VoltJockey: Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-core Frequencies

Pengfei Qiu[1,2,3], Dongsheng Wang[1,2], Yongqiang Lyu[2*], Gang Qu[3]
[1]Department of Computer Science and Technology, Tsinghua University, Beijing, China.
[2]Beijing National Research Center for Information Science and Technology, Tsinghua University, Beijing, China.
[3]Dept. of Electrical and Computer Engineering & Institute for Systems Research, Univ. of Maryland, College Park, USA.
qpf15@mails.tsinghua.edu.cn,{wds,luyq}@mail.tsinghua.edu.cn,gangqu@umd.edu

ABSTRACT

ARM TrustZone builds a trusted execution environment based on the concept of hardware separation. It has been quite successful in defending against various software attacks and forcing attackers to explore vulnerabilities in interface designs and side channels. The recently reported CLKscrew attack breaks TrustZone through software by overclocking CPU to generate hardware faults. However, overclocking makes the processor run at a very high frequency, which is relatively easy to detect and prevent, for example by hardware frequency locking.

In this paper, we propose an innovative software-controlled hardware fault-based attack, *VoltJockey*, on multi-core processors that adopt dynamic voltage and frequency scaling (DVFS) techniques for energy efficiency. Unlike CLKscrew, we manipulate the voltages rather than the frequencies via DVFS unit to generate hardware faults on the victim cores, which makes VoltJockey stealthier and harder to prevent than CLKscrew. We deliberately control the fault generation to facilitate differential fault analysis to break TrustZone. The entire attack process is based on software without any involvement of hardware. We implement VoltJockey on an ARM-based *Krait* processor from a commodity Android phone and demonstrate how to reveal the AES key from TrustZone and how to breach the RSA-based TrustZone authentication. These results suggest that VoltJockey has a comparable efficiency to side channels in obtaining TrustZone-guarded credentials, as well as the potential of bypassing the RSA-based verification to load untrusted applications into TrustZone. We also discuss both hardware-based and software-based countermeasures and their limitations.

CCS CONCEPTS

• **Security and privacy → Security in hardware**; **Embedded systems security**; **Hardware attacks and countermeasures**.

*Corresponding author.

CCS '19, November 11–15, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6747-9/19/11…$15.00
https://doi.org/10.1145/3319535.3354201

KEYWORDS

TrustZone; fault injection attack; voltage manipulation; low voltage error; dynamic voltage and frequency scaling (DVFS).

ACM Reference Format:
Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, Gang Qu. 2019. VoltJockey: Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-core Frequencies. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS'19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3319535.3354201

1 INTRODUCTION

TrustZone technology extends the hardware and software architectures of ARM-based system-on-chip (SoC) to provide an isolated trusted execution environment for a wide array of devices, including handsets, tablets, wearable devices, and enterprise systems. Almost all of the current ARM-based processors are designed to enable TrustZone. For example, there are more than one billion smart phones worldwide equipped with TrustZone. Meanwhile, more and more applications have been developed taking advantage of TrustZone's security such as payment protection, digital rights management, bring your own device (BYOD), and various secured enterprise solutions [66].

TrustZone utilizes the hardware-assured security methodologies that have intrinsically tight interfaces to direct software executions for a trusted and secure execution environment. Current attacks to TrustZone mainly rely on the implementation vulnerabilities of software interfaces, such as secure monitor call (SMC), interrupt request (IRQ), fast interrupt request (FIQ), shared memory access, and trusted application-specific calls [66]. Amending unsafe functions is effective to defend against those attacks. In addition to the software vulnerability-based attacks, it has also been reported successful attacks by exploiting hardware imperfections such as side channel [49] and hardware fault injection attack [65]. Although side channel attacks can only steal sensitive information, the hardware fault injection attacks can be deployed to invalidate the security measures of TrustZone and allow the attackers to gain control of it. For example, the CLKscrew attack [65] breaks TrustZone by overclocking CPU with software. However, this attack is also easy to be addressed by hardware frequency locking.

In this study, we propose an innovative hardware-fault injection method to break TrustZone. Our method manipulates the voltage of the multi-cores processor to induce hardware faults and hence earns the name *VoltJockey*. The basic idea is to lower the voltage deliberately such that fault could occur because of the vulnerability driven by the hardware dynamic voltage and frequency scaling

(DVFS) technique [14]. Compared to the CLKscrew attack [65], which motivates our work and is the most relevant, VoltJockey has the following two advantages: 1) it does not rely on the overclocking mechanism or manipulating frequencies; 2) the entire attack process is carried out under the normal functions of DVFS, and the glitch voltages are also valid voltages for DVFS. Both make VoltJockey a more practical and dangerous attack than CLKscrew.

Although changing voltage was mentioned in the original CLKscrew paper, the authors also acknowledged that the system crashes when they manipulate the voltage. Indeed, they have reported that when the device is set to any voltage outside the range $0.6V$ to $1.17V$, it either reboots or freezes [65]. This suggests that the voltage-change based attack they have experimented follows the same idea of CLKscrew. That is, running the system at voltage level outside the normal range allowed by DVFS. This could be the reason of their failure in extending CLKscrew attack from overclocking to "overVoltage". Even if they manage to implement such "overVoltage" attack, it can be easily detected and prevented by voltage guardians or locking techniques. From this point of review, VoltJockey is different from CLKscrew.

VoltJockey exploits the voltage management-related vulnerabilities of DVFS. The DVFS technique aims at saving energy by enabling software to adjust frequencies and voltages of cores. It has been employed by most of the modern commercial processors including those by ARM and Intel. Processors can reliably work at a fixed clock frequency with a range of voltage levels. However, if the voltage is inappropriately low, the processor could not reach the expected frequency and this will cause delay in delivering data from one function/component to another. Thus, next function/component might be using the incorrect data and results in wrong functional behaviors and/or hardware faults. Similarly, if the voltage is inappropriately high, the processor might also work with unpredictable functional behaviors and thus introduced unexpected errors to the running applications on it. Both kinds of faults make it possible for the attackers to analyze sensitive data or alter software outputs with differential fault analysis techniques.

We validate VoltJockey on an ARM-based *Krait* processor, whose voltage is controlled by a common hardware regulator. The attack is launched as follows: we first provide a high frequency to a victim core and a low frequency to an attacker core, and then give the processor a transitory voltage that is sufficiently high for the attacker core to perform tasks correctly but is not for the victim core to work normally. As a consequence, such glitch voltage will trigger functional errors into the victim core without bringing any hazardous impacts to the attacker core. The error output will be collected for differential fault analysis. We implement this and successfully obtain the AES key protected by TrustZone and circumvent the RSA-based signature verification so we can load any applications into TrustZone.

Our main contributions in this paper can be summarized as:

(1) We propose a novel software-controlled hardware fault-based attack to TrustZone, the VoltJockey, which manipulates the voltages of DVFS-enabled multi-core processors to breach TrustZone protections.
(2) We introduce a methodology for analyzing and exploiting the vulnerability of multi-core voltage-frequency management of DVFS to perform the VoltJockey attack, which can realize precise injections of the low-voltage faults into TrustZone procedures.
(3) We validate VoltJockey on an ARM-based *Krait* processor by breaking AES and RSA in TrustZone. The experiments successfully obtain the encryption key of AES and load untrusted applications into TrustZone by invalidating the RSA verification.

The remainder of this paper is organized as follows. Section 2 introduces the essential preliminaries of TrustZone, DVFS, and voltage management of the ARM-based *Krait* micro architecture. Section 3 describes the overall methodology and basic idea of the VoltJockey. Section 4 gives the major challenges and corresponding technical details in implementing the attack. Section 5 and Section 6 demonstrate the real applications of VoltJockey on the AES key extraction and RSA output alteration in TrustZone, respectively. At the end of the paper, Section 7 states the related work, Section 8 analyzes the possible countermeasures against the VoltJockey, and Section 9 finally concludes the paper.

2 PRELIMINARIES

In this section, we introduce the basics on TrustZone, DVFS, and voltage management of ARM-based *Krait* micro architecture, respectively.

2.1 ARM TrustZone

For enhancing the confidentiality, integrity, and availability of trusted applications, ARM develops TrustZone technology, which extends the hardware and software architectures of SoC to create an trusted execution environment [2]. The processor can work in two worlds, normal world and secure word. TrustZone virtualizes a physical core as a virtual logical core and a virtual secure core, which execute in a time-sliced fashion and perform the normal-world and secure-world programs, separately. A trusted application (also expressed as a trustlet) can be loaded from normal world into secure world with associated kernel drivers as long as it passes the verification by the RSA-based authentication procedure [2].

Although TrustZone enhances the security of ARM-based platforms to a remarkable extent, some software vulnerabilities can still be exploited to breach it. For example, the privilege escalation attacks can be launched if the necessary boundary check is missed when invoking secure services [51, 57, 64]; the downgrade attacks can be implemented if the system does not limit firmware versions [15, 30]. However, those vulnerabilities can easily be fixed by software upgrading. In contrast, the attacks that utilize hardware vulnerabilities may be more difficult to be prevented due to the upgrading overheads to both hardware and software. Such attacks include side channel attacks [16, 49] and hardware fault injection attacks [65]. Our proposed VoltJockey belongs to this category.

2.2 Dynamic Voltage and Frequency Scaling

The energy consumption is the integral of instantaneous dynamic power over time. And, the instantaneous power is proportional to the product of frequency and voltage quadratic[1] [41]. In order

[1]Formally, the formula of dynamic power P, voltage V, and frequency F for an electronic component that has a capacitative load C is: $P = \alpha CV^2F$.

to improve the energy efficiency, most of the current processors enable the DVFS extension [14], which dynamically adjusts the frequency and voltage of processors cores based on the real time computation load. On one hand, the integrated circuits are designed with variable voltages and corresponding frequencies available on-chip at real time. On the other hand, the regulator driver provided by hardware distributors directs hardware and exposes interfaces for privileged software[2] to change the frequency and voltage [41]. The vendor-stipulated voltage/frequency operating performance points (OPPs) [65] attach a frequency with a particular voltage that guarantees the processor works error-freely and energy-efficiently under that frequency. System managers can configure the CPU frequency manually with driver commands. However, no command can be utilized for them to modify the CPU voltage directly.

2.3 Voltage Management of *Krait* Architecture

Krait [13] is a processor micro architecture designed by *Qualcomm* with ARMv7-A instruction set architecture and is fabricated in 24 nanometers. It is largely embedded in *Qualcomm* processors, such as, *MSM8960*, *MSM8x60A*, *MSM8x30*, *MSM8960 Pro*, *APQ8064*, and *APQ8084AB*. Our verification experiments are mainly enforced on Google Nexus 6, which has an *APQ8084AB* processor. Figure 1 illustrates the voltage management framework in *Krait* architecture. The SoC usually connects to several hardware peripherals such as memory, camera, sensor, audio, display, etc. They are usually sourced from different vendors and may work under different rated voltages. It is impractical to provide a fixed voltage to all the peripherals. Therefore, the *Krait* SoC provides a power management integrated circuit (PMIC) (*PMA8084* in Nexus 6 [32]) that integrates several hardware regulators to provide different voltages to the hardware peripherals.

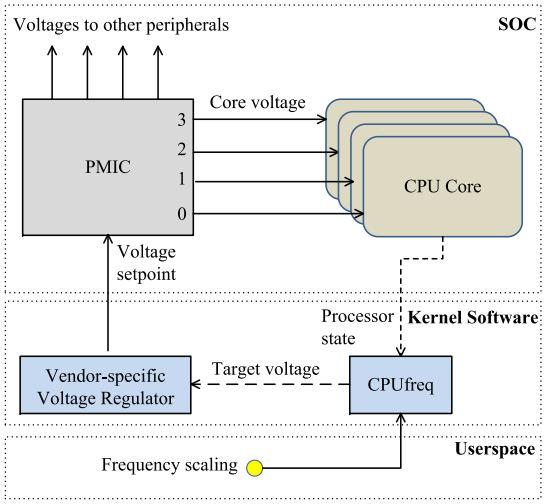


Figure 1: The voltage management of *Krait* architecture.

There are two significant drivers in the software stack of voltage regulation of the Android systems. Firstly, *Qualcomm* supplies a vendor-specific regulator driver [33, 39] to direct the output of PMIC and wrap the voltage related operations. Secondly, the common DVFS [38] driver updates the core frequency on the requirement of energy saving and performance, which will result in the core voltage modification indirectly. However, all cores share the same hardware regulator, and thus have the same working voltage which can support a range of different frequencies for the cores to operate with. This is the hardware vulnerability that our proposed VoltJockey will exploit.

[2]The kernel module of *CPUfreq* acts as the DVFS driver in Linux and Android systems. The core frequencies can be changed in the power plan in Windows system.

3 VOLTAGE-BASED HARDWARE FAULT AND OVERVIEW OF VOLTJOCKEY

Voltage is one of the key factors to ensure the correct functions of a circuit. In this section, we discuss the fact that inappropriate voltages can violate the timing constraints and introduce unexpected outputs in digital circuits. Based on the hardware faults introduced by manipulated voltages, we propose the attack of VoltJockey.

3.1 Timing Constraints of a Circuit

A digital circuit usually consists of multitudinous electronic components. For an input, one electronic component spends a certain time to give a stable and unambiguous output. Therefore, proper timing constraint should be satisfied to fulfill the valid information processing in the digital circuit. In fact, debugging the timing constraint is a very important and essential step when designing and validating a sequential circuit.

Figure 2 demonstrates the timing constraint for an example circuit that starts and ends with a sequential electronic element (flip-flop (FF) in this example), separately. The middle logic components transfer the output of the first FF to the input of the last FF. We assume the two FFs are triggered by the clock pulse's rising edges. First, we make the following definitions.

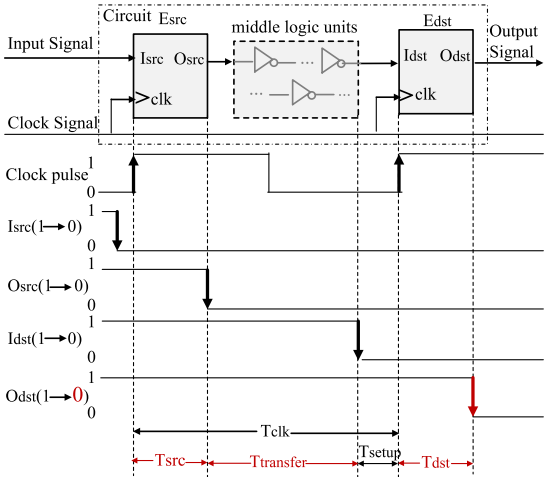


Figure 2: The timing constraint of a circuit case. The rising edge of clock controls the validation of sequential units.

- T_{clk} means the clock period of synchronous clock pulse, which also reflects the circuit frequency.

- The input signal (I_{dst}) of the last FF (E_{dst}) should be held for a period of T_{setup} to be stable before the next rising edge of clock pulse.
- T_{src} denotes the latency for the first sequential unit (E_{src}) to give a steady output after receiving the rising edge of clock signal.
- $T_{transfer}$ represents the transmission time from the output (O_{src}) of E_{src} to I_{dst}, which is also the execution time of the middle logic components.

For a settled frequency, T_{clk} is a constant. Besides, for a determinate E_{dst}, T_{setup} is invariable. Therefore, in order to make sure the O_{dst} is as expected, T_{src} and $T_{transfer}$ should be limited in equation (1) with a very small time constant T_ϵ.

$$T_{src} + T_{transfer} \leq T_{clk} - T_{setup} - T_\epsilon \tag{1}$$

3.2 Triggering Hardware Faults by Inappropriate Voltages

If the provided voltage is lower than the expected one, T_{src} and $T_{transfer}$ will increase, which might violate the timing constraint in equation (1). Figure 3 demonstrates the signal transitions when T_{src} and $T_{transfer}$ increase. O_{dst} will remain unchanged since the expected I_{dst} is not prepared at the next rising edge of the clock signal, which might trigger hardware faults and lead to bit-flip.

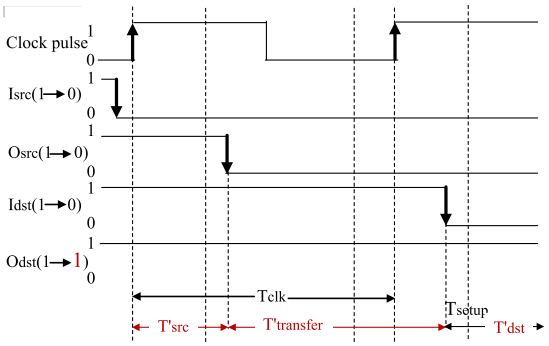


Figure 3: The timing constraint may be violated if T_{src} and $T_{transfer}$ are increased due to inappropriately low voltage against the designated frequency.

Increasing the voltage of the circuit may damage the stability of electronic units and introduce hardware faults, too. For example, if E_{src} becomes unstable due to the fact that a high voltage is imposed, O_{src} may be reversed and O_{dst} will remain at value 1. In this situation, hardware faults are introduced. This is elaborated with full details in Figure 11 in the Appendix.

3.3 Attack Method

If both the frequency and voltage of a core are independent from the other cores, attackers can induce hardware faults by fixing the victim core's frequency and choosing a low voltage for the victim core. However, most of the current processors that enable DVFS are designed with a shared hardware voltage regulator. Changing the voltage of victim core will also modify the voltage of other cores.

The clock frequency of a core is determined by the delay of the underlying circuit. A high voltage can provide small circuit delay and thus result in high frequency. DVFS manages multiple discrete frequencies and thus requires different voltages. The required voltage differentials provide the basic vulnerability for our attack. Figure 4 shows the basic idea of how VoltJockey works. The attacker procedure is executed on the core that has a low frequency and the victim procedure is performed on the core that has a high frequency. The attacker procedure provides a short-lived glitch voltage that is harmless for the attacker core but could be detrimental for the victim core to trigger hardware faults into the victim procedure. For example, to attack the AES function that is securely protected by TrustZone, the attacker can invoke the AES and induce errors into the middle state matrix. The encryption key can be maliciously stolen. To invalidate the signature authentication when an application is being loaded into TrustZone, the attacker can fault the public modulus of the RSA decryption and change the final output to cheat the authentication.

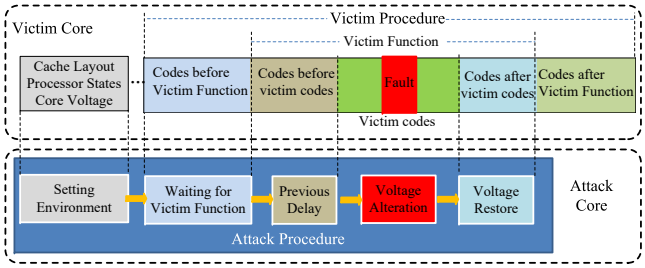


Figure 4: The overview of VoltJockey attack. ① Necessary preparations for providing a suitable voltage glitch environment; ② Attacker procedure waits for the victim function to be invoked; ③ Attacker procedure waits for the target codes to be executed; ④ Attacker procedure alters the core voltage to induce hardware faults; ⑤ Recovering the core voltage.

3.3.1 Assumption. In the attack model of VoltJockey, we make the following two assumptions

- We assume that the voltage of the target multi-cores processor can be adjusted by software. This is actually a common practice used by a wide range of processors for efficient energy saving by DVFS.
- We assume that the attackers have privileges to configure the processor voltage. This requires the attackers to first acquire the root privilege for some systems, which is a well-studied topic and there are many ways to achieve that.

3.3.2 Attacking parameters. The attackers need to determine the following critical parameters to launch the proposed attack:

$$F_{fault} = \{F_a, F_v, V_l, V_b, T_w, T_p, T_d\}$$

The meaning of each parameter is listed in Table 1.

3.3.3 Attacking procedure. The attackers can introduce hardware faults into the victim procedure by the following five steps.

Table 1: The descriptions of attacking parameters

Para.	Meaning
F_a	The frequency of attacker core.
F_v	The frequency of victim core.
V_l	The glitch voltage that induces hardware faults.
V_b	The baseline voltage before and after the fault injection.
T_w	The waiting time before the victim function is executed.
T_p	The previous delay before the attacker procedure can change the processor voltage to the glitch voltage.
T_d	The duration time that the glitch voltage should be kept to ensure a successful fault injection.

(a) ***Preparation setup***. A suitable voltage glitch environment should firstly be prepared before the attack is carried out. First of all, the victim core should be configured with a high frequency and all the other cores should be set to low frequencies. Secondly, the attacker procedure should initialize the processor with a fixed and secure voltage. Finally, all the residual states in the target device should be cleared. This includes cache cache layouts, branch prediction table, interrupt vector table, and status registers, etc. We show how to clear the residual states in section 4.2.

(b) ***Waiting for victim function***. In normal situations, the victim function that VoltJockey would inject faults into is a small part of the victim procedure, and it is uncertain when the victim function would be running. Therefore, the attacker procedure needs to wait for the target function to be invoked by monitoring the intermediate executions of victim procedure before it can induce desired faults. This will be further discussed in section 4.7.

(c) ***Waiting for appropriate injection points***. The goal of the injected hardware faults is to influence a small part of instructions and data of the victim function, and the portion of the affected codes should be kept as small as possible. Therefore, the fault injection points should be elaborately controlled. The waiting time before the fault injection can be conducted is called as the *previous delay*. We show our experiment data about the previous delay on faulting AES and RSA in section 4.11.

(d) ***Voltage manipulation***. This step is responsible for triggering hardware faults. The value of glitch voltage and its lasting duration are these two factors to take controllable hardware faults. It is very important to find the proper voltage values and their lasting time that will result in desired data changes. More details about the two factors on attacking AES and RSA are discussed in section 4.12.

(e) ***Restoring voltage***. The ultimate goal of VoltJockey is to obtain sensitive data or tamper the functions of victim procedure other than crashing the victim core. Therefore, the attacker procedure needs to restore the processor voltage to resume the victim procedure normally as soon as the fault injection is done.

4 TECHNICAL CHALLENGES OF VOLTJOCKEY

In this section, we present the major technical challenges that the VoltJockey addresses, which gives the most important technical details and experiment probings to implement such an attack.

4.1 Parallel Execution

In VoltJockey, the attacker procedure and the victim procedure are diverse processes and executed in parallel, which also is the most common attack scenario. System libraries provide functions to support the binding of one thread to a dedicated core. Besides, OS supplies operations[3] for a user to fix one task on a special core.

4.2 Residual States

The system's residual states before performing the victim procedure can affect the execution time of instructions of the victim procedure. This is mainly reflected from three sides. Firstly, the data access when cache hits will take less time than cache misses, and therefore influences the instruction finish time. Secondly, the branch predictions may also result in the execution time variations of instructions. Thirdly, the unfinished tasks or interrupts may enforce the victim core to emit the victim procedure. Therefore, clearing the residual states is practically significant for the attacker procedure to acquire the precise injection points of hardware faults. In this study, we first flush the cache memories to clear the existing data, and then execute the victim procedure for several times to fill cache and set processor state registers with the victim procedure-related data. In such circumstances, the branch prediction table will also be highly correlated to the victim procedure. Moreover, we also close the IRQ and FIQ interrupts that target at the victim core in voltage manipulation to shield the influences of interrupts.

4.3 Legal Voltage Enforcement

In Android systems, the OPPs are defined in the device-related *dtsi* file [36]. Figure 5 shows an example OPPs (black curve) extracted from Google Nexus 6, which gives the available frequencies and corresponding secure voltages. VoltJockey needs to bypass the restraint of OPPs to gain the power to manipulate the voltages. We analyze the kernel codes of the software stack of DVFS in the system of Nexus 6 and find that the top level driver is responsible for the establishment of frequency table and frequency selection. In addition, the vendor-specific frequency and voltage regulator drivers change the frequency and voltage of processor cores with the request of the top level software and do not care if the request parameters are in the frequency table. This makes it very easy to break the legal frequency-voltage enforcement for voltage manipulation.

4.4 Voltage Threshold

Qualcomm implants a *low dropout linear regulator* (LDO) mode for the *krait* architecture, which delivers a stable and regulated voltage (voltage threshold) to cores for avoiding the processor's malfunctions. If the requested voltage is less than the voltage threshold, the vendor-specific voltage regulator driver [33] will turn the processor voltage mode into LDO, and the processor will be provided the stabilized threshold voltage rather than the less one. The low voltage threshold is defined in the regulator property description file [35] and is read in the power probe stage of the voltage regulator driver. In this study, we change the voltage threshold by revising

[3]Linux provides a command of *taskset* for a normal user to assign one task on an appointed core and the command is supported after Android 6.0. The core binding can be achieved in the task manager in Windows systems.

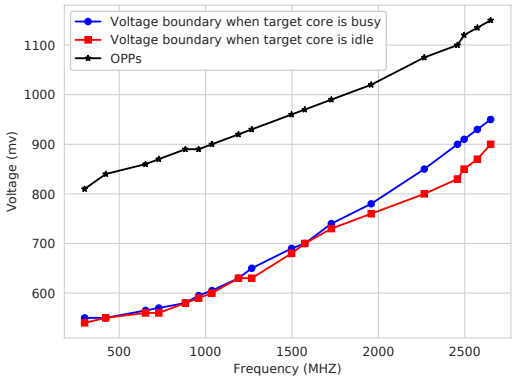


Figure 5: The manipulable voltages of Nexus 6 processor.

the power probe function since the influence range is smaller than modifying the property description file.

The high voltage threshold is hardened in the last byte of the voltage setpoint [34]. The step width of core voltage is $0.005V$ in Nexus 6, which means that the core voltage is theoretically restricted from $0V$ to $1.275V$. We find out that the processor is always restarted immediately when the core voltage is set to a value that is larger than $1.275V$. Therefore, it is actually unpractical to use high voltage to take faults with software.

4.5 Voltage Down-tuning Restriction

As shown in section 2.3, all the processor cores have a common voltage value. In order to ensure the high-frequency cores perform tasks error-freely when tuning the voltage, the voltage regulator driver is designed to only select the higher one between the request voltage and the voltage that corresponds to the highest core frequency, in other words, no down-tuning voltages are accepted by the regulator unless the frequency is lowered. In this study, we also cancel such restriction by revising the regulator driver.

4.6 VoltJockey Kernel

We alter the voltage regulator driver to bypass the limits above. Besides, we insert a restriction mechanism into the driver to prevent other programs from modifying the core voltage once we have changed it. User applications cannot invoke the voltage-related operations exposed by the voltage regulator driver due to a low privilege. However, we want the attacker procedure to run in userspace in order to make the attack more salable and extendable. Therefore, we develop a voltage manipulator kernel module (the VoltJockey kernel) to invoke the hardware regulator driver and provide user-application interfaces.

The module does not need to be integrated into the kernel image directly, and it can work in a loadable manner. In order to maintain the timing stability of the voltage manipulating process for a better fault injection precision, we implement most of the operations of the five attack steps mentioned earlier into the VoltJockey kernel. It first initializes the processor with a safe voltage and then clears the residual states. Next, it waits for the victim function and the injection points, and then conducts the low voltage glitches. Finally, the working environment is cleaned and reset to the original.

4.7 Victim Procedure Monitoring

The fault injection process is right subsequent to the time when the victim function is invoked. The attacker also needs to detect if the expected data changes are achieved to conclude whether the fault injection is successful. Therefore, the execution data of the victim procedure is required to be visible to the attacker procedure. However, the attacker procedure cannot directly read the data of victim procedure as they are different processes and are separated by the operating system (OS). In this study, we use cache side-channel attacks to speculate the victim procedure's data. There are several side-channel technologies available in the domain, for instance, *prime+probe* [58], *flush+reload* [70], *evict+reload* [50], and *flush+flush* [40]. Lipp et al. verified that the cache side channel attacks are effective for TrustZone and read T-table of AES executed in TrustZone successfully [49]. In VoltJockey, we monitor the instruction executions of the victim procedure by the method of *prime+probe*.

4.8 Timing

The previous delay and voltage glitch duration are time-sensitive factors. However, they are always a few cycles for the appropriate fault injections in real attacks. Therefore, the timing functions[4] provided by OS cannot fit the time precision demand. In the VoltJockey kernel module, we utilize the specified loops of `NOP` operations to count the execution cycles. In general, a `NOP` operation does nothing except for spending a cycle, it is a wonderful operation for timing in the clock accuracy. Besides, we embed the clock counting instructions in the required locations to eliminate the time consumption of function calls or returns. Moreover, the timing instructions are programmed to be not optimized by the compiler.

4.9 Voltage Manipulation

When inducing hardware faults, the abnormal voltages may make the system reboot. Besides, some attempts may fail. In order to avoid the system from rebooting and improve the attack efficiency and reliability, we employ the following techniques in VoltJockey.

- *Making the irrelevant cores busy.*

Figure 6 shows the minimum duration for the successful hardware fault injection with respect to different glitch voltages on the Nexus 6 processor. When the irrelevant cores (the processor cores except the attacker and victim cores) are idle (shut down), as the black curve shows, the minimum glitch duration is much longer than the other two situations shown by the red and blue curves. The reason might be that more power can be supplied to the victim core and the hardware faults are not easy to induce. When all the irrelevant cores are working with the normal workload (around 30%-50% core usage), as the red curve shows, the minimum duration of glitch voltages is also longer than those of the blue curve, which shows the case that all the irrelevant cores are busily working with high workloads (above 80% core usage). The reason might similarly be that more power can be supplied to the victim core when most

[4]In the Linux kernel, *udelay*, *mdelay*, *ndelay*, and *msleep* are widely utilized as the timekeeping functions.

cores are not consuming much power with low workloads and the fault injection is harder than that in the situation with most cores working with high workloads.

It should be noticed that it is not recommended to make all the irrelevant cores work at high frequencies because the hardware faults may also be induced into them. In this study, we let the irrelevant cores work at the same low frequency as attacker core.

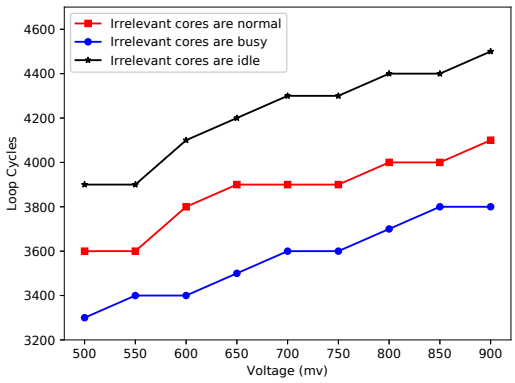


Figure 6: The minimum duration (number of NOP loops) of glitch voltage for the successful hardware fault injections. In this experiment, the frequency of attacker core and all the irrelevant cores is $0.42GHZ$ and the victim core's frequency is $2.65GHZ$.

- *Lower baseline voltage.*

From Figure 6, we can conclude that lower fault-injecting voltages result in easier fault injection process (less injection time). However, a remarkable voltage drop is also very easy to cause system reboot and interrupts the attack. In this study, we find that the switch from the baseline voltage to glitch voltage is more reliable and successful when their gap is reduced. Therefore, we configure the baseline voltage of the processor as the lowest voltage that is just available for all the cores.

- *Proper temperature.*

It is significant for the VoltJockey to select a proper working temperature to improve the injection reliability although processors are designed to fit for an extensive range of working temperatures. We find that an appropriately high temperature is very helpful for the injection success. In this study, we run a computing-heavy mission to raise the CPU temperature to the range of $35°C$ to $40°C$ before attacking.

We select a frequency from frequency table for the victim core that is idle and obtain the boundary voltages that will make the system stable. Moreover, we run a computing task that will make the victim core busy all the time and observe the boundary voltages again. Figure 5 displays the experiment results. The red line shows the lowest safe voltages for system being stable when the victim core is idle, and the blue line is the boundary voltages when the victim core is busy. As we can see from Figure 5, the voltage differentials among different frequencies are obvious. The boundary voltages when the core is vacant are higher than it is busy, especially in high frequency, this may because that busy states cost more energy. For a certain frequency, voltages between the black line and blue line are safe, voltages between the blue line and red line are likely to induce hardware faults, and voltages below the red line can take hardware faults in a very high possibility. Therefore, we can select the voltages below red line as the attack voltages.

4.10 Attacker Procedure Safety

Distributing the two different procedures to different cores can skirt crashes of the attacker procedure taken by the hardware faults of victim core. However, the glitch voltage may also cause the attacker core faulty especially when the frequency of attacker core is not lower than the victim core. Figure 5 indicates that the minimum acceptable voltage for a core is positive correlated to the frequency and it is possible to select a voltage that is safe for the low-frequency but harmful for the high-frequency. Fortunately, the frequencies of different cores are independent. Before attacking, we set the attacker core with a low frequency and the victim core with a high frequency. Then, we select the voltage that is safe for the attacker core but harmful for the victim core as the reason for hardware faults, which guarantees that the attacker procedure cannot be influenced in each voltage glitch trial.

4.11 Previous Delay

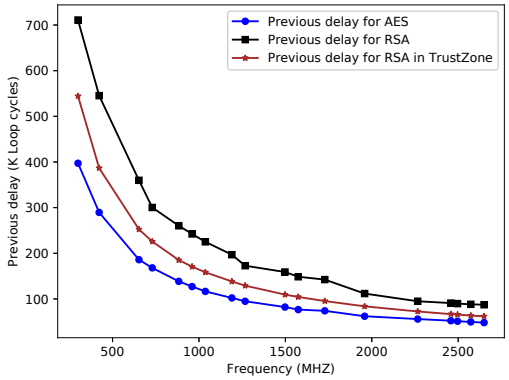


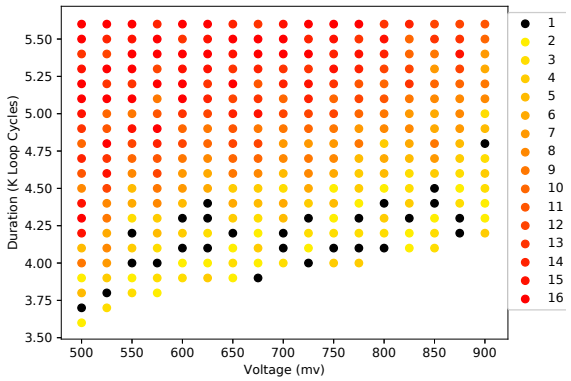
Figure 7: The previous delay for AES and RSA with different victim core frequencies. In this figure, the frequency of attacker core is $0.42GHZ$ and core voltage is $0.6V$.

The previous delay controls the fault injection sites. It is mainly determined by four factors: 1) the program codes of victim function; 2) the frequency of victim core; 3) the frequency of attacker core; 4) the core voltage. For a fixed victim function, peculiarly an encryption function, its implementations are commonly public; it is convenient to analyze the execution of the victim function and acquire the best fault injection points in advance. In this study, we implement an AES encryption function based on S-box and a RSA decryption function based on the Android cryptography library. We obtain the NOP loop cycles from the start of AES to the `MixColumn`

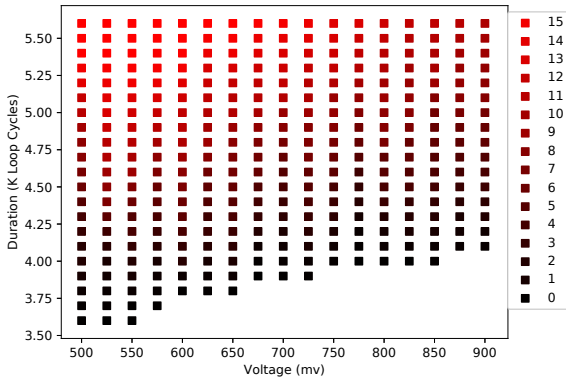
operation of the seventh round with different victim core frequencies but the same core voltage. Besides, we count the time from the start of RSA to instructions that transfer big-endian public modulus to little-endian modulus. Figure 7 shows the relationships between previous delay and the frequency of victim core for AES and RSA.

4.12 Glitch Voltage and Duration Pairs

Although voltage is the source of hardware faults, the malicious actions are decided by both voltage and its duration time. Our aim is to inject controllable hardware faults into the input state matrix of the eighth round of AES and the public modulus of RSA. Observing the data modifications caused by different voltage-duration pairs can help us select appropriate attacking parameters.



(a) Number of byte errors on the input of the eighth round of AES.



(b) Number of byte faults on the public modulus of RSA. We divide the numbers with 16 to show them in figure legend obviously.

Figure 8: The number of byte alterations with different glitch voltages and duration. In this figure, the attacker core frequency is $0.42GHZ$ **and victim core frequency is** $2.65GHZ$**.**

We fix the core frequencies, and then exert different core voltages and keep the voltage for various duration time. Next, we acquire the number of induced byte errors in the state matrix before the eighth round of AES and the public modulus of RSA. For each voltage-duration pair, we test five times and plot the average number of modified bits as the color-varying dots/squares in the Figure 8.

5 ATTACKING TRUSTZONE AES

In this section, we first present the experiments of applying the VoltJockey attack on the AES performed in the normal world, in which the encryption key can be obtained. Then, we carry out the attack to the AES running in the TrustZone, and the key can also be obtained successfully. Without loss of generality, the target AES is a 128-bits block cipher, which encompasses 10 rounds operations on the plaintext to generate corresponding ciphertext. There are four operations in each round except for the last round: 1) `ShiftRows` cyclically shifts the bytes in each row of the state matrix by a certain offset; 2) `SubBytes` replaces the plaintext bytes with corresponding S-box bytes; 3) `AddRoundKey` combines each element of the state matrix with the related key bytes using bitwise exclusive OR (XOR); 4) `MixColumns` blends every column of the state matrix with a linear transformation. The last round skips `MixColumns`.

The target device is a Google Nexus 6 mobile phone and the system version number is *LMY48M*. The key extraction is achieved on a DELL XPS laptop that has a virtual machine with 2G memories and two cores, in which Ubuntu 16.04 is performed.

5.1 Differential Fault Analysis on AES with a Single Fault

Brutely forcing the key of AES is impractically time-expensive, especially when the key is long enough. However, researchers have shown that the key hypotheses can be largely reduced if some errors are introduced into the middle execution of AES [5, 11, 23, 29, 67]. Triggering hardware faults to generate errors is a convenient and efficient choice. In this study, we employ the key extraction method proposed by Tunstall et al. [67] who verified that the key hypotheses can be reduced a lot using a two-stages algorithm when a single random byte fault is introduced into the input of the eighth round.

If a byte error happens in the input of the eighth round in a 128-bits AES, the functions `SubBytes`, `ShiftRows`, and `AddRoundKey` of the eighth round will record this error into the corresponding location of state matrix, and function `MixColumns` will next propagate this error to one column of state matrix. This will produce four byte-errors to the input of the ninth round. Similarly, the four byte-errors will be spread to the complete state matrix and cause sixteen errors to the input of the tenth round. There is no `MixColumns` operation in the tenth round, the other three functions replace, shift and XOR an element of state matrix once a time. Therefore, a byte error of input of the eighth round can be exactly propagated to the entire output bytes. By analyzing four sets of equations between the output bytes and the input bytes of the eighth round, the attacker will be able to have 2^{32} key assumptions for the encryption key.

The function `KeyExpansion` generates a key for next round using this round key, and the key schedule is invertible. Therefore, the ninth round key can be expressed by the key of the tenth round. Based on their relationship, attackers can further reduce the key conjectures to 2^8 [67]. In this study, we merge the key hypotheses for different ciphertext pairs to further scale down the searching space.

Table 2: Success rate with diverse parameters in AES attack.

Parameter		Normal world			TrustZone		
Volt (V)	Dura. (Loops)	#Error occurs	#One byte error	Succ. rate	#Error occurs	#One byte error	Succ rate
0.50	3700	54	1	1%	48	3	3%
0.55	3900	33	2	2%	37	1	1%
0.60	4100	56	3	3%	51	2	2%
0.65	4200	35	1	1%	62	5	5%
0.70	4200	26	2	2%	28	3	3%
0.75	4100	47	1	1%	58	2	2%
0.80	4400	45	2	2%	30	1	1%
0.85	4500	48	2	2%	46	2	2%
0.90	4800	32	1	1%	42	1	1%
Average		41.8	1.7	1.7%	44.7	2.2	2.2%

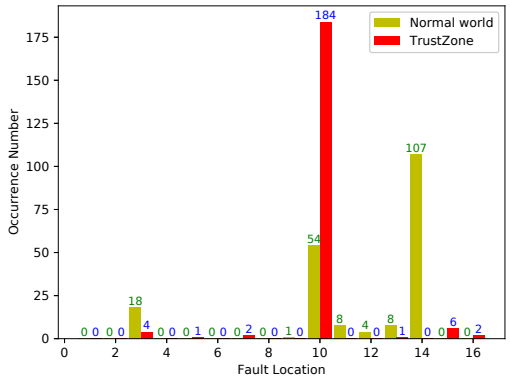


Figure 9: The distributions of modified bytes in AES attacks.

5.2 Normal World Attack

We select nine different parameters that take a single-byte error from Figure 8(a) to maliciously analyze the AES key of the normal world. We do not consider the step of waiting for the victim function due to the operations on plaintext in every round is fixed. We try the attack on AES for 100 times for each parameter and use the percentage of attack success to evaluate the effects of the parameter, which can be found in Table 2. In this attack, the frequency of attacker core is $0.42GHZ$ and victim core has a frequency of $2.65GHZ$. From the experiment results, we can observe that it is more likely to succeed when the glitch voltage is $0.6V$ and the duration is 4100 NOP loops. We then obtain the distributions of modified bytes in the input of the eighth round by conducting VoltJockey to acquire the AES key for 200 times with the attack parameter of $\{F_a = 0.42GHZ,\ F_v = 2.65GHZ,\ V_l = 0.6V,\ V_b = 1.055V,\ T_w = 0,\ T_p = 48132,\ T_d = 4100\}$. The distributions of modified bytes are illustrated with the yellow bar in Figure 9. We can see that the byte errors obtained by certain parameters are relatively stable.

5.3 Secure World Attack

TrustZone core uses the same physical core as the application core, and therefore is also vulnerable to the threats from VoltJockey like normal world. System users cannot specify a customized program to be executed in TrustZone, even privileged managers. In this study, we utilize the vulnerability described in [31] to execute AES in the TrustZone and regain the attacking parameters to perform attacks.

5.3.1 Inserting AES into TrustZone. As with the normal world that contains a general OS, *Qualcomm* provides a secure environment operating system (QSEOS) for supplying several security services to normal applications by means of system-calls. QSEOS does not trust any information provided by procedures outside TrustZone and always authenticates it. However, the system-call vulnerability in [31] exposes risks for attackers to escalate privilege, and then execute arbitrary codes in TrustZone. We develop shellcodes based on our AES program and combine it with the exploit codes of the vulnerability [31] to push AES instructions into the code cave of `widevine` trustlet, which is a digital rights management application. When the exploit codes are performed, the execution flow of TrustZone will jump to the code cave to execute the AES encryption function.

5.3.2 Waiting time for the victim function. The exploit codes employ several memory probes to trigger the vulnerability and the number of memory probes is uncertain. Therefore, attacking AES running in TrustZone requires to wait for the start of AES. Regarding the execution time from the start of exploit codes to the start of AES in a specific run as the waiting time for the victim function is incorrect. However, once the vulnerability is activated, the execution of exploit codes is considerably stable. We monitor the execution time from the function call that invokes the injected codes to the start of AES with different victim core frequencies and depict the experimental results with the green curve in Figure 10.

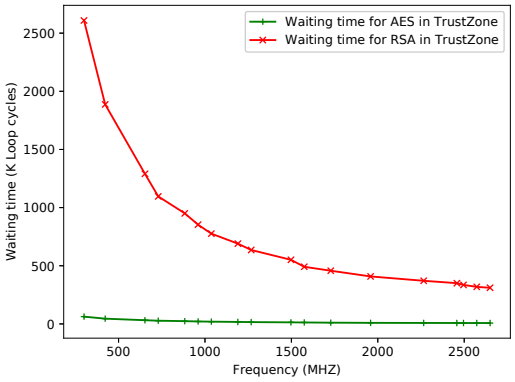


Figure 10: The waiting time for AES and RSA in TrustZone. In this experiment, the frequency of attacker core is $0.42GHZ$ and the core voltage is $0.6V$.

5.3.3 Attacking parameters. Although TrustZone adds some security-enhanced mechanisms and always checks an operation before it is performed, these measures are actually implemented in hardware and have negligible influences on the execution time of trustlets. Therefore, the attacking parameters in the normal world will be operative for breaking AES executed in TrustZone. In order to keep

the integrity of AES in TrustZone, we utilize the cache side channel attacks to continually check the state matrix.

5.3.4 Performing attack. We select the attacking parameters as them in the normal world attack to steal the encryption key and shows the success ratio in Table 2. The experiment on $\{F_a = 0.42GHZ, F_v = 2.65GHZ, V_l = 0.65V, V_b = 1.055V, T_w = 7680, T_p = 48132, T_d = 4200\}$ indicates that the success rate is about 5%, which is an acceptable value.

We examine the one-byte error distributions in the input of the eighth round by stealing AES key for 200 times with the above attacking parameters and illustrate them using the red bar in Figure 9. The byte errors are mainly concentrated on the tenth byte of state matrix due to the same attacking conditions. The error locations are not the same as those in the normal world because the working mechanisms of TrustZone is different from the normal world. When the error location is 10, it takes about 12 minutes to generate 1892 key hypotheses using differential fault analysis [43] with the laptop. Besides, the laptop spends around 8 minutes to generate 1159 key candidates when the fault location is 15. The two key hypotheses only have one common value, which is the encryption key.

6 BREACHING RSA-BASED TRUSTZONE AUTHENTICATION

In this section, we first provide the differential fault attack to the RSA decryption algorithm based on the signature verification function of an Android library *mincrypt* [37] by faulting the public modulus N. The attack makes it possible for the adversaries to revise the decryption result into a desired one. Next, we demonstrate the attack details of applying VoltJockey on RSA of the normal world and TrustZone respectively.

6.1 RSA Decryption and the Differential Attack

As one of the first public-key cryptosystems, RSA is vastly deployed on secure data transmissions, signature, and verification. However, RSA is still vulnerable to differential fault attacks [3, 4, 12, 65].

Given ciphertext C, public modulus N, public exponent e, RSA decrypts C to plaintext P through equation (2). Algorithm 1 shows the RSA decryption function RSA_Decoding (the public exponent e is limited to the widely used 65537 and 3) based on the Android library, which is an optimized implementation of equation (2) utilizing the Montgomery modular multiplication (function MonMul) whose output is calculated by operating the two integer parameters x and y, modulus N, and Montgomery radix r^{-1}(modular inverse of 2^{2048} (modulo N)) as: MonMul(x, y, N, r^{-1}) $\leftarrow x * y * r^{-1}$ *mod* N. What should be mentioned is that Algorithm 1 applies a machine word size radix *n0inv* (line 4) instead of r^{-1} to reduce the loop times and computation complexity of function MonMul.

$$P = C^e (mod\ N) \tag{2}$$

MonMul takes advantages of little-endian data to speed-up the multiplication operations. However, the input N and C are usually big-endian data. RSA_Decoding utilizes the function EndianInversion to convert N (line 5) and C (line 6) into little-endian N_{in} and C_{in} as well as transfer the little-endian multiplication result into

Algorithm 1 The RSA decryption algorithm

Input: Ciphertext, C; Public modulus, N; Public key, e
Output: Decoding plaintext, P

```
1:  procedure RSA_Decoding(C, N, e)
2:      r ← 2^2048
3:      R ← EndianInversion(r^2 mod N)
4:      n0inv ← 2^32 - ModuleInverse(N, 2^32)
5:      N_in ← EndianInversion(N)
6:      C_in ← EndianInversion(C)
7:      P_in ← MonMul(C_in, R, N_in, n0inv)
8:      P_in_temp ← P_in
9:      for i ∈ [0, bitlen(e) − 1) do
10:         P_in ← MonMul(P_in, P_in, N_in, n0inv)
11:     end for
12:     P_in ← MonMul(P_in, P_in_temp, N_in, n0inv)
13:     P_in ← MonMul(P_in, 1, N_in, n0inv)
14:     P ← EndianInversion(P_in)
15:     return P
16: end procedure
```

big-endian value as the decryption output (line 14). We append the implementation of EndianInversion in Algorithm 2 in Appendix.

Our attack goal is to guide the RSA_Decoding to output a desired plaintext $P(m)$, which is usually the hash of malicious message m in data verification applications. The tremendous computation complexity of factorizing the public modulus N is the security guarantee for RSA. Therefore, damaging N is a useful method to change the RSA behaviors. Function EndianInversion implements several byte-level OR and shift operations on the input to achieve the reversal of the input in machine word size. CLKscrew [65] proposes to induce hardware faults into this function to fault N and control the output. However, it requires the input r^{-1} (similar as *n0inv*) of MonMul is calculated based on the faulted N. Nevertheless, *n0inv* is computed before line 5 instead of after it in Algorithm 1. In this work, we extend the CLKscrew [65] to achieve the differential fault attack on Algorithm 1.

Similar with [65], we change N into N_m in line 5 and fabricate ciphertext $C_m^{'}$ to get the desired plaintext $P(m)$. N_m acts as the public modulus of the broken decryption function, it should be factorable and can be factorized in finite time. The Pollard's ρ algorithm [44] can be used to factorize N_m with sufficiently small factors of up to 60 bits. Besides, the Lenstra's Elliptic Curve factorization Method (ECM) is proposed to factorize a big number into up to 270 bits factors [47]. N_m does not need to exactly have two factors as traditional RSA algorithms and it sure will have multiple factors in real attacks since it is changed with random bytes errors. As long as it can be factorized, attackers can construct an RSA key pair $\{N_m, d_m, e\}$ for the broken RSA_Decoding with the Carmichael function [55]. Thus, they are able to encrypt $P(m)$ to get its ciphertext C_m with $C_m = P(m)^{d_m}$ *mod* N_m, where d_m is the private key. When $\{C_m, N_m, e\}$ is inputted into the RSA_Decoding, the result P_{in} of MonMul in line 7 will be as equation 3.

$$\begin{aligned} P_{in} \leftarrow C_m * (r^2\ mod\ N_m) * r_m^{-1}\ mod\ N_m \\ where\ (r_m^{-1} * r)\%N_m \equiv 1\%N_m \end{aligned} \tag{3}$$

If the broken RSA_Decoding decrypts the constructed input $C_m^{'}$ totally based on N_m and e, the output will be $P(m)$ if and only if $C_m^{'}$ equals to C_m. However, as we can see from Algorithm 1, N is used in line 3 and line 4, and thus is propagated to lines 7, 10, 12, and 13 before being altered in line 5. Therefore, the computation result $P_{in}^{'}$ of line 7 will be as equation 4 in real attacks.

$$P_{in}^{'} \leftarrow C_m^{'} * (r^2 \ mod \ N) * r^{-1} \ mod \ N_m \\ where \ (r^{-1} * r)\%N \equiv 1\%N \tag{4}$$

If $P_{in}^{'}$ equals to P_{in} and later MonMul is irrelevant to N, the output of RSA_Decoding will be $P(m)$. However, r_m^{-1} is hard to be equal with r^{-1} since N_m is different from N. Therefore, it is very tough to analyze $C_m^{'}$ if MonMul receives r^{-1} as one of the parameters. Luckily, Algorithm 1 is implemented based on *n0inv* instead of r^{-1}. *n0inv* is a machine word integer and means as equation 5.

$$n0inv = (-1/N)\%2^{32} \tag{5}$$

If *n0inv* that is calculated based on N_m equals to it based on N, the function MonMul after line 5 will be irrespective to N. In this situation, we can get the expected output $P(m)$ by simply making $P_{in}^{'} = P_{in}$, which is shown in the equation (6). The r^2, N, r_m^{-1}, P_{in}, and N_m are known values, it is convenient to calculate $C_m^{'}$.

$$C_m^{'} * (r^2 \ mod \ N) * r_m^{-1} \ mod \ N_m \equiv P_{in} \ mod \ N_m \tag{6}$$

In most situations, the injected hardware faults only influence a small continuous part of N. We generate the possible N_m with different lengths of continuous bits-errors and judge whether the *n0inv* based on them equals with it based on N. Table 3 gives the related data. For longer bits-error, it requires a mass of time to get all of the possible N_m and we do not test them. However, as shown in Table 3, for a specific N_m, it will have the same *n0inv* as N in a very high possibility. We develop a tool to check whether a N_m can be used in VoltJockey. Besides, we provide a tool to automatically generate $C_m^{'}$ with N, N_m, and desired $P(m)$.

Table 3: The equal rate of *n0inv* based on N and possible N_m.

Len of bit-error	4	8	12	16	20	24
#Possible N_m	7680	65280	696150	8388480	106954650	1426063275
#Available N_m	7560	64260	683880	8257410	104857755	1392574380
#Equal ration(%)	98.44	98.44	98.24	98.43	98.04	97.65

6.2 Normal World Attack

We run Algorithm 1 in the normal world and provide different external conditions to the victim core that performs it. We insert additional program codes to check the changes on the public modulus N during the execution. Figure 8(b) shows the number of bit modifications on N with different voltages and duration.

In this attack, we do not consider the waiting time for victim function as the target function is determined and there are no irrelevant codes are executed. According to our attack attempts on the victim RSA decryption function that uses the public modulus of *Widevine* (section C in Appendix), it has a good result when the glitch voltage is $0.65V$ and duration time is 3800 `NOP` loops. Among 500 attempts, 117 of them successfully inject faults into function EndianInversion, and 23 of them (about 4.6%) result in factorable N_m. In all of the factorable N_m, 18 of them are equal.

6.3 Secure World Attack

Verifying a trustlet requires four RSA-based authentications, which form a certificate chain. When loading a trustlet into TrustZone, the trusted execution environment (TEE) [24] authenticates the certificate chain by utilizing the RSA decryption on certificates and recognizes whether the decryption results are the same as code hashes. Tang et al. [65] described the signature decryption function in TEE, which is very similar to Algorithm 1. The main difference is that its MonMul is based on r^{-1}. What is more, r^{-1} relies on N_{in} instead of N. In other words, if hardware faults create a factorable N_m, we will be able to construct the poisonous input ciphertext.

6.3.1 Waiting time for the victim function. Different to the pure normal-world RSA decryption, authenticating a trustlet requires several calculations. Therefore, finding the best fault injection point is critical to improve the attack reliability. RSA_Decoding is loaded into memory with a constant start address. We utilize the *Prime+Probe* cache side-channel attack [49] to monitor the instruction executions when loading *widevine* into TrustZone in order to position the start time of the last RSA_Decoding. First, we load a block of dummy instructions from the attacker procedure to fill up the cache that RSA_Decoding will be mapped to. Then, the attacker procedure's dummy instructions and the victim procedure's RSA are executed on different cores in parallel. Because the processor cores share the same cache and the dummy instructions take place of the RSA's cache sets, the dummy instructions will miss in the cache if the processor starts to execute the last RSA_Decoding. The *Prime+Probe* can know the cache-missing event because the runtime of the instruction is increased. Therefore, we can finally know the right start time of the RSA_Decoding. We plot the waiting time with diverse frequencies in Figure 10 .

6.3.2 Attacking parameters. The signature decryption algorithm in TEE is invariant and the control flow transitions are fixed, therefore, the previous delay is almost stable. Since the function EndianInversion in TEE is loaded into a certain location, we time the duration time from the start of the last RSA decryption to the start of function EndianInversion to get the previous delay through the cache side signals. As shown in Figure 7, the previous delay for the RSA attack in normal world is different from it in TrustZone because the two RSA algorithms are actually different.

Our goal is to fault N into N_m. The two RSA algorithms have the same EndianInversion, therefore, the glitch voltage-duration pair that is usable for the attack in the normal world is also usable for the attack in TrustZone.

6.3.3 Performing attack. The location of N_m is hard-coded to an address 0x0FC8952C. We utilize the vulnerability exploit codes in AES attack to read N_m. The shell codes are implemented to transfer N_m into normal world for accessing outside TrustZone. Note that the relevant code sections for the signature validation are identical across versions [65]. In this attack, we first use VoltJockey to trigger hardware faults into the last RSA authentication to acquire an available N_m when loading the *widevine* trustlet. Then, we build a

signature based on the hash of malicious codes and N_m. Next, we replace the old signature with the new one. Finally, we perform our attacks with the same parameters to introduce the output of the last RSA certificate to be the desired hash at loading step.

According to our tests on different attacking parameters, the parameter of $\{F_a = 0.42GHZ,\ F_v = 2.65GHZ,\ V_l = 0.65V,\ V_b = 1.055V,\ T_w = 61942,\ T_p = 87267,\ T_d = 3800\}$ results in the best performance. Among 200 attempts, 73 of them successfully introduce faults to N, 21 of which are factorable and 15 of them are the same value. We use the N_{in} that has 2 bytes (byte 146 and 147) changed by faults to construct the attack signature C'_m, which is shown in section C in Appendix. An untrusted application can be successfully loaded into TrustZone in 94 attempts on average.

7 RELATED WORKS

Fault injections [1] are originally applied as the system reliability verification technologies. However, they are also effective in malicious attacks, especially combine with other attack methods, such as differential fault attack (DFA) [9, 10, 68]. The faults can be induced by hardware (changing the working environment with supererogatory hardware units, such as, frequency [21, 69], voltage [4, 6], temperature [42], light laser beam [18], ion radiation [62], sound [52], electromagnetic [20], et al.), software [25, 28, 61], or simulation [27, 46]. In this section, we review the related voltage-based and frequency-based fault injection attacks in this domain.

7.1 Voltage-based Fault Injection Attacks

Selmane et al. [63] might be the first ones to experimentally realize a practical fault attack by underpowering a smart card that preserves an AES function using a peripheral power supply, which is based on the random errors model invented by G. Piret and J.J. Quisquater [53]. Barenghi et al. [4] proposed a fault model based on the effects of underfeeding the power supply for an ARM processor and accomplished the attacks on the implementations of RSA primitives, in which the voltage supplied to the ARM processor is kept constantly lower than the normal voltages driven by Agilent 3631A. Barenghi et al. [6] also proposed an AES attack induced by low voltage-based faults, which is implemented by employing an operational amplifier to change the voltage supplied to the ARM-based development board. As an extension to the two works above [4, 6], Barenghi et al. [7] proposed a general low voltage-based fault attacks to break software implementations of AES and RSA.

Compared to the existing low voltage-based fault injection attacks, the VoltJockey method is more pragmatic and usable. The VoltJockey is totally controlled by software instead of using any auxiliary hardware units. Besides, we successfully conduct VoltJockey on a commercial device and demonstrate that the attack is dangerous to the secure execution environment. Moreover, the attack does not influence other applications running on non-victim cores.

7.2 Frequency-based Fault Injection Attacks

The basic idea of frequency-based fault injection attacks [21, 48, 59, 69] is that the expected output of a circuit may not be ready for the use in next unit if the frequency is too high to be met. The attacker can then obtain target data based on the wrong output that is influenced by the error. However, most of them require the frequency is changed by extra hardware units. Tang et al. proposed the Clkscrew [65] that uses overclocking technique to induce hardware faults with software. They verified that the ARM/Android devices do not lock core frequencies and the frequency of a core can be configured by another core in DVFS. Clkscrew exploits the above vulnerability of DVFS to overclock ARM processors and induce hardware faults.

However, the most significant prerequisite of CLKscrew is that the processor can be overclocked, which can be easily mitigated by frequency locking in which the frequency is locked in a certain range. In this study, the VoltJockey is proposed to be another usable fault-injection attack method, which exploits the legal voltage differentials corresponding to different frequencies. The voltage differences currently exist in most processors using the DVFS technique, no matter whether they are frequency locked or not.

8 COUNTERMEASURES AND EXTENSIONS

In this section, we discuss the possible hardware-based and software-based countermeasures to the VoltJockey, and also look forward to some future extensions of this study.

8.1 Hardware-based Countermeasures

8.1.1 Limiting voltage. Limiting the core voltage to break the voltage differentials is effective to prevent VoltJockey. The simplest method is to fix the core voltage and prevent all modifications on it with a watchdog circuit. This is effective but will introduce more energy wastes. Similarly, assigning a dedicated frequency to all cores is useful but is not economical for energy, either. What is more, the hardware can be designed to always configure the core voltage as the maximum voltage that corresponds to all the current core frequencies in the OPPs. The method can repel VoltJockey with less energy loss. However, the necessary alternations on hardware need to be implemented to limit the core voltage.

8.1.2 Abandoning shared hardware regulator. It is useful to address VoltJockey by ensuring every core has its own hardware voltage regulator. Besides, the voltage of a core can only be alterd by the software running on it. This countermeasure requires that the cross-core voltage setting should be prevented and the corresponding software-based regulator drivers should be redesigned. Another shortcoming of this design is that the energy optimization is achieved on a local core rather than the entire processor.

8.1.3 Separating the secure core from application cores. Taking malicious actions on the secure execution environment is the most common target of VoltJockey since a high privilege is required to induce hardware faults. The secure cores can be redesigned and are separated from application cores, like *CP15* co-processor in the ARM platforms. However, the design will be resource waste and enrich the hardware cost. Besides, the hardware regulator and corresponding drivers of secure cores should be distinguished with them of application cores.

8.2 Software-based Countermeasures

8.2.1 Integrity verification for regulator driver. In this study, we break some security mechanisms (section 4.4 and section 4.5) formerly enforced by the software regulator driver, which might be mitigated by some integrity-verification methods. The credential

of the regulator driver can be generated and stored in the secure boot image before releasing the software system. When loading the system into devices or calling functions of the regulator driver, TEE regenerates the credential and compares it with that stored in the boot image. The credential can be accomplished with *RSA*, *SHA-256*, *MD5*, and so on. However, such a mechanism still relies on the OS and software systems to guarantee the security.

8.2.2 Trusted voltage monitor. We can develop a trustlet based on TrustZone to continually monitor the core voltage and frequencies. If the trustlet finds that the core voltage is out of predefined legal ranges, it triggers a warning or a privileged voltage recover operation or just a system reboot. However, the trade-off between the voltage detection rate/reliability and power consumption may still leave the possibility for attackers to find feasible injection points.

8.3 Future Extensions

The VoltJockey may also be promising in other related fields.

8.3.1 Attacking other cryptosystems. In addition to AES and RSA, many other cryptosystems that have been theoretically proved to be vulnerable to DFA, like DES [56], 3G-SNOW [19], ECC [8], SMS4 [22] and Grain family [60], may become the next targets.

8.3.2 Being a general attack helper. VoltJockey might also be employed to achieve malicious data modifications, which may help attackers prepare vulnerabilities for other attacks. For example, OS can be misled to update software from a disguise website if the update source is changed and the RSA-based verification is evaded; Internet data will be sent to another destination if it is modified at packaging; the control flow [54] might be changed if the destinations of direct or indirect jump instructions are altered, or the conditions of conditional instructions are modified.

8.3.3 Attacking Intel software-guard extensions (SGX). SGX [26, 45] extends the hardware and software architectures of Intel-based SoC for providing a trusted execution environment. The trust base of SGX is only bound to CPU [17]. Nevertheless, DVFS is enabled by Intel processors and the voltage differences between disparate frequencies remain subsistence even though most of Intel processors do not enable overclocking technology[5]. Therefore, the VoltJockey might also be effective for attacking Intel SGX.

9 CONCLUSION

ARM TrustZone is a system-wide secure technology for a wide variety of devices. Most of the current attack techniques to TrustZone are software vulnerability-based. In this paper, we propose the VoltJockey, a hardware fault-based attack to TrustZone using software-controlled voltage manipulation, which exploits the voltage management vulnerability of DVFS. We deploy the VoltJockey on an ARM-based *Krait* multi-core processor, whose core frequencies can be different but the processor voltage is controlled with a shared hardware regulator. Experiments show that the attack method can successfully acquire the key of AES protected by TrustZone, and guide the RSA-based signature verification to output desired plaintexts, which suggests a comparable efficiency to side channels in obtaining TrustZone-guarded credentials, as well as a potential capability in cheating the RSA-based verification chain. At the end of the paper, we also give detailed analyses and discussions on the possible countermeasures to the attack and look forward to other potential and promising extensions of this work.

[5]The processors whose version number ended with *K* support overclocking, and some processors ended with *X* and *E* enable overclocking. However, the most deployed ultra-low-voltage processors whose version number ended with *U* cannot be overclocked.

ACKNOWLEDGMENTS

We gratefully thank the anonymous reviewers for their helpful comments and Xizhen Huang for her works in the experiments. We appreciate Mr. Adrian Tang's kind suggestions in the personal communications regarding the experiments. This work was supported in part by the National Key Research and Development Plan of China under Grant No. 2016YFB1000303 and the Guangdong Province Key Project of Science and Technology under Grant No. 2018B010115002.

REFERENCES

[1] Jean Arlat, Martine Aguera, Louis Amat, Yves Crouzet, J-C Fabre, J-C Laprie, Eliane Martins, and David Powell. 1990. Fault injection for dependability validation: a methodology and some applications. *IEEE Transactions on Software Engineering* 16, 2 (Feb 1990), 166–182. https://doi.org/10.1109/32.44380

[2] A ARM. 2009. *Security technology building a secure system using trustzone technology (white paper).* ARM.

[3] Feng Bao, Robert H Deng, Yongfei Han, A Jeng, A Desai Narasimhalu, and T Ngair. 1997. Breaking public key cryptosystems on tamper resistant devices in the presence of transient faults. In *International Workshop on Security Protocols*. Springer, Berlin, Heidelberg, 115–124.

[4] Alessandro Barenghi, Guido Bertoni, Emanuele Parrinello, and Gerardo Pelosi. 2009. Low Voltage Fault Attacks on the RSA Cryptosystem. In *2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE Computer Society Press, Lausanne, Switzerland, 23–31. https://doi.org/10.1109/FDTC.2009.30

[5] Alessandro Barenghi, Guido M Bertoni, Luca Breveglieri, Mauro Pellicioli, and Gerardo Pelosi. 2010. Fault attack on AES with single-bit induced faults. In *2010 Sixth International Conference on Information Assurance and Security*. IEEE, Atlanta, GA, USA, 167–172. https://doi.org/10.1109/ISIAS.2010.5604061

[6] A. Barenghi, G. M. Bertoni, L. Breveglieri, M. Pellicioli, and G. Pelosi. 2010. Low voltage fault attacks to AES. In *2010 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. IEEE, Anaheim, CA, USA, 7–12.

[7] Alessandro Barenghi, Guido M Bertoni, Luca Breveglieri, and Gerardo Pelosi. 2013. A fault induction technique based on voltage underfeeding with application to attacks against AES and RSA. *Journal of Systems and Software* 86, 7 (2013), 1864–1878.

[8] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. 2012. Fault Injection Attacks on Cryptographic Devices: Theory, Practice, and Countermeasures. *Proc. IEEE* 100, 11 (Nov 2012), 3056–3076.

[9] Eli Biham and Adi Shamir. 1997. Differential fault analysis of secret key cryptosystems. In *Advances in Cryptology — CRYPTO '97*, Burton S. Kaliski (Ed.). Springer, Berlin, Heidelberg, 513–525.

[10] Eli Biham and Adi Shamir. 2012. *Differential cryptanalysis of the data encryption standard*. Springer Science & Business Media, Berlin/Heidelberg, Germany.

[11] Johannes Blömer and Jean-Pierre Seifert. 2003. Fault Based Cryptanalysis of the Advanced Encryption Standard (AES). In *Financial Cryptography*. Springer, Berlin, Heidelberg, 162–181.

[12] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. 1997. On the Importance of Checking Cryptographic Protocols for Faults. In *Advances in Cryptology — EUROCRYPT '97*, Walter Fumy (Ed.). Springer, Berlin, Heidelberg, 37–51.

[13] Klug Brian and Lal Shimpi Anand. 2011. *Qualcomm's New Snapdragon S4: MSM8960 & Krait Architecture Explored*. Qualcomm. https://www.anandtech.com/show/4940/qualcomm-new-snapdragon-s4-msm8960-krait-architecture

[14] Aaron Carroll and Gernot Heiser. 2010. An Analysis of Power Consumption in a Smartphone. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference (USENIXATC'10)*. USENIX Association, Berkeley, CA, USA, 21–21. http://dl.acm.org/citation.cfm?id=1855840.1855861

[15] Yue Chen, Yulong Zhang, Zhi Wang, and Tao Wei. 2017. Downgrade Attack on TrustZone. *arXiv preprint arXiv* 1707.05082 (2017), 26.

[16] Haehyun Cho, Penghui Zhang, Donguk Kim, Jinbum Park, Choong-Hoon Lee, Ziming Zhao, Adam Doupé, and Gail-Joon Ahn. 2018. Prime+Count: Novel Cross-world Covert Channels on ARM TrustZone. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC '18)*. ACM, New York, NY, USA, 441–452. https://doi.org/10.1145/3274694.3274704

[17] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016, 086 (2016), 1–118.

[18] Franck Courbon, Philippe Loubet-Moundi, Jacques J. A. Fournier, and Assia Tria. 2014. Adjusting Laser Injections for Fully Controlled Faults. In *Constructive Side-Channel Analysis and Secure Design*, Emmanuel Prouff (Ed.). Springer International Publishing, Cham, 229–242.

[19] Blandine Debraize and Irene Marquez Corbella. 2009. Fault Analysis of the Stream Cipher Snow 3G. In *2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE, Lausanne, Switzerland, 103–110. https://doi.org/10.1109/FDTC.2009.33

[20] Amine Dehbaoui, Jean-Max Dutertre, Bruno Robisson, and Assia Tria. 2012. Electromagnetic Transient Faults Injection on a Hardware and a Software Implementations of AES. In *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography(FDTC)*, Vol. 00. IEEE, Leuven, Belgium, 7–15. https://doi.org/10.1109/FDTC.2012.15

[21] Jeroen Delvaux and Ingrid Verbauwhede. 2014. Fault Injection Modeling Attacks on 65 nm Arbiter and RO Sum PUFs via Environmental Changes. *IEEE Transactions on Circuits and Systems I: Regular Papers* 61, 6 (June 2014), 1701–1713.

[22] ZB Du, Zhen WU, Min WANG, and JT Rao. 2015. Improved chosen-plaintext power analysis attack against SM4 at the round-output. *Journal on Communications* 36, 10 (2015), 85–91.

[23] Pierre Dusart, Gilles Letourneux, and Olivier Vivolo. 2003. Differential Fault Analysis on A.E.S. In *Applied Cryptography and Network Security*, Jianying Zhou, Moti Yung, and Yongfei Han (Eds.). Springer, Berlin, Heidelberg, 293–306.

[24] Jan-Erik Ekberg, Kari Kostiainen, and N Asokan. 2013. Trusted execution environments on mobile devices. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, ACM, Berlin, Germany, 1497–1498.

[25] Sho Endo, Takeshi Sugawara, Naofumi Homma, Takafumi Aoki, and Akashi Satoh. 2011. An on-chip glitchy-clock generator for testing fault injection attacks. *Journal of Cryptographic Engineering* 1, 4 (21 Oct 2011), 265. https://doi.org/10.1007/s13389-011-0022-y

[26] Ben Fisch, Dhinakaran Vinayagamurthy, Dan Boneh, and Sergey Gorbunov. 2017. Iron: functional encryption using Intel SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, ACM, Dallas, Texas, USA, 765–782.

[27] Peter Folkesson, Sven Svensson, and Johan Karlsson. 1998. A comparison of simulation based and scan chain implemented fault injection. In *Digest of Papers. Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing (Cat. No.98CB36224)*. IEEE, Munich, Germany, Germany, 284–293. https://doi.org/10.1109/FTCS.1998.689479

[28] Aurélien Francillon and Claude Castelluccia. 2008. Code Injection Attacks on Harvard-architecture Devices. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS '08)*. ACM, New York, NY, USA, 15–26. https://doi.org/10.1145/1455770.1455775

[29] Christophe Giraud. 2005. DFA on AES. In *Advanced Encryption Standard – AES*, Hans Dobbertin, Vincent Rijmen, and Aleksandra Sowa (Eds.). Springer, Berlin, Heidelberg, 27–41.

[30] Github 2015. *Qsee privilege escalation exploit using prdiag* commands*. Github. https://github.com/laginimaineb/cve-2015-6639

[31] Github 2016. *The exploit codes of CVE-2016-2431*. Github. https://github.com/laginimaineb/cve-2016-2431

[32] Google 2014. *Qualcomm Krait PMIC regulator driver source code*. Google. https://android.googlesource.com/kernel/msm/+/android-msm-shamu-3.10-lollipop-mr1/arch/arm/mach-msm/krait-regulator-pmic.c

[33] Google 2014. *Qualcomm Krait PMIC voltage regulator source code*. Google. https://android.googlesource.com/kernel/msm/+/android-msm-shamu-3.10-lollipop-mr1/arch/arm/mach-msm/krait-regulator.c

[34] Google 2014. *Qualcomm SPM source code*. Google. https://android.googlesource.com/kernel/msm/+/android-msm-shamu-3.10-lollipop-mr1/arch/arm/mach-msm/spm-v2.c

[35] Google 2014. *The regulator* dtsi *source file in Google Nexus 6*. Google. https://android.googlesource.com/kernel/msm/+/android-msm-shamu-3.10-lollipop-mr1/arch/arm/boot/dts/qcom/apq8084-regulator.dtsi

[36] Google 2015. *The* dtsi *source file in Google Nexus 6*. Google. https://android.googlesource.com/kernel/msm/+/android-msm-shamu-3.10-lollipop-mr1/arch/arm/boot/dts/qcom/apq8084.dtsi

[37] Google 2015. *Mincrypt library*. Google. https://android.googlesource.com/platform/system/core.git/+/android-6.0.1_r1/libmincrypt

[38] Google 2015. *Qualcomm Krait CPUfreq driver source code*. Google. https://android.googlesource.com/kernel/msm/+/android-msm-shamu-3.10-lollipop-mr1/drivers/cpufreq/cpufreq.c

[39] Google 2015. *Qualcomm Krait PMIC frequency driver source code*. Google. https://android.googlesource.com/kernel/msm/+/android-msm-shamu-3.10-lollipop-mr1/drivers/clk/qcom/clock-krait.c

[40] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez (Eds.). Springer International Publishing, Cham, 279–299.

[41] Inki Hong, Darko Kirovski, Gang Qu, Miodrag Potkonjak, and Mani B Srivastava. 1999. Power optimization of variable-voltage core-based systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 18, 12 (Dec 1999), 1702–1714. https://doi.org/10.1109/43.811318

[42] Michael Hutter and Jörn-Marc Schmidt. 2013. The temperature side channel and heating fault attacks. In *International Conference on Smart Card Research and Advanced Applications*. Springer, Cham, 219–235.

[43] Philipp Jovanovic. 2013. *Differential fault analysis framework for AES128*. Github. https://github.com/Daeinar/dfa-aes

[44] Jonathan Katz, Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. 1996. *Handbook of applied cryptography*. CRC press, Boca Raton, FL.

[45] Sebastian Krieter, Tobias Thiem, and Thomas Leich. 2019. Using Dynamic Software Product Lines to Implement Adaptive SGX-enabled Systems. In *Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems*. ACM, ACM, Leuven, Belgium, 9.

[46] Dongwoo Lee and Jongwhoa Na. 2009. A Novel Simulation Fault Injection Method for Dependability Analysis. *IEEE Design Test of Computers* 26, 6 (Nov 2009), 50–61. https://doi.org/10.1109/MDT.2009.135

[47] Hendrik W Lenstra Jr. 1987. Factoring integers with elliptic curves. *Annals of mathematics* 126, 3 (1987), 649–673.

[48] Yang Li, Kazuo Sakiyama, Shigeto Gomisawa, Toshinori Fukunaga, Junko Takahashi, and Kazuo Ohta. 2010. Fault sensitivity analysis. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, Springer, Berlin, Heidelberg, 320–334.

[49] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 549–564. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/lipp

[50] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *2015 IEEE Symposium on Security and Privacy*. IEEE, San Jose, CA, USA, 605–622. https://doi.org/10.1109/SP.2015.43

[51] Huasong Meng, Vrizlynn L.L. Thing, Yao Cheng, Zhongmin Dai, and Li Zhang. 2018. A survey of Android exploits in the wild. *Computers & Security* 76 (2018), 71 – 91. https://doi.org/10.1016/j.cose.2018.02.019

[52] Thomas S. Messerges. 2000. Using Second-Order Power Analysis to Attack DPA Resistant Software. In *Cryptographic Hardware and Embedded Systems — CHES 2000*, Çetin K. Koç and Christof Paar (Eds.). Springer, Berlin, Heidelberg, 238–251.

[53] Gilles Piret and Jean-Jacques Quisquater. 2003. A Differential Fault Attack Technique against SPN Structures, with Application to the AES and Khazad. In *Cryptographic Hardware and Embedded Systems - CHES 2003*, Colin D. Walter, Çetin K. Koç, and Christof Paar (Eds.). Springer, Berlin, Heidelberg, 77–88.

[54] Pengfei Qiu, Yongqiang Lyu, Jiliang Zhang, Dongsheng Wang, and Gang Qu. 2018. Control flow integrity based on lightweight encryption architecture. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 7 (2018), 1358–1369.

[55] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. 2016. Flip Feng Shui: Hammering a Needle in the Software Stack. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 1–18.

[56] Matthieu Rivain. 2012. *Differential Fault Analysis of DES*. Springer, Berlin, Heidelberg, 37–54.

[57] Dan Rosenberg. 2014. Qsee trustzone kernel integer over flow vulnerability. In *Black Hat conference*. Blackhat, Las Vegas, NV, USA, 26.

[58] Majid Sabbagh, Yunsi Fei, Thomas Wahl, and A Adam Ding. 2018. SCADET: A Side-Channel Attack Detection Tool for Tracking Prime-Probe. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, IEEE, San Diego, CA, USA, 1–8.

[59] Dhiman Saha, Debdeep Mukhopadhyay, and Dipanwita Roy Chowdhury. 2009. A Diagonal Fault Attack on the Advanced Encryption Standard. *IACR Cryptology ePrint Archive* 2009 (01 2009), 581.

[60] Santanu Sarkar, Subhadeep Banik, and Subhamoy Maitra. 2015. Differential Fault Attack against Grain Family with Very Few Faults and Minimal Assumptions. *IEEE Trans. Comput.* 64, 6 (June 2015), 1647–1657. https://doi.org/10.1109/TC.2014.2339854

[61] Mark Seaborn and Thomas Dullien. 2015. Exploiting the DRAM rowhammer bug to gain kernel privileges. *Black Hat* 15 (2015), 71.

[62] Norbert Seifert, Vinod Ambrose, B Gill, Q Shi, R Allmon, C Recchia, S Mukherjee, N Nassif, J Krause, J Pickholtz, et al. 2010. On the radiation-induced soft error performance of hardened sequential elements in advanced bulk CMOS technologies. In *2010 IEEE International Reliability Physics Symposium*. Blackhat, Anaheim, CA, USA, 188–197. https://doi.org/10.1109/IRPS.2010.5488831

[63] Nidhal Selmane, Sylvain Guilley, and Jean-Luc Danger. 2008. Practical Setup Time Violation Attacks on AES. In *2008 Seventh European Dependable Computing Conference*. Blackhat, Kaunas, Lithuania, 91–96.

[64] Di Shen. 2015. Exploiting Trustzone on Android. *Black Hat USA* 1, 1-7 (2015), 7.

[65] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. 2017. CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 1057–1074.

[66] C Josh Thomas and Nathan Keltner. 2014. Reflections on Trusting TrustZone. In *Black Hat conference*. Blackhat, Las Vegas, NV, USA, 33.

[67] Michael Tunstall, Debdeep Mukhopadhyay, and Subidh Ali. 2011. Differential Fault Analysis of the Advanced Encryption Standard Using a Single Fault. In *Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication*. Springer, Berlin, Heidelberg, 224–233.

[68] Qian Wang, An Wang, Gang Qu, and Guoshuang Zhang. 2017. New Methods of Template Attack Based on Fault Sensitivity Analysis. *IEEE Transactions on Multi-Scale Computing Systems* 3, 2 (April 2017), 113–123. https://doi.org/10.1109/TMSCS.2016.2643638

[69] Qian Wang, An Wang, Liji Wu, Gang Qu, and Guoshuang Zhang. 2015. Template attack on masking AES based on fault sensitivity analysis. In *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, Vol. 00. Blackhat, Washington, DC, USA, 96–99. https://doi.org/10.1109/HST.2015.7140245

[70] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 719–732. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

A FAULTS INDUCED BY A HIGH VOLTAGE

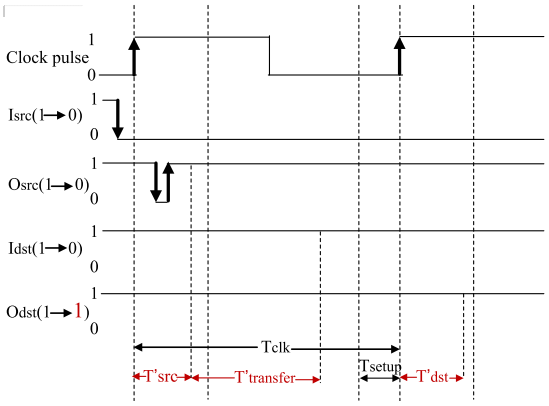


Figure 11: The bit-flip will be inserted if E_{src} becomes unstable because of a high voltage.

B THE IMPLEMENTATION OF FUNCTION ENDIANINVERSION IN ALGORITHM 1

Algorithm 2 The endian reversal algorithm

Input: The variable that needs to be endian reversed, V
Output: The reversed bytes sequence, S

function EndianInversion(V)
  $S \leftarrow \{0\}$
  for each $i \in [0, bytelen(V)/4 - 1]$ **do**
    $S_{temp} \leftarrow V[i * 4] \ll 24$
    $S_{temp} \leftarrow (V[i * 4 + 1] \ll 8) \mid S_{temp}$
    $S_{temp} \leftarrow (V[i * 4 + 2] \gg 8) \mid S_{temp}$
    $S_{temp} \leftarrow (V[i * 4 + 3] \gg 24) \mid S_{temp}$
    $Index \leftarrow bytelen(V) - (i + 1) * 4$
    $S[Index, Index + 3] \leftarrow S_{temp}$;
  end for
  return S
end function

C AN EXAMPLE ERROR ON THE PUBLIC MODULUS OF THE `WIDEVINE` TRUSTLET

Original N:
0xc44dc735f6682a261a0b8545a62dd13df4c646a5ede482cef858925baa1811fa
0284766b3d1d2b4d6893df4d9c045efe3e84d8c5d03631b25420f1231d8211e23
22eb7eb524da6c1e8fb4c3ae4a8f5ca13d1e0591f5c64e8e711b3726215cec59ed
0ebc6bb042b917d445288**8791**5fdf764df691d183e16f31ba1ed94c84b476e74b
488463e85551022021763af35a64ddf105c1530ef3fcf7e54233e5d3a4747bbb17
328a63e6e3384ac25ee80054bd566855e2eb59a2fd168d3643e44851acf0d118f
b03c73ebc099b4add59c39367d6c91f498d8d607af2e57cc73e3b5718435a8112
3f080267726a2a9c1cc94b9c6bb6817427b85d8c670f9a53a777511b **Corrupted N_{in}:**
0x...8cb3...
Factors of corrupted N_{in}:
0x11, 0x033b, 0x377, 0x010819f1285c6b307a82beba93d7c496488...
Private key d_m:
0x062cde999954a9ced6840f2b04ae4d4187baa01a5044c0242c70dbe...