

Exploring Power Fuzzing in Embedded Systems: Architecture, Challenges, and Enhancements

by

Kavish Mehta

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2024

© Kavish Mehta 2024

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

[Embedded Systems \(ES\)](#) are becoming increasingly prevalent across various industries, playing an important role in everything from critical infrastructure to consumer electronics. However, their resource-constrained nature and complex interactions with the physical world make them susceptible to security vulnerabilities. Fuzzing, a technique that feeds random or mutated data to a program to uncover software bugs and vulnerabilities, has emerged as a powerful tool for improving embedded system security.

This thesis explores the concept of power fuzzing, a specialized fuzzing approach that focuses on capturing variations in the power consumption of the [Target System \(TS\)](#) as feedback. We examine the power fuzzing structure, highlighting the different events triggered during fuzzing and the inherent variability associated with these events. The thesis also addresses challenges in data capture and the limitations of the [Target System \(TS\)](#).

Furthermore, this thesis proposes two enhancements to improve the effectiveness of power fuzzing architectures: (1) Hardware Trigger and (2) [Profile and Fine-Tune \(PnFT\)](#) Approach. These enhancements aim to address the aforementioned challenges and contribute to a more robust security testing methodology for [Embedded Systems \(ES\)](#).

Acknowledgments

I extend my deepest gratitude to my supervisor, Professor Sebastian Fischmeister, for his unwavering support and invaluable guidance throughout my graduate studies. His motivation and encouragement were instrumental in shaping my academic journey at the University of Waterloo. Special thanks to Murray Dunne and Arthur Grisel-Davy, whose insightful feedback was crucial in overcoming challenges and enhancing the quality of my research.

I am profoundly grateful to my family for their continuous support and strength during challenging times. My heartfelt thanks to Rakshit Sharma, Abhijith Sharma, and Srijan Pabbi for their engaging brainstorming sessions, and to everyone in the ESG lab for their constant motivation and support.

Dedication

I would like to dedicate this thesis to my father, Mr. Rupesh Mehta; my mother, Mrs. Kinnari Mehta; and my sister, Ms. Sakshi Mehta

Table of Contents

Author’s Declaration	ii
Abstract	iii
Acknowledgments	iv
Dedication	v
List of Figures	ix
List of Tables	xi
List of Abbreviations	xii
1 Introduction	1
1.1 Motivation	2
1.2 Thesis Outline	2
2 Background	3
2.1 Embedded Systems (ES)	3
2.1.1 Classification of Embedded Systems (ES)	4
2.1.2 Security of Embedded Systems (ES)	5
2.2 Fuzzing	6

2.2.1	Types of Fuzzing	7
2.2.2	Embedded Systems (ES) Fuzzing	8
2.3	Side-Channel Analysis (SCA)	9
3	Abstract Overview of the Power Fuzzing Architecture	11
3.1	Functional View	11
3.2	Timeline Diagram	13
3.2.1	Sequence of Events	15
3.2.2	Analyzing Time Variability (ϵ_i):	19
4	Challenges	29
4.1	Experimental Setup	30
4.1.1	Hardware	30
4.1.2	Software	32
4.2	Data Capture	33
4.2.1	Capturing Data at the Right Instant	34
4.2.2	Capturing Data for the Right Duration	38
4.2.3	Capturing the Right Amount of Data	41
4.2.4	Capturing the Right Resolution of Data	44
4.3	Variable Target Properties	47
4.3.1	Background Processes	47
4.3.2	Input Message Properties	48
4.3.3	Performance Variability Over Extended Runtime	51
4.4	Lack of Standard Analysis Methods	51
5	Proposed Enhancements	53
5.1	Hardware Trigger	53
5.2	Profile and Fine-Tune (PnFT) Approach	56

6 Conclusion	59
References	61

List of Figures

3.1	Functional View of Power Fuzzing Architecture	12
3.2	Timeline diagram of Power Fuzzing Architecture	14
3.3	Universal Asynchronous Receiver-Transmitter (UART) frame stack [5] . . .	17
3.4	Variability ϵ_1 Cause and Effect Diagram	21
3.5	Variability ϵ_2 Cause and Effect Diagram	23
3.6	Variability ϵ_3 Cause and Effect Diagram	25
3.7	Variability ϵ_4 Cause and Effect Diagram	27
3.8	Variability ϵ_5 Cause and Effect Diagram	28
4.1	Power Consumption of RaspBerryPi (RPi) for ‘ls’ [55] Command	35
4.2	Power Consumption of RPi for ‘ls’ [55] Command over Multiple Runs . . .	36
4.3	Histogram Representation of Sample Points at Which Event-3 (E_3) Begins When ‘ls’ [55] Command Is Sent to the RPi Development Board 500 Times. Power traces were collected for 100,00 Sample Points at 1 MHz	37
4.4	Execution Time of ‘ls’ [55] Command on RPi Development Board for 100 Runs at Low and High CPU Loads	39
4.5	Power Consumption of RPi when subjected to ‘ls’ [55], ‘ps’ [56] and ‘uname -a’ [59] Commands	40
4.6	Trend in Total Power P_t When Averaging Multiple Traces for 4 Captures Conducted at Different Times	43
4.7	Representation of Original Power Trace If It Was Captured at Different Sampling Frequencies. The Original Power Trace Is Power Consumption of RPi When Subjected to the ‘ls’ [55] Command	46

4.8	Power consumption Pattern for RPi when Subjected to (1) “echo ‘abc’ ” [53] Command and (2) ‘pwd’ [57] Command	49
5.1	Improved Functional View of Power Fuzzing Architecture	55

List of Tables

4.1	Specification of Lenovo ThinkCentre M90t Workstation	30
4.2	Specification of RPi 3B Development Board	31
4.3	Specification of Teledyne ADQ32-PCIe Digitizer	32
4.4	Key Roles played by Background Processes in an ES	47

List of Abbreviations

ADC Analog-to-Digital Converter [32](#)

AI Artificial Intelligence [1](#)

ARIMA AutoRegressive Integrated Moving Average [52](#)

ARM Advanced-RISC Machine [23](#), [24](#), [26](#), [32](#)

CAN Controller-Area Network [10](#), [15](#), [21](#), [54](#)

CBIG Command-Based Input Grammar [15](#), [20](#)

CLT Central Limit theorem [37](#)

CPU Central-Processing Unit [4](#), [8](#), [17](#), [18](#), [20](#), [23–26](#), [28](#), [30](#), [31](#), [34](#), [35](#), [38](#), [42](#), [44](#), [47](#), [50](#), [51](#), [57](#)

DMA Direct-memory Access [18](#), [22](#), [24](#)

ECUs Electronic-Control UnitS [10](#), [15](#)

EMI Electro-Magnetic Interferences [24](#)

EoM End of Message [54](#)

ES Embedded Systems [iii](#), [xi](#), [1–10](#), [13](#), [15](#), [17](#), [18](#), [22](#), [25](#), [29](#), [31](#), [41](#), [47](#), [51](#), [53–57](#), [59](#), [60](#)

GPIO General-Purpose Input Output [31](#)

GPOS General Purpose Operating Systems [4](#), [15](#)

GSM Global System for Mobile Communications 15

I/O Input-Output 18, 20, 26–28, 38

I2C Inter-Integrated Circuit 31, 54

IoT Internet Of Things 1, 3–5, 15, 31, 32

IPC Inter-Process Communication 26, 44

ISRs Interrupt-Service Routines 26

MBIG Message-Based Input Grammar 15, 20

OS Operating System 4, 12, 15–17, 20–22, 26, 31, 32, 35, 37–39, 42, 44, 50

PCH Power Capture Hardware 13, 30, 31, 54

PCIe Peripheral Component Interconnect Express 32

PCU Power Collection Unit 13, 31

PnFT Profile and Fine-Tune iii, 56, 60

PTU Power Translation Unit 13, 31, 32

RAM Random-Access Memory 4, 8, 17, 18, 20, 24, 28, 30, 31, 57

RMS Root Mean Square 41

RPi RaspBerryPi ix–xi, 30–32, 34–40, 42, 45, 46, 49, 50

SBIG Script-Based Input Grammar 15, 20

SCA Side-Channel Analysis 2, 9, 10, 33

SNR Signal-to-Noise Ratio 41

SoC System-On-Chip 34

SoM Start of Message 16, 17, 34

SPI Serial Peripheral Interface 31, 54

TS Target System [iii](#), [1](#), [2](#), [6–9](#), [11–13](#), [15–18](#), [20–25](#), [27–31](#), [33–35](#), [38](#), [41](#), [42](#), [44](#), [45](#), [47–54](#), [56–60](#)

TTL Transistor-Transistor Logic [31](#)

UART Universal Asynchronous Receiver-Transmitter [ix](#), [16](#), [17](#), [21](#), [22](#), [31](#), [34](#), [50](#), [54](#)

USB Universal Serial Bus [30–32](#), [54](#)

ZT Zero Trust [5](#)

Chapter 1

Introduction

ES have become integral to modern technology, designed to perform specific functions with high reliability. These systems are widely used in consumer electronics, process control systems, aircraft, automotive systems, and numerous other applications where their dependability is high. As important components in the expanding Internet Of Things (IoT) landscape, ES facilitate seamless connectivity and data exchange, enhancing efficiency and innovation across industries [24]. Moreover, they play an important role in supporting the development and deployment of Artificial Intelligence (AI), which is rapidly advancing as a transformative technology [73]. The integration of ES in AI applications enables real-time data processing and decision-making, driving advancements in automation, machine learning, and intelligent systems. As the reliance on ES grows, ensuring their security becomes increasingly critical. Undocumented behaviour of these systems can lead to significant risks, highlighting the need for robust testing methodologies.

Fuzzing, a dynamic testing technique, has gained importance in identifying security flaws within the software by providing random or malformed inputs to TS. Fuzzing is beneficial for testing ES due to their resource-constrained nature. These systems possess just enough computing power to perform their dedicated tasks and often cannot support additional security features. Fuzzing targets specific components, optimizing resource usage and prioritizing critical areas. It also enables the gradual identification of vulnerabilities, providing iterative improvements without high resource allocation.

1.1 Motivation

Implementing fuzzing in [ES](#) presents challenges. Its effectiveness relies on analyzing feedback such as crash reports, logs, and unexpected behaviour — capabilities typically available in general computing platforms but limited in [ES](#) due to resource constraints. To overcome this issue, we need to use [Side-Channel Analysis \(SCA\)](#) to monitor some other aspects of the [ES](#) as feedback without affecting the functioning of the system itself. Power fuzzing is an effective fuzzing technique because it leverages power consumption data to identify vulnerabilities in [ES](#). By analyzing variations in power usage during different operational states, it uncovers hidden behaviours and potential security flaws. This approach allows for non-intrusive testing, providing insights into the system’s response to diverse inputs without altering its functionality. This thesis explores the power fuzzing architecture in detail and its associated challenges and suggests some enhancements to implement this technique better to fuzz [ES](#).

1.2 Thesis Outline

In this thesis, we begin by introducing the underlying problem, followed by background information to aid the reader’s understanding. The background chapter clarifies the concept of [ES](#), discussing their classification and security aspects. It then explains fuzzing, its various types, and its application to [ES](#), concluding with an overview of [SCA](#).

Chapter [3](#) delves into the power fuzzing architecture, detailing the different components and their roles and providing a functional overview. We explore the key events within the power fuzzing process, explaining the activities between them to elaborate the overall workflow. This chapter concludes by examining the timing variability among these events and the reasons behind each variation.

Chapter [4](#) offers a detailed examination of the challenges in the power fuzzing architecture. We describe the experimental setup used to obtain results, discuss the significance of ideal power trace capture in the fuzzing process, and outline the difficulties in obtaining high-quality power traces. Additionally, we address the challenges posed by fluctuations in target properties beyond our control and the absence of standardized analysis methods.

Chapter [5](#) proposes two enhancements to the power fuzzing architecture and process. The first enhancement involves using nonintrusive methods to collect power traces with precise timing, mitigating some timing variability. The second enhancement suggests employing an informed fuzzing approach to fine-tune the process for each [TS](#), optimizing its effectiveness.

Chapter 2

Background

Understanding the terminology and background information is important for a complete grasp of the power fuzzing architecture. This section provides essential contextual information to ensure clarity and understanding of the concepts discussed.

2.1 Embedded Systems (ES)

ES are specialized hardware and software solutions that perform specific functions, often interfacing directly with the physical world through their peripherals [72]. They exist independently or typically form part of larger structures like IoT, finding application across various industries and products. There is a fundamental difference between ES and other computing systems: ES integrate computation with physical constraints. This integration means that the powerful separation of computation (software) from physical elements (platform and environment), a central principle that has enabled advancements in computing science, does not apply to embedded systems [35]. It is important to note that both the hardware and software of ES are designed precisely to their requirements, avoiding unnecessary overhead in size, power consumption, and cost to execute their tasks efficiently [105]. Beyond computing power, time responsiveness is critical for ES, as many require deterministic handling of real-time events [67].

2.1.1 Classification of Embedded Systems (ES)

The diversity and specificity of ES applications contribute to the complexity of classification. One might begin by categorizing ES based on the industries they are used in, such as aerospace, transportation, and manufacturing. However, this initial categorization requires further improvement. Within each industry, the complexity of tasks varies, leading to different requirements on the ES. Simpler applications utilize devices with limited memory and processing capabilities. In contrast, more demanding tasks require ES that are less constrained, often capable of replacing basic computer functions. Consequently, we must further categorize these systems based on the computing power of the onboard microcontroller [86]. One of the better ways to classify ES is based on whether they use Operating System (OS) or not [24]; this categorization is a broader classification which encompasses all the minor categorizations.

- **ES Running OS:** ES that use OS are required to manage complex and computationally demanding tasks. Tasks that require operator involvement, network management, and performing common computing operations run on a General Purpose Operating Systems (GPOS) [84]. These OS need to be customized to the corresponding hardware and require multi-core Central-Processing Unit (CPU) and GigaBytes of Random-Access Memory (RAM) to run efficiently. BusyBox [83] is a great example of GPOS and is widely used to develop consumer-grade IoT products. On the other hand, there are a lot of applications that are time-critical and require strict reliability and determinism, which is not provided by GPOS [81]. Such applications use ES with Embedded OS, which are tailored to serve the specific needs of the application. These OS are light-weight compared to GPOS and require just MegaBytes of RAM and single-core CPU to run smoothly. FreeRTOS [1] is an example of Embedded OS.
- **ES not Running OS:** They are typically employed to perform less critical and simpler tasks. These systems are designed with very limited computing and storage capabilities, making them ideal for low-power applications. They use interrupt-driven or polling methods for control, which allows them to respond to events in real-time with minimal resource usage [67]. Their design focuses on efficiency and cost-effectiveness, ensuring that they can operate effectively within the constraints of their intended use. GPS dongle is an example of such as ES [24].

2.1.2 Security of Embedded Systems (ES)

With an exponentially growing number of devices entering the [IoT](#) space, rapid advancements in technology and ease of accessibility, information security has become a huge concern for general-purpose systems, encouraging significant investment in technologies to ensure the confidentiality, integrity, and availability of these systems across various layers [98]. Researchers are advocating the concept that every system should independently safeguard itself without relying on external security mechanisms. N. F. Syed et al. [93] in their comprehensive survey on the [Zero Trust \(ZT\)](#) security paradigm, describe the fundamental principles of [ZT](#) and explore various options for its successful implementation, including advanced techniques for authentication, access control, encryption and micro-segmentation, while advocating for its growing need. General-purpose systems, with their ample resources, can integrate these advanced security technologies without compromising operational efficiency.

However, the scenario is different for [ES](#). These systems have been designed with a focus on functional requirements, often neglecting security considerations. Design priorities have typically centred on cost, performance, and power efficiency, with minimal regard for security. The limited hardware resources of embedded systems often prevent the integration of security technologies. Consequently, the [IoT](#) space is affected by numerous security vulnerabilities [69]. In this connected world, enhancing device security requires strengthening the weakest link: [ES](#). A comprehensive study by Srivaths Ravi et al. [79] discusses the significant security challenges faced by modern [ES](#), emphasizing that security is not merely an add-on but a critical design consideration alongside cost, performance, and power. They highlight the “security processing gap,” “battery gap” and “assurance gap” as key issues and explore recent advancements and future research opportunities in secure embedded system design. Designers must adopt new architectural paradigms that prioritize security from the outset. This includes incorporating hardware security modules, cryptographic algorithms, security protocols, and the latest secure versions of software libraries and kernels.

The development of secure products provides a competitive edge in commercial terms. However, the testing process, which aims to uncover backdoors or undocumented behaviour, is one of the most resource-intensive stages of the development lifecycle. As a result, traditional testing methodologies are increasingly being improved with newer approaches capable of identifying more vulnerabilities in less time. Fuzzing has emerged as one of the most effective techniques for this purpose [67]. According to the “IEC 62443-4 Practice 5, SVV-3 Vulnerability testing” standard, fuzzing is mandated for external interfaces [24]. The standard stipulates that fuzzing should be an obligatory step in every

new development. The ISO/SAE 21434 standard [60] establishes a framework of technical requirements for cybersecurity and risk management to secure motor vehicles throughout their life cycle. It includes fuzz testing as a test method suitable for automated execution. In the context of constrained environments like ES, external testing remains the only feasible method for identifying vulnerabilities. ES typically lack the capacity to run additional security software. Therefore, fuzzing presents an attractive alternative, offering the ability to test an embedded TS without requiring extra software on the device or detailed knowledge of its internal operations.

2.2 Fuzzing

Fuzzing, also known as fuzz testing, is an automated testing technique that provides invalid, unexpected, or random data as inputs to a target. The primary goal of fuzzing is to identify vulnerabilities, crashes, and bugs within TS applications by stressing them with malformed inputs [104]. This technique is particularly effective in uncovering security vulnerabilities that can be exploited by attackers, making it a crucial component of modern system security practices. Fuzzing systematically injects various input variations into a TS or application. These inputs can include random data, boundary values, or specifically crafted payloads designed to test the robustness of the TS. The behaviour of the TS is then monitored for any anomalies such as crashes, memory leaks, or unexpected outputs.

Fuzzing involves several key phases that test the TS for undocumented behaviour. The first phase is **Input Generation**, where a large set of random, malformed, or semi-random inputs are created to simulate potential attack vectors. This phase often involves techniques such as mutation or generation-based methods to ensure diverse input coverage [24]. The second phase is **Execution**, where these inputs are fed into the TS, and its behaviour is monitored for any anomalies or crashes. This phase can be automated to test numerous inputs in a short period rapidly. The third phase is **Monitoring and Data Collection**, where detailed information about the TS's response to each input is collected. This includes power traces, log files, and any other relevant data that can indicate abnormal behaviour. The fourth phase is **Analysis**, where the collected data is analyzed to identify patterns, anomalies, or crashes that could indicate vulnerabilities. Advanced analysis techniques, including machine learning and signal processing, can be used to sift through large datasets efficiently. Finally, the **Reporting** phase involves documenting the identified vulnerabilities, their potential impacts, and recommendations for remediation. This phase is crucial for communicating findings to developers and stakeholders, enabling them to address the security flaws effectively.

2.2.1 Types of Fuzzing

Just like for [ES](#), we can categorize fuzzing techniques based on various parameters. These include input generation techniques, analysis techniques, and execution techniques. However, in this subsection, we will use a widely accepted classification method that categorizes fuzzing techniques based on prior knowledge of the target and input. This approach allows us to differentiate between different fuzzing methodologies and understand their unique advantages and challenges in the context of the information available about the [TS](#) and the nature of the input data.

- **Black-Box Fuzzing:** It treats the [TS](#) as an opaque entity without knowledge of its internal workings. The fuzzer generates inputs blindly using random mutation limited by some rules without considering the structure or behaviour of the [TS](#) [28]. This approach simulates an external attack scenario where the attacker has no insight into the system’s implementation. While black-box fuzzing is relatively easy to set up and does not require access to source code or detailed system documentation, it often lacks efficiency and thoroughness. The fuzzer might miss subtle vulnerabilities or fail to reach deeper states of the system due to its random and uninformed nature. However, it is a practical choice when source code is unavailable or when quick, broad testing is needed [61].
- **White-Box Fuzzing:** It leverages complete knowledge of the [TS](#)’s internals, including source code, design documents, and architecture [18]. This detailed understanding allows the fuzzer to generate targeted and intelligent inputs. Techniques such as symbolic execution and static analysis are often employed to guide the input generation process, ensuring complete coverage and the identification of subtle vulnerabilities [32]. White-box fuzzing is highly effective but requires significant resources, including access to source code and computational power. Its setup is complex, involving deep integration with the development environment, but the results are usually more thorough and precise compared to black-box fuzzing.
- **Grey-Box Fuzzing:** It strikes a balance between black-box and white-box approaches by utilizing partial knowledge of the [TS](#). It involves monitoring run-time behaviour, such as code coverage, to inform and optimize input generation [50, 61]. Grey-box fuzzers often use light-weight instrumentation to gather information about which parts of the code are executed with each input. This feedback mechanism helps the fuzzer explore more code paths and discover vulnerabilities more efficiently than black-box fuzzing. Grey-box fuzzing provides a good compromise, offering improved

effectiveness over black-box fuzzing while avoiding the high complexity and resource demands of white-box fuzzing. It is particularly useful in situations where partial information about the system is available, such as binary-level instrumentation or when developers can provide high-level insights without full source code access.

2.2.2 Embedded Systems (ES) Fuzzing

Fuzzing is a well-established technique in software testing. In general software applications, fuzzing works by automatically generating many inputs to test the software’s robustness and security. This approach is highly effective in identifying buffer overflows, input validation errors, and other security flaws in desktop applications, and other software systems. The primary reason for fuzzing’s success in software testing is the relatively straightforward environment in which software operates, where inputs and outputs can be easily controlled and monitored [11]. However, when it comes to ES, traditional fuzzing methods face some limitations and challenges, making it difficult to apply these techniques directly. Eisele, Max, et al. [25] present a systematic review of fuzzing approaches for ES, highlighting the complexity posed by varied architectures and hardware peripherals. They provide a formal definition of “embedded fuzzing” and categorize existing methods, culminating in a comprehensive table of 42 entries that underscores significant opportunities for future research due to current limitations.

ES fuzzing aims to test the software stored in the ES, commonly known as firmware, which controls the hardware. However, due to target connectivity constraints, only applications might be fuzzed instead of the entire firmware. Finding debugging ports for all TS poses a significant challenge. Researchers might have to locate the corresponding firmware online and employ emulation-based fuzzing. This approach, however, might not reveal all vulnerabilities present in the TS. Another challenge in ES fuzzing is the limited observability. Monitoring fuzzing results is crucial, yet crash detection in ES is notably more challenging compared to traditional software fuzzing. Traditional software systems have various detection mechanisms, such as error messages, security warnings, and system logs, which enable easy crash detection. In contrast, ES firmware often lacks fault generation mechanisms, making it difficult for a fuzzer to detect a fault or crash [103]. Furthermore, many embedded devices do not have output devices, preventing users from noticing crash detections. The limited computational resources in ES, including CPU power, RAM, and storage, further complicate the fuzzing process. These constraints limit the capabilities for debugging and logging, making it difficult to monitor the internal state and behaviour of the ES during fuzzing.

Addressing these challenges, [SCA](#) presents a promising solution to the limited observability problem in [ES](#) fuzzing. By analyzing indirect information such as power consumption or execution time, [SCA](#) can provide insights into the internal state of the [ES](#) without needing explicit output mechanisms. This method can significantly enhance the ability to detect crashes and other anomalies, thereby improving the effectiveness of fuzzing in [ES](#).

2.3 Side-Channel Analysis (SCA)

[SCA](#) is a technique used to gather information from a computing device by observing its physical or operational characteristics rather than relying on its logical output. These characteristics, often called side channels, can include power consumption, electromagnetic emissions, timing information, and acoustic signals. Kocher, Paul C. [47] introduced the first [SCA](#) with execution time as the side-channel. [SCA](#) takes advantage of the fact that the physical implementation of an algorithm may leak information about the internal state of the device, which can be analyzed to infer sensitive data, uncover backdoors, or gain insights into the device’s behaviour [76].

In the context of [ES](#) fuzzing, [SCA](#) can provide a powerful means of overcoming the limitations posed by traditional fuzzing methods. Since [ES](#) often lack robust mechanisms for outputting fault information and have limited computational resources, conventional crash detection methods may fall short. [SCA](#) helps address these challenges by offering an indirect way to monitor the system’s internal state and detect anomalies [41]. By analyzing side channels such as power consumption or electromagnetic emissions, we can detect subtle changes in the system’s behaviour that may indicate the presence of faults, crashes, or other anomalies. This enhances the fuzzing process’s observability, even in resource-constrained environments. In their study, G. T. Becker et al. [8] present three verification methods to detect software plagiarism in [ES](#) without access to the source code, utilizing side-channel information obtained during execution. Khan, Haider Adnan, et al. [44] propose a novel malware detection system for critical [ES](#) that utilizes electromagnetic side-channel signals to detect malicious activity, achieving 100% accuracy for DDoS and ransomware detection and near-perfect accuracy for stealthier code modification from up to 3 meters away. [SCA](#) does not require modifying the [TS](#) or relying on its built-in debugging capabilities. This non-intrusive nature makes it suitable for a wide range of [ES](#), including those with limited computational power or without output interfaces. Additionally, [SCA](#) can detect a broader range of faults by capturing the physical indications of these faults through side channels, providing a more complete assessment of the system’s security and reliability. In their study, Y. Huang et al. [37] address the challenge of detecting

hardware Trojans by proposing a novel [SCA](#) test generation approach based on multiple excitations of rare switching (MERS). Their method significantly increases Trojan detection sensitivity, achieving higher detection coverage through scalable statistical test generation and judicious test set reordering.

Power consumption stands out among the various side channels available for analysis as one the most effective for [ES](#) fuzzing. Power consumption measurement is relatively easy to perform and does not require specialized equipment. Most [ES](#) can be interfaced with power measurement tools, making this side channel widely applicable. M. Dunne and S. Fischmeister [23] propose using power trace monitoring as an additional feedback channel for fuzzing vehicle cyber-physical systems. By constructing a hardware-in-the-loop system, they demonstrate the effectiveness of this black-box approach in detecting responses from [Electronic-Control UnitS \(ECUs\)](#) on a [Controller-Area Network \(CAN\)](#) bus, even identifying hidden backdoors and undocumented functionality with high probability. Gamaarachchi, Hasindu, and Harsha Ganegoda [30] constructed a testbed for power analysis attacks to facilitate future studies and developed novel methods to breach the Speck cryptographic algorithm within an hour, advocating the effectiveness of using power as side-channel. Power traces also often contain much information about the system’s internal operations. Different instructions and states within the [ES](#) can produce distinct power consumption patterns, allowing detailed analysis of the system’s behaviour [90]. Power measurements can be taken at high sampling rates, providing fine-grained temporal resolution. This lets us capture detailed power consumption patterns corresponding to specific operations within the [ES](#), facilitating precise fault detection. Moreover, power consumption directly correlates with the [ES](#)’s computational activity. As the system executes different tasks, processes data, or encounters faults, these activities display as variations in power consumption. This makes power analysis particularly effective for monitoring and understanding the [ES](#)’s behaviour during fuzzing.

Chapter 3

Abstract Overview of the Power Fuzzing Architecture

System architecture describes the organized framework used to define and visualize a system's behaviour, structure, and components, as well as their interactions and constraints. It guides the design and development process by acting as a blueprint outlining the parts of the system and how they interact [6]. The abstract overview of power fuzzing architecture provides a high-level understanding of the components and workflow involved in power fuzzing. This chapter outlines the fundamental elements and their interactions within the power fuzzing process, providing a better understanding of the architecture's general structure, underlying events and functionality.

3.1 Functional View

The functional view helps us outline a system's primary functions and the interactions between these functions. It focuses on what the system does and how the parts collaborate to perform the required operations. This view abstracts the implementation details, providing a high-level understanding of the system's capabilities and workflows [22]. Figure 3.1 shows the functional block diagram of the power fuzzing architecture. The architecture comprises three main devices responsible for processing messages, transmitting messages, and capturing power data, along with the TS that is being tested. Below is a brief overview of the devices, components, and functionality.

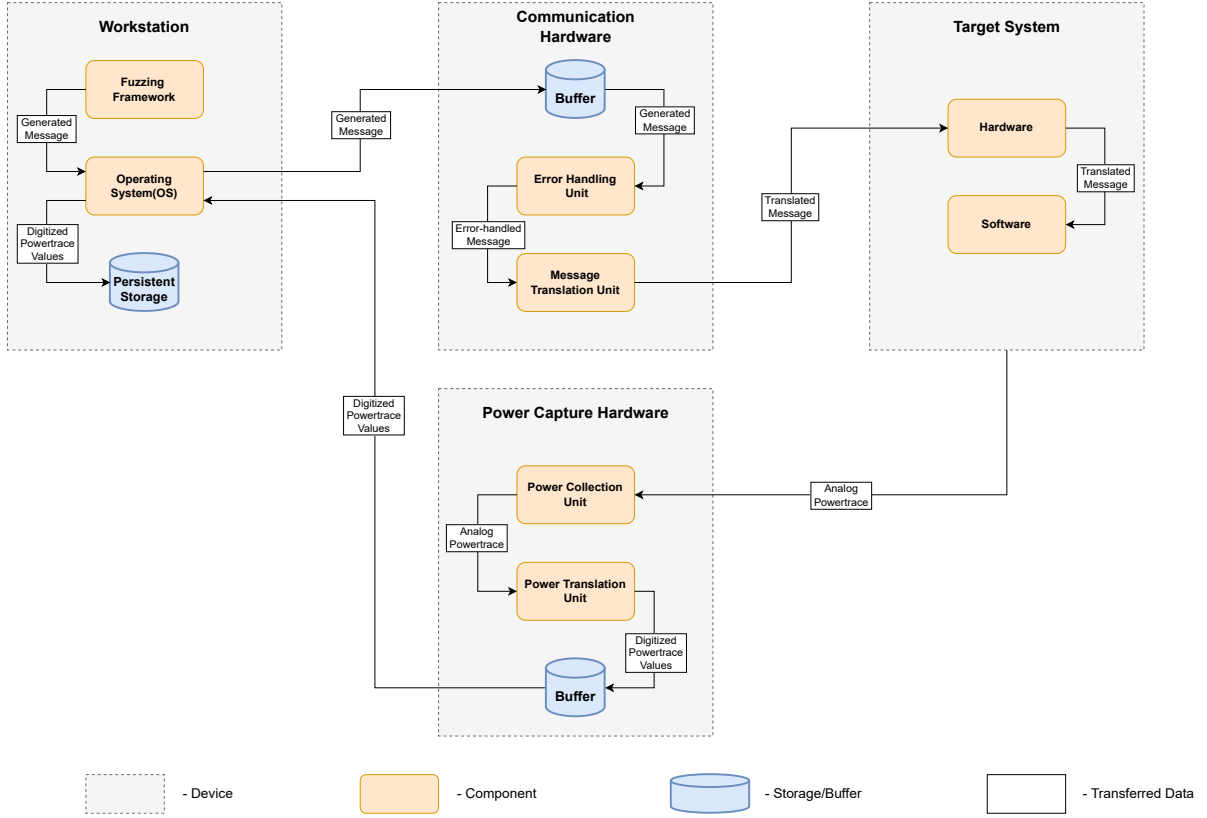


Figure 3.1: Functional View of Power Fuzzing Architecture

- **Workstation:** In a power fuzzing architecture, the workstation serves mainly two functions: (1) generating the required input message and (2) storing captured power data. The fuzzing framework generates the required input message accepted by the TS; upon generating the message, it sends it to the OS, which processes the message and sends it to the communication hardware. The OS is also responsible for receiving the power data from the Power Capture Hardware and storing it for further analysis.
- **Communication Hardware:** The TS generally cannot understand the message transmitted by the workstation. The Communication hardware translates the incoming message into a format the TS accepts by adding the required protocol-specific headers and control information. It also performs error detection and correction to ensure data integrity and synchronizes its own timing with the incoming data streams [38, 66]. In addition to these functions, the communication hardware may perform

buffering, temporarily storing data to manage flow control and prevent loss during high-speed transmissions.

- **Power Capture Hardware (PCH)**: Accurate capture and processing of power traces are important aspects of the power fuzzing architecture. The **PCH** ensures precision in capturing power data. The **Power Collection Unit (PCU)** detects and captures variations in the **TS**'s power consumption and can amplify the analog power trace if necessary. The **Power Translation Unit (PTU)** then maps these analog values to digital values for further processing by the workstation.
- **Target System (TS)**: The **TS** is an important element in the power fuzzing architecture. It is the **ES**, whose behaviour and power consumption are being analyzed. The **TS** receives input messages generated by the fuzzing framework, processes these messages, and performs the required tasks. Monitoring the **TS**'s power consumption during these operations helps identify vulnerabilities and inefficiencies.

3.2 Timeline Diagram

A timeline diagram is a visualization tool to show the temporal sequencing of activities within the power fuzzing process. This diagram shows each step's sequential order and duration, helping identify challenges and optimize process efficiency.

Figure 3.2 shows the power fuzzing architecture timeline diagram. E_i represents the different events, T_i denotes the specific time points at which each event occurs, D_i represents the duration between these time points, and the ϵ_i signifies the variability associated with each time duration. The power fuzzing architecture can be represented by a sequence of six events. Understanding these events and their timing is crucial for identifying the sources of variability and potential points of failure or inefficiency within the system. Subsection 3.2.1 provides a brief overview of these events to understand the power fuzzing architecture better.

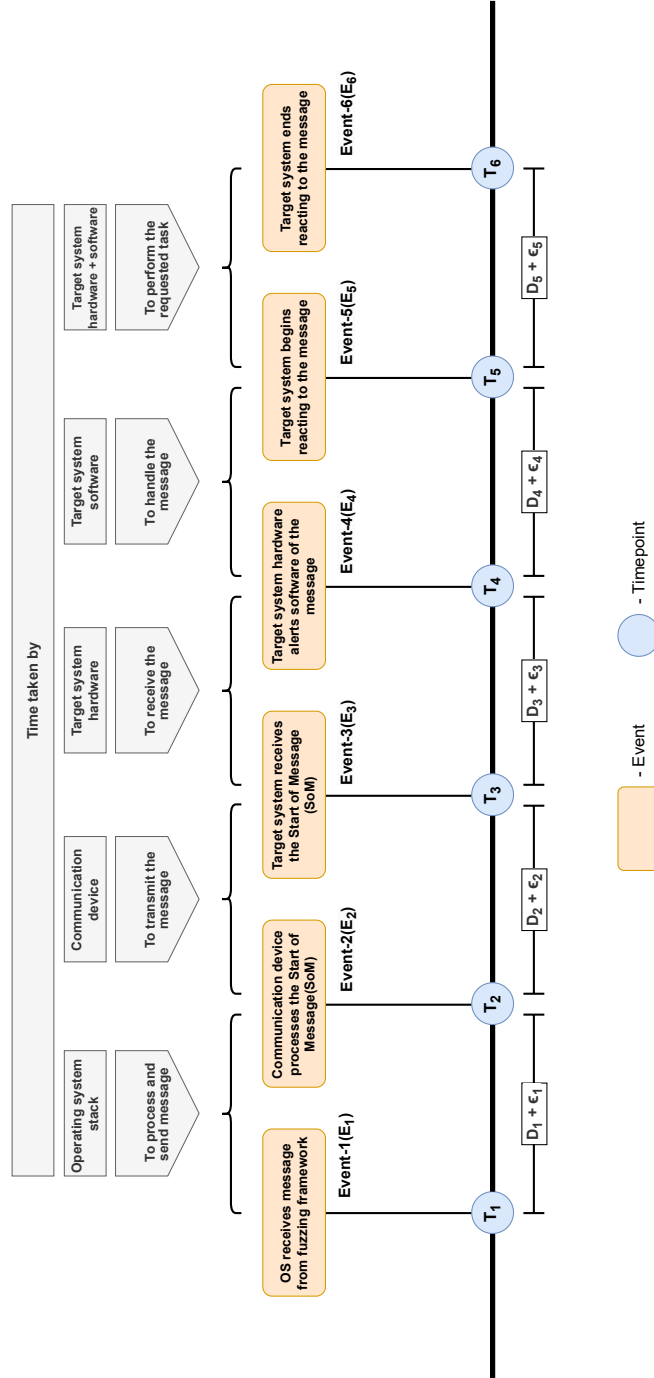


Figure 3.2: Timeline diagram of Power Fuzzing Architecture

3.2.1 Sequence of Events

Event-1 (E_1)

The fuzzing framework begins the process by creating messages based on the predefined input grammar specific to the TS. ES inputs might vary in structure depending on the system's complexity and use case. The Input grammar specifies the structure and rules defining the format of inputs provided to a system. Understanding input grammar is essential in the context of power fuzzing for ES, as it defines how inputs should be constructed to test the system effectively [61].

We can divide the input grammar into the following types:

1. **Command-Based Input Grammar (CBIG):** ES uses CBIG to perform actions through specific commands. These commands follow a predefined syntax and are usually simple text strings that the system interprets and executes.

For example, modems and Global System for Mobile Communications (GSM) modules use 'AT' commands, a standardized set of instructions for controlling modems [3]. Each command begins with the prefix 'AT', followed by a specific command keyword (AT+<command>)

2. **Message-Based Input Grammar (MBIG):** ES communicates through structured messages using MBIG. These messages typically consist of headers, payloads, and sometimes footers, following specific formatting rules.

For example, The CAN bus protocol uses messages to communicate between different ECUs in a vehicle [27]. Each message has an identifier and a data field that enables the system to request data from or instruct an ECUs to perform a certain task.

3. **Script-Based Input Grammar (SBIG):** SBIG involves writing scripts in a specific scripting language to perform tasks or tests. These scripts can be complex and are often used in ES that perform composite operations.

For example, IoT devices running GPOS like Linux or Windows use shell scripts or batch files to perform tasks. These scripts can perform various operations, from simple commands to complex workflows.

The fuzzing framework sends the message to the TS once it is generated. This is achieved through the OS, which provides a standardized interface for communication between software applications and hardware components. The fuzzing framework packages

the generated message and prepares it for transmission by invoking an OS system call. The OS system call is a mechanism that allows the fuzzing framework to request a service from the OS. In this context, it sends the generated message to the target system. Upon receiving the system call, the OS processes the request. This involves several steps, including validating the input, scheduling the transmission, and managing the underlying hardware resources. The OS then sends the message to the communication device.

Event-2 (E_2)

The communication hardware begins receiving message from the OS. It starts by detecting **Start of Message (SoM)**, indicating that a new message is incoming. The detection of the SoM triggers the communication device to prepare for data reception. Below-mentioned is a brief overview of operations performed by a communication hardware with UART communication protocol as an example.

1. **Protocol-Handling:** This operation involves interpreting the protocol-specific headers and control information to ensure the message adheres to the expected format and structure.

For example, when the communication hardware receives a UART message, it must recognize the start bit, then read and process the data bits according to the configured frame format as shown in Figure 3.3.

2. **Synchronization:** Communication hardware syncs its own timing with the incoming data stream to ensure accurate data reception.

For example, UART communication is asynchronous, meaning there is no shared clock signal between the sender and receiver. Instead, both must agree on the baud rate (bits per second) to ensure proper timing.

3. **Buffering:** It involves temporarily storing received data in memory buffers before processing. This helps manage data flow and check if upcoming parts of the message are correctly aligned and interpreted.

For example, UART devices often have hardware buffers (FIFO - First In, First Out) to store incoming data before it is sent to the TS.

4. **Error Detection and Correction:** This operation involves verifying the integrity of the received data to ensure it has not been corrupted during transmission.

For example, if the parity bit, as shown in Figure 3.3, is enabled, the receiver calculates the parity of the received data bits and compares it to the received parity bit. A parity error is flagged if there is a mismatch, indicating possible data corruption.

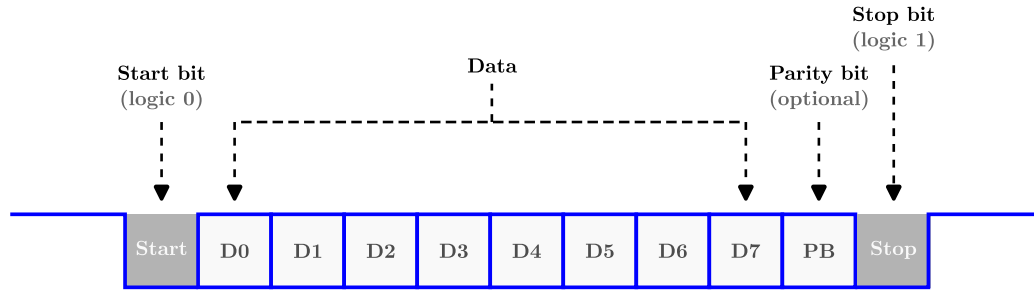


Figure 3.3: UART frame stack [5]

Dedicated communication hardware is essential because different protocols have unique requirements for timing, data structure, and error checking. It is necessary to translate the OS messages into a format that the ES can understand and respond to correctly. Once this translation is achieved, the message is transmitted to the TS

Event-3 (E_3)

TS begins receiving the translated message and detects SoM indicating arrival of new data. The TS hardware takes time to copy the entire message from the communication buffer into RAM or directly to the CPU for processing [70]. This operation involves various steps, which are outlined below.

1. **Buffering:** The communication device temporarily stores the incoming message in an internal buffer as it is received. This buffer acts as a holding area for the data until it can be transferred to the system's main memory or CPU.
2. **Interrupt Generation:** Once the buffer reaches a certain threshold or the entire message is received, the hardware generates an interrupt to notify the CPU that a new message is available for processing.

3. **Data Transfer to RAM/CPU:** The system begins data transfer from the communication buffer to the system's RAM or directly to the CPU registers. A Direct-memory Access (DMA) controller transfers the data from the buffer to RAM, reducing CPU usage in the data transfer process. Alternatively, the CPU might directly read the data from the buffer and write it into RAM or process it immediately if the data size is manageable.

Event-4 (E_4)

The TS software is informed of the new message and prepares to process it. The software is designed to handle tasks efficiently, but it may not always react to a new message immediately [70]. In the context of ES software, many factors can cause this delay:

1. **Context Switching:** If the CPU is currently occupied with other tasks, the system might need to perform a context switch. This involves saving the CPU's current state and loading the state needed to handle the new message. Context switching introduces latency as the system transitions from its current task to handling the new message.
2. **CPU Scheduling:** The software scheduler decides when the CPU should handle the new message based on its priority and the current workload. If the CPU is busy with higher-priority tasks, the handling of the new message might be delayed.
3. **Resource Availability:** The software may require access to certain resources, such as memory or Input-Output (I/O) interfaces, which are currently unavailable or used for other tasks. It may need to wait until those resources become available before it can continue processing the message.

Event-5 (E_5) and Event-6 (E_6)

E_5 and E_6 represent the start and end of the processing phase of the received message respectively. Each message is associated with a task of varying complexity. The TS hardware and software work together to execute the task efficiently. The majority of the task execution occurs within the application code and includes performing computations, data manipulations, and other specific operations required by the task [77]. During task execution, the application code must handle any encountered errors. Once the main task is

completed, the application performs post-processing steps such as logging results, updating states, or sending responses. The application code also manages these post-processing activities.

3.2.2 Analyzing Time Variability (ϵ_i):

We can observe from the timing diagram in Figure 3.2 that there is a time duration D_i between each event E_i occurring at timepoint T_i . Subsection 3.2.1 briefly discussed the operations associated with each event that contributed to this time duration D_i . We can expect that every time duration has a variability ϵ_i associated with it. This variability represents the uncertainty of the time duration between each event. Each variability ϵ_i is unique, and this Subsection aims to analyze the factors causing this variability in time duration.

Understanding the variability in time during the power fuzzing process is crucial for identifying and addressing the challenges that arise. Theoretically, the cause of variability in time duration between events can be broadly divided into two categories: noisy and controllable. The goal behind looking for sources of variability is to identify all controllable sources, isolate their effect on the total variability, and then figure out as many noisy sources as possible to prevent their effects as best as possible.

One effective method for visualizing and analyzing the factors contributing to time variability is the Ishikawa diagram, also known as a cause-and-effect diagram. This tool helps to systematically identify potential causes of variability and categorize them, making it easier to pinpoint specific areas for improvement [14]. Creating an effective Ishikawa diagram requires a structured approach to identify and organize all potential causes of a problem.

1. **Step-1:** Clearly define the problem or effect we want to analyze. In our case, the goal is to address the cause contributing to the variability in the time duration between events in a power fuzzing architecture. This will be the head of our diagram.
2. **Step-2:** The next step is to identify the main categories of causes that contribute to the problem and tailor the categories to fit the specific aspects of the problem if necessary.
3. **Step-3:** Finally, we need to brainstorm appropriate causes and group similar causes within each category.

Causes of Variability ϵ_1

This variability occurs in the time duration D_1 , which represents the time between [Event-1](#) (E_1) and [Event-2](#) (E_2). The time duration between these two events is associated with the generation of the message by the fuzzer and pre-processing of the message by the [OS](#) before transmitting the message further. This variation occurs due to the reasons outlined below.

1. **Message Property:** As mentioned in Subsection [3.2.1](#), the grammar of the input message may vary based on [TS](#) use-case and requirements. [CBIG](#) have simpler, more straightforward syntax compared to [MBIG](#), which may include headers, footers and payloads, making it more complex. [SBIG](#) complexity can range from simple to complex based on the task it performs. The time needed to process and format messages correctly can increase or decrease based on the variation in length and structure of the input message to the [TS](#) [\[102\]](#).
2. **OS Handling:** The fuzzing framework invokes an [OS](#) system call to send the message, and the time taken for the [OS](#) to process this call can vary. System calls involve transitioning from user mode to kernel mode, which adds overhead [\[99\]](#). The [OS](#) manages hardware resources, such as [CPU](#) and [RAM](#), and the efficiency of this management can also affect the time taken to send a message. Alongside this, the [OS](#) schedules the transmission of the message, balancing it with other tasks and processes running on the system, which can introduce delays depending on the system load [\[88\]](#). A. Beifuß et al. [\[10\]](#) show that the Linux kernel running on a [CPU](#) with a 3.3 GHz frequency can take up to 535 ns to process network packets.
3. **Hardware Efficiency:** The hardware interface at the source might also perform some operation, which might introduce a delay in transmission. The system often buffers data before transmitting it, and the time taken to buffer data can vary based on buffer size and availability. The efficiency of hardware drivers in managing data transmission affects overall performance [\[99\]](#). Drivers need to handle data transfer requests and manage hardware resources efficiently to reduce processing delays. L. Abeni et al. [\[2\]](#) present that memory and [I/O](#) operations in a Linux-based [OS](#) can take up to a few milliseconds to complete, depending upon the task.
4. **System Load:** In modern computers, this is generally not a cause of delay, but with the introduction of complex [OS](#), it might cause delays in performing required tasks. Other processes running simultaneously on the system can compete for [CPU](#), [RAM](#), and other resources, introducing delays. Background tasks, such as system

maintenance and updates, can consume resources and affect system performance as well.

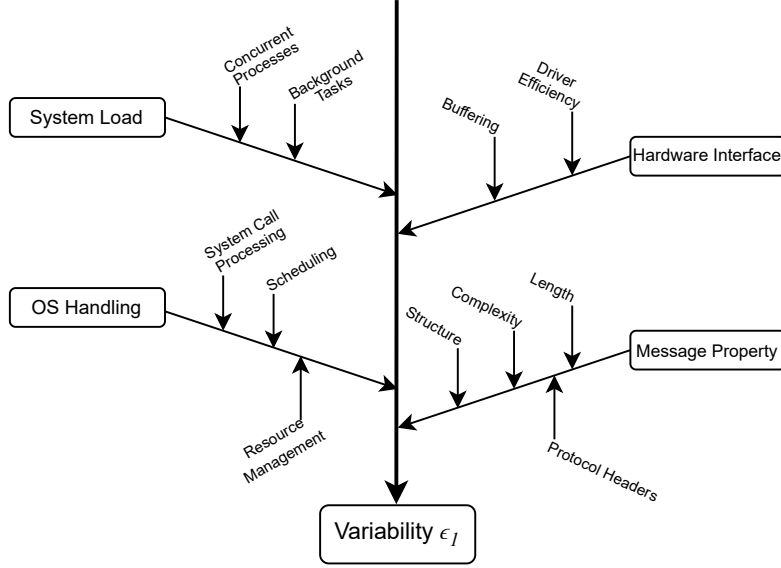


Figure 3.4: Variability ϵ_1 Cause and Effect Diagram

Causes for Variability ϵ_2

The communication hardware's task is to translate the incoming message from the OS into a format that the TS understands. The communication hardware performs various operations during this process, as mentioned in the previous Subsection 3.2.1. Multiple factors can introduce variability in the time duration it takes for the communication device to translate the message, which has been mentioned below.

1. **Protocol-Handling:** Messages belonging to each communication protocol require a different amount of processing time [102]. More complex protocols with detailed headers and control information require more processing time than simpler protocols. The specific configuration of the frame format also introduces variability in the processing time. For instance, a UART frame might be simple with start, data, parity, and stop bits (c.f. Figure 3.3), while protocols like CAN or Ethernet involve more detailed frame structures [27, 94] and require more time to translate.

2. **Synchronization:** Some communication protocols have clock synchronization, while others do not. Generally, synchronized systems benefit applications requiring high data rates, low latency, and predictable performance [42]. In contrast, asynchronous systems offer simplicity, flexibility, and scalability, making them suitable for less timing-critical applications. Proper synchronization is important for accurate data reception [42]. In asynchronous protocols, any variation in the transfer rate requires re-synchronization, introducing variability. Moreover, differences in clock speeds between the sender and receiver can cause misalignment, necessitating periodic re-synchronization to ensure data integrity, contributing to timing variability as the system works to maintain synchronization. W. Elmenreich and M. Delvai [26] show in their study how commercial UARTs display timing inaccuracies due to arithmetic errors in their baud rate settings and potential send jitter. The instability of a real clock causes baud rate drift and clock deviations, requiring synchronization every few seconds.
3. **Error Management:** Many communication protocols in the scope of ES have robust error-checking mechanisms. This is important to maintain the integrity of transmitted data. The error-checking features add to the processing time as each data byte must be checked for errors. The complexity of these mechanisms directly impacts the time taken for error detection [91]. When the system detects errors, it requires additional time to handle them. This might involve requesting re-transmissions or applying correction algorithms.
4. **Buffering:** The communication device translates every byte of the message received from the OS before transmitting it to the TS, and hence, it is required to store this data into a buffer [78, 29, 65]. The size and efficiency of the hardware buffers are important in managing data flow. Larger buffers can hold more data, potentially reducing the need for immediate processing but also increasing latency if not managed efficiently. J. Li and J. Shen [49] show that managing DMA buffer size in a UART to WLAN converter reduces transmission errors and maintains high-speed data transfer, preventing delays of several seconds at low speeds. If the incoming data rate exceeds the processing rate, buffers may overflow, leading to data loss and the need for re-transmissions. This overflow causes variability as the system deals with managing and recovering from these events.

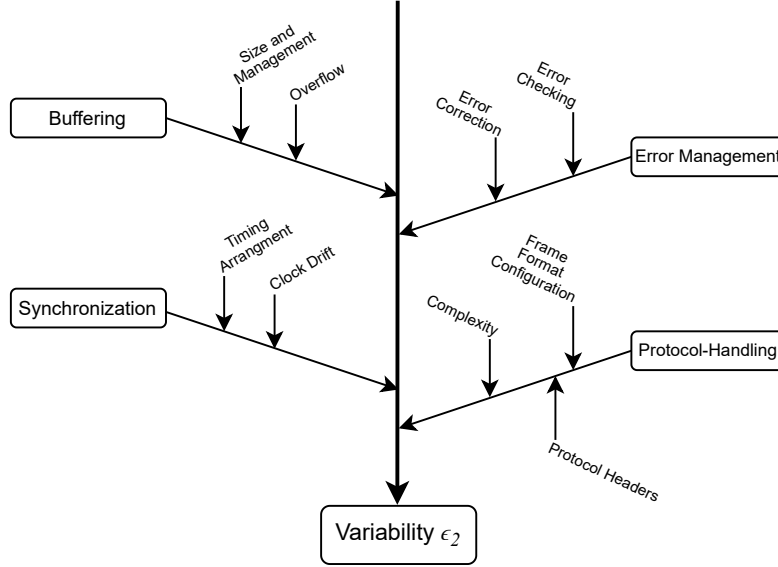


Figure 3.5: Variability ϵ_2 Cause and Effect Diagram

Causes for Variability ϵ_3

Once the **TS** hardware receives the message, it must perform various operations to prepare the message for processing. The time duration these operations take is dependent on the hardware capabilities of the **TS**. Multiple factors can bring variability in message processing time by the **TS** hardware.

1. **Interrupt Generation:** The time the hardware takes to generate an interrupt and for the **CPU** to acknowledge and handle it can vary based on the system's interrupt-handling capabilities and current **CPU** load. For instance, The **Advanced-RISC Machine (ARM)** Cortex-M4's interrupt latency is around 12 clock cycles (approximately 150 ns at 80 MHz), and context switching takes about 6 clock cycles [101], with variability introduced by current **CPU** load and task priority. Interrupt handlers can take up to a few microseconds to complete [9]. If multiple interrupts are occurring, the priority of the interrupt can affect how quickly it is serviced. Additionally, a high frequency of interrupts can cause delays in handling each interrupt [80], leading to variability in processing times.

2. **Buffering:** As mentioned previously in Subsection 3.2.1, buffering can cause variability in the timing duration between events. In the scope of TS hardware, different buffer sizes can lead to varying amounts of data being stored before processing. Smaller buffers may require more frequent transfers, while larger buffers might introduce delays if they take longer to fill [42]. If the buffer overflows due to high incoming data rates, it can cause data loss or necessitate re-transmission, introducing variability [45]. Inefficient buffer management algorithms can also delay storing and retrieving data.
3. **Data Transfer to RAM/CPU:** The received message is either stored in the RAM for later processing or is sent to the CPU for immediate processing. If the data is sent to RAM, the efficiency of the DMA controller in transferring data can introduce variability. For instance, the DMA controller in the ARM Cortex-M4 can achieve data transfer rates up to 20 MBs^{-1} , whereas the DMA controller in the ARM Cortex-M3 can achieve data transfer rates only up to 15 MBs^{-1} [101]. This results in a 33% reduction in CPU load in the ARM Cortex-M4 compared to the Cortex-M3. Moreover, the time taken to access RAM can vary due to memory latency [75], which is influenced by factors like memory type, current load, and memory controller efficiency, contributing to the variability. If the data is directly sent to the CPU, the current CPU load and transfer task's priority can introduce processing delays.
4. **Environment Factors:** Finally, ambient temperature variations can affect the TS hardware and impact the performance of electronic components [68]. Higher temperatures can lead to thermal throttling, where the system slows down to prevent overheating. Benoit-Cattin et al. [12] show that active cooling with a hysteresis-based fan prevents thermal throttling in edge devices, improving CNN throughput by up to 90% compared to scenarios without cooling. Electro-Magnetic Interferences (EMI) from nearby electronic devices or equipment can disrupt signal integrity and communication between hardware components [85]. This can lead to errors in data transmission, requiring re-transmissions and causing delays in processing.

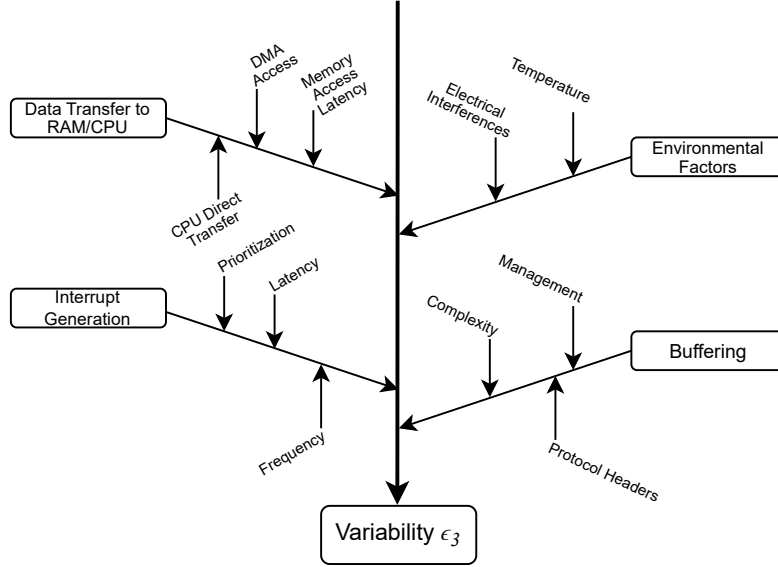


Figure 3.6: Variability ϵ_3 Cause and Effect Diagram

Causes for Variability ϵ_4

As highlighted in Subsection 3.2.1, the TS software may or may not process a message as soon as it is notified about it from the TS hardware due to various software-related overheads. Several factors can introduce variability in the time it takes for the software to begin processing the message. These factors are mainly related to the nature of the ES and the specifics of the TS software and hardware interactions. The factors have been categorized and outlined below.

1. **CPU Scheduling and Load:** The CPU scheduler determines the order in which tasks are executed based on their priority and the current system workload. If the CPU is busy with higher-priority tasks, the processing of the new message may be delayed until those tasks are completed. The variability in CPU scheduling depends on the scheduling algorithm used and the current load on the system. Additionally, context-switching occurs when the CPU must switch from one task to another. This process involves saving the state of the current task and loading the state of the new task. It introduces latency because the system must pause the current task, save its state, load the state of the new task, and then begin execution. This delay can vary

depending on the complexity of the tasks and the efficiency of the context-switching mechanism. K. Baynes et al. [7], from their simulation bed, show that for different embedded OSs and even simple schedulers, CPU load could vary execution time of tasks in the order of milliseconds.

2. **Resource Availability:** The software may need to access specific memory locations that are currently in use or require allocation. Waiting for memory access or allocation can vary based on number background processes running and the number of tasks contending for the same resource. A similar scenario is possible when the system tries to access I/O interfaces.
3. **Interrupt Handling:** The time taken to handle hardware interrupts notifying the software of the new message may vary due to various factors such as nested interrupts, interrupt prioritization, and lengthy Interrupt-Service Routines (ISRs). ISRs that take a long time to execute due to complex processing requirements can block other interrupts and delay the overall system response [80]. Additionally, the overhead associated with saving and restoring the CPU state during interrupt handling, as well as resource contention among different ISRs, can further contribute to the variability in response times. Jian Yang et al. [100] demonstrate that the execution time for different ISRs in a Linux kernel can vary by a few nanoseconds and by several milliseconds between two different kernel versions.
4. **Software Efficiency:** The efficiency of the software code in handling the new message is important to handle incoming messages quickly. Poorly optimized code can increase the time needed to begin processing the message. The overhead associated with creating and terminating threads to handle the new message contributes to the time delay. Additionally, the time required for processes to communicate with each other through Inter-Process Communication (IPC) mechanisms like message queues, pipes, or shared memory introduces variability in processing time [74]. There is also the overhead associated with making system calls. Frequent system calls or system calls with high overhead can slow down the processing of the message. Se-Ho Park et al. [74] present the performance of embedded Linux and Windows on an ARM platform, showing that completion times can vary by 2-5 μ s between the OSs and typically take several microseconds to complete.

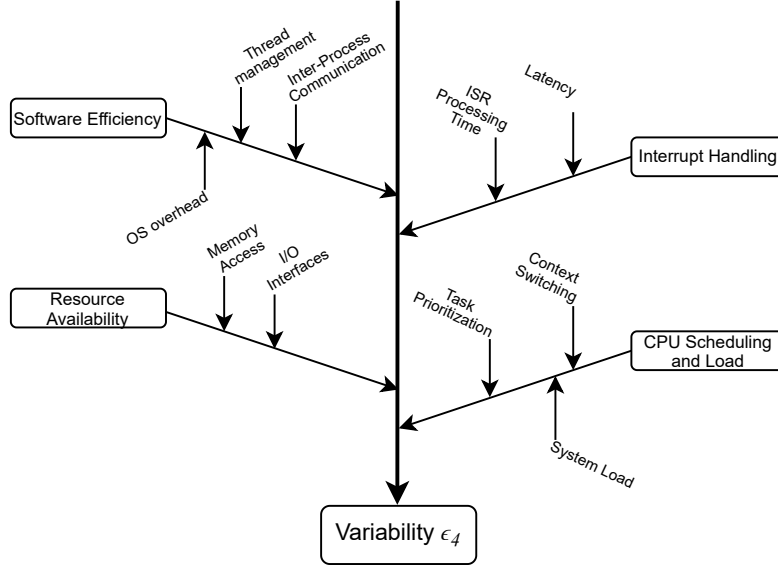


Figure 3.7: Variability ϵ_4 Cause and Effect Diagram

Causes for Variability ϵ_5

Every input message to the TS triggers the execution of a different task. The time required to complete these tasks can vary significantly due to the complexity and nature of the operations involved, making it difficult to quantify the variability accurately. Several factors influence this variability in task execution time, as listed below.

1. **Task:** Different tasks require varying amounts of computational power. Complex algorithms, extensive data processing, or heavy calculations will take more time. Tasks involving large data sets or extensive data manipulation will take longer to complete as well. In systems with real-time constraints, certain tasks may be prioritized over others. The time to complete the current task can vary based on its priority relative to other running tasks. All this can lead to variability in time duration to execute a task.
2. **I/O Operations:** Interaction with peripheral devices can introduce delays. The speed and efficiency of these devices, as well as the bus architecture, affect the overall timing [97]. Time taken for data transfer over communication interfaces can also

introduce variability. If tasks involve reading from or writing to disk, the variability in disk I/O speeds can affect timing. Tasks that depend on network communication can be affected by network latency and throughput issues.

3. **Error Handling:** Error handling can introduce time delays and variations during message transmission. Similarly, detecting errors during task execution will require additional time to correct them. This includes retry mechanisms, data validation, and recovery procedures. Handling unexpected conditions or exceptions during task execution can bring delays in task execution. Moreover, logging detailed error messages and diagnostic information can take additional time [64], especially if the logging mechanism is synchronous.
4. **System Resources:** Task execution requires the involvement of both TS hardware and software. Hence any delay in these resources for processing the message can cause variability in time duration to process the message. If multiple tasks or processes share the CPU, the current CPU load can cause the time to react to a message to vary. Delays in accessing RAM or other memory components can also introduce variability. If the task involves frequent memory reads/writes, this can also significantly impact timing.

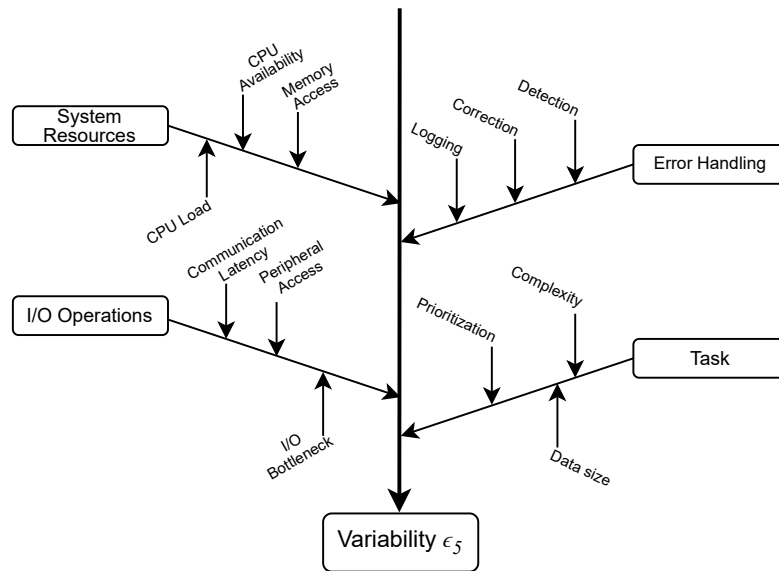


Figure 3.8: Variability ϵ_5 Cause and Effect Diagram

Chapter 4

Challenges

Chapter 3 provided a broad overview of the elements and events associated with the power fuzzing architecture. We can observe that power fuzzing requires the efficient collaboration of hardware and software components to fuzz an [ES](#) effectively. While the power fuzzing architecture is a powerful tool for identifying vulnerabilities in [ES](#), it faces several significant challenges.

These challenges mainly arise from the timing variability between different events of the power fuzzing architecture as shown in Figure 3.2, hardware limitations of the different devices shown in Figure 3.1 and the various uncontrollable properties of the [TS](#) itself. Timing variability can lead to inconsistencies in test results, making it difficult to identify vulnerabilities accurately. Hardware limitations, such as the precision of power measurement devices and the processing capabilities of the fuzzing framework, can constrain the effectiveness of the power fuzzing process. Additionally, the [TS](#) may exhibit unpredictable behaviour due to environmental factors, inherent design complexities, and other uncontrollable properties, further complicating the fuzzing efforts.

This section elaborates on some of the key challenges posed by the current power fuzzing architecture. Each challenge is detailed and explained through experiments conducted on an [ES](#) to provide a clearer understanding of the issues. By highlighting these challenges, we can identify areas for improvement and potential solutions to enhance the overall effectiveness of power fuzzing techniques. The insights gained from these experiments will contribute to developing more robust and reliable fuzzing strategies for [ES](#).

4.1 Experimental Setup

This section provides an overview of the hardware and software components used to gather power traces for visualizing and understanding the challenges posed by a traditional power fuzzing architecture. The setup comprises various components, each playing an important role in ensuring the accurate capture and analysis of power traces. The experimental setup follows the architecture specified in Section 3.1. We detail the specifications and roles of each component, including the workstation, communication hardware, [TS](#), and [PCH](#), to highlight their contributions to the overall experiment and to address the inherent challenges effectively.

4.1.1 Hardware

Workstation:

It performs major tasks of generating input messages, storing power traces and analyzing power traces. Lenovo ThinkCentre M90t Desktop was selected as the computing platform, and its specifications are mentioned in Table 4.1. The multi-core [CPU](#) and substantial [RAM](#) ensure efficient execution of input generation scripts and concurrent processing of long power traces. The high storage capacity allows for the storage of a vast set of power traces. Furthermore, the latest generation [Universal Serial Bus \(USB\)](#) ports ensure continuous data transfer to the [TS](#).

Components	Specifications
RAM	64 GB
CPU	Intel(R) Core(TM) i5 -12600
Storage	1 TB
USB Ports	USB-A 3.2 Gen 1 and Gen 2

Table 4.1: Specification of Lenovo ThinkCentre M90t Workstation

[Target System \(TS\):](#)

[RPI 3B](#) development board is used as the [TS](#). The specifications of the [RPI 3B](#) development board are detailed in Table 4.2. This board supports multiple communication protocols,

making it highly versatile for various applications. Additionally, it supports Embedded-Linux, which closely resembles the OS used in consumer IoT products on the market. This makes the RPi 3B a good choice as a TS for our experiments, providing a realistic and practical environment to test and analyze power fuzzing techniques.

Components	Specifications
RAM	1 GB
CPU	ARM(R) Cortex-A53
Storage	Variable
GPIO Pins	40
Communication Protocols	Wi-Fi, Bluetooth, UART, SPI, I2C

Table 4.2: Specification of RPi 3B Development Board

Communication Hardware:

UART communication is chosen for the interaction between the workstation and the TS. UART is a widely adopted communication protocol, particularly for debugging ES. Many consumer-grade IoT products feature exposed UART pins specifically for testing purposes, making it an ideal choice for our experimental setup. Workstations communicate using USB, while ES, like the RPi 3B, rely on Transistor-Transistor Logic (TTL) level serial communication. Hence, we require to use a USB-TTL converter to bridge the communication between the workstation and RPi. Our experimental setup uses the DFRobot DFR0065 USB-TTL converter. It features a compact design, plug-and-play functionality, and compatibility. It also has dual TTL logic level (3.3V/ 5V) support required by different TS.

Power Capture Hardware (PCH):

Our PCH is based on the current measurement of the TS inspired by the setup implemented by M. Dunne et al. [23]. Current flow is one of the most frequently used parameters for assessing and diagnosing the operational efficiency of electronic systems [20]. Monitoring current flow allows for accurate measurement of power consumption and enables the identification of irregularities or spikes that can indicate the presence of bugs or vulnerabilities in the system [33]. As mentioned in section 3.1, PCH consists of two components: (1) PCU and (2) PTU. In our experimental setup, the PCU consists of a shunt resistor board

attached to a 2-stage fully differential amplifier circuit. The shunt resistor is placed in the current path and develops a voltage across itself proportional to the current passing through it. Because the shunt resistor should not affect the current flow significantly, it is often quite small. As a result, the voltage developed across the shunt resistor is also quite small. Hence, we pass this small voltage drop through an amplifier circuit that amplifies the input voltage by two times. This amplified analog voltage needs to be converted into a digital representation, which is handled by the [PTU](#). Teledyne ADQ32-PCIe digitizer is used as the [PTU](#) in our experimental setup. It is a high-performance [Analog-to-Digital Converter \(ADC\)](#) that provides accurate and precise digital representation of analog signals. It has a substantial on-board storage to store long power traces and connects to the [Peripheral Component Interconnect Express \(PCIe\)](#) slot of the workstation, enabling fast data transfer for further analysis. Specifications of the digitizer have been provided in [Table 4.3](#).

Components	Specifications
ADC Resolution	12 bit
Maximum Sampling Frequency	2.5 GHz
Input Voltage Range	0.5 V_{pp}
On-Board Memory	8 GB

Table 4.3: Specification of Teledyne ADQ32-PCIe Digitizer

4.1.2 Software

The Lenovo ThinkCentre M90t workstation has been equipped with Linux-based Ubuntu [OS](#) [34]. Ubuntu is known for its stability and reliability, making it a good choice for running long-term experiments without unexpected crashes or interruptions. Its efficient resource management ensures better performance, minimizing the impact of background processes. The necessary Teledyne drivers to communicate with the ADQ32-PCIe digitizer, [USB](#) device drivers and dependencies to run python and C++ codes have also been installed in the Ubuntu [OS](#).

The [RPi](#) 3B development board is equipped with Raspbian [OS](#) [36]. It is optimized for the [RPi](#) hardware. It provides a user-friendly environment tailored to [RPi](#)'s [ARM](#) architecture, making it ideal for prototyping and developing [IoT](#) projects. Its lightweight design ensures efficient performance on the limited resources of the [RPi](#), while maintaining compatibility with various applications and peripherals.

4.2 Data Capture

The quality of captured data is crucial in effectively performing any [SCA](#). The feedback to the fuzzer needs to be accurate and reliable to analyze and detect undesirable behavior of the system. Inaccurate data can lead to false positives or missed vulnerabilities, undermining the integrity of the analysis. Moreover, the resolution and granularity of the captured data are essential for identifying subtle variations in power consumption that may indicate potential security flaws.

In the scope of power fuzzing, good data would mean good quality of captured power traces. An ideal power trace would have the following features:

1. **High Sampling Rate:** The power trace should be captured at a sufficiently high sampling rate to accurately capture the transient power fluctuations caused by data-dependent operations within the target device. A higher sampling rate allows for better resolution and detection of subtle power variations [\[62\]](#).
2. **Low Noise:** The power trace should have low noise and interference from external sources. Noise can obscure the data-dependent power signatures, making it difficult to extract useful information [\[46\]](#). Proper shielding, filtering, and noise reduction techniques are essential.
3. **Precise Synchronization:** The power trace should be precisely synchronized with executing the target operations or instructions within the [TS](#). Accurate synchronization is crucial for aligning the power trace with specific operations and extracting meaningful information [\[62\]](#).
4. **Sufficient Trace Length:** The power trace should capture the duration of the target operations or algorithms of interest. A trace that is too short may miss important power signatures or fail to capture the complete execution flow [\[90\]](#).
5. **Multiple Traces:** Ideally, multiple power traces should be collected for the same operations or algorithms. This allows statistical analysis, averaging, and noise reduction techniques to improve the signal-to-noise ratio and increase confidence in the extracted information [\[17\]](#).

Capturing an ideal power trace with the desired features poses significant challenges in a practical power fuzzing setup. Hardware and software interactions at various stages of the power fuzzing architecture introduce timing variability, making it difficult to obtain good-quality power traces. Subsection [3.2.2](#) explains this timing

variability in detail. This section provides a comprehensive overview of the challenges posed by the timing variability between different events in capturing power trace data.

4.2.1 Capturing Data at the Right Instant

One of the challenges in power fuzzing architecture is capturing data at the precise moment when significant events occur. Timing variability between events poses a significant hurdle. Each event in the power fuzzing timeline (c.f. Figure 3.2) has associated durations (D_i) and variabilities (ϵ_i). These variabilities cause deviations when the events occur, making it difficult to synchronize data capture perfectly. For instance, even minor delays in detecting the SoM or transferring data from communication buffers to the CPU can result in missed or inaccurate power traces. Aligning the exact moment of data capture with the occurrence of critical events is important for good-quality power trace data. Still, it is difficult to achieve this in a real-world setting, as we will observe in this Subsection.

Figure 4.1 illustrates the power consumption pattern of the RPi development board when subjected to the ‘ls’ [55] command. This power trace was collected at a frequency of 1 MHz for 100,000 sample points, representing a total time period of 100 ms. The portion of the power trace circled in orange represents the time duration D_3 between Event-3 (E_3) and Event-4 (E_4). The System-On-Chip (SoC) on the RPi includes a Mini-UART peripheral that handles general-purpose UART communication [13]. This peripheral directly interfaces with the CPU. The time the Mini-UART peripheral takes to receive and process the message depends on the number of bytes received. The longer the message, the more time it takes to synchronize, error-check, and buffer the message. Consequently, this results in higher power consumption for a longer duration. In our case, since the ‘ls’ [55] message is just two bytes long, we observe a spike in power consumption for only a short period of time. The portion of the power trace circled in green represents the time duration D_5 between Event-5 (E_5) and Event-6 (E_6). This is the total duration for the TS to react to the received message. In this scenario, it is the time taken by the SoC on the RPi development board to process the ‘ls’ command. This time duration also varies based on the complexity of the task to be performed, its priority, and the availability of hardware resources. Figure 4.1 also shows multiple significant spikes in power consumption of the RPi occurring at different time points. These spikes are undesirable and occur due to the various periodic and aperiodic background processes running simultaneously, which consume CPU cycles. These background processes are essential for the basic functioning of the TS. Interrupting or halting them could potentially damage the TS or disrupt its normal operations.

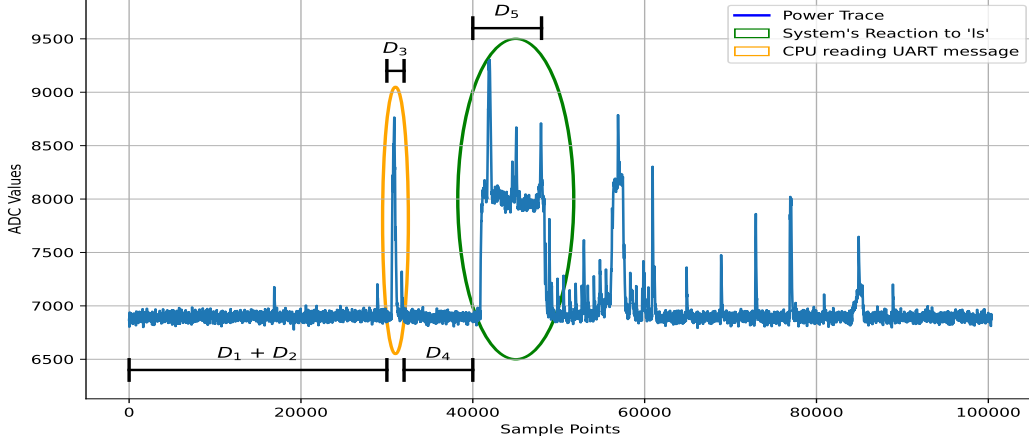


Figure 4.1: Power Consumption of RPi for 'ls' [55] Command

Now, let us examine the portion of the power trace before the occurrence of [Event-3](#) (E_3), annotated by time duration $D_1 + D_2$. This portion represents the time it takes for the workstation's OS to process and send the generated message to the communication hardware (c.f. [Event-1](#) (E_1)) and the time it takes for this communication hardware to translate the generated message and send it to the TS (c.f. [Event-2](#) (E_2)). This portion of the power trace is undesirable to us as we are only interested in the region of the power trace circled in green, representing the actual reaction of the TS to the received message. In the current power fuzzing architecture, power trace data collection is started just before sending the generated message. Hence, this captures the time it takes for [Event-1](#) (E_1) and [Event-2](#) (E_2) to complete. It is difficult to gauge how long these two events take to complete due to various factors mentioned in Subsection 3.2.2. Figure 4.2 depicts the variability $\epsilon_1 + \epsilon_2$ associated with time duration $D_1 + D_2$ (c.f. Figure 3.2). We can observe that the amount of time it takes to process, translate and send the 'ls' [55] message varies by a visible amount. During this time, when the TS has not received any message, it consumes only the default amount of power, as shown in Figure 4.1. However, even in this idle state, multiple background processes run simultaneously. These processes consume CPU cycles and can impact the TS power consumption. If we start capturing power traces while a background process is running, the spike in power consumption might appear in the starting portion of the power trace. This is visible in the 2nd and 4th captures in Figure 4.2. These undesirable spikes can lead to incorrect classification of power traces during further analysis.

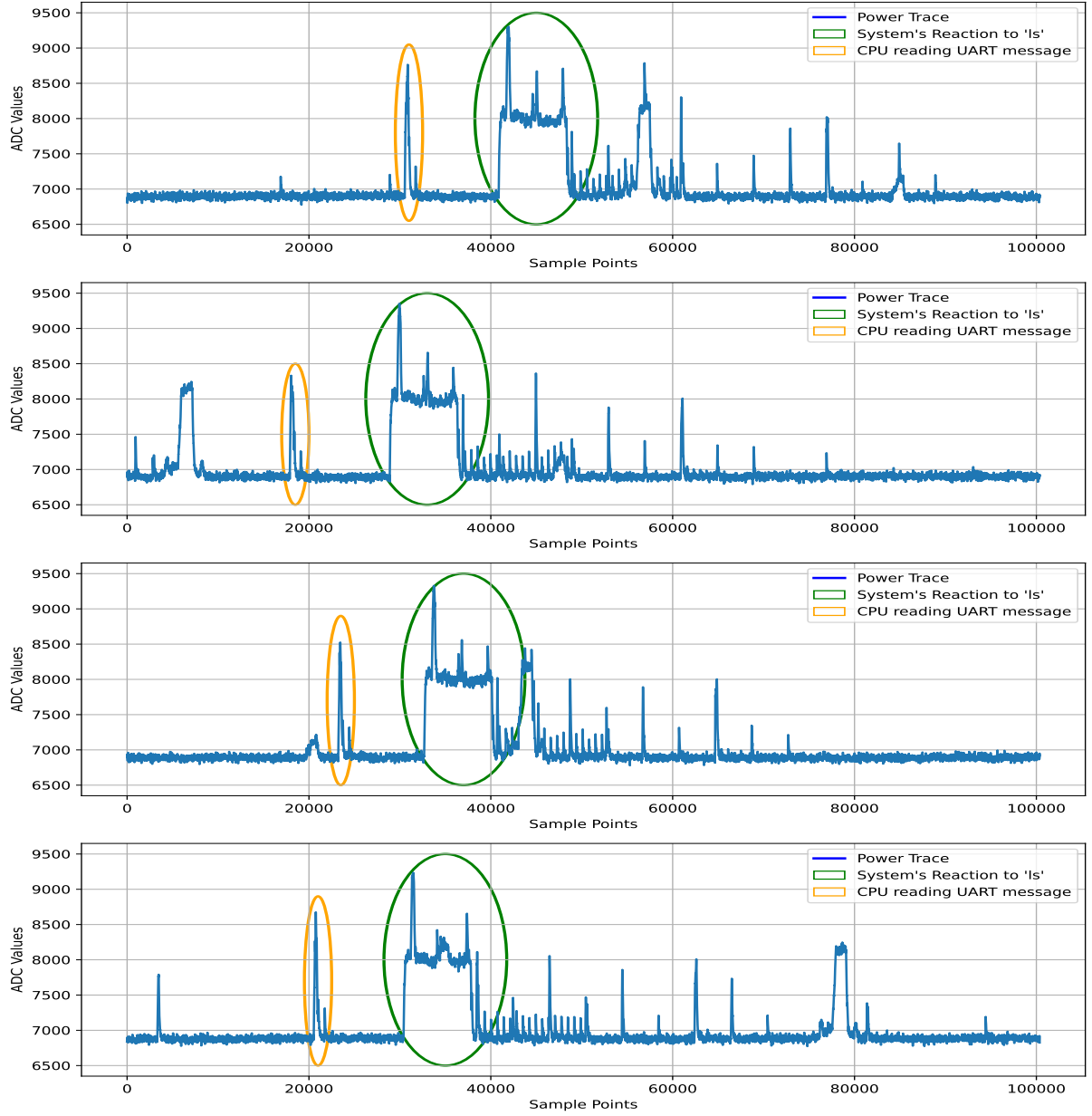


Figure 4.2: Power Consumption of RPi for 'ls' [55] Command over Multiple Runs

Figure 4.3 gives us a better understanding of the variability in time duration for [Event-1](#) (E_1) and [Event-2](#) (E_2) to complete over multiple transmissions of the ‘ls’ [55] command. We can observe that for 500 transmissions of the ‘ls’ command to the [RPI](#) development board in the given experimental setup, the mean completion time ($D_1 + D_2$) for [Event-1](#) (E_1) and [Event-2](#) (E_2) is 33.63 ms. The standard deviation of 9.42 ms indicates visible variability ($\epsilon_1 + \epsilon_2$) in the completion times, suggesting that some executions take considerably longer or shorter than the average. The minimum observed completion time is 11.80 ms, while the maximum is 51.72 ms, resulting in a range of 39.92 ms. The spread is five times larger than the total execution time for ‘ls’ [55] command, further emphasizing the variability ($\epsilon_1 + \epsilon_2$) in the completion times. We also observe that the distribution is not Gaussian, which appears inconsistent with the [Central Limit theorem \(CLT\)](#) [48]. However, this arises because the variability ($\epsilon_1 + \epsilon_2$) is derived from multiple independent factors with non-identical distributions. The [CLT](#) holds true only when these factors are identically distributed. Lindeberg’s theorem [51] extends the [CLT](#) to accommodate non-identically distributed variables, provided they satisfy Lindeberg’s conditions. Given the complex interdependencies and non-deterministic nature of [OS](#) behaviour, it is improbable that [OS](#) variabilities strictly adhere to the Lindeberg conditions. Quantifying the impact of these various factors would make for an interesting future research topic.

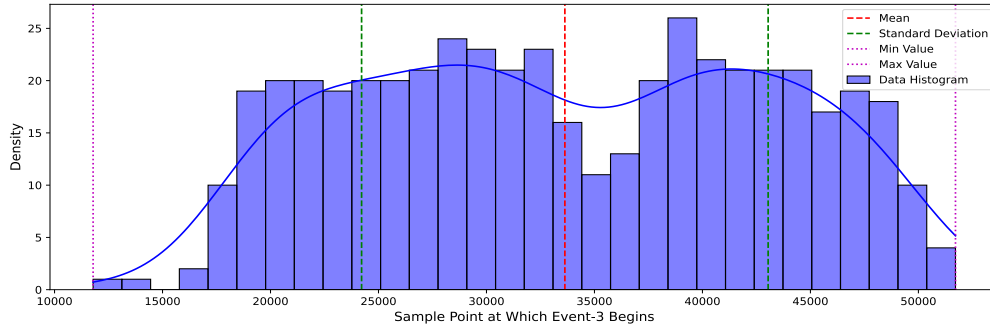


Figure 4.3: Histogram Representation of Sample Points at Which [Event-3](#) (E_3) Begins When ‘ls’ [55] Command Is Sent to the [RPI](#) Development Board 500 Times. Power traces were collected for 100,00 Sample Points at 1 MHz

The observed variability differs for different experimental setups. It is also possible that we might see an extended range of variability for more number of runs. We cannot control the workstation [OS](#)’s basic functioning or the communication hardware’s limitations, making this timing variability unavoidable and somewhat unpredictable.

4.2.2 Capturing Data for the Right Duration

In the previous Subsection, we visualized how the variability $\epsilon_1 + \epsilon_2$ in time duration $D_1 + D_2$ needed to complete [Event-1](#) (E_1) and [Event-2](#) (E_2) affects our ability to capture data at the right instant. In this Subsection, we are going to focus on variability ϵ_5 occurring in time duration D_5 between events [Event-5](#) (E_5) and [Event-6](#) (E_6). As shown in [Figure 4.1](#), we only need the portion of the power trace circled in green for our analysis, while the other regions would introduce noise into the data. Even if we manage to isolate this crucial portion of the power trace from the rest, we won't consistently capture the correct duration every time due to two major reasons: (1) The [TS](#)'s reaction time for the same message is not consistent, and (2) The [TS](#)'s reaction time varies for different messages.

Let us begin by examining the first reason, as mentioned in [Subsection 3.2.2](#): running a task efficiently requires the availability of system resources at the right moment. However, due to the presence of multiple background processes, there is a contention for the system resources, and it might take longer than expected to complete a task. [Figure 4.4](#) shows the variations in time duration it takes to execute a particular task over a period of multiple runs. This data was collected by subjecting the [RPI](#) development board to the 'ls' [\[55\]](#) message 100 times. In our experimental setup, the [RPI](#) is running Raspbian [OS](#) [\[36\]](#), which is based on the Linux kernel. This allows us to use the in-built 'time' command to measure the duration of time a command takes to execute [\[58\]](#). The 'time' command accounts for the user [CPU](#) time and the system [CPU](#) time consumed by the input command. We can observe that even at low [CPU](#) load of 1%, the time [RPI](#) takes to execute 'ls' [\[55\]](#) command is not very consistent, it varies between 8.4ms and 9.2ms indicating a timing variability of $\epsilon_5 = 0.8\text{ms}$. When the [CPU](#) load is high at 60%, we see a 25% increase in the execution time of the 'ls' command as compared to when the [CPU](#) load was low at 1%. This increase in execution time can be linked with the delays in [I/O](#) operations and context-switching in the [OS](#) as mentioned in [Subsection 3.2.2](#). Alongside this, we can see that variability ϵ_5 in the execution time of the 'ls' [\[55\]](#) command has also increased. At a high [CPU](#) load of 60%, execution time of 'ls' command varies between 10.6ms and 12.2ms indicating a timing variability $\epsilon_5 = 1.6\text{ms}$, which is a 60% increase in variability compared to the variability at a low [CPU](#) load of 1%. These observations indicate that irrespective of the [CPU](#) load, the execution time for a particular task is inconsistent and would only vary more with increased [CPU](#) load. Hence, there is a chance we capture more than the required amount of trace and introduce noise in the captured data making it a challenge to capture good-quality data trace.

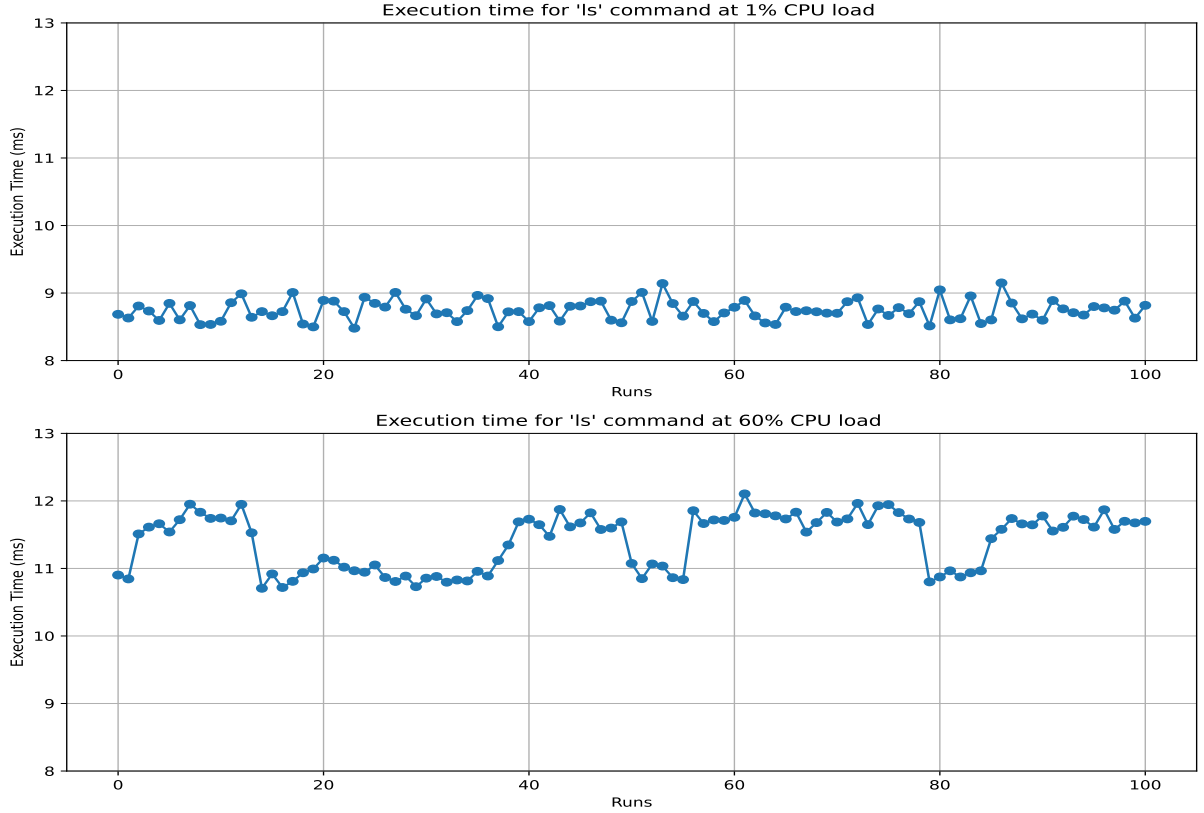


Figure 4.4: Execution Time of 'ls' [55] Command on RPi Development Board for 100 Runs at Low and High CPU Loads

Moving onto the second major reason mentioned in the first paragraph of this Subsection. Every input message performs a task of varying complexity. Based on the amount of system resources required a task could take longer or shorter time to execute. Figure 4.5 shows execution times of three commands: (1) 'ls' [55], (2) 'ps' [56] and (3) 'uname -a' [59] accepted by the Raspbian OS on the RPi development board. The region of power trace circled in green represents the execution time for the three commands and the difference is clearly visible. The 'ps' command's need to access extensive process information results in the longest execution time, while the 'ls' command's dependency on file metadata retrieval places it in the middle. The 'uname -a' command's straightforward nature and minimal system calls contribute to its quick execution. Similarly, every command requires

a different amount of time to finish executing, making it challenging to measure and document the execution time for each command accurately. This variability introduces further complexity when capturing the right portion of the power trace for analysis.

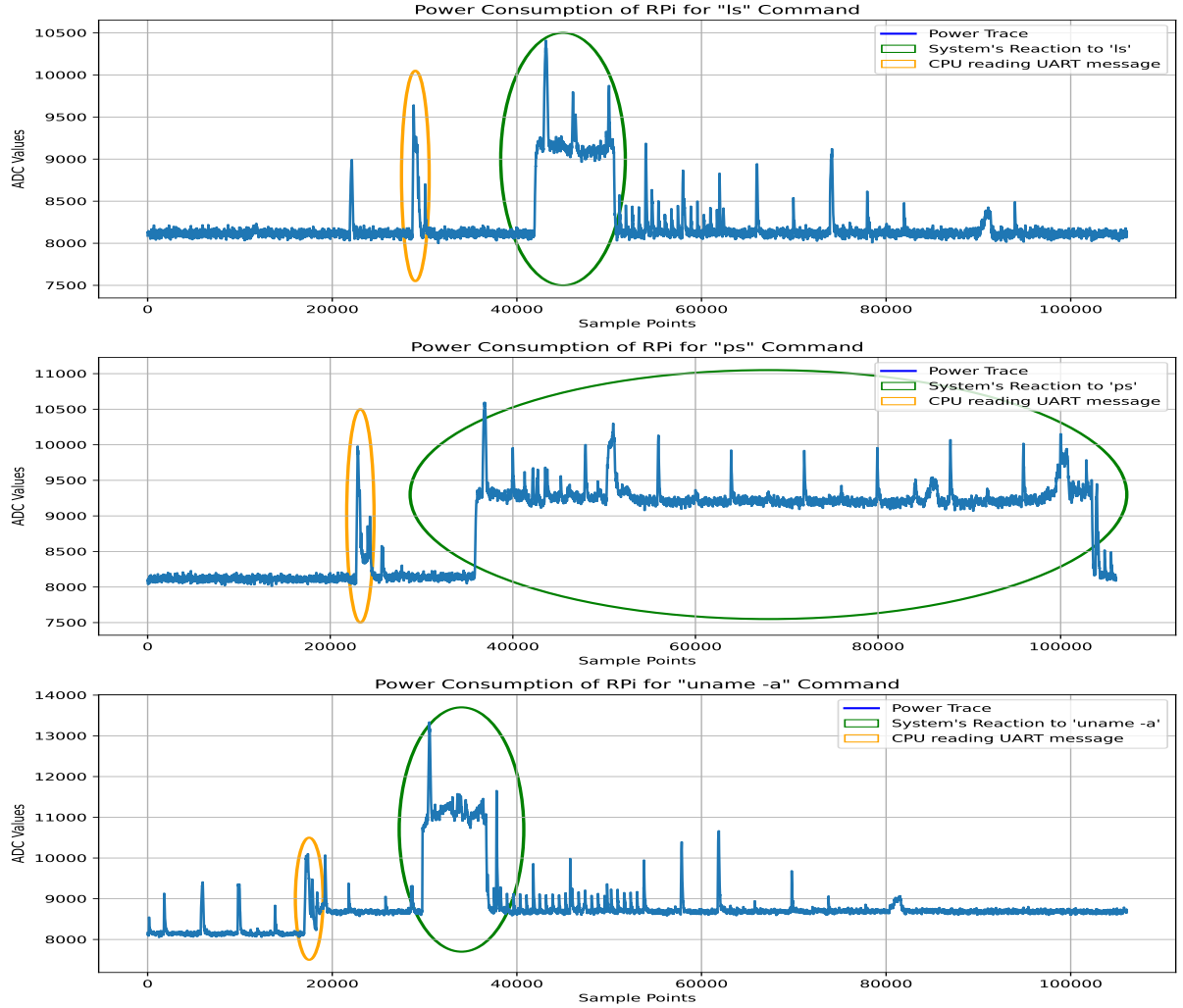


Figure 4.5: Power Consumption of RPi when subjected to 'ls' [55], 'ps' [56] and 'uname -a' [59] Commands

4.2.3 Capturing the Right Amount of Data

Capturing a single power trace can often yield noisy data, making it difficult to extract meaningful insights about the TS's behaviour. To solve this issue, one of the approaches is to collect multiple traces and average them to reduce the noise and highlight the relevant features. The question then arises: how many traces are necessary to achieve a reliable average? This Subsection explains why answering this question within the scope of power fuzzing for ES is challenging.

Signals consist of a persistent desired pattern or component and a variable undesired noise component. Averaging multiple traces helps to smooth out random noise and transient variations that can obscure the true power consumption patterns [21]. In mathematical terms, averaging multiple traces enhances the Signal-to-Noise Ratio (SNR). SNR measures the ratio of signal power to background noise power, typically expressed in decibels dB . A high SNR means the signal is clear and easily identifiable, while a low SNR means the signal is obscured by noise, making it challenging to discern or recover [96]. The total power P_t of a signal is an important feature for calculating SNR. We can calculate it by summing the absolute squares of its time-domain samples and dividing by the signal length or, equivalently, by squaring its Root Mean Square (RMS) level [63] and can be represented with Equation 4.1.

$$P_t = \frac{1}{M} \sum_{m=0}^{M-1} |s[m]|^2, \quad (4.1)$$

Where $S[m]$ represents the m^{th} sample point of the digitized signal, and M is the total number of sample points in the signal. For signals containing distinct components and Gaussian noise, the total power P_t equals the sum of the signal power P_s and the noise power P_n . After averaging N samples and accounting for noise reduction [21], the power P_{M-avg} is adjusted as described in the corresponding Equation 4.2.

$$P_{M-avg} = P_s + \frac{P_n}{N}, \quad (4.2)$$

Equation 4.2 indicates that averaging 'N' traces reduces the noise power (P_n) by a factor of 'N'. This implies that the more traces we average, the lower the total power (P_t) of the signal. Consequently, the overall SNR improves, allowing us to obtain a clearer and more accurate representation of the TS's power consumption patterns. From a statistical perspective, the law of large numbers [82] tells us that the more samples we collect, the closer our average will be to the true mean of the underlying distribution. However, there

is a point of diminishing returns where additional traces provide minimal improvement in the accuracy of the average. Determining the optimal number of traces involves balancing the benefits of noise reduction against the practical constraints of time and computational resources.

Figure 4.6 shows the trend in the total power of the power trace signal of the RPi versus the number of traces averaged. To obtain this trend, we subjected the RPi to the 'ls' [55] command 500 times continuously and stored the power traces for analysis. We repeated this process four times under the same environmental conditions at different times to verify if we could reproduce the same trend. We can observe the expected trend in the first capture: the total power decreases as we average more power traces. After averaging 190 traces, the improvement in the total power of the signal diminishes, with the trend showing a slight negative impact as we average more traces, indicating that collecting more traces is not beneficial. In the case of the fourth capture, we see a continuous decreasing trend as we approach 500 power traces. This indicates that collecting more traces might prove beneficial for improving signal quality.

The second and third captures don't follow the decreasing trend consistently. In the second capture, the trend is maintained until 80 traces, after which there is a sharp increase in total power for 100 averaged traces. Following this, we observe a short-period sawtooth trend. Similarly, in the third capture, a sharp increase occurs after only 30 traces, continuing until 60 traces, followed by a stretched sawtooth trend. This sharp increase in total power, followed by a sawtooth pattern, indicates one or more of the following: (1) noise dominated the majority of the power trace, obscuring the actual pattern; (2) the power traces shifted significantly on the x-axis and y-axis; (3) some power traces failed to capture the actual pattern. Consequently, many power traces collected during the second and third captures need to be discarded and cannot be used for averaging. The x-axis shift in power traces relates to the challenge mentioned in Subsection 4.2.1. Additionally, during the second and third captures, the Raspbian OS [36] on the RPi likely executed other background processes, heating the CPU and causing delays in executing the 'ls' [55] command. This increased the power consumption of the RPi development board, resulting in higher variations on the x and y axes of the power trace. Non-stationary signals exhibit time-varying statistical properties, causing their power to evolve over time. When averaging such signals, the total power does not remain constant or decrease predictably as with stationary signals; instead, it fluctuates as the intrinsic characteristics of the signal and noise vary. Due to inherent TS properties, this uncertainty in capturing high-quality power traces each time hinders our ability to determine the optimal number of traces to capture for averaging.

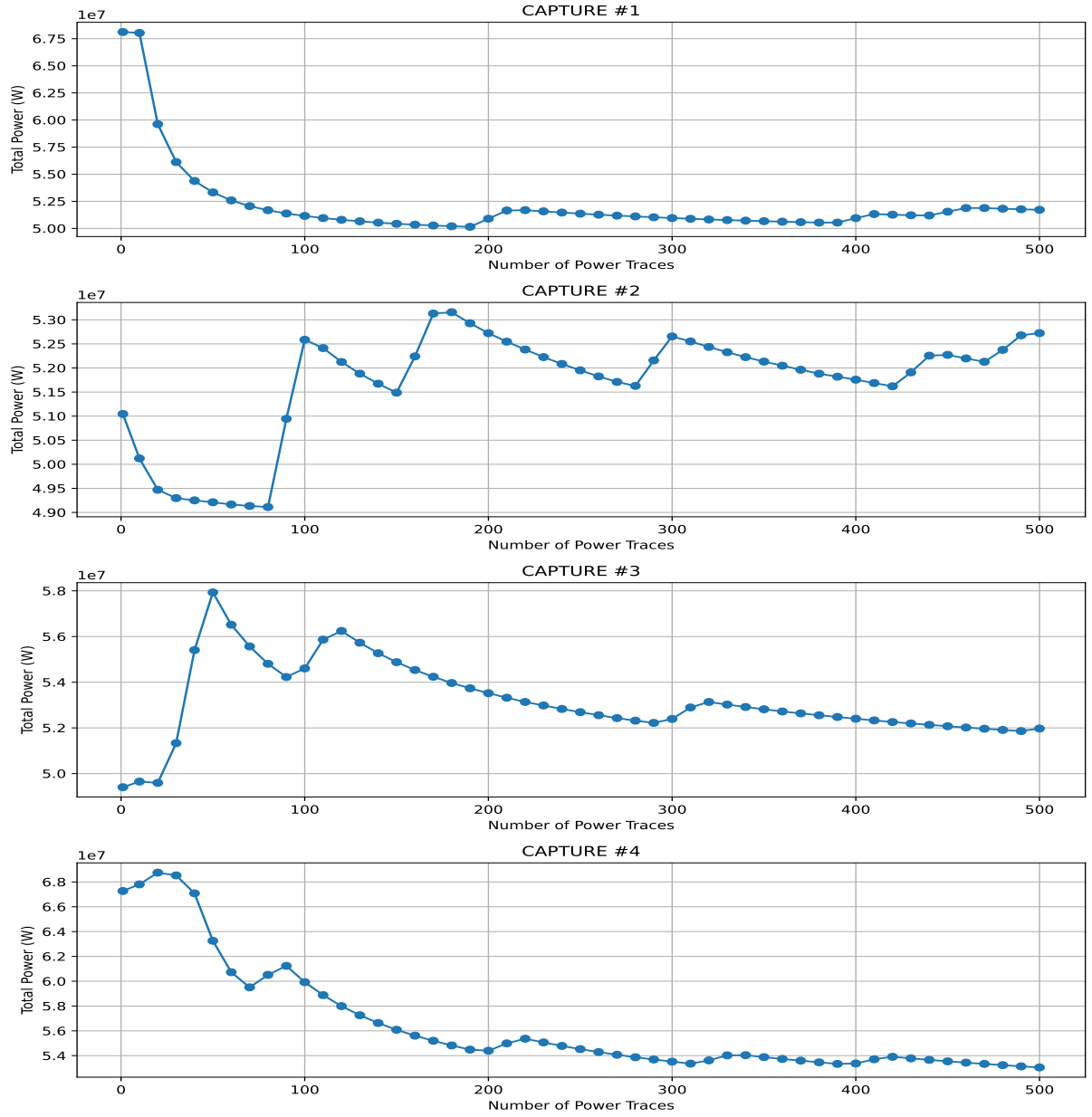


Figure 4.6: Trend in Total Power P_t When Averaging Multiple Traces for 4 Captures Conducted at Different Times

4.2.4 Capturing the Right Resolution of Data

The resolution of the captured data determines how finely we can observe the power consumption variations of the [TS](#). High-resolution data capture allows us to detect subtle changes in power consumption that may indicate specific operations or potential vulnerabilities. However, achieving the right resolution poses several challenges.

- **Hardware Limitation:** The bandwidth of the data acquisition system impacts the maximum sampling rate that can be used. Limited bandwidth can restrict the resolution of the captured data, potentially causing important details to be missed [\[89\]](#). The sampling rate is a key factor in determining the resolution of the captured data. A higher sampling rate can capture more details of the power consumption profile, but it requires more storage space and processing power [\[89\]](#).
- **Noisy Data:** In Subsection [4.2.3](#), we observed that noise in power trace signals could significantly degrade the quality of power traces, diminishing the effectiveness of noise reduction techniques such as averaging. High-resolution data is often more susceptible to noise, which can obscure meaningful patterns in the power trace [\[16\]](#). The [TS CPU](#) and [OS](#) operate with numerous background processes that introduce variability in power consumption. T. K. Tan et al. [\[95\]](#) provide a comprehensive analysis of the energy consumption in embedded [OS](#). They introduce a macro-modeling approach to quantify the energy overheads of various [OS](#) services and processes, highlighting the impact of context switching, timer interrupt handling, and [IPC](#) on overall power consumption. Capturing power traces at a high resolution amplifies these variations, making it difficult to isolate the relevant data. This issue is compounded by the fact that the sampling rate must be sufficiently high to capture the detailed power consumption patterns associated with specific operations or potential vulnerabilities. However, too high a sampling rate increases the likelihood of recording noise and undesired power spikes. Additionally, since the power consumption pattern is inconsistent every time, determining an effective sampling rate that balances capturing desired patterns and variations in the power trace while minimizing noise is challenging.
- **Unsynchronised Data:** Figure [4.2](#) illustrates the impact of variability ϵ_1 on the time required to execute [Event-1 \(\$E_1\$ \)](#) and [Event-2 \(\$E_2\$ \)](#). Due to this variability, the pattern corresponding to the actual execution of the input message shifts on the x-axis. This shift complicates the determination of an optimal sampling rate for data capture. Selecting a very low sampling rate can result in capturing a significant

amount of noise and undesirable sections of the power trace after the message execution. Conversely, selecting a very high sampling rate might only capture the section of the power trace preceding the actual execution, missing critical information.

Figure 4.7 illustrates the importance of selecting the appropriate sampling rate for data capture in power fuzzing. The maximum number of sample points we can capture using the digitizer in our experimental setup is 100,000. When we capture the original power trace at 1 MHz for 100,000 sample points, it represents a total time of 100 ms, allowing us to clearly observe RPi's reaction to the 'ls' [55] command. However, this also results in capturing many noisy sections of the power trace after executing the command. If we increase the sampling frequency to 1.25 MHz for 100,000 sample points, representing a total time of 80 ms, we can reduce some of the noisy parts. Further increasing the sampling frequency to 2 MHz, representing a total time of 50 ms, allows us to partially capture RPi's reaction to the 'ls' [55] command while also capturing significant noise that reflects the state of the RPi before sending the message. Lastly, capturing the power trace at a sampling frequency of 5 MHz for 100,000 sample points, representing a total time period of 20 ms, results in capturing only noise without any meaningful reaction of the RPi to the 'ls' [55] command. In summary, capturing the right data resolution in power fuzzing involves balancing the need for detailed power traces with the limitations of the data acquisition system. An optimal sampling rate ensures that we capture the essential patterns in the power trace, reflecting the TS's reaction to the input message without overwhelming the data with irrelevant noise.

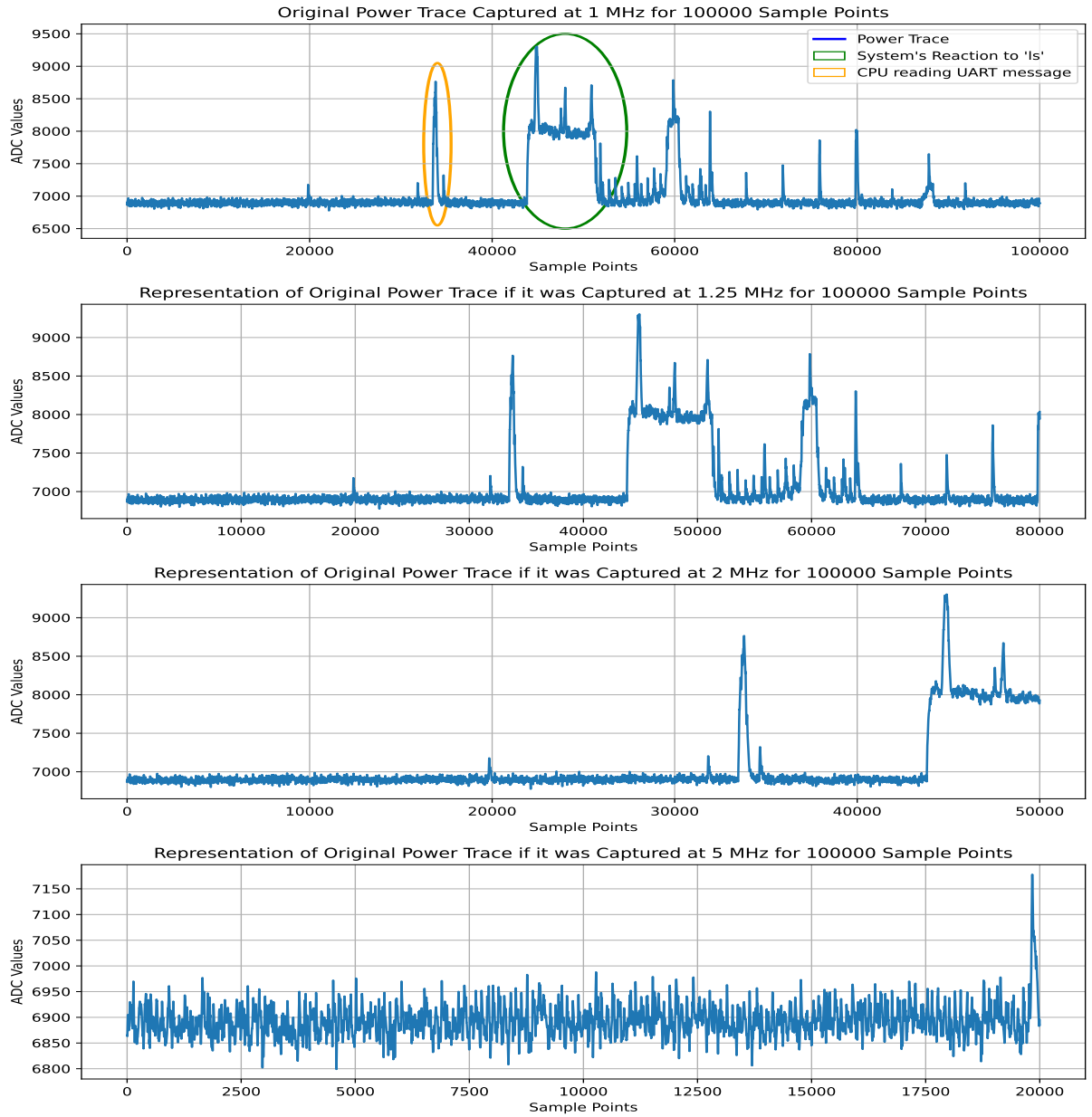


Figure 4.7: Representation of Original Power Trace If It Was Captured at Different Sampling Frequencies. The Original Power Trace Is Power Consumption of RPi When Subjected to the 'ls' [55] Command

4.3 Variable Target Properties

The intrinsic properties of the [TS](#) can greatly influence the power consumption patterns and subsequently affect the accuracy of the power trace analysis. Factors such as the [TS](#)'s hardware characteristics, operational state, and environmental conditions all contribute to variability in power traces. Understanding and accounting for these variable [TS](#) properties is important for obtaining reliable and meaningful data during power fuzzing. This section delves into the various target properties that introduce variability and explores how they impact the overall effectiveness of the power fuzzing process.

4.3.1 Background Processes

A significant source of variability in power consumption is the background processes running on the [TS](#). Background processes introduce noise into the power traces, complicating the analysis and making it difficult to isolate the power consumption pattern associated with the specific task being fuzzed. Even when the [TS](#) appears idle, these background processes can consume [CPU](#) cycles and power, leading to visible and significant fluctuations in the power trace data as seen previously in [Figure 4.1](#). These processes are essential for the system's basic functioning and cannot be avoided. The key roles played by the background processes [\[15\]](#), along with their descriptions, have been listed in [Table ??](#); these include tasks such as system management, peripheral control, and event handling. Except

Key Roles	Description
System monitoring	Monitor system health, resource usage, and performance metrics.
Communication	Manage network connections, handle data transmission, or sync data with remote servers
Scheduling	Schedule and execute periodic tasks
Event handling	Listen for and respond to system events or external triggers
Logging	Handle system logging and error reporting
Device Driver Management	Manage hardware interfaces and device drivers.

Table 4.4: Key Roles played by Background Processes in an [ES](#)

for logging and device driver management, all the tasks listed in [Table 4.4](#) are high-priority

tasks. The [TS](#) software can switch to executing these high-priority tasks even in the middle of running the current task. This means that while the system processes the fuzzing input message, a higher-priority task might preempt it, resulting in an incorrect power consumption pattern. This preemption can significantly distort the power trace, leading to inaccurate analysis and unreliable results in the fuzzing process.

Moreover, the timing and frequency of background processes are often unpredictable. Based on their frequency, they can be divided into two categories: (1) periodic and (2) aperiodic. Periodic background processes occur at regular intervals, making them predictable and relatively easier to manage when processing power trace data. In contrast, aperiodic background processes occur at random intervals, posing a greater challenge due to their unpredictability. This unpredictability complicates the isolation of relevant power consumption patterns, leading to difficulties in accurately analyzing the data.

Finally, from a fuzzing perspective, background processes might be crucial. The primary goal of a fuzzing experiment is to detect vulnerabilities and bugs by subjecting the [TS](#) to a large number of random, malformed, or semi-random inputs. Important and less important background processes might be linked to our fuzzing inputs, and we might only discover vulnerabilities or bugs when these processes are running. While background processes pose a challenge in terms of power consumption variability, they are beneficial for comprehensive fuzzing. This duality makes it difficult to decide whether to disable background processes during fuzzing experiments. Therefore, balancing the inclusion of background processes is crucial.

4.3.2 Input Message Properties

Input messages play a critical role in power fuzzing experiments, as their properties can significantly impact the effectiveness and accuracy of the analysis. In Subsection [3.2.1](#), we examined how the input message's grammar can vary based on the type and complexity of the [TS](#) it serves. For a fuzzing experiment to be effective, we must fuzz the entire grammar space of the input message. This comprehensive approach is essential to detect all possible vulnerabilities and bugs in the [TS](#). This Subsection explores two key challenges related to our ability to fuzz the entire grammar space of the input: (1) the computational load of messages and (2) the limitations posed by certain commands.

Computationally Light Messages

One major challenge arises when the input messages are not computationally heavy. When a message does not require substantial processing power, the resulting power consumption pattern might be indistinguishable from the background noise. This lack of a distinguishable pattern makes it difficult to detect bugs or vulnerabilities in the TS. Figure 4.8 shows the power consumption pattern of RPi when it is subjected to two computationally light commands (1) ‘echo’ [53] and (2) ‘pwd’ [57]. The power traces were collected for 100,000 sample points at a frequency of 1 MHz, representing a total time period of 100 ms.

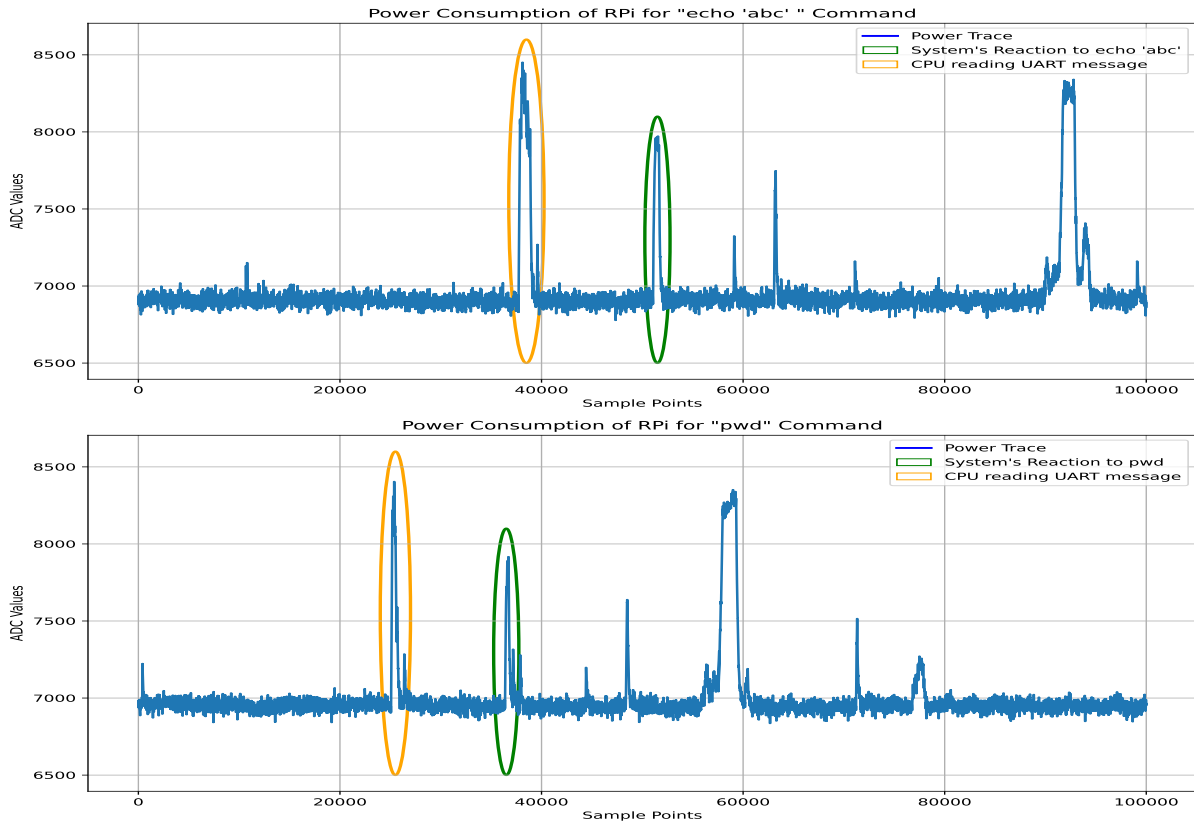


Figure 4.8: Power consumption Pattern for RPi when Subjected to (1)“echo ‘abc’ ” [53] Command and (2) ‘pwd’ [57] Command

We can see that the execution times for the ‘echo’ [53] and ‘pwd’ [57] commands, highlighted in green within the power trace, are very short and lack any distinguishable pattern. These traces closely resemble background noise spikes. In contrast, the spike caused by background noise and the CPU reading the message from UART, highlighted in orange, exhibits a more significant pattern. This undesirable pattern can negatively impact the averaging of traces, potentially leading to false classification of the power traces.

Blocking Commands

Another challenge is the inability to test commands that block the communication channel between the fuzzing setup and the TS. In our experimental setup, we send commands to the RPi development board using UART, which is a critical interface for sending and receiving messages between the fuzzing setup and RPi. However, certain commands can interfere with the UART interface, preventing further communication until they are fully executed or timed out. This blockage disrupts the continuous flow of messages necessary for effective fuzzing and data capture.

For instance, on a RPi running Raspbian OS [36], the command “cat /dev/ttyS0” [52] can block the UART communication. This command continuously reads from the specified UART port (ttyS0 in this case) and does not terminate until manually interrupted. As a result, any subsequent attempts to send messages through the same UART port would fail, as the port remains occupied by the blocking command. This situation disrupts the power fuzzing process by preventing new inputs from being delivered to the TS, thereby hindering the detection of vulnerabilities that require a sequence of varied inputs. Another example is a command like “find / -type f” [54]. This command searches for all files in the filesystem. Depending on the number of files and the filesystem’s size, this process can take a considerable amount of time to complete. Because the command takes so long to execute, the interval at which new commands can be sent is significantly extended. This delay reduces the throughput of the fuzzing process, making it less efficient. Furthermore, in a typical fuzzing experiment, it is necessary to collect many power traces of the same command to average out noise and identify consistent patterns. When a command takes an excessively long time to execute, the total time required to collect a sufficient number of traces becomes impractically long. This limits the feasibility of testing such commands and reduces the overall effectiveness of the fuzzing experiment.

4.3.3 Performance Variability Over Extended Runtime

The performance of a [TS](#) can degrade significantly over extended periods of runtime, posing substantial challenges for power fuzzing. This performance degradation can affect the accuracy of power trace analysis and the overall effectiveness of the fuzzing process.

The first major reason for this is the unavoidable heating of [CPU](#). As the [TS](#) executes a continuous stream of commands, the [CPU](#) generates heat, which can alter its power consumption characteristics and obscure the patterns we aim to capture and analyze. When the [CPU](#) temperature rises, the [TS](#)'s thermal management mechanisms may throttle the [CPU](#) to prevent overheating. This throttling results in inconsistent execution times and power consumption patterns, complicating the correlation between power traces and specific commands. Depending on the [CPU](#)'s temperature, the same command might exhibit different power consumption patterns, making it difficult to identify genuine vulnerabilities or anomalies.

Hardware variability is another factor that influences the power fuzzing process. Different units of the same model of the [TS](#) can exhibit variations in power consumption due to manufacturing differences, component tolerances, and aging effects. These variations can affect the reproducibility and consistency of power trace data. Aging effects also contribute to hardware variability. Over time, components such as capacitors, resistors, and integrated circuits degrade, leading to changes in their electrical characteristics. This degradation can cause shifts in the power consumption patterns of the [TS](#), making it difficult to compare power traces collected at different times.

4.4 Lack of Standard Analysis Methods

This section dives into the challenges in power fuzzing architecture due to the absence of standardized analysis methods for interpreting power traces. This lack of standardization hinders the consistency, reproducibility, and comparability of results across different studies and experiments, ultimately affecting the reliability of power fuzzing as a method for identifying vulnerabilities in [ES](#).

Power traces are a type of time series data, capturing the power consumption of a [TS](#) over time. Each power trace consists of numerous data points collected at specific intervals, representing the electrical power usage at each moment. In theory, time series analysis methods should be applicable to power traces to identify patterns, trends, and anomalies. However, traditional time series analysis techniques often fall short when

applied to power traces due to several unique challenges inherent in power consumption data. Traditional time series analysis methods, such as [AutoRegressive Integrated Moving Average \(ARIMA\)](#) models [87], seasonal decomposition [19], and exponential smoothing [31], are designed to handle relatively clean and stationary data with predictable patterns. Power traces, however, are typically noisy, non-stationary, and exhibit complex patterns influenced by various factors such as the system’s workload, background processes, and environmental conditions [62]. These characteristics make it difficult to apply standard time series techniques effectively. Power traces also often contain high-frequency noise, transient spikes, and abrupt changes that are not easily modelled by conventional time series approaches. The variability in power consumption due to hardware and software interactions adds another layer of complexity.

Power trace analysis involves examining the power consumption patterns of a [TS](#) to identify anomalies and potential vulnerabilities. However, researcher’s and practitioners’ methodologies for analyzing these power traces vary widely. These inconsistencies can stem from differences in experimental setups, data acquisition techniques, signal processing methods, and analytical tools used. Some studies may use simple statistical methods to identify deviations in power consumption, while others might employ more sophisticated techniques such as machine learning algorithms [43] or signal processing methods like wavelet transforms and Fourier analysis [40]. The variability in these approaches makes it difficult to establish a common framework for power trace analysis, leading to disparate results that are hard to compare or reproduce.

The lack of standardized methods also affects the efficacy of power fuzzing. Without a unified approach to analyzing power traces, it becomes challenging to benchmark the performance of different fuzzing techniques or to validate the findings of a particular study. Inconsistent preprocessing and filtering of power trace data can result in the misinterpretation of noise as meaningful signals or the overlooking of subtle anomalies that indicate genuine vulnerabilities. Consequently, this undermines the confidence in power fuzzing results and limits its adoption as a robust security testing technique.

Chapter 5

Proposed Enhancements

Chapter 4 detailed the various limitations and challenges associated with the traditional power fuzzing architecture. The current architecture restricts the ability to capture high-quality data, and the inherent properties of the TS augments these issues. While we cannot gain significant control over the inherent properties of the TS and have to treat such variations as noise, we can, however, address some of the challenges related to data capture. This chapter aims to present two approaches to mitigate these issues, thereby enhancing the overall effectiveness of power fuzzing. By focusing on improving data capture techniques, we can better isolate meaningful patterns from noise, leading to more accurate and reliable fuzzing results.

5.1 Hardware Trigger

Figure 3.2 depicts the various events occurring in a power fuzzing architecture. In the context of ES fuzzing, we focus on the time duration D_5 occurring between Event-5 (E_5) and Event-6 (E_6), as this represents the execution phase of a command. Variations during this phase indicate potential vulnerabilities or bugs. Currently, as shown in Figure 4.1, the power trace captures variations in the power consumption of the TS during Event-1 (E_1), Event-2 (E_2), Event-3 (E_3), and Event-4 (E_4), leading to a total time variability $\epsilon_T = \epsilon_1 + \epsilon_2 + \epsilon_3 + \epsilon_4 + \epsilon_5$. This results in the collection of noisy and shifted power traces. The time duration D_4 required for Event-4 (E_4) to complete is influenced by the hardware and software limitations of the TS, and the variability ϵ_4 remains beyond our control. However, we can mitigate other timing variabilities by precisely adjusting the instance at which we start collecting the power traces. This approach will help isolate the

execution phase more effectively, reducing noise and enhancing the quality of the captured power data.

The proposed approach involves a hardware setup that monitors communication between the communication module and the **TS**, capturing activity between **Event-3** (E_3) and **Event-4** (E_4). This monitoring is non-invasive and does not interfere with actual communication. Figure 5.1 shows the placement of the hardware setup in the current power fuzzing architecture. Widely used communication protocols in **ES**, such as **UART**, **SPI**, **I2C**, **CAN**, **USB**, and Ethernet, are all serial protocols, meaning they transfer data one bit at a time [92, 39]. Each protocol employs a specific format that generally includes a start bit, stop bit, parity bit, and the message itself. Based on this structure, the hardware setup implements a simple inference logic as mentioned in Algorithm 1. When the hard-

Algorithm 1 Inference Logic

```

Comm = Communication_Channel()
Trig = Trigger()
while True do
    Msg = Comm.Read_Bit()
    if Msg.End_Of_Message() == True then
        Trig.send()
    else
        Pass
    end if
end while

```

ware setup detects the **End of Message** (EoM), it indicates that the **TS** has also received it, barring any environmental losses. Upon receiving the **EoM**, the **TS**'s only remaining task is to react to the message. At this point, the hardware setup sends a trigger to the **PCH** to begin collecting the power trace. The total variability in this trace is now $\epsilon_T = \epsilon_4 + \epsilon_5$, eliminating the variability $\epsilon_1 + \epsilon_2 + \epsilon_3$ from the trace by not capturing power data during **Event-1** (E_1), **Event-2** (E_2), and **Event-3** (E_3). This method significantly enhances the quality of the power trace by ensuring that the efficiency of the workstation in generating fuzzing inputs and the communication hardware in translating and sending messages do not affect the trace. Furthermore, this approach is protocol-agnostic, as it only collects power traces at the **EoM**, regardless of the communication protocol's specific message structure and timing.

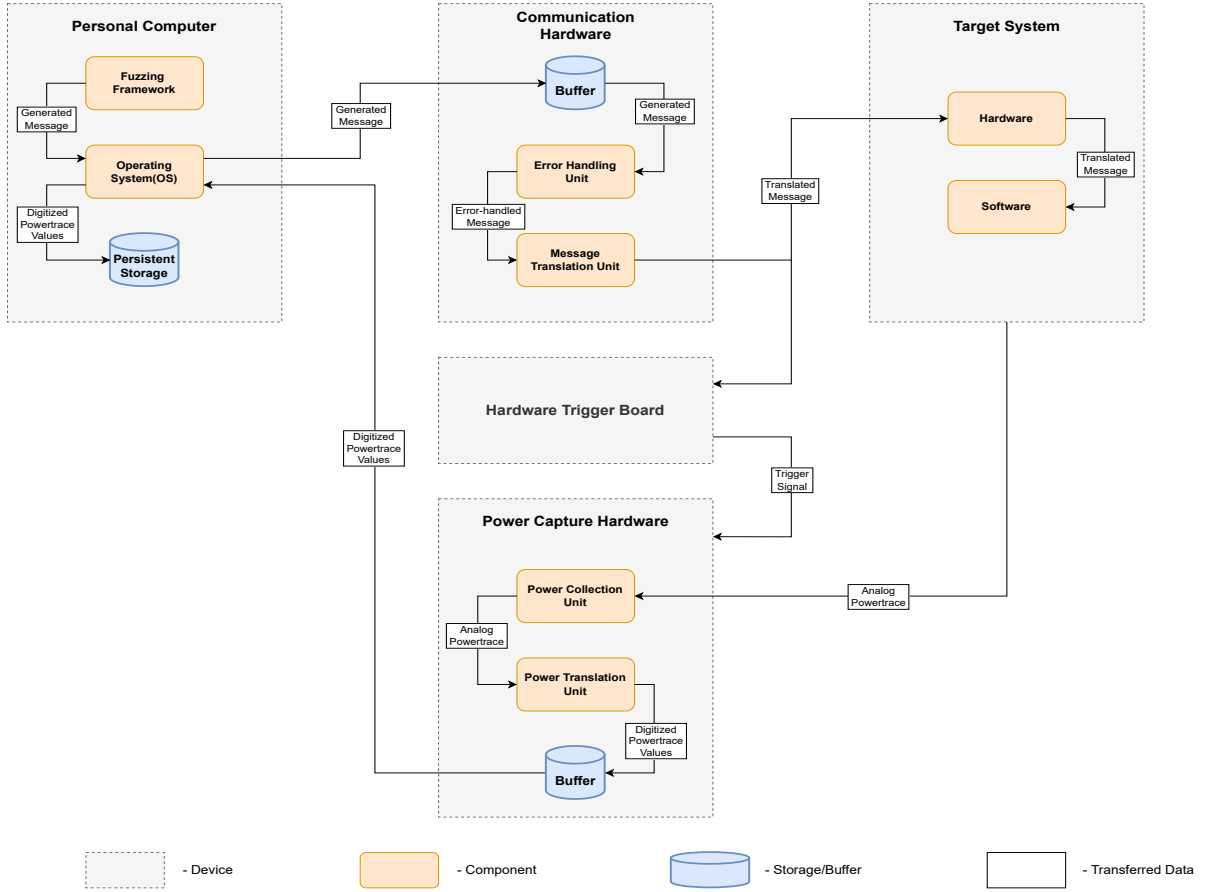


Figure 5.1: Improved Functional View of Power Fuzzing Architecture

This approach effectively resolves the challenge of [Capturing Data at the Right Instant](#) by aligning power traces precisely, thus eliminating the need for extensive shift correction before analysis. This precision enables the use of various pattern recognition techniques to isolate relevant portions of the trace, which enhances both the quality and reliability of the data collected. As a result, this method substantially improves the accuracy and efficiency of subsequent analyses in [ES](#) fuzzing, providing a clearer view of potential vulnerabilities and system behaviours.

5.2 Profile and Fine-Tune (PnFT) Approach

The hardware trigger effectively addresses the challenge of data capture, reducing dependency on the limitations of data generation and translation process. However, it does not solve the issues of capturing the right amount and resolution of data. These challenges occur due to the variable target properties beyond our control. To tackle the challenge of [Capturing the Right Amount of Data](#), one might consider various signal processing techniques, such as spectral analysis and filtering, to distinguish between useful and irrelevant traces. However, these methods typically require parameter settings that depend on extensive domain knowledge, which we lack regarding our [TS](#) [71, 4]. This knowledge gap also limits our ability to determine the optimal resolution for power trace data capture, further complicating the analysis and potentially obscuring significant findings.

To effectively address these challenges, an informed approach to fuzzing [ES](#) is important. Attempting to fuzz a [TS](#) without carefully adjusting parameters such as power trace capture frequency, resolution, and duration can yield inaccurate interpretations and flawed analyses. Therefore, adopting a methodical approach is essential to enhance the effectiveness of [ES](#) fuzzing processes significantly. The [PnFT](#) approach serves this purpose, providing a structured framework for improved fuzzing outcomes. The steps involved in this approach are outlined below:

1. **Profile the Target:** First, we conduct a thorough analysis of the [ES](#) to understand its architecture, functionality, and constraints. This involves identifying communication protocols and hardware and software specifications. Profiling enables us to gather essential information that informs the next phases, ensuring that fuzzing is targeted and effective. The key aspects of profiling are mentioned below:
 - **Firmware Analysis:** By examining the firmware, we can identify potential entry points for fuzzing and understand the software structure. We can understand the file format and any headers to grasp the structure and memory layout. Explore available documentation to understand the system’s expected behaviour. If possible, analyze the code (source or disassembled) to identify functions for targeted fuzzing. Finally, examine how the firmware interacts with peripherals and determine their expected input ranges.
 - **Communication Protocols:** [ES](#) rarely operate in isolation. They interact with sensors, actuators, and other devices to perform their tasks. Understanding how these conversations happen is vital for effective fuzzing. Communication

protocols define the language used by the [ES](#) to exchange information with external components. By identifying the protocols used by the [ES](#), we can craft fuzzing inputs that mimic real-world communication scenarios. This allows us to test the system’s ability to handle unexpected data or invalid messages that might be encountered during normal operation, potentially exposing vulnerabilities.

- **Hardware Characteristics:** [ES](#) are resource-constrained environments. Unlike a powerful desktop computer, they often have limited memory, processing power, and specific hardware components tailored for their function. It is important to understand the hardware characteristics of the [ES](#). This includes investigating the [CPU](#), [RAM](#) and other processing components on-board to set expectations before fuzzing and tailor our fuzzing approach. By understanding these limitations, we can prevent our fuzzing attempts from overwhelming the system’s resources or causing unintended hardware interactions that might damage the device.

Gathering complete information about a [TS](#) can be challenging due to factors such as prior research, proprietary constraints, and available online documentation. However, by collecting even small pieces of information and systematically connecting them, one can gain sufficient insights to make educated guesses about the [TS](#)’s functioning. This method enables a deeper understanding of the system, facilitating a more informed and effective approach to fuzzing. By leveraging available data and contextual clues, we can optimize the analysis and testing strategies.

2. **Fine-Tune the Setup:** After profiling the [TS](#), it is essential to tailor the fuzzing parameters and environment to align with the specific characteristics and limitations of the [ES](#). By carefully adjusting these components, we can enhance the precision of fuzzing, improving the likelihood of uncovering subtle vulnerabilities that may otherwise remain hidden. We can consider the below-mentioned aspect to fine-tune the fuzzing setup:

- **Adjust Input Generation:** Based on the profiling data, we can customize the input generation to target specific functions or vulnerabilities in the firmware. Use formats and patterns that reflect the target’s communication protocols and expected input structures.
- **Set Fuzzing Parameters:** We can optimize parameters such as mutation rate, input size, and fuzzing speed to suit the [ES](#)’s hardware limitations. This ensures

the system remains stable during fuzzing while maximizing the likelihood of triggering vulnerabilities.

- **Configure Power Trace Capture:** We can determine the appropriate sampling frequency and resolution for power traces based on the system's processing capabilities. Use insights from profiling to decide the duration and timing of power trace captures, focusing on desirable execution phases.

Each [TS](#) presents unique characteristics, requiring a tailored approach to fine-tuning the fuzzing setup. Based on the information gathered during the profiling phase, it's essential to iteratively adjust multiple parameters to identify the optimal combination for the specific [TS](#) under test. This iterative process ensures that the fuzzing parameters and settings are well-aligned with the system's architecture and vulnerabilities. Continuous refinement enhances the effectiveness of the fuzzing process, ultimately leading to more accurate detection of potential security flaws.

Chapter 6

Conclusion

In this thesis, we explored the internal workings of power fuzzing architecture, focusing on its core components and the sequence of events that shape its operation. By analyzing these events, we identified significant variabilities that can impact the efficacy of fuzzing processes. These variabilities occur due to the various factors associated with the different devices in the power fuzzing architecture. They highlight the challenges inherent in capturing precise and accurate power trace data, which is crucial for effective vulnerability detection in [ES](#).

One of the primary challenges discussed was the difficulty in data capture due to the limitations posed by the computing platform and the [TS](#). Hardware and software characteristics inherent to the workstation used for fuzzing and the communication hardware that translates the messages can hinder our ability to assess the operational efficiency of the fuzzing process accurately. These limitations can introduce potential delays within the power fuzzing architecture, complicating the ideal timeframe for data capture and the optimal data capture duration. This ambiguity creates uncertainty regarding the most opportune moment to capture data and the appropriate length of the data capture window. Furthermore, the [TS](#)'s unknown properties contribute to the challenge of capturing the most relevant data. The lack of enough knowledge about the [TS](#)'s characteristics hinders the selection of appropriate data capture parameters, potentially resulting in the capture of insufficient or irrelevant data.

Another challenge discussed was the fluctuating nature of [TS](#) properties. These properties, often unpredictable and uncontrollable, introduce noise that can obscure meaningful results. This noise complicates the interpretation of power traces, making it challenging to distinguish between genuine vulnerabilities and mere anomalies in data. Furthermore, the

current fuzzing space is hampered by the lack of standardized analysis methods. This absence of a uniform approach hinders the consistency, reproducibility and comparability of results across various studies and experiments. The inconsistency introduced by disparate analysis techniques ultimately undermines the reliability of power fuzzing as a method for uncovering vulnerabilities within [ES](#).

To address these challenges, we proposed two enhancements to the traditional power fuzzing framework. The first enhancement involved the use of a hardware trigger mechanism designed to synchronize data capture with specific events in the [TS](#). This mechanism aims to reduce the variability in power traces, improving data quality and the overall reliability of the fuzzing process.

The second enhancement focused on optimizing the duration and resolution of data capture. By implementing a tailored approach [PnFT](#) that considers the unique characteristics of each [TS](#), we can fine-tune the fuzzing setup to match the system’s behaviour better. This methodology not only enhances the precision of vulnerability detection but also streamlines the fuzzing process, ensuring that the insights gained are both relevant and actionable. Together, these enhancements form a robust framework for advancing the field of power fuzzing in [ES](#).

References

- [1] FreeRTOS—Market Leading RTOS (Real Time Operating System) for Embedded Systems With Internet of Things Extensions. <https://www.freertos.org>.
- [2] L. Abeni, A. Goel, C. Krasic, J. Snow, and J. Walpole. A Measurement-Based Analysis of the Real-Time Performance of Linux. In *Proceedings. Eighth IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 133–142, 2002.
- [3] Meraj Alam. AT Commands. *International Journal of Scientific and Research Publications*, 5(5), 2015.
- [4] Andreas Antoniou. *Digital Signal Processing*. McGraw-Hill, 2016.
- [5] Aticleworld. Difference between UART and USART(UART vs USART), 2024. URL https://aticleworld.com/difference-between-uart-and-usb/#google_vignette.
- [6] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 2003. ISBN 978-0-321-15495-8. Google-Books-ID: mdiIu8Kk1WMC.
- [7] K. Baynes, C. Collins, E. Fiterman, Brinda Ganesh, P. Kohout, C. Smit, T. Zhang, and B. Jacob. The Performance and Energy Consumption of Embedded Real-Time Operating Systems. *IEEE Transactions on Computers*, 52(11):1454–1469, 2003.
- [8] Georg T. Becker, Daehyun Strobel, Christof Paar, and Wayne Burleson. Detecting Software Theft in Embedded Systems: A Side-Channel Approach. *IEEE Transactions on Information Forensics and Security*, 7(4):1144–1154, 2012.
- [9] Pete Beckman, Kamil Iskra, Kazutomo Yoshii, Susan Coghlan, and Aroon Nataraj. Benchmarking the Effects of Operating System Interference on Extreme-Scale Parallel Machines. *Cluster Computing*, 11:3–16, 2008.

- [10] Alexander Beifuß, Daniel Raumer, Paul Emmerich, Torsten M. Runge, Florian Wohlfart, Bernd E. Wolfinger, and Georg Carle. A Study of Networking Software Induced Latency. In *2015 International Conference and Workshops on Networked Systems (NetSys)*, pages 1–8, 2015.
- [11] Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier. Finding Software Vulnerabilities by Smart Fuzzing. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 427–430. IEEE, 2011.
- [12] Théo Benoit-Cattin, Delia Velasco-Montero, and Jorge Fernández-Berni. Impact of Thermal Throttling on Long-Term Visual Inference in a CPU-Based Edge Device. *Electronics*, 9(12):2106, 2020.
- [13] Boardcom. BCM2837B0 Datasheet. <https://pdf1.alldatasheet.com/datasheet-pdf/view/1572344/BOARDCOM/BCM2837B0.html>.
- [14] C Botezatu, Ionut Condrea, Bogdan Oroian, Adelina Hrițuc, Mihaela Ețcu, and Laurentiu Slătineanu. Use of the Ishikawa Diagram in the Investigation of Some Industrial Processes. In *IOP Conference Series: Materials Science and Engineering*, volume 682, page 012012. IOP Publishing, 2019.
- [15] Kelly C Bourne. *Application Administrators Handbook: Installing, Updating and Troubleshooting Software*. Newnes, 2013.
- [16] C Byrne. Noise in High-Resolution Signal Processing. *posted on Research-Gate July, 20:2021*, 2021.
- [17] Xiaomin Cai, Renfa Li, Shijie Kuang, and Jinhui Tan. An Energy Trace Compression Method for Differential Power Analysis Attack. *IEEE Access*, 8:89084–89092, 2020.
- [18] Chen Chen, Baojiang Cui, Jinxin Ma, Runpu Wu, Jianchao Guo, and Wenqian Liu. A systematic review of fuzzing techniques. *Computers & Security*, 75:118–137, 2018.
- [19] Robert B Cleveland, William S Cleveland, Jean E McRae, Irma Terpenning, et al. STL: A Seasonal-Trend Decomposition. *J. Off. Stat*, 6(1):3–73, 1990.
- [20] Brian Dempsey. Fundamentals of Current Measurement Part 1: Current Sense Resistors, 2021. Accessed: 2024-06-25.
- [21] John Dempster. Chapter Six - Signal Analysis and Measurement. In John Dempster, editor, *The Laboratory Computer*, Biological Techniques Series, pages 136–171. Academic Press, London, 2001.

- [22] L. Dobrica and E. Niemela. A Survey on Software Architecture Analysis Methods. *IEEE Transactions on Software Engineering*, 28(7):638–653, July 2002. ISSN 1939-3520. doi: 10.1109/TSE.2002.1019479. Conference Name: IEEE Transactions on Software Engineering.
- [23] Murray Dunne and Sebastian Fischmeister. Powertrace-based Fuzzing of CAN Connected Hardware. In *2022 IEEE International Conference on Cyber Security and Resilience (CSR)*, pages 239–244, 2022.
- [24] Maialen Eceiza, Jose Luis Flores, and Mikel Iturbe. Fuzzing the Internet of Things: A Review on the Techniques and Challenges for Efficient Vulnerability Discovery in Embedded Systems. *IEEE Internet of Things Journal*, 8(13):10390–10411, 2021.
- [25] Max Eisele, Marcello Maugeri, Rachna Shriwas, Christopher Huth, and Giampaolo Bella. Embedded fuzzing: A Review of Challenges, Tools, and Solutions. *Cybersecurity*, 5(1):18, 2022.
- [26] W. Elmenreich and M. Delvai. Time-Triggered Communication with UARTs. In *4th IEEE International Workshop on Factory Communication Systems*, pages 97–104, 2002.
- [27] Mohammad Farsi, Karl Ratcliff, and Manuel Barbosa. An Overview of Controller Area Network. *Computing & Control Engineering Journal*, 10(3):113–120, 1999.
- [28] Xiaotao Feng, Ruoxi Sun, Xiaogang Zhu, Minhui Xue, Sheng Wen, Dongxi Liu, Surya Nepal, and Yang Xiang. Snipuzz: Black-box Fuzzing of IoT Firmware via Message Snippet Inference. In *Proceedings of the 2021 ACM SIGSAC conference on computer and communications security*, pages 337–350, 2021.
- [29] Future Technology Devices International Limited (FTDI). FT201XS USB to I2C Bridge. <https://ftdichip.com/products/ft201xs/>.
- [30] Hasindu Gamaarachchi and Harsha Ganegoda. Power Analysis Based Side Channel Attack. *arXiv preprint arXiv:1801.00932*, 2018.
- [31] Everette S Gardner Jr. Exponential Smoothing: The State of the Art. *Journal of forecasting*, 4(1):1–28, 1985.
- [32] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based Whitebox Fuzzing. In *Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation*, pages 206–215, 2008.

- [33] Hackenberg, Daniel and Ilsche, Thomas and Schöne, Robert and Molka, Daniel and Schmidt, Maik and Nagel, Wolfgang E. Power Measurement Techniques on Standard Compute Nodes: A Quantitative Comparison. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 194–204, 2013.
- [34] Matthew Helmke. *Ubuntu Unleashed*. Sams publishing, 2012.
- [35] Thomas A Henzinger and Joseph Sifakis. The Embedded Systems Design Challenge. In *International Symposium on Formal Methods*, pages 1–15. Springer, 2006.
- [36] Gerry Howser and Gerry Howser. Raspberry Pi Operating System. *Computer Networks and the Internet: A Hands-On Approach*, pages 119–149, 2020.
- [37] Yuanwen Huang, Swarup Bhunia, and Prabhat Mishra. Scalable Test Generation for Trojan Detection Using Side Channel Analysis. *IEEE Transactions on Information Forensics and Security*, 13(11):2746–2760, 2018.
- [38] Secure Ideas. Hardware Hacking: Interfacing to UART with Your Computer, 2024. URL <https://www.secureideas.com/blog/hardware-hacking-interfacing-to-uart-with-your-computer>.
- [39] Mirko Jakovljevic and Astrit Ademaj. Ethernet Protocol Services for Critical Embedded Systems Applications. In *29th Digital Avionics Systems Conference*, pages 5–B. IEEE, 2010.
- [40] Chengbin Jin and Yongbin Zhou. Enhancing Non-Profiled Side-Channel Attacks by Time-Frequency Analysis. *Cybersecurity*, 6(1):15, 2023.
- [41] Ismet Burak Kadron, Nicolás Rosner, and Tevfik Bultan. Feedback-Driven Side-Channel Analysis for Networked Applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 260–271, 2020.
- [42] Kamal. *Embedded Systems: Architecture, Programming and Design*. McGraw-Hill Science/Engineering/Math, 2006.
- [43] Farid Kenarangi and Inna Partin-Vaisband. Exploiting Machine Learning Against On-Chip Power Analysis Attacks: Tradeoffs and Design Considerations. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 66(2):769–781, 2018.

- [44] Haider Adnan Khan, Nader Sehatbakhsh, Luong N Nguyen, Milos Prvulovic, and Alenka Zajić. Malware Detection in Embedded Systems Using Neural Network Model for Electromagnetic Side-Channel Signals. *Journal of Hardware and Systems Security*, 3:305–318, 2019.
- [45] N Abid Ali Khan, N Sankariah, and PA Govindacharyulu. Best Practices of Fixing Extreme Embedded Software Bugs in Designing Hard Real-time System.
- [46] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *Advances in Cryptology—CRYPTO’99: 19th Annual International Cryptology Conference Santa Barbara, California, USA, August 15–19, 1999 Proceedings 19*, pages 388–397. Springer, 1999.
- [47] Paul C Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology—CRYPTO’96: 16th Annual International Cryptology Conference Santa Barbara, California, USA August 18–22, 1996 Proceedings 16*, pages 104–113. Springer, 1996.
- [48] Sang Gyu Kwak and Jong Hae Kim. Central limit theorem: the cornerstone of modern statistics. *Korean journal of anesthesiology*, 70(2):144–156, 2017.
- [49] Ji Li and Jianhua Shen. The Implementation and Optimization of Data Conversion between UART and WLAN. In *2010 Second International Conference on Networks Security, Wireless Communications and Trusted Computing*, volume 2, pages 365–368, 2010.
- [50] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. Fuzzing: State of the Art. *IEEE Transactions on Reliability*, 67(3):1199–1218, 2018.
- [51] J. W. Lindeberg. Eine neue Herleitung des Exponentialgesetzes in der Wahrscheinlichkeitsrechnung. *Mathematische Zeitschrift*, 15:211–225, 1922.
- [52] Linuxize. Linux Cat Command. <https://linuxize.com/post/linux-cat-command/>, 2019.
- [53] Linuxize. Linux Echo Command. <https://linuxize.com/post/echo-command-in-linux-with-examples/>, 2019.
- [54] Linuxize. Linux Find Command. <https://linuxize.com/post/how-to-find-files-in-linux-using-the-command-line/>, 2019.

- [55] Linuxize. Linux Ls Command. <https://linuxize.com/post/how-to-list-files-in-linux-using-the-ls-command/>, 2019.
- [56] Linuxize. Linux Ps Command. <https://linuxize.com/post/ps-command-in-linux/>, 2019.
- [57] Linuxize. Linux Pwd Command. <https://linuxize.com/post/current-working-directory/>, 2019.
- [58] Linuxize. Linux Time Command. <https://linuxize.com/post/linux-time-command/>, 2019.
- [59] Linuxize. Linux Uname Command. <https://linuxize.com/post/uname-command-in-linux/>, 2019.
- [60] Georg Macher, Christoph Schmittner, Omar Veledar, and Eugen Brenner. ISO/SAE DIS 21434 Automotive Cybersecurity Standard - In a Nutshell. In *Computer Safety, Reliability, and Security. SAFECOMP 2020 Workshops: DECSoS 2020, DepDevOps 2020, USDAI 2020, and WAISE 2020, Lisbon, Portugal, September 15, 2020, Proceedings 39*, pages 123–135. Springer, 2020.
- [61] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering*, 2019.
- [62] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. Power Analysis Attacks: Revealing the Secrets of Smart Cards. 31:45–50, 2008.
- [63] MathWorks. Measure the Power of a Signal. <https://www.mathworks.com/help/signal/ug/measure-the-power-of-a-signal.html>.
- [64] Memfault. Device Logging: Techniques and Best Practices. <https://interrupt.memfault.com/blog/device-logging>.
- [65] Microchip Technology Inc. MCP2515 Family Data Sheet. <https://ww1.microchip.com/downloads/aemDocuments/documents/APID/ProductDocuments/DataSheets/MCP2515-Family-Data-Sheet-DS20001801K.pdf>.
- [66] Mikroe. FT201X USB I2C SLAVE IC Datasheet, version 1.5, 2024. URL https://download.mikroe.com/documents/datasheets/FT201X_datasheet.pdf.

- [67] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices. In *NDSS*, 2018.
- [68] V. Narayanan and Y. Xie. Reliability Concerns in Embedded System Designs. *Computer*, 39(1):118–120, 2006.
- [69] Nataliia Neshenko, Elias Bou-Harb, Jorge Crichigno, Georges Kaddoum, and Nasir Ghani. Demystifying IoT Security: An Exhaustive Survey on IoT Vulnerabilities and a First Empirical Look on Internet-Scale IoT Exploitations. *IEEE Communications Surveys & Tutorials*, 21(3):2702–2733, 2019.
- [70] Tammy Noergaard. *Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers*. Newnes, 2012.
- [71] Alan V Oppenheim, Alan S Willsky, and Syed Hamid Nawab. *Signals & Systems*. Pearson Educación, 1997.
- [72] Dorottya Papp, Zhendong Ma, and Levente Buttyan. Embedded Systems Security: Threats, Vulnerabilities, and Attack Taxonomy. In *2015 13th Annual Conference on Privacy, Security and Trust (PST)*, pages 145–152. iee, 2015.
- [73] Daniel Parente. The Future of Embedded Systems in the Age of AI and IoT. <https://medium.com/daniel-parente/the-future-of-embedded-systems-in-the-age-of-ai-and-iot-1f86a6ca1795>, 2020.
- [74] Se-Ho Park, Yong-suk Park, and Jun-Rim Choi. Performance Analysis of Embedded OS in ARM Platform. In *2012 International Conference on ICT Convergence (ICTC)*, pages 51–52. IEEE, 2012.
- [75] David A Patterson and John L Hennessy. *Computer Organization and Design ARM edition: The Hardware Software Interface*. Morgan kaufmann, 2016.
- [76] Stjepan Picek, Annelie Heuser, Alan Jovic, Simone A. Ludwig, Sylvain Guilley, Domagoj Jakobovic, and Nele Mentens. Side-Channel Analysis and Machine Learning: A Practical Perspective. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 4095–4102, 2017.
- [77] Andy D Pimentel, LO Hertzbetger, Paul Lieverse, Pieter Van Der Wolf, and EE Deprettere. Exploring Embedded-Systems Architectures with Artemis. *Computer*, 34(11):57–63, 2001.

- [78] Prolific Technology Inc. PL2303HXD USB to Serial Bridge Controller Datasheet. https://www.prolific.com.tw/UserFiles/files/ds_pl2303HXD_v1_4_4.pdf.
- [79] Srivaths Ravi, Anand Raghunathan, Paul Kocher, and Sunil Hattangady. Security in Embedded Systems: Design Challenges. *ACM Trans. Embed. Comput. Syst.*, 3(3):461–491, aug 2004.
- [80] John Regehr and Usit Duongsaa. Preventing Interrupt Overload. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '05, page 50–58, New York, NY, USA, 2005. Association for Computing Machinery.
- [81] Sebastian Reichelt, Jan Stoess, and Frank Bellosa. A Microkernel API for Fine-Grained Decomposition. In *Proceedings of the Fifth Workshop on Programming Languages and Operating Systems*, pages 1–5, 2009.
- [82] Pál Révész. *The Laws of Large Numbers*, volume 4. Academic Press, 2014.
- [83] Gene Sally. BusyBox. In *Pro Linux Embedded Systems*, pages 293–307. Springer, 2010.
- [84] Daniel Sangorrin, Shinya Honda, and Hiroaki Takada. Dual Operating System Architecture for Real-Time Embedded Systems. *Jul*, 6:1–24, 2010.
- [85] Jayaprakash Selvaraj, Gökçen Yılmaz Dayanıklı, Neelam Prabhu Gaunkar, David Ware, Ryan M. Gerdes, and Mani Mina. Electromagnetic Induction Attacks Against Embedded Systems. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ASIACCS '18, page 499–510, New York, NY, USA, 2018. Association for Computing Machinery.
- [86] KV Shibu. *Introduction to Embedded Systems*. Tata McGraw-Hill Education, 2009.
- [87] Robert H Shumway, David S Stoffer, Robert H Shumway, and David S Stoffer. ARIMA Models. *Time Series Analysis and Its Applications: With R Examples*, pages 75–163, 2017.
- [88] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary. In *NDSS*, 2019.

- [89] Spectrum Instrumentation. Advantages of High Resolution in High Bandwidth Digitizers. https://spectrum-instrumentation.com/applications/product_notes/PN_Advantages_of_High_Resolution_in_High_Bandwidth_Digitizers.php.
- [90] Philip Sperl and Konstantin Böttinger. Side-channel Aware Fuzzing. In *Computer Security—ESORICS 2019: 24th European Symposium on Research in Computer Security, Luxembourg, September 23–27, 2019, Proceedings, Part I 24*, pages 259–278. Springer, 2019.
- [91] William Stallings. *Data and Computer Communications*. Pearson Education India, 2007.
- [92] Wojciech Staszewski, Adam Jabłoński, and Kajetan Dziędziech. A Survey of Communication Protocols in Modern Embedded Condition Monitoring Systems. *Diagnostyka*, 19, 2018.
- [93] Naeem Firdous Syed, Syed W. Shah, Arash Shaghaghi, Adnan Anwar, Zubair Baig, and Robin Doss. Zero Trust Architecture (ZTA): A Comprehensive Survey. *IEEE Access*, 10:57143–57179, 2022.
- [94] JP Talledo. Design and Implementation of an Ethernet Frame Analyzer for High Speed Networks. In *15th International Conference on Electronics, Communications and Computers (CONIELECOMP’05)*, pages 171–176. IEEE, 2005.
- [95] T.K. Tan, A. Raghunathan, and N.K. Jha. Embedded Operating System Energy Analysis and Macro-modeling. In *Proceedings. IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 515–522, 2002.
- [96] TechTarget. Signal-to-noise ratio (SNR) definition. <https://www.techtarget.com/searchnetworking/definition/signal-to-noise-ratio>.
- [97] Guangmo Tong and Cong Liu. Supporting Read/Write Applications in Embedded Real-time Systems via Suspension-aware Analysis. In *2014 International Conference on Embedded Software (EMSOFT)*, pages 1–10, 2014.
- [98] Teng Xu, James B Wendt, and Miodrag Potkonjak. Security of IoT Systems: Design Challenges and Opportunities. In *2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 417–423. IEEE, 2014.
- [99] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing New Operating Primitives to Improve Fuzzing Performance. In *Proceedings of the 2017*

- ACM SIGSAC conference on computer and communications security*, pages 2313–2328, 2017.
- [100] Jian Yang, Yu Chen, Huayong Wang, and Bibo Wang. A Linux Kernel with Fixed Interrupt Latency for Embedded Real-Time System. In *Second International Conference on Embedded Software and Systems (ICESSE'05)*, pages 8 pp.–, 2005.
 - [101] J Yiu. The Definitive Guide to ARM® Cortex®-M3 and Cortex®-M4 Processors. Newnes, 2013.
 - [102] Bo Yu, Pengfei Wang, Tai Yue, and Yong Tang. Poster: Fuzzing IoT Firmware via Multi-stage Message Generation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS '19*, page 2525–2527, New York, NY, USA, 2019. Association for Computing Machinery.
 - [103] Joobeom Yun, Fayozbek Rustamov, Juhwan Kim, and Youngjoo Shin. Fuzzing of Embedded Systems: A Survey. *ACM Computing Surveys*, 55(7):1–33, 2022.
 - [104] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: A Survey for Roadmap. *ACM Computing Surveys (CSUR)*, 54(11s):1–36, 2022.
 - [105] Richard Zurawski. *Embedded Systems Handbook: Embedded Systems Design and Verification*. CRC press, 2018.