Hardware Security Flaws and their Impact on Software Security

Anthony Philippe Christoforou

A report submitted as part of the requirements for the degree
of Computer Science
at the Faculty of Science
University of Geneva
Geneva, Switzerland

April 2024

Supervisor Prof. Eduardo Solana

Abstract

Disclaimer

The research and discussions presented in this thesis are intended solely for educational purposes. The case studies, including the examination of the "fusee-gelee" vulnerability within the Nintendo Switch console, are explored to contribute to the academic understanding of hardware security and side-channel resistances. Under no circumstances should the content of this thesis be used to engage in unlawful activities, including the hacking or modification of devices such as the Nintendo Switch. The author and academic institution do not condone unauthorized hacking, do not provide guidance for engaging in such activities, and are not liable for any actions taken by individuals who misuse the information provided.

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

> Companies spend millions of dollars on firewalls, encryption, and secure access devices, and it's money wasted because none of these measures address the weakest link in the security chain: the people who use, administer, operate and account for computer systems that contain protected information.
>
> *Kevin Mitnick*,
> The Art of Deception: Controlling the Human Element of Security

1.1 Background

The importance of hardware security has escalated in our increasingly digital world, where the proliferation of smart devices makes every aspect of our lives interconnected and, potentially, vulnerable. This surge in connectivity has broadened the attack surface for malicious actors, making hardware security a critical pillar of our digital infrastructure's integrity.

Video game consoles, such as the Nintendo Switch, epitomize the sophisticated nature of modern hardware. These devices are not merely platforms for entertainment but intricate ecosystems comprising proprietary software, custom hardware components, and online services. They embody a blend of performance, entertainment, and connectivity, making them a prime target for exploitation.

The Nintendo Switch, in particular, presents an intriguing case study in hardware security. Its popularity and unique design have attracted attention not only from millions of users worldwide but also from individuals and groups looking to exploit potential vulnerabilities for various purposes, ranging from piracy to the customization of the device beyond the manufacturer's intended limitations. The discovery of the Fusee Gelee vulnerability, a significant exploit within the Switch's boot ROM, highlights the ongoing tension between hardware manufacturers, who strive to secure their devices, and the hacker community, which continually seeks to find and exploit vulnerabilities.

To understand the gravity of such exploits, one must consider the broader implications of hardware vulnerabilities. Unlike software flaws, which can often be patched through updates, vulnerabilities at the hardware level can be more challenging to address. They may require physical recalls or rely on mitigation strategies that can only minimize the risk rather than eliminate it. The presence of such vulnerabilities underscores the necessity of robust hardware security measures not only to protect intellectual property and user data but also to maintain trust in digital ecosystems.

1.2 Thesis Statement

The core of this thesis is encapsulated in the exploration of the Fusee Gelee exploit, a significant vulnerability within the Nintendo Switch's security architecture. This exploit does not merely represent a singular flaw within a popular gaming console; it symbolizes the broader challenges and implications inherent in securing complex hardware systems in the digital age. The statement of this thesis posits that:

The detailed examination of the Fusee Gelee exploit serves as a critical case study, shedding light on the broader issues surrounding hardware vulnerabilities, their potential impacts on various digital ecosystems, and the evolving landscape of hardware security measures.

This statement underscores the exploit's role as a window into understanding the dynamics between hardware designers and the hacker community. It emphasizes the need to delve into the technical specifics of such vulnerabilities to appreciate their severity and potential ramifications. The analysis of Fusee Gelee is not confined to its technical execution but extends to understanding its implications on the security posture of similar devices and systems.

The case of Fusee Gelee is particularly illuminating because it involves exploiting a vulnerability in the boot ROM of the Nintendo Switch, a component that is fundamental to the device's operation and, crucially, cannot be modified once manufactured. This immutability makes mitigating the exploit particularly challenging and highlights the

importance of foresight and robust security measures in hardware design.

The significance of this exploit—and the reason it is central to this thesis—is twofold:

1. **Technical Insight**: It provides a deep technical insight into the nature of hardware vulnerabilities, particularly those embedded deeply within a system's architecture, inaccessible to conventional software patches.

2. **Broader Implications**: It prompts a discussion on the broader implications for the security of interconnected devices. As devices become increasingly complex and integral to personal and professional life, understanding and mitigating such vulnerabilities become paramount.

In sum, the thesis statement frames the Fusee Gelee exploit as a pivotal case study for understanding the intricacies of hardware security and the continuous efforts required to protect against evolving threats.

The primary objectives of this thesis are designed to build a comprehensive understanding of hardware vulnerabilities, with a particular focus on the Fusee Gelee exploit within the Nintendo Switch, and to evaluate mitigation strategies that can be employed against such vulnerabilities. The objectives are outlined as follows:

1. **Comprehensive Overview of Hardware Vulnerabilities**: To conduct a thorough literature review that maps the landscape of hardware vulnerabilities, categorizing them based on their nature, origin, and impact. This review aims to establish a foundational understanding of the challenges in hardware security, setting the stage for a deeper exploration of specific exploits like Fusee Gelee.

2. **In-depth Analysis of the Fusee Gelee Exploit**: To dissect the Fusee Gelee exploit in detail, examining how it was discovered, its technical mechanisms, and how it manages to circumvent the Nintendo Switch's security measures. This analysis will provide insights into the exploit's workings, offering a case study of how a single vulnerability can have significant ramifications.

3. **Assessment of Mitigation Strategies**: To evaluate existing strategies employed to mitigate hardware vulnerabilities, focusing on their applicability and effectiveness in the context of the Fusee Gelee exploit. This will involve an examination of both reactive measures taken post-discovery and proactive strategies that can be integrated into the design and manufacturing processes to prevent similar vulnerabilities.

1.2.1 Detailed Breakdown of Objectives

- **Objective 1**: The literature review will encompass academic papers, security conference proceedings, and industry whitepapers to create a taxonomy of hardware vulnerabilities. This will include discussions on side-channel attacks, fault injection, hardware Trojans, and more, providing a broad perspective on the types of challenges faced in securing hardware.
- **Objective 2**: The analysis of Fusee Gelee will be technical, involving an examination of the Tegra X1's boot ROM, the role of the BootROM bug, and how the vulnerability is exploited to run arbitrary code. It will also cover the implications of such an exploit, from the perspective of both security professionals and end-users.
- **Objective 3**: The assessment will cover specific mitigation strategies, such as secure boot, hardware patches, and the use of Trusted Execution Environments (TEEs). It will critically analyze the effectiveness of Nintendo's responses and general practices in the industry for preventing and responding to hardware vulnerabilities.

1.2.2 Supporting Concepts and Tools

To achieve these objectives, the thesis will leverage various concepts and tools, including:

- **Reverse Engineering**: Techniques and tools for reverse engineering will be discussed, as they are crucial for uncovering and understanding hardware vulnerabilities.
- **Cryptography**: The role of cryptographic measures in securing hardware, particularly in the context of secure boot processes and data protection.
- **Security Frameworks**: Examination of frameworks and standards for hardware security, such as the Trusted Computing Group's guidelines and the Common Criteria for Information Technology Security Evaluation.

1.3 Scope

While the Fusee Gelee exploit within the Nintendo Switch serves as the focal point of this thesis, it is crucial to delineate the boundaries of the discussion to maintain a focused and coherent analysis. The scope of this thesis includes the points talked about in the Objectives

1.3.1 Limitations

To ensure a focused analysis, the thesis will not cover:

- **Software Vulnerabilities**: While recognizing that hardware and software security are often intertwined, this thesis will limit its discussion to hardware vulnerabilities and the specific intersection with software only where relevant to the Fusee Gelee exploit.
- **Comprehensive Survey of All Hardware Vulnerabilities**: Given the vast and evolving nature of hardware vulnerabilities, the thesis will not provide an exhaustive survey of all known hardware vulnerabilities but will instead highlight those most relevant to the context of the Nintendo Switch and similar consumer electronics.
- **Detailed Technical Solutions**: While mitigation strategies will be discussed, the thesis will not delve into the detailed technical design of specific security solutions, focusing instead on the conceptual and strategic levels.

1.3.2 Rationale for Scope

The chosen scope ensures that the thesis remains manageable while providing valuable insights into a significant area of hardware security. By focusing on the Fusee Gelee exploit, the thesis leverages a specific, well-documented case to explore broader themes and challenges in hardware security, making it both relevant and accessible to a wider audience, including those not deeply versed in hardware engineering.

1.3.3 Implications of Scope

By adhering to this scope, the thesis aims to contribute to the discourse on hardware security by:

- Providing a detailed case study of a significant exploit, offering insights that can inform both academic research and practical security measures.
- Highlighting the ongoing challenges in securing hardware against increasingly sophisticated exploits, underscoring the need for continued innovation and vigilance in hardware design and security practices.
- Encouraging a broader discussion on the balance between hardware security, functionality, and user freedom, particularly in consumer electronics where these factors are in constant tension.

This scoped approach allows for a thorough exploration of the chosen topic while acknowledging the vast and complex nature of hardware security as a field, thereby setting a clear direction for the research and analysis that follows.

1.4 Structure

The thesis is meticulously organized to navigate through the complexities of hardware security with a spotlight on the Fusee Gelee exploit.

In the State of the Art in Hardware Security Flaws chapter, the thesis delves into the historical context of hardware vulnerabilities, categorizing common types, discussing their impacts and implications, and reviewing standard mitigation strategies. This chapter also spotlights the ongoing challenges in the field and potential research directions, laying a comprehensive foundation for the focused exploration of the Fusee Gelee exploit that follows.

The third chapter, dedicated to The Nintendo Switch and the Fusee Gelee Exploit, explores the security architecture of the Nintendo Switch, detailing the discovery and technical specifics of the Fusee Gelee vulnerability and its broader implications for hardware security.

Next chapter would be on methodology and practical analysis, the subsequent section combines an outline of the research approach and experimentation ethics with a thorough documentation of replicating the Fusee Gelee exploit. This includes a detailed account of the experimental setup, execution, and a critical analysis of the findings, integrating methodological rigor with practical insights.

Nintendo's Response and Industry Implications is examined next, where the focus shifts to the countermeasures adopted by Nintendo in response to the exploit, evaluating their effectiveness and discussing their broader ramifications for the gaming industry and the domain of hardware security at large.

The thesis progresses to Alternative Mitigation Strategies, offering a critical assessment of Nintendo's approach and proposing potential alternative strategies for addressing similar vulnerabilities, considering their feasibility, advantages, and limitations.

Concluding the thesis, the Conclusion chapter summarizes the key findings from the exploration of hardware security issues, the Fusee Gelee exploit analysis, and the evaluation of mitigation strategies. It articulates the thesis's contribution to the field of hardware security and suggests directions for future research.

Chapter 2

State of the Art in Hardware Security Flaws

> The only truly secure system is one that is powered off, cast in a block of concrete and sealed in a lead-lined room with armed guards.
>
> *Edward Amoroso*

2.1 Historical Overview

The journey of hardware security has evolved significantly over the years, from its initial focus on safeguarding military and space exploration equipment to protecting consumer electronics against sophisticated attacks. This evolution can be broadly categorized into several key phases:

1. **Early Developments**: Initially, hardware security was predominantly driven by the needs of government and military applications. The focus was on ensuring the reliability and security of semiconductors in environments subject to extreme conditions, such as outer space or high-altitude flights. Techniques like **radiation hardening**[15] were developed to protect these systems against environmental challenges, including radiation and temperature fluctuations. For example, the use of Silicon on Insulator (SOI) technology in semiconductor fabrication improved resistance to radiation effects.

2. **Commercialization and Consumer Devices**: With the advent of consumer electronics, hardware security expanded to include protection against piracy and

unauthorized access. Digital Rights Management (DRM) became crucial in devices like cable set-top boxes and gaming consoles. This era saw the emergence of **content protection schemes** and the corresponding development of countermeasures to bypass these protections.[4]

3. **Remote Hardware Vulnerabilities**: The discovery of vulnerabilities that could be exploited remotely marked a significant shift in cybersecurity concerns. Notably, the **Rowhammer attack** exemplifies this transition. Traditionally, hardware attacks were assumed to require physical access. However, Rowhammer can be initiated remotely by leveraging code that induces bit flips in a device's DRAM, affecting adjacent rows. Such an attack was demonstrated on various architectures, including Intel's Sandy Bridge, Ivy Bridge, Haswell, and AMD's Piledriver systems, by executing a specific pattern of assembly instructions:

```
code1a:
    mov (X), %eax
    mov (Y), %ebx
    clflush (X)
    clflush (Y)
    mfence
    jmp code1a
```

a. Induces errors

```
code1b:
    mov (X), %eax
    clflush (X)


    mfence
    jmp code1b
```

b. Does not induce errors

Figure 2.1: Assembly code snippets for Rowhammer attack[6]

This attack sequence strategically causes DRAM cells to leak charges into adjacent cells, overcoming the inherent electrical isolation between them, leading to bit flips. For instance, this code sequence resulted in numerous bit flips, which varied across different microarchitectures, as shown in Table 2.1 from a study by Kim et al.[6].

Bit-Flip	Sandy Bridge	Ivy Bridge	Haswell	Piledriver
'0' → '1'	7,992	10,273	11,404	47
'1' → '0'	8,125	10,449	11,467	12

Table 2.1: Bit-flips induced by disturbance on a 2GB module

The ability to induce such bit flips remotely through crafted payloads, like malicious JavaScript on a web page, has elevated Rowhammer from a theoretical concern to a practical cybersecurity threat. The implications of such a vulnerability are profound: systems could potentially be compromised without the attacker ever physically touching

the hardware. This shifts the landscape of system security, emphasizing the need for vigilant memory management and robust protective mechanisms in both hardware design and system software.

4. **Modern Challenges**: Today, hardware vulnerabilities like **Spectre and Meltdown**[16, 12] have shown that even fundamental hardware design principles can introduce security risks. These vulnerabilities exploit speculative execution—a performance feature in modern CPUs—to leak sensitive information. Speculative execution is used by CPUs to predict future execution paths and prematurely execute instructions. This can increase performance but also introduces the possibility of leaking information if the prediction is incorrect and the speculative execution has side effects that are not fully discarded.

   For instance, Spectre attacks trick the processor into executing instructions that should not have been executed, exploiting the latency in the branch prediction mechanism of the CPU. The processor's speculative execution feature is then leveraged to perform operations that leave observable side effects such as changes in cache state, even if the speculative results are discarded. These side effects can be monitored to infer sensitive data like cryptographic keys or personal information. [8]

```
1    if (index < simpleByteArray.length) {
2        index = simpleByteArray[index | 0];
3        index = (((index * TABLE1_STRIDE)|0) &
↪   (TABLE1_BYTES-1))|0;
4        localJunk ^= probeTable[index|0]|0;
5    }
```

Listing 2.1: Exploiting Speculative Execution via JavaScript.[8]

Alongside **Spectre**, the **Meltdown** vulnerability has revealed critical risks inherent in performance optimization techniques employed by modern CPUs. Specifically, Meltdown circumvents memory isolation guarantees by exploiting out-of-order execution, a feature used by CPUs to speed up processing. This exploitation allows an attacker to read all memory on a system, even without any permissions.
Meltdown is based on a fundamental hardware behavior involving out-of-order execution, where CPUs execute instructions out of their planned sequence for efficiency. When the CPU processes an instruction that should not be executed, it discards the result to maintain correct program operation. However, the discarded results can affect the cache, leading to a potential side-channel that can be exploited.

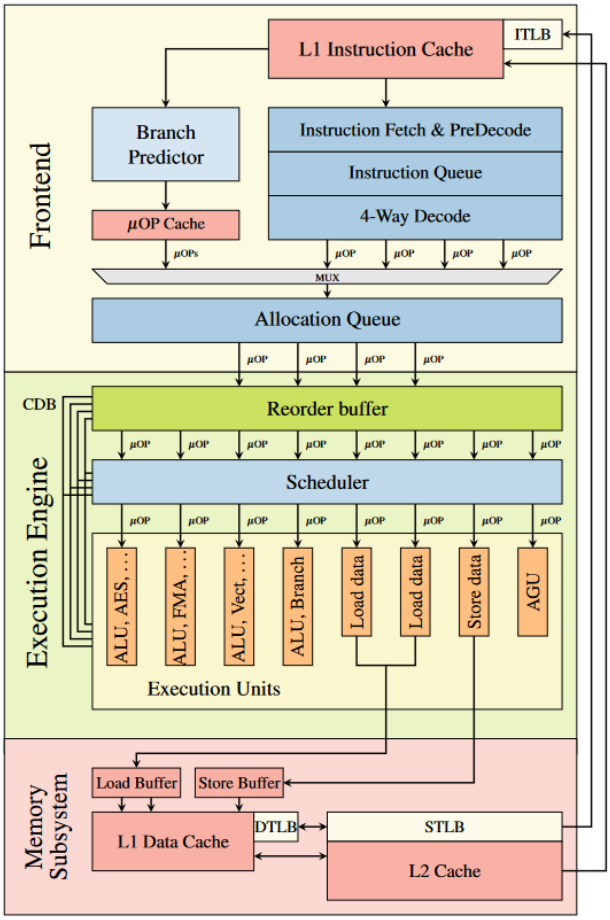


Figure 2.2: Simplified illustration of a single core of the Intel's Skylake microarchitecture.[10]

Essentially, Meltdown breaks the foundational security boundary that segregates the kernel's memory space from user processes. Attackers can take advantage of Meltdown to access not just the kernel memory but the entire physical memory of the host machine, potentially exposing sensitive data across user processes and virtual machines. The issue is pervasive across numerous Intel CPUs and potentially other processors.[10]

```
1       ; rcx = kernel address
2       ; rbx = probe array
3       retry:
4         mov al, byte [rcx]
5         shl rax, 0xc
6         jz retry
7         mov rbx, qword [rbx + rax]
```

Listing 2.2: Meltdown Attack Assembly Code.[10]

Unlike Spectre, Meltdown does not need to be tailored to a specific victim's environment, nor does it rely on any form of software vulnerability, making it widely exploitable on affected systems.

2.1.1 Milestones

- **1980s-1990s**: Radiation hardening techniques developed for space and military use.
- **Early 2000s**: Rise of consumer electronics security with DRM and content protection.
- **2014**: Discovery of the Rowhammer vulnerability, illustrating a shift towards remote exploitability of hardware.
- **2018**: Spectre and Meltdown vulnerabilities exposed, highlighting deep-seated issues in CPU design.

Diving deeper into the "Types of Hardware Flaws" section, we'll explore the various categories of hardware vulnerabilities, providing a more nuanced understanding of these threats through detailed examples, code snippets, and references to academic and industry sources.

2.2 Types of Hardware Flaws

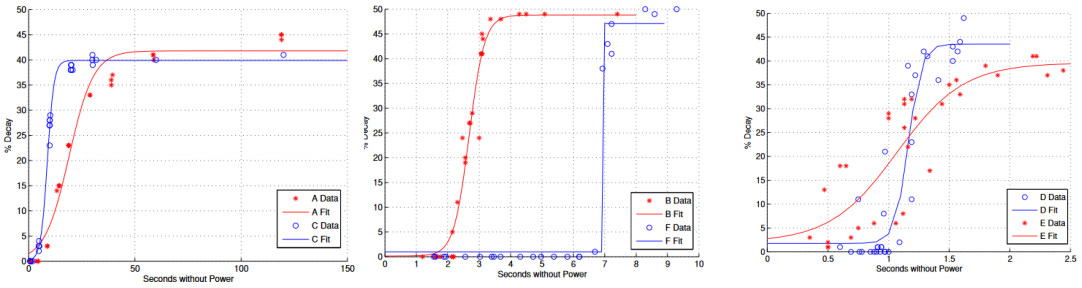
Hardware vulnerabilities can manifest in numerous forms, each exploiting different aspects of physical design, implementation, and operational behavior. These vulnerabilities are typically categorized into three primary types: physical vulnerabilities, side-channel attacks, and fault injection attacks.

2.2.1 Physical Vulnerabilities

Physical vulnerabilities are those that necessitate direct interaction with or access to the hardware device. They can exploit inherent design flaws or result from malicious physical modifications.

- **Cold Boot Attacks**: A striking example of a physical vulnerability is the cold boot attack[5], where sensitive data such as cryptographic keys are retrieved from RAM after a device is powered off. Data remanence (residual physical representation of data that has been nominally erased or removed) in DRAM can persist for seconds to minutes at room temperature, and cooling the chips can extend

this period significantly, allowing attackers to reboot the system with a custom loader and extract the remaining data.



(a) Machine A and C Data (b) Machine B and F Data (c) Machine D and E Data

Figure 2.3: Decay rates at different temperatures showing extended data retention as temperature decreases.[5]

Post-cooling, an attacker uses a custom memory imaging tool to read the remaining data by rebooting the system with a minimal kernel that dumps the memory contents. The primary goal of that attack being the extraction of cryptographic keys from the decayed memory images, which are then used to decrypt sensitive data.

```
# Pseudocode for a cold boot attack scenario
def cold_boot_attack(ram_content, search_patterns):
  # Simulate cooling of RAM to preserve data
  cooled_ram = cool_ram(ram_content)
  # Search for cryptographic keys or sensitive data in the cooled
  ↪ RAM content
  for pattern in search_patterns:
      if pattern in cooled_ram:
        print("Sensitive data found:", pattern)
```

Listing 2.3: Pseudocode for a cold boot attack scenario

- **Hardware Implantation**: Another form of physical vulnerability involves tampering with hardware components to introduce malicious functionality. For instance, adding a small, inconspicuous chip to a motherboard can create a backdoor for attackers to access or manipulate the device remotely.

2.2.2 Side-channel Attacks

Side-channel attacks exploit indirect effects of system operations, such as timing, power consumption, and electromagnetic emissions, to infer sensitive information without breaching the system's logical security boundaries.

Timing Attacks

The seminal work of Kocher (1996)[9] on timing attacks offers a comprehensive examination of the vulnerabilities inherent in cryptographic systems due to variations in execution time during cryptographic operations such as modular exponentiation used in algorithms like RSA and Diffie-Hellman.

These attacks measure the time required for cryptographic operations, using variations to deduce secret keys. For instance, by measuring the time it takes for a server to respond to varying encrypted messages, an attacker can infer details about the encryption keys.

Kocher models the timing attack as a signal detection problem, where the 'signal' is the timing variation caused by the specific exponent bit, and 'noise' consists of measurement inaccuracies and variations from unknown exponent bits. Extensive statistical analysis is utilized, focusing on the probability distribution function F, which encapsulates the expected timing variations due to specific bits. The attack effectiveness can be expressed by the equation:

$$P(\mathit{correct}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(T-\mu)^2}{2\sigma^2}\right)$$

where T is the observed timing, μ is the mean timing for the correct guess, and σ^2 is the variance.

In practical scenarios, Kocher suggests simplifying the attack by avoiding the computation of F. Instead, the focus shifts to analyzing the variance of timing measurements adjusted for each guessed exponent bit. If the guess is correct, the variance of these adjusted measurements will be lower than those adjusted for incorrect guesses. The variance comparison is calculated as follows:

$$\mathrm{Var}(\mathit{correct}) < \mathrm{Var}(\mathit{incorrect})$$

This method of variance analysis serves as a crucial mechanism for efficiently distinguishing between correct and incorrect guesses of the secret key bits.

The paper includes experimental results using the RSA encryption algorithm implemented with the RSAREF toolkit, confirming that correct guesses about exponent bits consistently result in lower timing variances, thus validating the theoretical model. Kocher also discusses the potential for adapting the timing attack methodology to other cryptographic operations, underscoring its flexibility and broad applicability.

```
def perform_timing_attack(modexp, n, base, public_exponent):
    timings = []
    guessed_exponent = 0

    for bit_position in range(number_of_bits(n)):
        best_time = float('inf')
        best_bit = None

        for bit in [0, 1]:
            test_exponent = set_bit(guessed_exponent, bit_position,
            ↪  bit)

            start_time = current_time()
            modexp(base, test_exponent, n)
            elapsed_time = current_time() - start_time

            if elapsed_time < best_time:
                best_time = elapsed_time
                best_bit = bit

        guessed_exponent = set_bit(guessed_exponent, bit_position,
        ↪  best_bit)
        timings.append((bit_position, best_time))

    return guessed_exponent, timings

def number_of_bits(n):
    return n.bit_length()

def set_bit(number, position, value):
    mask = 1 << position
    return (number & ~mask) | (value << position)

def current_time():
    import time
    return time.time()
```

Listing 2.4: Pseudocode for a timing attack

Power Analysis Attacks

By monitoring the power usage of a device, attackers can gain insights into the data being processed. Simple Power Analysis (SPA) and Differential Power Analysis (DPA) are two common methods, with DPA being particularly effective at extracting cryptographic keys from seemingly innocuous power usage patterns. We'll delve into how these attacks work by analyzing the work of Kocher et al. (1999)[7].

- **SPA**: In Single Power Analysis (SPA), we observe the power consumption of a device to infer the operations being executed. SPA can reveal significant information about the execution path of cryptographic algorithms. In SPA, the observable feature is the power consumption, which correlates with the physical operations of a device.

  This type of analysis can detect significant operations within cryptographic algorithms, such as DES, by observing the distinct power signatures corresponding to each phase of the operation, notably the permutations and conditional operations based on the secret key[7].

```
def square_and_multiply(base, exponent, modulus):
    # Convert the exponent to its binary representation
    binary_exponent = bin(exponent)[2:]
    result = 1
    for bit in binary_exponent:
        result = (result * result) % modulus
        if bit == '1':
            result = (result * base) % modulus

    return result
```

Listing 2.5: Pseudocode for a Single Power Analysis attack using 'square and multiply' algorithm

- **DPA**: Differential Power Analysis uses statistical techniques to extract secret keys by analyzing power consumption data from multiple operations. In DPA, we focus on the mean difference of grouped data based on hypothetical intermediate values. Given a set of power traces T_i and a hypothesis function $H(k, x)$ that predicts power consumption based on key guess k and input x, the differential trace D is calculated as:

$$D_k[j] = \frac{1}{|G_0|} \sum_{i \in G_0} T_i[j] - \frac{1}{|G_1|} \sum_{i \in G_1} T_i[j]$$

  where G_0 and G_1 are sets of indices classified by whether $H(k, x_i)$ predicts low

or high power consumption, respectively.

```
def dpa_attack(traces, key_guesses):
  high_group = []
  low_group = []
  for trace, key_guess in zip(traces, key_guesses):
      if predict_high(key_guess):
          high_group.append(trace)
      else:
          low_group.append(trace)
  mean_high = np.mean(high_group, axis=0)
  mean_low = np.mean(low_group, axis=0)
  return mean_high - mean_low
```

Listing 2.6: Pseudocode for a Differential Power Analysis attack

In this pseudocode, `traces` is a list of power consumption traces, and `key_guesses` is a list of key hypotheses. The function `predict_high` decides the grouping based on a prediction model using the key guess. The differential trace, computed as the difference between the means of these groups, helps identify the correct key guess by highlighting variations in the power consumption corresponding to different key bits.

Fault Injection Attacks

Fault injection attacks deliberately induce operational errors to bypass security mechanisms or corrupt the execution of processes, exploiting these faults for unauthorized access or data extraction.

- **Voltage Glitching**: A technique used to manipulate the physical operating conditions of electronic devices in order to induce faults. These faults can be exploited to bypass security measures or corrupt the device's usual execution flow. In the context of security research, voltage glitching is often applied to cryptographic devices to either bypass security checks or extract secret keys.
  The basic idea behind voltage glitching involves momentarily altering the device's power supply to disrupt its normal operation. This disruption can cause the device to skip instructions, execute incorrect instructions, or produce erroneous data.
  In practical scenarios, such as the one explored by Moradi et al.[14] in their research on FPGA bitstream encryption vulnerabilities, voltage glitching is used to manipulate the execution of cryptographic algorithms, allowing attackers to bypass security checks or interfere with the encryption process.

```
def voltage_glitching_attack(target_operation):
  successful = False
```

```
while not successful:
  apply_voltage_drop()
  result = target_operation()

  if check_for_errors(result):
    exploit_errors(result)

  successful = True
```

Voltage glitching poses significant security risks, particularly for devices that handle sensitive information like cryptographic keys.
As discussed by Bittner et al. (2021)[2], voltage glitching remains a potent attack vector against electronic devices, especially as hardware becomes increasingly miniaturized and integrated.

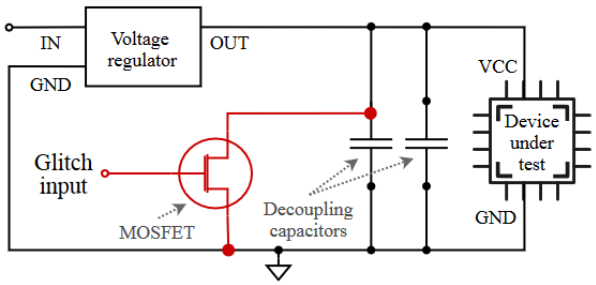


Figure 2.4: Voltage glitching attack on a microcontroller[2]

Voltage glitching has been adapted to target advanced microarchitectures and security-enforced environments. Modern devices often incorporate complex power management features that can be exploited to introduce glitches more subtly and effectively (fusee-gelee being a notable example of a voltage glitching attack on the Nintendo Switch which is the main focus of this paper).

- **Clock Glitching**: Similar to voltage glitching, clock glitching involves momentarily altering the system clock's frequency or timing, disrupting the sequence of operations and potentially allowing attackers to manipulate or bypass processes. It exploits vulnerabilities in the timing mechanisms of digital circuits, like those identified in [17]. This study demonstrates that high-resolution photonic emission analysis can physically characterize the intrinsic behavior of timing-based security mechanisms, such as those found in arbiter Physical Unclonable Functions (PUFs), by measuring the minute delays within the circuit with great precision. The relevance of this technique to clock glitching lies in its ability to manipulate and observe the effects of slight deviations in clock frequency or pulse timing,

which can induce errors or alter the behavior of the security mechanism. This method provides a potent example of how even robust security designs can be undermined by exploiting their fundamental physical properties—highlighting a critical area for enhancing resistance to clock glitching attacks

```
def clock_glitching_attack(target_device):
    original_clock = target_device.clock
    while not achieved_goal:
        glitched_clock = induce_clock_glitch(original_clock)
        target_device.clock = glitched_clock
        if target_device.malfunctions():
            exploit_malfunction(target_device)
        target_device.clock = original_clock
```

Listing 2.7: Pseudocode for a clock glitching attack

2.3 Mitigation Strategies

Mitigation strategies in hardware security encompass a wide array of techniques, from design-phase interventions to post-incident responses. These strategies are essential for reducing the risk and impact of hardware vulnerabilities.

2.3.1 Preemptive Measures

Secure Hardware Design: Embedding security features at the design level can significantly reduce vulnerabilities. This includes the adoption of design practices that inherently minimize security risks, such as:

- **Redundant Design**: Implementing redundant circuits to provide fallback options in case of failure or tampering.

```
    # Pseudocode for a simple redundant design in hardware
    ↪ security
    def execute_secure_operation(operation, redundancy_level=2):
        results = []
        for _ in range(redundancy_level):
            result = operation()
            results.append(result)
        if all_equal(results):
            return results[0]
        else:
```

```
10              raise SecurityException("Discrepancy detected in
                ↪  redundant operations")
```

- **PUFs**: Utilizing unique physical characteristics of the hardware as cryptographic keys or identifiers, enhancing security against cloning and tampering.
  In their exploration of RO-PUFs[11], Maiti detailed the deployment of ring oscillator PUFs on a large array of FPGA chips to analyze the viability of PUFs in achieving high levels of hardware security through uniqueness and reliability. RO-PUFs utilize the minute variations in the oscillation frequency of identically laid-out ring oscillators, which arise due to unavoidable process variations during chip manufacturing. These frequencies are sensitive to environmental conditions and operational variations, which in turn, affect the reliability of the PUF's response.
  The uniqueness of a PUF is measured by the inter-die Hamming distance, which should ideally be close to 50%. Maiti demonstrated that RO-PUFs achieve an average inter-die Hamming distance of 47.31%, which indicates a high level of uniqueness in responses across different chips.
  The reliability is estimated as the average intra-die Hamming distance i.e. $HD(R, R')$ over x samples of the same PUF, given by:

$$\frac{1}{x}\sum_{y=1}^{x}\frac{HD(R_i, R'_{i,y})}{n} \times 100\%$$

  where $R'_{i,y}$ is the y-th sample ofR'_i.

Hardware-assisted Security: Modern processors have integrated features to bolster security at the hardware level, notably Intel's Software Guard Extensions (SGX)[3] and AMD's Secure Encrypted Virtualization (SEV)[13]. These technologies enhance the protection of sensitive code and data within secure enclaves or encrypted virtual machines, shielding them from potentially compromised privileged software.

- **Intel SGX**: Intel's Software Guard Extensions (SGX) provide a way to increase the security of software systems on platforms where the privileged software, like the operating system or hypervisor, might be compromised. SGX achieves this by allowing sensitive computations to take place within a protected area of execution, called an enclave, which is designed to be tamper-resistant even from privileged code running on the same machine.
  based on the paper by Costan and Devadas[3], SGX designates a region of memory known as Processor Reserved Memory (PRM), where the sensitive code and data reside. This region is protected by the CPU such that no non-enclave memory access can occur, including those from the kernel, hypervisor, System Management

Mode (SMM), and Direct Memory Access (DMA) from peripherals.
Within PRM, there's a structure called the Enclave Page Cache (EPC), consisting of 4 KB pages that store the actual code and data of enclaves. System software, which is not fully trusted, is responsible for assigning these pages to enclaves, but the CPU ensures through the Enclave Page Cache Metadata (EPCM) that each page is only associated with a single enclave.
The SGX framework aims to enable secure remote computation by allowing users to run security-sensitive applications on a remote computer, potentially operated by an untrusted party, with integrity and confidentiality guarantees. Users can safely upload encrypted data and code for processing in a secure enclave, with the knowledge that the enclave is protected against tampering or unauthorized access, even from the host system's privileged software.

- **AMD SEV**: AMD's Secure Encrypted Virtualization (SEV) is a hardware feature designed to encrypt the memory of virtual machines (VMs), aiming to protect the memory contents against unauthorized access, even from privileged software like hypervisors.
  From the paper by Mofrad et al.[13], SEV is built upon AMD's Memory Encryption Technology, which employs a dedicated AES engine within the system on a chip (SoC) to encrypt and decrypt memory pages on the fly, without noticeable performance impact to the user.
  Each VM under SEV has its own unique encryption key, which is managed by the AMD Secure Processor, an ARM Cortex-A5 core within the SoC. This ensures that even if one VM's security is compromised, the others remain protected due to the use of separate keys.
  SEV aims to protect against both direct memory attacks (such as cold-boot attacks) and indirect attacks that exploit the hypervisor (such as side-channel attacks).
  Unlike Intel SGX, which provides memory integrity protection, AMD SEV does not protect the integrity of the encrypted VM memory. However, the use of AMD's secure processor to manage encryption keys keeps these keys out of the reach of the hypervisor or any other privileged code on the host.
  While SEV provides robust protection against many types of attacks, it does not completely eliminate the risk of security breaches. For example, side-channel attacks and other sophisticated exploits remain a concern.
  AMD has responded to security vulnerabilities with firmware updates, indicating that while SEV provides a significant security advantage, it requires ongoing maintenance and updates to ensure the highest level of protection.

Side-channel Resistance: Side-channel resistance is crucial in cryptographic implementations to prevent leakage of sensitive information through unintended channels. Techniques such as constant-time programming and differential power analysis (DPA) resistance are pivotal to enhancing the security of cryptographic systems.
The primary defense against timing attacks is ensuring that operations execute in constant time. Timing attacks exploit the variable execution times of operations depending on secret values. Just like the OpenSSL 'memcmp' function[1].

```
int constant_time_memcmp(const void* a, const void* b, size_t size)
↪ {
    const unsigned char* _a = (const unsigned char*)a;
    const unsigned char* _b = (const unsigned char*)b;
    unsigned char result = 0;
    for (size_t i = 0; i < size; i++) {
        result |= _a[i] ^ _b[i];
    }
    return result == 0;
}
```

Listing 2.8: Simplified constant-time comparison function in C for understanding.

DPA attacks involve analyzing power consumption patterns during cryptographic operations to infer secret keys. Countermeasures include balancing power consumption across different operations and using randomization techniques to mask the power signature. Cryptographic algorithms can be modified to exhibit uniform power consumption, or additional circuitry can be integrated to disguise the actual power use patterns.
Modern cryptographic modules integrate side-channel resistant features, including noise generators and dual-rail logic, to obscure the relationship between the cryptographic operations and the physical emissions like power or electromagnetic signals.
Dedicated hardware elements, such as Hardware Security Modules (HSMs) or Trusted Platform Modules (TPMs), often include designs that inherently resist side-channel attacks, thereby safeguarding the cryptographic processes they handle.

2.3.2 Reactive Strategies

Firmware and Software Updates: Patching vulnerabilities is a common approach to mitigate discovered flaws. This includes updates to:

- **Microcode**: Processor microcode updates can mitigate certain vulnerabilities at the cost of performance.
  Source: Intel's response to Spectre and Meltdown vulnerabilities with microcode updates.

- **Device Firmware**: Updating firmware can address vulnerabilities in peripherals and embedded devices.
  Source: "Best Practices for Firmware Updates", NIST Special Publication 800-193.

Hardware Recalls and Replacements: In cases where software cannot fully mitigate a vulnerability, hardware modifications or recalls may be necessary.

- **TPM Recalls**: The Infineon TPM vulnerability required a hardware fix for certain keys generated by the flawed library.
  Source: "Infineon RSA Key Generation Issue", Infineon Technologies Advisory, 2017.

Isolation and Virtualization: Employing hardware and software techniques to isolate potentially vulnerable components or sensitive operations from the rest of the system.

- **Virtualization-Based Security (VBS)**: Uses hardware virtualization features to isolate security-critical parts of the OS.
  Source: "How Virtualization-Based Security Powers Windows Defender", Microsoft Tech Community.

2.4 Current Challenges in Hardware Security

Complexity and Integration Modern hardware's complexity, characterized by billions of transistors and multifaceted systems on a chip (SoCs), introduces numerous security vulnerabilities. The integration of diverse functionalities, from wireless communication to cryptographic processing units, within single chips, like those seen in smartphones and IoT devices, amplifies the risk of cross-component vulnerabilities.

- **Source**: Kocher et al.'s seminal paper on Spectre and Meltdown vulnerabilities showcases how complex CPU architectures can lead to significant security challenges (Kocher et al., "Spectre Attacks: Exploiting Speculative Execution," 2019).

Scaling of Preemptive Measures As hardware technology evolves, maintaining and scaling preemptive security measures becomes increasingly challenging. The design and implementation of security features must not only address current threats but also anticipate future vulnerabilities, all while adhering to stringent performance and power consumption constraints.

- **Example**: The development of Trusted Execution Environments (TEEs) like ARM's TrustZone, which must evolve to address new attack vectors while maintaining compatibility and performance across generations of hardware.

Post-Quantum Cryptography The potential of quantum computing to break current cryptographic systems poses a significant threat to hardware security, particularly in secure communication and data protection. Developing quantum-resistant cryptographic algorithms and integrating them into hardware is an urgent priority.

- **Source**: The National Institute of Standards and Technology (NIST)'s ongoing efforts to standardize post-quantum cryptographic algorithms highlight the importance of this research area (NIST Post-Quantum Cryptography Standardization Project).

Reactive Strategy Limitations The inherent limitations of hardware in accommodating reactive security strategies necessitate innovative approaches to vulnerability management post-production. The rigidity of hardware makes addressing discovered vulnerabilities complex and costly.

- **Example**: The recall of FPGAs and SoCs due to the discovery of critical vulnerabilities, such as the vulnerability in Infineon's TPM chips, demonstrates the challenges and costs associated with reactive strategies in hardware security.

2.4.1 Future Research Directions

Advanced Materials and Fabrication Techniques Research into new materials like graphene or advanced silicon fabrication techniques could yield hardware with inherent security features, such as resistance to tampering or enhanced electromagnetic shielding.

- **Source**: A study on the use of graphene in creating tamper-resistant circuits presents a promising avenue for secure hardware design (Lee et al., "Graphene for Secure Hardware," IEEE Transactions on Emerging Topics in Computing, 2020).

AI and Machine Learning for Security Integrating AI and machine learning directly into hardware security mechanisms can offer dynamic, adaptive defenses against complex and evolving threats. This includes anomaly detection systems that can identify unusual behavior indicative of a security breach at the hardware level.

- **Example**: Google's use of machine learning in its Titan M security chip to detect abnormal patterns and prevent insider attacks illustrates the potential of AI in enhancing hardware security.

Homomorphic Encryption Hardware Developing specialized hardware that supports homomorphic encryption operations natively can revolutionize data privacy, enabling secure computation on encrypted data without decryption, a critical capability for cloud computing.

- **Source**: Research by Gentry and others on fully homomorphic encryption lays the groundwork for this transformative approach to data security (Gentry, "A Fully Homomorphic Encryption Scheme," Stanford University, 2009).

Secure Hardware Lifecycle Management Ensuring hardware security from design to decommissioning involves secure supply chain practices, reliable firmware updates, and safe disposal methods to prevent data leakage from discarded devices.

- **Example**: The Secure Device Lifecycle Management (SDLM) approach, which encompasses secure provisioning, updates, and end-of-life processes, is vital for maintaining security throughout a device's lifespan.

By elaborating on these challenges and research directions with detailed examples and authoritative sources, this section offers a comprehensive overview of the state of hardware security. It highlights the ongoing need for innovation and adaptation in the face of an ever-evolving threat landscape and technological advancements.

Bibliography

[1] */Docs/Man1.1.1/Man3/CRYPTO_memcmp.Html*. URL: https://www.openssl.org/docs/man1.1.1/man3/CRYPTO_memcmp.html (visited on 04/10/2024).

[2] Otto Bittner et al. *The Forgotten Threat of Voltage Glitching: A Case Study on Nvidia Tegra X2 SoCs*. Aug. 16, 2021. arXiv: 2108.06131 [cs]. URL: http://arxiv.org/abs/2108.06131 (visited on 04/15/2024). preprint.

[3] Victor Costan and Srinivas Devadas. *Intel SGX Explained*. 2016. URL: https://eprint.iacr.org/2016/086 (visited on 04/10/2024). preprint.

[4] *Digital Rights Management*. In: *Wikipedia*. Mar. 25, 2024. URL: https://en.wikipedia.org/w/index.php?title=Digital_rights_management&oldid=1215486134 (visited on 04/15/2024).

[5] J. Alex Halderman et al. "Lest We Remember: Cold-Boot Attacks on Encryption Keys". In: *Communications of the ACM* 52.5 (May 2009), pp. 91–98. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/1506409.1506429. URL: https://dl.acm.org/doi/10.1145/1506409.1506429 (visited on 04/10/2024).

[6] Yoongu Kim et al. "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)*. 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA). Minneapolis, MN, USA: IEEE, June 2014, pp. 361–372. ISBN: 978-1-4799-4394-4 978-1-4799-4396-8. DOI: 10.1109/ISCA.2014.6853210. URL: http://ieeexplore.ieee.org/document/6853210/ (visited on 04/10/2024).

[7] Paul Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis". In: *Advances in Cryptology — CRYPTO' 99*. Ed. by Michael Wiener. Berlin, Heidelberg: Springer, 1999, pp. 388–397. ISBN: 978-3-540-48405-9. DOI: 10.1007/3-540-48405-1_25.

[8] Paul Kocher et al. *Spectre Attacks: Exploiting Speculative Execution*. arXiv.org. Jan. 3, 2018. URL: https://arxiv.org/abs/1801.01203v1 (visited on 03/07/2024).

[9] Paul C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: *Advances in Cryptology — CRYPTO '96*. Ed. by

Neal Koblitz. Berlin, Heidelberg: Springer, 1996, pp. 104–113. ISBN: 978-3-540-68697-2. DOI: 10.1007/3-540-68697-5_9.

[10] Moritz Lipp et al. *Meltdown*. arXiv.org. Jan. 3, 2018. URL: https://arxiv.org/abs/1801.01207v1 (visited on 03/07/2024).

[11] Abhranil Maiti et al. "A Large Scale Characterization of RO-PUF". In: *2010 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. 2010 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST). June 2010, pp. 94–99. DOI: 10.1109/HST.2010.5513108. URL: https://ieeexplore.ieee.org/document/5513108 (visited on 04/10/2024).

[12] *Meltdown (Security Vulnerability)*. In: *Wikipedia*. Mar. 27, 2024. URL: https://en.wikipedia.org/w/index.php?title=Meltdown_(security_vulnerability)&oldid=1215853086 (visited on 04/15/2024).

[13] Saeid Mofrad et al. "A Comparison Study of Intel SGX and AMD Memory Encryption Technology". In: *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*. HASP '18: Hardware and Architectural Support for Security and Privacy. Los Angeles California: ACM, June 2, 2018, pp. 1–8. ISBN: 978-1-4503-6500-0. DOI: 10.1145/3214292.3214301. URL: https://dl.acm.org/doi/10.1145/3214292.3214301 (visited on 04/15/2024).

[14] Amir Moradi et al. "On the Vulnerability of FPGA Bitstream Encryption against Power Analysis Attacks: Extracting Keys from Xilinx Virtex-II FPGAs". In: *Proceedings of the 18th ACM Conference on Computer and Communications Security*. CCS '11. New York, NY, USA: Association for Computing Machinery, Oct. 17, 2011, pp. 111–124. ISBN: 978-1-4503-0948-6. DOI: 10.1145/2046707.2046722. URL: https://dl.acm.org/doi/10.1145/2046707.2046722 (visited on 04/10/2024).

[15] *Radiation Hardening*. In: *Wikipedia*. Feb. 9, 2024. URL: https://en.wikipedia.org/w/index.php?title=Radiation_hardening&oldid=1205501084 (visited on 04/15/2024).

[16] *Spectre (Security Vulnerability)*. In: *Wikipedia*. Apr. 15, 2024. URL: https://en.wikipedia.org/w/index.php?title=Spectre_(security_vulnerability)&oldid=1218972533 (visited on 04/15/2024).

[17] Shahin Tajik et al. "Physical Characterization of Arbiter PUFs". In: *Cryptographic Hardware and Embedded Systems – CHES 2014*. Ed. by Lejla Batina and Matthew Robshaw. Berlin, Heidelberg: Springer, 2014, pp. 493–509. ISBN: 978-3-662-44709-3. DOI: 10.1007/978-3-662-44709-3_27.