, and demonstrate why $S_A^{'}$ is *not trivially* derived the same way as S_A.

Cryptanalysis. The goal is to derive $S_A^{'}$ (as input to DECRYPTSIG) given an expected corrupted modulus N_A, the original vendor's modulus N, and the signature of the attack code, S_A. For brevity, all line references in this section refer to Algorithm 1. The key observation is that after being derived from FLIPENDIANNESS at Line 4, N_{rev} is next used by MONTMULT at Line 15. Line 15 marks the beginning of the modular exponentiation of the input signature, and thus, we focus our analysis here.

First, since we want DECRYPTSIG($S_A^{'}$, e, N) to result in $H(C_A)$ as dictated by Equation 2, we begin by analyzing the invocation of DECRYPTSIG that will lead to $H(C_A)$. If we were to run DECRYPTSIG with inputs S_A and N_A, DECRYPTSIG(S_A, e, N_A) should output $H(C_A)$. Based on the analysis of this invocation of DECRYPTSIG, we can then characterize the output, $x_{desired}$, of the operation at Line 15 of DECRYPTSIG(S_A, e, N_A) with Equation 4. We note that the modular inverse of r is computed based on N_A at Line 5, and so we denote this as r_A^{-1}.

$$x_{desired} \leftarrow S_A \cdot (r^2 \bmod N_A) \cdot r_A^{-1} \bmod N_A \qquad (4)$$

Next, suppose our CLKSCREW fault is delivered in the operation DECRYPTSIG($S_A^{'}$, e, N) such that N is corrupted to N_A at Line 4. We note that while N is faulted to N_A at Line 4, subsequent instructions continue to indirectly use the original modulus N because R is derived based on the uncorrupted modulus N at Line 3. Herein lies the complication. The attack signature $S_A^{'}$ passed into DECRYPTSIG gets converted to the Montgomery representation at Line 15, where *both* moduli are used:

$$x_{fault} \leftarrow \text{MONTMULT}(S_A^{'}, r^2 \bmod N, N_A, r_A^{-1})$$

We can then characterize the output, x_{fault}, of the operation at the same Line 15 of a faulted DECRYPTSIG($S_A^{'}$, e, N) as follows:

$$x_{fault} \leftarrow S_A^{'} \cdot (r^2 \bmod N) \cdot r_A^{-1} \bmod N_A \qquad (5)$$

By equating $x_{fault} = x_{desired}$ (*i.e.* equating results from (4) and (5)), we can reduce the problem to finding $S_A^{'}$ for constants $K = (r^2 \bmod N) \cdot r_A^{-1}$ and $x_{desired}$, such that:

$$S_A^{'} \cdot K \bmod N_A \equiv x_{desired} \bmod N_A$$

Finally, subject to the condition that $x_{desired}$ is divisible[13] by the greatest common divisor of K and N_A, denoted as $gcd(K, N_A)$, we use the Extended Euclidean Algorithm[14] to solve for the attack signature $S_A^{'}$, since there exists a constant y such that $S_A^{'} \cdot K + y \cdot N_A = x_{desired}$. In summary, we show that the attack signature $S_A^{'}$ (to be used as an input to DECRYPTSIG($S_A^{'}$, e, N)) can be derived from N, N_A and S_A.

[13] We empirically observe that $gcd(K, N_A) = 1$ in our experiments, thus making $x_{desired}$ trivially divisible by $gcd(K, N_A)$ for our purpose.

[14] The Extended Euclidean Algorithm is commonly used to compute, besides the greatest common divisor of two integers a and b, the integers x and y where $ax + by = gcd(a, b)$.

5.3 Timing Profiling

Each trustlet app file on the Nexus 6 device comes with a certificate chain of four RSA certificates (and signatures). Loading an app into Trustzone requires validating the signatures of all four certificates [49]. By incrementally corrupting each certificate and then invoking the loading of the app with the corrupted chain, we measure the operation of validating one certificate to take about 270 million cycles on average. We extract the target function FLIPENDIANNESS from the binary firmware image and execute it in the non-secure environment to measure its length of execution. We profile its invocation on a 256-byte buffer (the size of the 2048-bit RSA modulus) to take on average 65k cycles.

To show the feasibility of our attack, we choose to attack the validation of the fourth and final certificate in the chain. This requires a *very precise* fault to be induced within in a 65k-cycle-long targeted period within an entire chain validation operation that takes 270 million x 4 = 1.08 billion cycles, a duration that is four orders of magnitude longer than the targeted period. Due to the degree of precision needed, it is thus crucial to find a way to determine a reliable time anchor (see Steps ❷ / ❸ in § 3.5) to guide the delivery of the fault.

Cache profiling To determine approximately which region of code is being executed during the chain validation at any point in time, we leverage side-channel-based cache profiling attacks that operate across cores. Since we are profiling code execution within Trustzone in a separate core, we use recent advances in the cross-core instruction- and data-based *Prime+Probe*[15] cache attack techniques [31, 40, 62]. We observe that the cross-core profiling of the instruction-cache usage of the victim thread is more reliable than that of the data-cache counterpart. As such, we adapt the instruction-based *Prime+Probe* cache attack for our profiling stage.

Within the victim code, we first identify the code address we want to monitor, and then compute the set of memory addresses that is congruent to the cache set of our monitored code address. Since we are doing instruction-based cache profiling, we need to rely on executing instructions instead of memory read operations. We implement a loop within the fault injection thread to continuously execute dynamically generated dummy instructions in the cache-set-congruent memory addresses (the *Prime* step) and then timing the execution of these instructions (the *Probe* step) using the clock cycle counter. We determine a threshold for the cycle count to indicate that the associated cache lines have been evicted. The eviction patterns of the monitored cache set provides an indication that the monitored code address has been executed.

[15] Another prevalent class of cross-core cache attacks is the *Flush+Reload* [61] cache attacks. We cannot use the *Flush+Reload* technique to profile Trustzone execution because *Flush+Reload* requires being able to map addresses that are shared between Trustzone and the non-secure environment. Trustzone, by design, prohibits that.

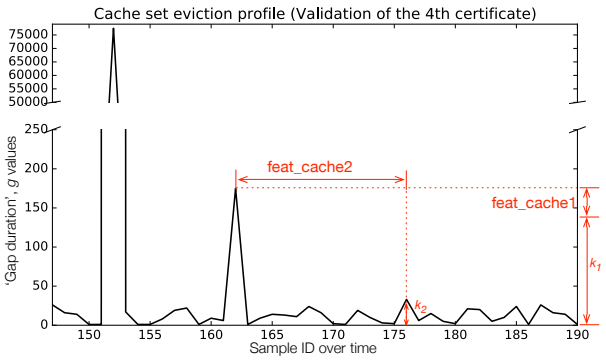

Figure 11: Cache eviction profile snapshot with cache-based features.

> **ATTACK ENABLER (TZ-SPECIFIC) #6**: Memory accesses from the non-secure world can evict cache lines used by Trustzone code, thereby enabling *Prime+Probe*-style execution profiling of Trustzone code.

While we opt to use the *Prime+Probe* cache profiling strategy in our attack, there are alternate side-channel-based profiling techniques that can also be used to achieve the same effect. Other microarchitectural side channels like branch predictors, pipeline contention, prefetchers, and even voltage and frequency side channels can also conceivably be leveraged to profile the victim execution state. Thus, more broadly speaking, the attack enabler #6 is the presence of microarchitectural side channels that allows us to profile code for firing faults.

App-specific timing feature. For our timing anchor, we want a technique that is more fine-grained. We devise a novel technique that uses the features derived from the eviction timing to create a proxy for profiling program phase behavior. First, we maintain a global incrementing count variable as an approximate time counter in the loop. Then, using this counter, we track the duration between consecutive cache set evictions detected by our *Prime+Probe* profiling. By treating this series of *eviction gap duration* values, g, as a time-series stream, we can approximate the execution profile of the chain validation code running within Trustzone.

We plot a snapshot of the cache profile characterizing the validation of the fourth and final certificate in Figure 11. We observe that the beginning of each certification validation is preceded by a large spike of up to 75,000 in the g values followed by a secondary smaller spike. From experimentation, we found that FLIPENDIANNESS runs after the second spike. Based on this obser-

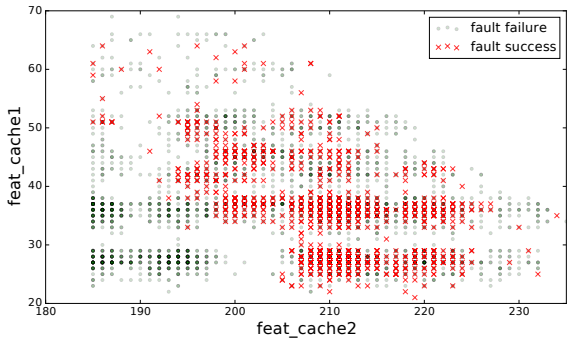

Figure 12: Observed faults using the timing features.

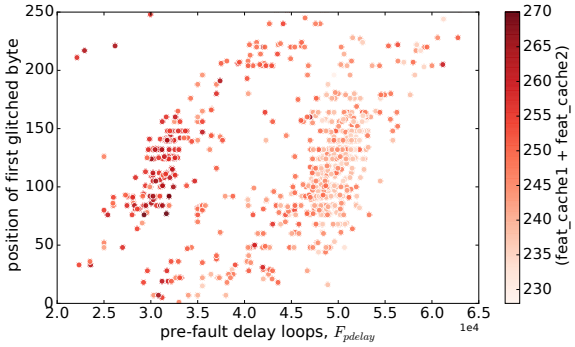

Figure 13: Variability of faulted byte(s) position.

vation, we change the profiling stage of the attack thread to track two hand-crafted timing features to characterize the instantaneous state of victim thread execution.

Timing anchor. We annotate the two timing features on the cache profile plot in Figure 11. The first feature, *feat_cache1*, tracks the length of the second spike minus a constant k_1. The second feature, *feat_cache2*, tracks the cumulative total of g after the second spike, until the $g > k_2$. We use a value of $k_1 = 140$ and $k_2 = 15$ for our experiments. By continuously monitoring values of g after the second spike, the timing anchor is configured to be the point when $g > k_2$.

To evaluate the use of this timing anchor, we need a means to assess when and how the specific invocation of the FLIPENDIANNESS is faulted. First, we observe that the memory buffer used to store N_{rev} is hard-coded to an address `0x0FC8952C` within Trustzone, and this buffer is not zeroed out after the validation of each certificate. We downgrade the firmware version to `MMB29Q` (Feb, 2016), so that we can leverage a Trustzone memory safety violation vulnerability [17] to access the contents of N_{rev} after the fourth certificate in the chain has been validated[16]. Note that this does not affect the normal operation of the chain validation because the relevant code sections for these operations is identical across version `MMB29Q` (Feb, 2016) and `MOB31S` (Jan, 2017).

With this timing anchor, we perform a grid search for the faulting parameters, $F_{\text{freq\_hi}}$, F_{dur} and F_{pdelay} that can best induce faults in FLIPENDIANNESS. The parameters $F_{\text{freq\_hi}} = 3.99GHz$ and $F_{\text{dur}} = 1$ are observed to be able to induce faults in FLIPENDIANNESS reliably. The value of the pre-fault delay parameter F_{pdelay} is crucial in controlling the type of byte(s) corruption in the target memory buffer N_{rev}. With different values of F_{pdelay}, we plot the observed faults and failed attempts based on the values of *feat_cache1* and *feat_cache2* in Figure 12.

[16] We are solely using this vulnerability to speed up the search for the faulting parameters. They can be replaced by more accurate and precise side-channel-based profiling techniques.

Each faulting attempt is considered a success if any bytes within N_{rev} are corrupted during the fault.

Adaptive pre-delay. While we see faults within the target buffer, there is some variability in the position of the fault induced within the buffer. In Figure 13, each value of F_{pdelay} is observed to induce faults across all parts of the buffer. To increase the precision in faulting, we modify the fault to be delivered based on an adaptive F_{pdelay}.

5.4 Fault Model

Based on the independent variables *feat_cache1* and *feat_cache2*, we build linear regression models to predict F_{pdelay} that can best target a fault at an intended position within the N_{rev} buffer. During each faulting attempt, F_{pdelay} is computed only when the timing anchor is detected. To evaluate the efficacy of the regression models, we collect all observed faults with the goal of injecting a fault at byte position 141. Figure 14 shows a significant clustering of faults around positions 140 - 148.

More than 80% of the faults result in 1-3 bytes being corrupted within the N_{rev} buffer. Many of the faulted values suggest that instructions are skipped when the fault occurs. An example of a fault within a segment of the buffer is having corrupted the original byte sequence from `0xa777511b` to `0xa777`**`7777`**.

5.5 Putting it together

We use the following faulting parameters to target faults to specific positions within the buffer: $F_{\theta,\,\text{RSA}} = \{F_{\text{volt}} = 1.055V,\ F_{\text{pdelay}} = \text{adaptive},\ F_{\text{freq\_hi}} = 3.99GHz,\ F_{\text{dur}} = 1,\ F_{\text{freq\_lo}} = 2.61GHz\}$.

Factorizable modulus N_A. About 20% of faulting attempts (1153 out of 6000) result in a successful fault within the target N_{rev} buffer. This set of faulted N values consists of 805 unique values, of which 38 (4.72%) are factorizable based on the algorithm described in § 5.2. For our attack, we select one of the factorizable N_A,

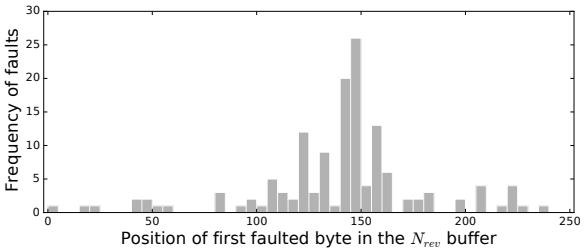

Figure 14: Histogram of observed faults and where the faults occur. The intended faulted position is 141.

where two bytes at positions 141 and 142 are corrupted. We show an example of this faulted and factorizable modulus in Appendix A.4.

Actual attack. Using the above selected N_A, we embed our attack signature $S_A{}'$ into the *widevine* trustlet. Then we conduct our CLKSCREW faulting attempts while invoking the self-signed app. On average, we observe one instance of the desired fault in 65 attempts.

6 Discussion and Related Works

6.1 Applicability to other Platforms

Several highlighted attack enablers in preceding sections apply to other leading architectures. In particular, the entire industry is increasingly moving or has moved to fine-grained energy management designs that separate voltage/frequency domains for the cores. We leave the exploration of these architectures to future research.

Intel. Intel's recent processors are designed with the base clock separated from the other clock domains for more scope of energy consumption optimization [32,35]. This opens up possibilities of overclocking on Intel processors [23]. Given these trends in energy management design on Intel hardware and the growing prevalence of Intel's Secure Enclave SGX [34], a closer look at whether the security guarantees still hold is warranted.

ARMv8. The ARMv8 devices adopt the ARM *big.LITTLE* design that uses non-symmetric cores (such as the "big" Cortex-A15 cores, and the "LITTLE" Cortex-A7 cores) in same system [36]. Since these cores are of different architectures, they exhibit different energy consumption characteristics. It is thus essential that they have separate voltage/frequency domains. The use of separate domains, like in the 32-bit ARMv7 architecture explored in this work, expose the 64-bit ARMv8 devices to similar potential dangers from the software-exposed energy management mechanisms.

Cloud computing providers. The need to improve energy consumption does not just apply to user devices; this extends even to cloud computing providers. Since 2015, Amazon AWS offers EC2 VM instances [16] where power management controls are exposed within the virtualized environment. In particular, EC2 users can fine-tune the processor's performance using P-state and C-state controls [8]. This warrants further research to assess the security ramifications of such user-exposed energy management controls in the cloud environment.

6.2 Hardware-Level Defenses

Operating limits in hardware. CLKSCREW requires the hardware regulators to be able to push voltage/frequency past the operating limits. To address this, hard limits can be enforced within the regulators in the form of additional limit-checking logic or e-fuses [55]. However, this can be complicated by three reasons. First, adding such enforcement logic in the regulators requires making these design decisions very early in the hardware design process. However, the operational limits can only be typically derived through rigorous electrical testing in the post-manufacturing process. Second, manufacturing process variations can change operational limits even for chips of the same designs fabricated on the same wafer. Third, these hardware regulators are designed to work across a wide range of SoC processors. Imposing a one-size-fits-all range of limits is challenging because SoC-specific limits hinder the portability of these regulators across multiple SoC. For example, the PMIC found on the Nexus 6 is also deployed on the Galaxy Note 4.

Separate cross-boundary regulators. Another mitigation is to maintain different power domains across security boundaries. This entails using a separate regulator when the isolated environment is active. This has two issues. First, while trusted execution technologies like Trustzone and SGX separate execution modes for security, the different modes continue to operate on the same core. Maintaining separate regulators physically when the execution mode switches can be expensive. Second, DVFS components typically span across the system stack. If the trusted execution uses dedicated regulators, this implies that a similar cross-stack power management solution needs to be implemented within the trusted mode to optimize energy consumption. Such an implementation can impact the runtime of the trusted mode and increase the complexity of the trusted code.

Redundancy/checks/randomization. To mitigate the effects of erroneous computations due to induced faults, researchers propose redesigning the application core chip with additional logic and timing redundancy [13], as well as recovery mechanisms [33]. Also, Bar-El *et al.* suggest building duplicate microarchitectural units and encrypting memory bus operations for attacks that target mem-

ory operations [13]. Luo *et al.* present a clock glitch detection technique that monitors the system clock signal using another higher frequency clock signal [41]. While many of these works are demonstrated on FPGAs [58] and ASICs [54], it is unclear how feasible it is on commodity devices and how much chip area and runtime overhead it adds. Besides adding redundancy, recent work proposes adding randomization using reconfigurable hardware as a mitigation strategy [59].

6.3 Software-Level Defenses

Randomization. Since CLKSCREW requires some degree of timing precision in delivering the faults, one mitigation strategy is to introduce randomization (via no-op loops) to the runtime execution of the code to be protected. However, we note that while this mitigates against attacks without a timing anchor (AES attack in § 4), it may have limited protection against attacks that use forms of runtime profiling for the timing guidance (RSA attack in § 5).

Redundancy and checks. Several software-only defenses propose compiling code with checksum integrity verification and execution redundancy (executing sensitive code multiple times) [13, 15]. While these defenses may be deployed on systems requiring high dependability, they are not typically deployed on commodity devices like phones because they impact energy efficiency.

6.4 Subverting Cryptography with Faults

Boneh *et al.* offer the first DFA theoretical model to breaking various cryptographic schemes using injected hardware faults [22]. Subsequently, many researchers demonstrate physical fault attacks using a range of sophisticated fault injection equipment like laser [24, 25] and heat [29]. Compared to these attacks including all known undervolting [14, 45] and overclocking [20] ones, CLKSCREW does not need physical access to the target devices, since it is initiated entirely from software. CLKSCREW is also the first to demonstrate such attacks on a commodity device. We emphasize that while CLKSCREW shows how faults can break cryptographic schemes, it does so to highlight the dangers of hardware regulators exposing software-access interfaces, especially across security trust boundaries.

6.5 Relation to Rowhammer Faults

Kim *et al.* first present reliability issues with DRAM memory [37] (dubbed the "Rowhammer" problem). Since then, many works use the Rowhammer issue to demonstrate the dangers of such software-induced hardware-based transient bit-flips in practical scenarios ranging from browsers [30], virtualized environments [50], privilege escalation on Linux kernel [52] and from Android apps [57]. Like Rowhammer, CLKSCREW is equally pervasive. However, CLKSCREW is the manifestation of a different attack vector relying on software-exposed energy management mechanisms. The complexity of these cross-stack mechanisms makes any potential mitigation against CLKSCREW more complicated and challenging. Furthermore, unlike Rowhammer that corrupts DRAM *memory*, CLKSCREW targets *microarchitectural* operations. While we use CLKSCREW to induce faults in memory contents, CLKSCREW can conceivably affect a wider range of computation in microarchitectural units other than memory (such as caches, branch prediction units, arithmetic logic units and floating point units).

7 Conclusions

As researchers and practitioners embark upon increasingly aggressive cooperative hardware-software mechanisms with the aim of improving energy efficiency, this work shows, for the first time, that doing so may create serious security vulnerabilities. With only publicly available information, we have shown that the sophisticated energy management mechanisms used in state-of-the-art mobile SoCs are vulnerable to confidentiality, integrity and availability attacks. Our CLKSCREW attack is able to subvert even hardware-enforced security isolation and does not require physical access, further increasing the risk and danger of this attack vector.

While we offer proof of attackability in this paper, the attack can be improved, extended and combined with other attacks in a number of ways. For instance, using faults to induce specific values at exact times (as opposed to random values at approximate times) can substantially increase the power of this technique. Furthermore, CLKSCREW is the tip of the iceberg: more security vulnerabilities are likely to surface in emerging energy optimization techniques, such as finer-grained controls, distributed control of voltage and frequency islands, and near/sub-threshold optimizations.

Our analysis suggests that there is unlikely to be a single, simple fix, or even a piecemeal fix, that can entirely prevent CLKSCREW style attacks. Many of the design decisions that contribute to the success of the attack are supported by practical engineering concerns. In other words, the root cause is not a specific hardware or software bug but rather a series of well-thought-out, nevertheless security-oblivious, design decisions. To prevent these problems, a coordinated full system response is likely needed, along with accepting the fact that some modest cost increases may be necessary to harden energy management systems. This demands research in a

number of areas such as better Computer Aided Design (CAD) tools for analyzing timing violations, better validation and verification methodology in the presence of DVFS, architectural approaches for DVFS isolation, and authenticated mechanisms for accessing voltage and frequency regulators. As system designers work to invent and implement these protections, security researchers can complement these efforts by creating newer and exciting attacks on these protections.

Acknowledgments

We thank the anonymous reviewers for their feedback on this work. We thank Yuan Kang for his feedback, especially on the case studies. This work is supported by a fellowship from the Alfred P. Sloan Foundation.

References

[1] Firmware update for Nexus 6 (shamu). https://dl.google.com/dl/android/aosp/shamu-mob31s-factory-c73a35ef.zip. Factory Images for Nexus and Pixel Devices.

[2] MSM Subsystem Power Manager (spm-v2). https://android.googlesource.com/kernel/msm.git/+/android-msm-shamu-3.10-lollipop-mr1/Documentation/devicetree/bindings/arm/msm/spm-v2.txt. Git at Google.

[3] Nexus 6 Qualcomm-stipulated OPP. https://android.googlesource.com/kernel/msm/+/android-msm-shamu-3.10-lollipop-mr1/arch/arm/boot/dts/qcom/apq8084.dtsi. Git at Google.

[4] QSEECOM source code. https://android.googlesource.com/kernel/msm/+/android-msm-shamu-3.10-lollipop-mr1/drivers/misc/qseecom.c. Git at Google.

[5] Qualcomm Krait PMIC frequency driver source code. https://android.googlesource.com/kernel/msm/+/android-msm-shamu-3.10-lollipop-mr1/drivers/clk/qcom/clock-krait.c. Git at Google.

[6] Qualcomm Krait PMIC voltage regulator driver source code. https://android.googlesource.com/kernel/msm/+/android-msm-shamu-3.10-lollipop-mr1/arch/arm/mach-msm/krait-regulator.c. Git at Google.

[7] Mobile Hardware Stats 2016-09. http://hwstats.unity3d.com/mobile/cpu.html, Sep 2016. Unity.

[8] AMAZON. Processor State Control for Your EC2 Instance. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/processor_state_control.html. Amazon AWS.

[9] ANATI, I., GUERON, S., JOHNSON, S., AND SCARLATA, V. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy (HASP)* (2013), vol. 13.

[10] ARM. c9, Performance Monitor Control Register. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0344b/Bgbdeggf.html. Cortex-A8 Technical Reference Manual.

[11] ARM. Security Technology - Building a Secure System using TrustZone Technology. *ARM Technical White Paper* (2009).

[12] ARM. Power Management with big.LITTLE: A technical overview. https://community.arm.com/processors/b/blog/posts/power-management-with-big-little-a-technical-overview, sep 2013.

[13] BAR-EL, H., CHOUKRI, H., NACCACHE, D., TUNSTALL, M., AND WHELAN, C. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE 94*, 2 (2006), 370–382.

[14] BARENGHI, A., BERTONI, G., PARRINELLO, E., AND PELOSI, G. Low voltage fault attacks on the rsa cryptosystem. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2009 Workshop on* (2009), IEEE, pp. 23–31.

[15] BARENGHI, A., BREVEGLIERI, L., KOREN, I., PELOSI, G., AND REGAZZONI, F. Countermeasures against fault attacks on software implemented AES: effectiveness and cost. In *Proceedings of the 5th Workshop on Embedded Systems Security* (2010), ACM, p. 7.

[16] BARR, J. Now Available - New C4 Instances. https://aws.amazon.com/blogs/aws/now-available-new-c4-instances/, jan 2015.

[17] BEAUPRE, S. TRUSTNONE - Signed comparison on unsigned user input. http://theroot.ninja/disclosures/TRUSTNONE_1.0-11282015.pdf.

[18] BERZATI, A., CANOVAS, C., AND GOUBIN, L. Perturbating RSA public keys: An improved attack. In *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)* (2008), Springer, pp. 380–395.

[19] BIHAM, E., CARMELI, Y., AND SHAMIR, A. Bug attacks. In *Annual International Cryptology Conference* (2008), Springer, pp. 221–240.

[20] BLÖMER, J., DA SILVA, R. G., GÜNTHER, P., KRÄMER, J., AND SEIFERT, J.-P. A practical second-order fault attack against a real-world pairing implementation. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2014 Workshop on* (2014), IEEE, pp. 123–136.

[21] BONEH, D. Twenty years of attacks on the RSA cryptosystem. *Notices of the American Mathematical Society (AMS) 46*, 2 (1999), 203–213.

[22] BONEH, D., DEMILLO, R. A., AND LIPTON, R. J. On the Importance of Checking Cryptographic Protocols for Faults. In *Proceedings of the 16th Annual International Conference on Theory and Application of Cryptographic Techniques* (Berlin, Heidelberg, 1997), EUROCRYPT'97, Springer-Verlag, pp. 37–51.

[23] BTARUNR. Rejoice! Base Clock Overclocking to Make a Comeback with Skylake. https://www.techpowerup.com/218315/rejoice-base-clock-overclocking-to-make-a-comeback-with-skylake, Dec 2015. TechPowerup.

[24] CANIVET, G., MAISTRI, P., LEVEUGLE, R., CLÉDIÈRE, J., VALETTE, F., AND RENAUDIN, M. Glitch and Laser Fault Attacks onto a Secure AES Implementation on a SRAM-Based FPGA. *Journal of Cryptology 24*, 2 (2011), 247–268.

[25] DOBRAUNIG, C., EICHLSEDER, M., KORAK, T., LOMNÉ, V., AND MENDEL, F. *Statistical Fault Attacks on Nonce-Based Authenticated Encryption Schemes*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016, pp. 369–395.

[26] EDGE, J. KS2012: ARM: Secure monitor API. https://lwn.net/Articles/513756/, Aug 2012.

[27] EKBERG, J.-E., AND KOSTIAINEN, K. Trusted Execution Environments on Mobile Devices. https://www.cs.helsinki.fi/group/secures/CCS-tutorial/tutorial-slides.pdf, Nov 2013. ACM CCS 2013 tutorial.

[28] GOOGLE. Multiplatform Content Protection for Internet Video Delivery. https://www.widevine.com/wv_drm.html. Widevine DRM.

[29] GOVINDAVAJHALA, S., AND APPEL, A. W. Using Memory Errors to Attack a Virtual Machine. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy (S&P)*, pp. 154–165.

[30] GRUSS, D., MAURICE, C., AND MANGARD, S. Rowhammer.js: A remote software-induced fault attack in javascript. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016, pp. 300–321.

[31] GRUSS, D., SPREITZER, R., AND MANGARD, S. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *24th USENIX Security Symposium (USENIX Security 15)* (Washington, D.C., 2015), USENIX Association, pp. 897–912.

[32] HAMMARLUND, P., KUMAR, R., OSBORNE, R. B., RAJWAR, R., SINGHAL, R., D'SA, R., CHAPPELL, R., KAUSHIK, S., CHENNUPATY, S., JOURDAN, S., ET AL. Haswell: The fourth-generation Intel core processor. *IEEE Micro*, 2 (2014), 6–20.

[33] HUU, N. M., ROBISSON, B., AGOYAN, M., AND DRACH, N. Low-cost recovery for the code integrity protection in secure embedded processors. In *Hardware-Oriented Security and Trust (HOST), 2011 IEEE International Symposium on* (2011), IEEE, pp. 99–104.

[34] INTEL. Intel Software Guard Extensions (Intel SGX). https://software.intel.com/en-us/sgx.

[35] INTEL. The Engine for Digital Transformation in the Data Center. http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/xeon-e5-brief.pdf. Intel Product Brief.

[36] JEFF, B. big.LITTLE system architecture from arm: Saving power through heterogeneous multiprocessing and task context migration. In *Proceedings of the 49th Annual Design Automation Conference (DAC)* (2012), ACM, pp. 1143–1146.

[37] KIM, Y., DALY, R., KIM, J., FALLIN, C., LEE, J. H., LEE, D., WILKERSON, C., LAI, K., AND MUTLU, O. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)* (June 2014), pp. 361–372.

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

[39] LENSTRA JR, H. W. Factoring integers with elliptic curves. *Annals of mathematics* (1987), 649–673.

[40] LIPP, M., GRUSS, D., SPREITZER, R., MAURICE, C., AND MANGARD, S. Armageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, 2016), USENIX Association, pp. 549–564.

[41] LUO, P., LUO, C., AND FEI, Y. System Clock and Power Supply Cross-Checking for Glitch Detection. Cryptology ePrint Archive, Report 2016/968, 2016. http://eprint.iacr.org/2016/968.

[42] MENEZES, A. J., VANSTONE, S. A., AND OORSCHOT, P. C. V. *Handbook of Applied Cryptography*, 1st ed. CRC Press, Inc., Boca Raton, FL, USA, 1996.

[43] MILLER, F. P., VANDOME, A. F., AND MCBREWSTER, J. *Advanced Encryption Standard*. Alpha Press, 2009.

[44] MONTGOMERY, P. L. Modular multiplication without trial division. *Mathematics of computation 44*, 170 (1985), 519–521.

[45] O'FLYNN, C. Fault Injection using Crowbars on Embedded Systems. Tech. rep., IACR Cryptology ePrint Archive, 2016.

[46] PALLIPADI, V., AND STARIKOVSKIY, A. The ondemand governor. In *Proceedings of the Linux Symposium* (2006), vol. 2, sn, pp. 215–230.

[47] PATTERSON, D. A., AND HENNESSY, J. L. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[48] QUALCOMM. Snapdragon S4 Processors: System on Chip Solutions for a New Mobile Age. https://www.qualcomm.com/documents/snapdragon-s4-processors-system-chip-solutions-new-mobile-age, jul 2013.

[49] QUALCOMM. Secure Boot and Image Authentication - Technical Overview. https://www.qualcomm.com/documents/secure-boot-and-image-authentication-technical-overview, Oct 2016.

[50] RAZAVI, K., GRAS, B., BOSMAN, E., PRENEEL, B., GIUFFRIDA, C., AND BOS, H. Flip feng shui: Hammering a needle in the software stack. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, 2016), USENIX Association, pp. 1–18.

[51] RIVEST, R. L., SHAMIR, A., AND ADLEMAN, L. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM 21*, 2 (1978), 120–126.

[52] SEABORN, M., AND DULLIEN, T. Exploiting the DRAM rowhammer bug to gain kernel privileges. *Black Hat* (2015).

[53] SHEARER, F. *Power Management in Mobile Devices*. Newnes, 2011.

[54] STAMENKOVIĆ, Z., PETROVIĆ, V., AND SCHOOF, G. Fault-tolerant ASIC: Design and implementation. *Facta universitatis-series: Electronics and Energetics 26*, 3 (2013), 175–186.

[55] STMICROELECTRONICS. E-fuses. http://www.st.com/en/power-management/e-fuses.html?querycriteria=productId=SC1532. How-swap power management.

[56] TUNSTALL, M., MUKHOPADHYAY, D., AND ALI, S. Differential Fault Analysis of the Advanced Encryption Standard using a Single Fault. In *IFIP International Workshop on Information Security Theory and Practices* (2011), Springer, pp. 224–233.

[57] VAN DER VEEN, V., FRATANTONIO, Y., LINDORFER, M., GRUSS, D., MAURICE, C., VIGNA, G., BOS, H., RAZAVI, K., AND GIUFFRIDA, C. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)* (Nov 2016).

[58] VELEGALATI, R., SHAH, K., AND KAPS, J.-P. Glitch Detection in Hardware Implementations on FPGAs Using Delay Based Sampling Techniques. In *Proceedings of the 2013 Euromicro Conference on Digital System Design* (Washington, DC, USA, 2013), DSD '13, IEEE Computer Society, pp. 947–954.

[59] WANG, B., LIU, L., DENG, C., ZHU, M., YIN, S., AND WEI, S. Against Double Fault Attacks: Injection Effort Model, Space and Time Randomization Based Countermeasures for Reconfigurable Array Architecture. *IEEE Transactions on Information Forensics and Security 11*, 6 (2016), 1151–1164.

[60] WEISER, M., WELCH, B., DEMERS, A., AND SHENKER, S. Scheduling for Reduced CPU Energy. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation (OSDI)* (1994).

[61] YAROM, Y., AND FALKNER, K. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX Security 14)* (2014), pp. 719–732.

[62] ZHANG, X., XIAO, Y., AND ZHANG, Y. Return-Oriented Flush-Reload Side Channels on ARM and Their Implications for Android Devices. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2016), pp. 858–870.

A Appendix

A.1 Timing Violation due to Undervolting

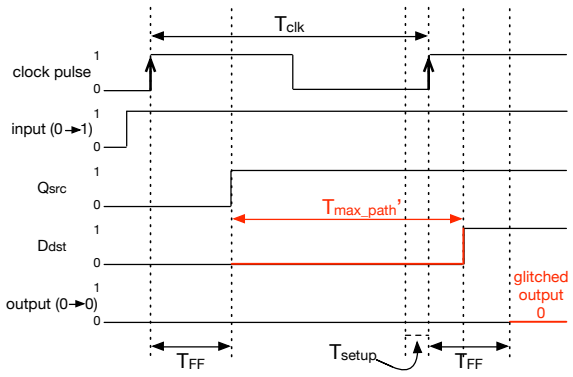

Figure 15: Glitch due to undervolting: Increasing propagation time of the critical path between the two consecutive flip-flops, clock period $T_{\text{max\_path}} \rightarrow T_{\text{max\_path}}{}'$ results in a bit-flip in output $1 \rightarrow 0$.

A.2 FLIPENDIANNESS Implementation

Algorithm 2 Reverse the endianness of a memory buffer.

```
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
```

A.3 Vendor-Stipulated vs Observed OPPs

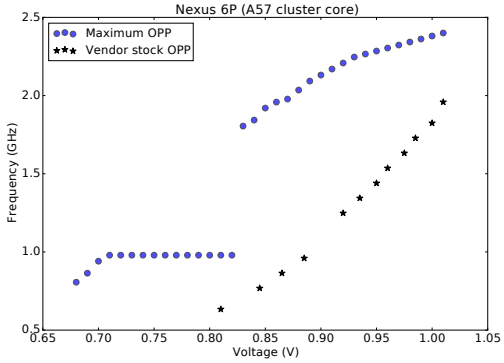

Figure 16: Vendor-stipulated vs maximum voltage/frequency OPPs for Nexus 6P.

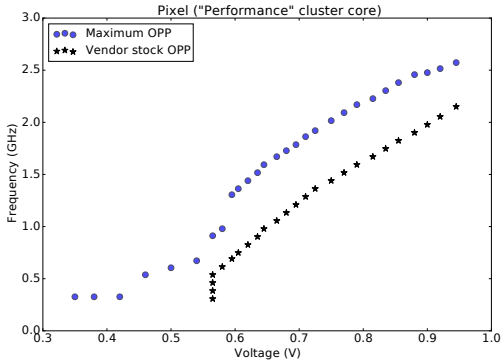

Figure 17: Vendor-stipulated vs maximum voltage/frequency OPPs for Pixel.

A.4 Example Glitch in RSA Modulus

Original Modulus N:

```
...f35a...
```

Corrupted Modulus N_A:

```
c44dc735f6682a261a0b8545a62dd13df4c646a5ede482cef85892
5baa1811fa0284766b3d1d2b4d6893df4d9c045efe3e84d8c5d036
31b25420f1231d8211e2322eb7eb524da6c1e8fb4c3ae4a8f5ca13
d1e0591f5c64e8e711b3726215cec59ed0ebc6bb042b917d445288
87915fdf764df691d183e16f31ba1ed94c84b476e74b488463e855
510220217 63a3a3a 64ddf105c1530ef3fcf7e54233e5d3a4747bbb
17328a63e6e3384ac25ee80054bd566855e2eb59a2fd168d3643e4
4851acf0d118fb03c73ebc099b4add59c39367d6c91f498d8d607a
f2e57cc73e3b5718435a81123f080267726a2a9c1cc94b9c6bb681
7427b85d8c670f9a53a777511b
```

Factors of N_A:

```
0x3, 0x11b, 0xcb9, 0x4a70807d6567959438227805b12a19...
```

Private Exponent d_A:

```
04160eecc648a3da19abdc42af4cfb41a798e5eb8b1b49c2c29...
```