A Lightweight Host-based Intrusion Detection System using a Hardware-Assisted Monitor to detect Wireless Attacks Targeting Constrained IoT Devices

Mohamed El-Bouazzati

COLLEGE MATHSTIC
DOCTORAL BRETAGNE
BRETAGNE OCEANE

Université Bretagne Sud

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ BRETAGNE SUD

ÉCOLE DOCTORALE N° 644
Mathématiques et Sciences et Technologies de l'Information et de la Communication en Bretagne Océane
Spécialité : *Informatique et Architectures Numériques*

Par
Mohamed EL BOUAZZATI

A Lightweight Host-based Intrusion Detection System using a Hardware-Assisted Monitor to detect Wireless Attacks Targeting Constrained IoT Devices

Thèse présentée et soutenue à Lorient, le 05 Decembre 2023
Unité de recherche : Lab-STICC UMR 6285
Thèse N° : 685

Rapporteurs avant soutenance :

Aurélien FRANCILLON	Professeur des Universités - EURECOM, Biot, FR
Vincent NICOMETTE	Professeur des Universités - LASS-CNRS, INSA, Toulouse, FR

Composition du Jury :

Président :	Daniela DRAGOMIRESCU	Professeur des Universités - LAAS-CNRS, INSA, Toulouse, FR
Examinateurs :	Aurélien FRANCILLON	Professeur des Universités - EURECOM, Biot, FR
	Pierre-François GIMENEZ	Maître de Conférences - IRISA, CentraleSupélec, Rennes, FR
	Vincent NICOMETTE	Professeur des Universités - LAAS-CNRS, INSA, Toulouse, FR
	Russell TESSIER	Professeur des Universités - University of Massachusetts, Amherst, USA
Dir. de thèse :	Guy GOGNIAT	Professeur des Universités - Lab-STICC, Université Bretagne Sud, Lorient, FR
Co-dir. de thèse :	Philippe TANGUY	Maître de Conférences - Lab-STICC, Université Bretagne Sud, Lorient, FR

Acknowledgement

Throughout my academic journey, I have been grateful for the support of a vast network of people whose advice and guidance were key in helping me to achieve my PhD.

I would like to express my deep gratitude for the continuing support of my thesis supervisors, Professor Guy Gogniat and Associate Professor Philippe Tanguy. Their advice and confidence in my abilities have been the keystone of my academic achievements.

Throughout my PhD, I also had the privilege of collaborating with an international laboratory and being hosted as a visiting researcher at the University of Massachusetts. I am very grateful to Professor Russell Tessier for his invaluable advice and support during and after this period. His mentorship has been crucial to the success of my research.

I extend my gratitude to the respected jury members : Daniela Dragomirescu of INSA Toulouse, France; Vincent Nicomette of INSA Toulouse, France; Russell Tessier of the University of Massachusetts, Amherst, MA, USA; Aurélien Francillon of EURECOM, Biot, France; and Pierre-François Gimenez of CentraleSupélec, Rennes, France. I sincerely appreciate the time they dedicated to reviewing this manuscript and the constructive remarks they provided. I would also like to thank Pascal Benoit and Bertrand Legal for having participated in my CSI (Comité de Suivi Individuel). Their scientific discussions and insightful advice played a crucial role in shaping my thesis.

My heartfelt thanks go to my friends and colleagues at Lab-STICC and other laboratories who have made a significant contribution to the productivity of my working environment. I am particularly grateful to Béatrice Guern, Virginie Guillet, Florence Palin and Noluenn Chauvin for their immeasurable help in various administrative aspects. Their support was crucial and I am grateful for their dedicated efforts.

My warmest thanks go to my whole family: my parents, my brothers, my sister and my wife for their unconditional love and support.

Acronyms

A

ASIC Application Specific Integrated Circuit.

B

BLE Bluetooth Low Energy.

C

CPU Central Processing Unit.

CPUs Central Processing Units.

CRC Cyclic Redundancy Check.

CSR Control Status Register.

D

DDoS Distributed Denial of Service.

DoS Denial of Service.

E

EWMA Exponentially Weighted Moving Average.

F

FPGA Field Programmable Gate Array.

FSM Finite State Machine.

H

HIDS Host-Based Intrusion Detection System.

HPC Hardware Performance Counter.

HPM Hardware Performance Monitoring.

HPMtracer Hardware Performance Monitoring Tracer.

I

IDS Intrusion Detection System.

IoT Internet of Things.

ISA Instruction Set Architecture.

L

LCL Lower Control Limit.

M

MAC Media Access Control.

MCU Microcontroller Unit.

MITM Man in the Midlle.

N

NIDS Network-Based Intrusion Detection System.

NwHPC Network Hardware Performance Counter.

O

OTA Over the Air.

P

PHY Physical.

PLR Packet Loss Rate.

R

RCE Remote Code Execution.

RF Radio Frequency.

RSSI Received Signal Strength Indicator.

S

SDR Software Defined Radio.

SF Spreading Factor.

SIMD Single Instruction Multiple Data.

SNR Signal Noise Ratio.

SoC System on Chip.

SoCs Systems on Chip.

U

UCL Upper Control Limit.

List of Figures

List of Tables

Table of Contents

Chapter 1

Introduction

1.1 Context

The Internet of Things (IoT) domain has experienced exceptional growth in recent decades and has become an integral part of the world. This has brought significant transformations to various aspects of our lives, enabling continuous data monitoring and real-time remote system control. IoT applications span across sectors, such as healthcare, smart homes, smart cities, agriculture, and industrial automation. Consumer IoT products are being increasingly integrated into daily routines. The spread of IoT devices has been remarkable, with billions now in use globally. For instance, in 2020, the European commission reported in [1] that 51% of individuals in Europe are using IoT devices such as smart TVs, game consoles, home audio systems, and smart speakers. The global IoT market reached 16.7 billion devices and is projected to grow to 29.7 billion by 2027 [2]. The economic impact of IoT is substantial, with worldwide consumer IoT revenue expected to rise from 107.2 billion euros in 2019 to 408.7 billion euros by 2030 [3].

The IoT domain is rapidly expanding in scale; however, along with this growth, there are increasing challenges related to security and data privacy. Cyberattacks targeting IoT devices have surged significantly, with millions of incidents reported annually. In 2016, the cybersecurity landscape was significantly affected by the Mirai botnet attack [4]. This attack strategically targets unsecured IoT devices by scanning a vast array of devices, including digital cameras, in search of open telnet ports. Subsequently, attackers attempted to gain access using default passwords. By successfully infiltrating these vulnerable devices, they effectively constructed a botnet and subsequently unleashed Distributed Denial of Service (DDoS) attacks. In 2022 alone, there were over 112 million IoT attacks worldwide, a stark increase compared to the approximately 32 million detected cases in 2018 [5]. Addressing the security challenges is crucial for ensuring the continued growth and success of the IoT.

The IoT environment's architecture is significant as highlighted in Figure 1.1, spanning

from the top where cloud servers manage vast amounts of data to the bottom, housing IoT gateways that collect data from devices. IoT protocols enable wireless connections between the gateways and devices. Finally, IoT devices, which are physical components, interact with the real world. IoT end-devices are a subset of IoT devices that possess limited resources in terms of performance, computing capabilities, memory capacity, and power consumption. The security of embedded systems within IoT end-devices has been targeted through various mechanisms. This can be addressed using several measures such as protecting physical access to devices, securing data privacy with cryptographic encryption, and enhancing communication security. Embedded systems in IoT devices now possess built-in communication capabilities, which serve as the primary entrance for data to the IoT network.

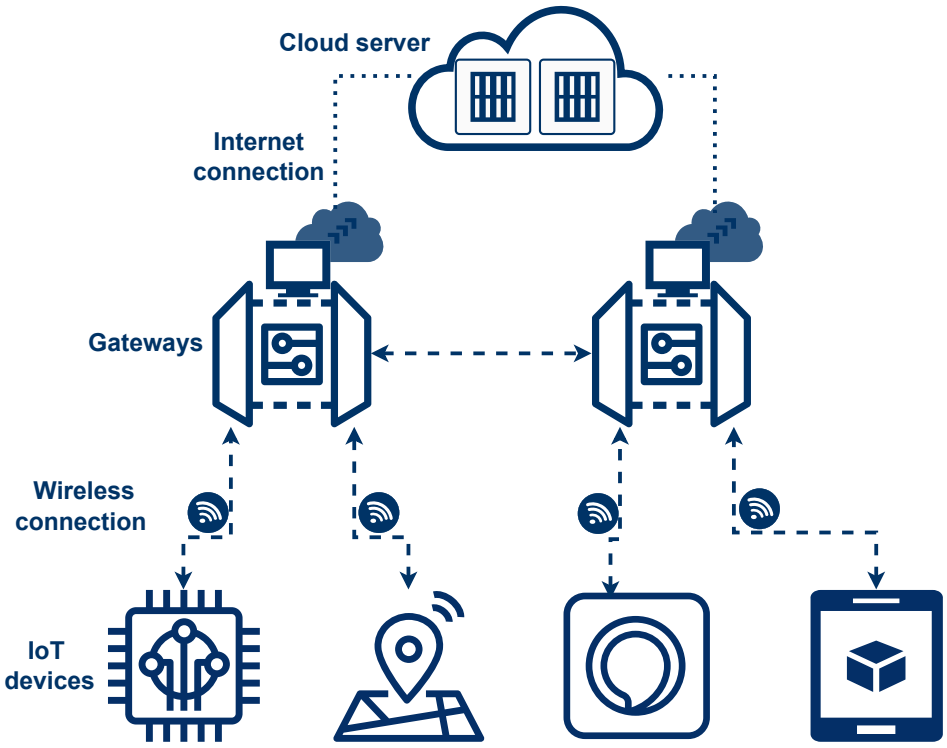


Figure 1.1 – IoT Environment Architecture

The built-in wireless communication capability is a significant entry point for security threats to IoT devices. It represents a crucial attack surface housing various vulnerabilities that attackers exploit to launch cyberattacks in the IoT environment. As an illustrative example, Forescout Research Labs uncovered 33 vulnerabilities, collectively referred to as AMNESIA:33 [6, 7]. These vulnerabilities pose a significant risk to millions of IoT devices

and affect four open-source TCP/IP IoT stacks. These vulnerabilities primarily result in memory corruption, giving attackers the ability to compromise devices, gain remote control, execute malicious code, initiate denial-of-service attacks, and steal data. Attackers can then potentially access various environments where these devices are deployed, often owing to common oversight during software development and product design phases, where security considerations are neglected. The emergence of high-performance, low-cost platforms and associated software [8] has also expanded the opportunities for attackers and simplified their access to the lower layers of communication networks. The potential economic situations of these vulnerabilities are considerable, given that open-source IoT stacks are widely utilized by numerous vendors in their products. Addressing these vulnerabilities requires the deployment of patches across millions of IoT devices. However, a challenging aspect of this situation is that some IoT devices cannot be patched due to technical unfeasibility.

In this thesis, we address the security challenges presented by IoT end-devices, with a particular emphasis on security threats affecting their wireless connectivity. The primary objective is to provide a robust security mechanism capable of detecting potential attacks.

1.2 Motivation

Embedded Systems on Chip (SoCs) in IoT end-devices include security mechanisms against cyberattacks, such as cryptographic accelerators for data encryption, secure boot procedures, and over-the-air update mechanisms. These are crucial for addressing the IoT device security challenges. However, IoT end-devices lack monitoring and detection mechanisms that can track system metrics and analyze behavior to identify malicious activities. Combining monitoring and detection with existing security measures can enhance the resilience of IoT devices to security challenges.

The attack surface of IoT end-devices has expanded significantly owing to their wireless connectivity. For example, jamming attacks disrupt IoT communication protocols, leading to Denial of Service (DoS) incidents. These attacks can present difficulties in countering them using existing protections and necessitate monitoring network metrics, such as Received Signal Strength Indicator (RSSI) and Signal Noise Ratio (SNR), to detect jammers. Another type of cyberattack involves packet injection, which exploits software-memory vulnerabilities. These attacks range from simple DoS attacks to the control of IoT devices. Protection against such attacks often requires access to IoT de-

vices for software patching or updates. Real-time monitoring and detection mechanisms can help to identify these vulnerabilities. However, implementing monitoring and detection security mechanisms in resource-constrained embedded IoT systems presents several challenges. The development of lightweight monitoring and detection methods against wireless attacks is promising for mitigating these issues.

1.3 Contributions and Organization of Manuscript

1.3.1 Contributions

This thesis comprises two significant contributions. It introduces a methodology and the associated experimental framework for simulating, emulating, and implementing wireless attacks, such as jamming and packet injection. This approach also includes dataset generation for microarchitectural and network metrics, which are essential for constructing a Host-Based Intrusion Detection System (HIDS) tailored for IoT end-devices. In addition, it introduces a lightweight IDS called *Diwall*. It was designed to detect packet injection and jamming attacks by monitoring microarchitectural events and RSSI metadata within the LoRaWAN stack.

In our first contribution, we introduce our methodology and the associated framework to study packet injection and jamming attacks that target IoT protocol stacks. This framework generates extensive datasets for both the simulated and real-world scenarios. We delve into the behavior of microarchitectural events during memory stack and heap buffer overflows, focusing on packet injection attacks. Additionally, we examined network metadata metrics such as RSSI and SNR to identify jamming attacks. Our research encompasses machine learning and statistical techniques applied to the generated datasets. Importantly, this comprehensive framework is adaptable for use with various IoT protocol stacks and Instruction Set Architecture (ISA) processors. This adaptability is possible because monitored microarchitectural and RSSI metadata are present in modern Central Processing Units (CPUs) and IoT protocol stacks. Within this methodology, we employ a simplified Media Access Control (MAC) layer that can be substituted with more complex MAC layers that we demonstrated using the LoRaWAN MAC layer. Similarly, the LoRa Physical (PHY) layer can be replaced with technologies such as Bluetooth or Zigbee to investigate the same wireless attacks that pose threats to the mentioned IoT stacks.

In our second contribution, we introduced *Diwall*, a hardware-based lightweight HIDS

that requires minimal software instructions for configuration and control. *Diwall* uses microarchitectural event analysis to identify packet injection attacks that exploit buffer overflow, and it utilizes RSSI metrics to detect jamming attacks. We implemented *Diwall* as a compact component within a RISC-V processor on an Field Programmable Gate Array (FPGA) board. In real-world scenarios, we evaluated the detection rates and achieved impressive levels of approximately 99.98%. Moreover, *Diwall* requires a minimal FPGA area overhead of approximately 14.30%, consuming 22.89% of LUTs and FFs, and does not negatively impact system performance.

1.3.2 Organization

The remainder of this manuscript is organized as follows.

Chapter 2: This chapter aims to establish a comprehensive background on security threats and mechanisms applied in SoC for Low-Power and Low-Data-Rate IoT end-devices. We begin by providing an overview of SoCs in both research and industry contexts. Subsequently, we delve into the security aspects of the SoC wireless connectivity for the considered waveforms. We also discuss the implementation approach with respect to the attack surface. In this chapter, we explore the potential of using intrusion detection for IoT end-devices, including related works, identified challenges, and requirements.

Chapter 3: In this chapter, we detail the methodology used in our framework to build *Diwall*, a lightweight HIDS designed for detecting attacks in the wireless connectivity of IoT end-devices. We outline the targeted threat model, our proposed approach, and the key features of *Diwall* in the context of related works. Furthermore, we investigate wireless attacks, jamming, and packet injection based on memory corruption in separate sections. For packet injection attacks, we utilize microarchitectural event datasets generated from simulations and apply machine learning classification techniques. Another section focuses on network metadata metrics, such as RSSI and SNR, in real communication scenarios, studying their use in detecting jamming attacks through statistical techniques. We also discuss the machine learning classification and statistical techniques used for the generation of *Diwall* detection models.

Chapter 4: In this chapter, we present the implementation details of *Diwall* on hardware using detection models for packet injection and jamming attacks. We elaborate on the detection models generated using the framework outlined in the previous chapter. Additionally, we provide insights into the *Diwall* architecture, configuration, resource utilization, and its impact on FPGA performance. We evaluate its detection rates and

present a practical use case of *Diwall* within a real IoT protocol stack.

Chapter 5: The final chapter is dedicated to concluding remarks on the obtained results, identifying limitations, and discussing the challenges encountered in this thesis. We also explore future research perspectives, address the limitations, and suggest potential improvements.

Chapter 2

System-on-Chip for IoT and Wireless Security

Introduction

This chapter covers background knowledge for understanding an SoC in the IoT domain. First, it provides an overview of the SoC subsystems for low-power, low-data-rate IoT end-devices. In addition, an identification and comparison of several features are described for SoCs used in industry and those used in research. Second, a focus is proposed on the principal wireless connectivity subsystem architectures for the low-power, low-data-rate waveforms considered in this study. Based on this knowledge, we highlight the security issues related to wireless connectivity, including existing vulnerabilities, attacks, and the proposed security mechanisms. Finally, this chapter addresses the potential of IoT intrusion detection systems as a security mechanism and the associated challenges. A concluding section is provided based on the lessons derived from the literature.

2.1 SoC for Low-Power and Low-Data-Rate End-Devices

2.1.1 SoC Overview

IoT SoCs are an integrated circuit designed to offer both processing and communication capabilities. To achieve this, various modules and subsystems were integrated. Figure 2.1 shows a general overview of the subsystems inside an SoC system.

SoC subsystems can be classified into four main categories according to their characteristics:

— **Application processor:** Generic CPU that hosts the firmware of the user.

— **Peripherals and connectivity:** It includes general-purpose inputs and outputs (GPIOs) and various communication front ends, such as UART, SPI and I2C,. . . .

They are used to interconnect the SoC with external modules and sensors. Hardware timers are also included.

— **Cryptographic accelerator:** An SoC is provided with the necessary dedicated hardware for implementing cryptographic algorithms, such as AES, RSA and SHA,....

— **Wireless connectivity:** An SoC incorporates modules designed to implement and process the lower layers of IoT protocol stacks (MAC and PHY layers). It also contains a built-in Radio Frequency (RF) transceiver.

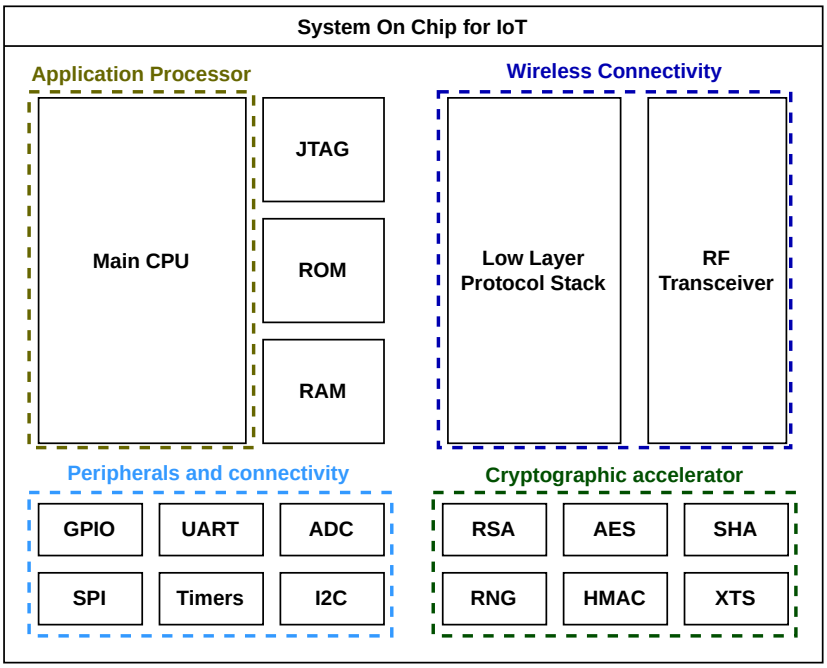


Figure 2.1 – Block Diagram of SoC for IoT end-devices

2.1.2 SoC in Industry

Many chip manufacturers in the industry introduced SoC solutions [9–11] for IoT a few years ago, in which wireless capabilities were built-in. Such SoCs integrate all the major components featured in Figure 2.1 on one chip. They provide a range of integrated network connectivity technologies, mainly commonly used IoT protocols, such as LoRaWAN, Bluetooth/Bluetooth Low Energy (BLE), and ZigBee. The built-in radio ensures dedicated wireless network access, removing the problem of compatibility issues.

They also reduce implementation costs, energy requirements, and complexity compared to traditional SoCs, requiring expensive hardware for wireless connectivity.

In industry, several SoCs feature built-in wireless connectivity. Here, we discuss only three types of SoCs designed with multi-protocol capabilities.

ESP32-H2 [11] is an SoC developed by Espressif Systems and is part of the ESP32 series. The SoC is powered by a single-core, 32-bit RISC-V processor. The Wireless connectivity supports the coexistence of radio protocols in the 2.4 GHz band, with each protocol having its own Application Specific Integrated Circuit (ASIC) baseband. Achieving a more flexible approach of wireless connectivity and adaptability when protocols need to be changed involves the integration of a dedicated network CPU. STM32WL55CC [9] is a STMicroelectronics SoC, which is part of the STM32WL series. It combines an Arm Cortex-M4 processor core as the main CPU and an Arm Cortex-M0+ core dedicated to sub-GHz wireless connectivity. It supports various sub-GHz IoT stacks and proprietary protocols. CC1352R [10], produced by Texas Instruments, features a multiband device specifically tailored for IoT and proprietary protocols in the sub-GHz and 2.4GHz frequencies. SoC CC1352R incorporates dual Arm Cortex-M4F/M0 cores, supporting a variety of protocol operations. Wireless connectivity is managed by an Arm Cortex-M0 core and has the ability to manage simultaneous protocols.

Security functions are provided by most chip manufacturers for their SoCs, typically using modules that provide similar security functions. ESP32-H2, STM32WL55CC, and CC1352R incorporate hardware security features, such as cryptographic acceleration (AES, SHA, RSA, and TRNG). Integrated hardware enables code authentication for a wide range of security services. These include secure boots, secure firmware update mechanisms, memory encryption/decryption, and protocol stack key generation.

Table 2.1 provides a summary of the three industry IoT SoCs discussed. The main focus of the comparison was wireless connectivity capabilities. Further details are provided in technical datasheets [9–11].

2.1.3 SoC in Research

Researchers and the open-source community have been involved in the development of various SoCs in different HDL languages [12, 13]. These SoCs are commonly designed to be highly scalable, allowing new functionalities to be added according to the specific requirements of the application. A significant number of these SoCs are based on the RISC-V architecture, an open-source ISA [14, 15]. The current focus in research-oriented

Table 2.1 – Industrial IoT SoCs Features Comparison

Features	CC1352R [10]	STM35WL55 [9]	ESP32-H2 [11]
Main CPU	Cortex M4F	Cortex M4 or M0+	32 bits RISC-V
Wireless connectivity	Cortex M0	Cortex M0+ or M4	ASIC Baseband
Frequency Band	2.4+sub-GHz	sub-GHz	2.4GHz
Concurrent Multiprotocol	✓	✗	✗
Radio Flexibility	✓	✓	✗
Radio/App Isolation	✓	✓	✗
Main Security	Secure Boot/Firmware update/Cryptography Acceleration		

SoCs is often on separate subsystems within the SoC, such as the processor architecture and peripheral subsystems. Hardware-based security accelerators or built-in mechanisms have also been discussed and proposed as separate subsystems [16, 17]. Some processors have been modified with ISA extensions [18] to incorporate instructions dedicated to specific tasks such as security or DSP. However, open-source platforms do not always offer wireless connectivity. They provide a multipurpose and highly flexible SoC that can be customized and expanded. Our discussion here is specifically centered on RISC-V-based SoCs. The reasons for this focus are the open-source nature of RISC-V and the extensive involvement of the community in both research and industrial applications.

The PULP (Parallel Ultra-Low Power) platform project [13] covers open-source SoC designs for energy-efficient computing. It provides multicore support and multiple IP cores for peripherals, such as UART, SPI, JTAG, and I2C. In addition, PULP supports the integration of hardware accelerators. The PULP SoC uses 32-bit RISC-V processors and can be configured to use either a 4-stage RI5CY pipeline [19] or 2-stage Zero-riscy pipeline [20]. Several SoCs optimized for different use cases are based on the PULP platform, such as PULPissimo [21] and PULPino [15], designed for IoT applications at the edge. They are available for RTL simulation, FPGA implementation, and ASICs. The CORE-V-MCU [14] is a modern 32-bit RISC-V-based open-source SoC for IoT devices. It was originally based on the PULP platform and has the same peripherals as the PULPissimo SoC. However, it has been extended to include an eFPGA to implement customs peripherals to improve the energy efficiency of the SoC.

LiteX [12] is an SoC building framework that provides a wide range of open-source components. It includes buses, simple IPs such as RAM and UART, complex IPs, and supports for various CPUs and ISAs. The framework supports mixed HDL integration

and compatibility with VHDL, Verilog, System Verilog, Migen, Spinal HDL, ... Taking advantage of LiteX's vast collection of open-source IPs and support for mixed HDLs, the design, simulation, and implementation of a custom SoC for FPGA boards are simplified with respect to the PULP platform [13].

Table 2.2 provides a comparable overview of available SoCs. It concentrates primarily on the open-source cores and peripherals included, CPU and ISA supported, and simulation and prototyping flows. Further details can be found in [12–15].

Table 2.2 – Comparison of Research SoCs Features

Features	PUPLino [15]	PULP [13]	CORE-V-MCU [14]	LiteX Framework [12]
CPU	RI5CY/Zero-riscy		CV32E40P+ eFPGA	Various CPUs/ISAs
Core Numbers	Single	Multi	Single	Multi/Single
Peripherals	UART, SPI, I2C	UART, SPI, I2C, CPI, JTAG		Simple + Complex IPs
BUS	APB + AXI4	μDMA+TCDMI+APB		Whishbone/AXI
Cache L1/L2	✗	✓	✓	✓
HDL support	System Verilog			Mixed HDL
Prototyping	RTL Simulation, FPGA, ASIC			RTL Simulation, FPGA

This section delves into the discussion of SoCs in both research and industry contexts. Additionally, we provide a concise overview of the SoC subsystems. Shifting our focus to the next section, we delve into various implementation approaches for wireless connectivity, with a specific emphasis on the considered IoT protocol stacks.

2.2 SoC Wireless Connectivity

2.2.1 IoT Protocol Stack

There are several main layers in an IoT protocol stack. First, the PHY layer manages modulation and demodulation, the physical transmission of data, and is used to establish connectivity physically. Another important layer is the MAC layer, where network packets are processed. It handles access to the physical medium and ensures that network packets are transmitted and received according to the standards. In addition to the MAC layer, there are additional upper layers for the application-specific protocols. They manage the requirements specific to different IoT applications. They provide the functionality required for data exchange and interaction with the user applications. IoT SoC wireless connectivity generally includes a physical layer and MAC layer.

Figure 2.2 presents a typical block diagram of an IoT protocol stack along with three other protocol stacks: BLE, LoRaWAN, and ZigBee. An IoT protocol can generally be considered to have three main levels: the physical, MAC, and application layers.

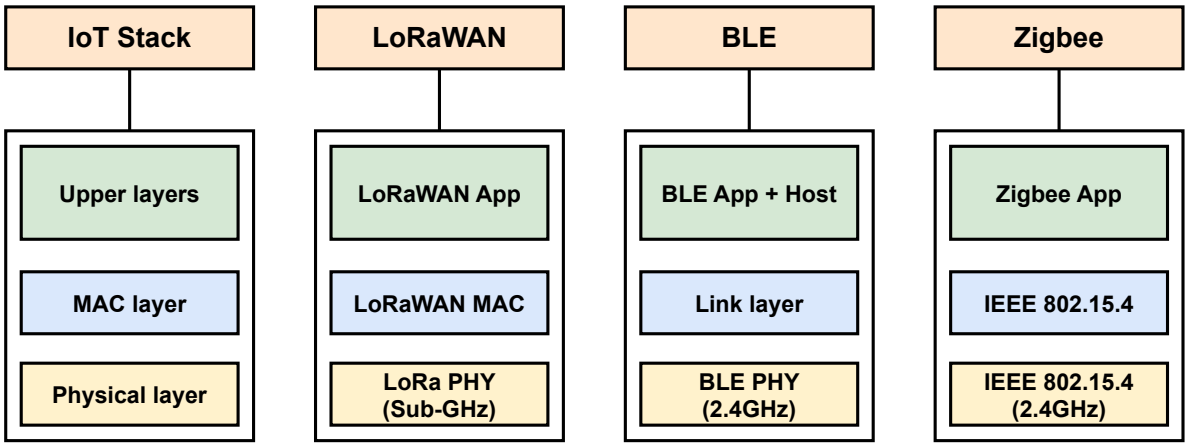


Figure 2.2 – IoT Stack Layers

2.2.2 Considered Waveforms

The focus of this research is on low-power IoT protocols that support low-data-rate. Targeting specific IoT end-devices with power constraints, IoT stack technologies are explored for implementation in low-power wide area networks (LPWANs) operating in the sub-GHz frequency bands. An example of such technology is LoRaWAN. In addition, low-data-rate protocols that operate in the 2.4 GHz frequency band, such as BLE and ZigBee, have also been studied.

LoRaWAN: LoRaWAN, or long-range wide-area network, is a MAC protocol designed for managing wireless communication between battery-powered IoT end-devices and gateways within a large-scale network [22]. This protocol forms a software layer that prescribes the interaction between the devices and the LoRa hardware. Built on top of the sub-GHz band LoRa physical layer [23], LoRaWAN is well suited for transmitting small payloads such as sensor data over extended distances. The key feature of LoRaWAN is its utilization of a 'star-of-stars' topology [24]. In this configuration, end-devices constitute a star pattern around the gateways, which then communicates with a central network server, generating another star pattern. This hierarchical structure facilitates long-range

communication while minimizing power consumption because each end-device communicates only with a proximate gateway rather than the central server. Moreover, gateways can be strategically positioned at elevated locations with broad coverage to optimize the range of communication. Therefore, LoRaWAN has emerged as an efficient solution for wide area networks (WANs), rendering it particularly applicable to use cases such as IoT applications.

BLE: BLE, or Bluetooth Low Energy, is an energy-efficient variant of traditional Bluetooth technology, it was first introduced as part of Bluetooth 4.0, in 2010 [25]. Tailored explicitly for IoT applications, BLE has been widely adopted in various sectors, such as smart homes, healthcare, and industrial automation, primarily because of its power efficiency and design tailored for short-range communications. Uniquely, BLE operates using a star topology, where a central 'master' device forms connections with multiple 'slave' devices. This master device is capable of sending and receiving data from any of its connected slave devices, but communication between the slave devices is not possible. This specific network structure provides an effective power management strategy because the slave devices can remain in sleep mode until the master initiates a connection. Such power conservation capabilities make BLE particularly suited for IoT devices that require extended operational periods for battery power.

ZigBee: ZigBee, a high-level wireless communication protocol, was purposefully designed for short-range, low-power communications [26]. Built upon the MAC and Physical layers of the IEEE 802.15.4 standard for wireless personal networks [27], it has a distinct topology adaptable to multiple network structures, such as mesh, star, and tree topologies. This distinct characteristic affords ZigBee with high resilience and network robustness, as it can reroute data through alternative paths if a node fails or a connection is lost. Moreover, ZigBee's various topological supports enhance its demand for diverse IoT applications. Uniquely, while ZigBee operates in the 2.4GHz band like Bluetooth, it is also optimized to function in the sub-GHz range in specific geographical regions, thereby expanding its usability.

2.2.3 Network Processor Architectures

In SoC architecture, wireless connectivity systems typically include a network processor to manage the PHY and MAC layers. To achieve these requirements, a range of

technologies, such as CPUs, FPGAs, and dedicated ASICs, are often used. The MAC layer can be implemented in software or hardware, whereas, for each protocol, a dedicated ASIC module is typically used to implement the PHY layer.

However, to improve the flexibility of wireless connectivity, it has recently been shown that the PHY layer can be implemented in software [28–31]. Figure 2.3 summarizes the implementation approaches and technologies for SoCs with wireless connectivity. In this figure, we present three distinct methods of implementing wireless connectivity. In SoC 1, the PHY layer is implemented using dedicated hardware specific to each protocol. Conversely, in SoCs 2 and 3, it is implemented using software through Software Defined Radio (SDR), leveraging diverse technologies, including FPGA and CPU. The SDR approach

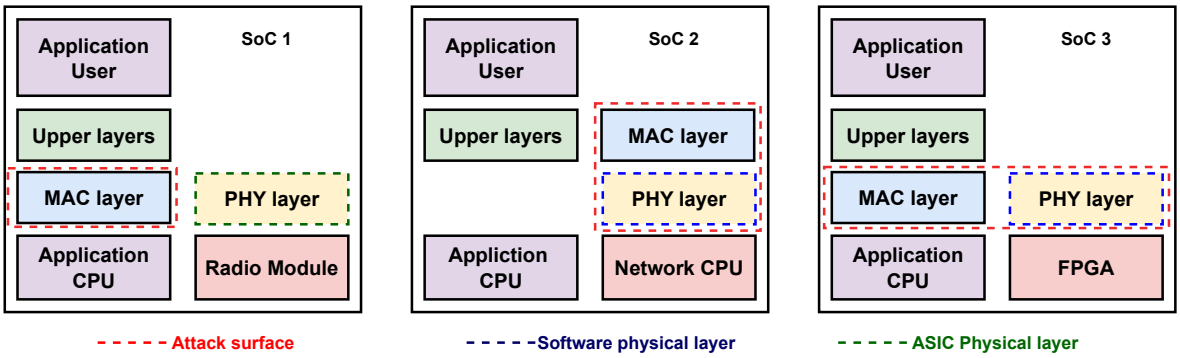


Figure 2.3 – Wireless Connectivity SoC Technologies

has the advantage of being reconfigurable and adaptable to a variety of protocols. Many SoCs with wireless capabilities support multiple protocols. The use of SDR for baseband digital processing is of interest to many researchers. It has inspired the development and implementation of a multitude of architectural approaches, each with its own particular strategies.

Using Hybrid FPGA: In [32], the authors demonstrated the implementation of an SoC with wireless connectivity, where the PHY layer is based on SDR technology. This platform, intended for Over the Air (OTA) programmable IoT devices, uses a Lattice FPGA and a 32-bit Cortex M4F Microcontroller Unit (MCU). The use of such hybrid systems (FPGA and/or MCU) in constrained IoT devices offers significant advantages in terms of computing performance and flexibility. However, these advantages do not extend to energy consumption, and this type of system is less efficient than dedicated hardware or low-power microcontrollers.

Using Dedicated CPU Architecture: The deployment of an SDR PHY layer on a dedicated architecture has become increasingly appealing owing to its consideration of the power consumption. For instance, the authors of [28] proposed an SDR baseband processor for IoT SoCs. This solution, based on a custom Single Instruction Multiple Data (SIMD) datapath, only consumes 1.3 mW of power. Alternatively, another approach [29] introduced a scalar datapath in a baseband processor for IoT SoCs, focusing on low power consumption. It achieved an impressive average energy consumption rate of 2.41 $nJ/cycle$ when operating at 3 MHz. While these dedicated architectures for SDR implementation effectively minimize power consumption, they require IoT protocols' waveform algorithms to be specifically coded in the assembly language for each dedicated processor. Moreover, these architectures lack the ability to utilize existing DSP libraries, such as ARM CMSIS DSP, which curtails their programmability.

Using Extended Generic CPU Architecture: Numerous studies have demonstrated the feasibility of using a generic CPU with ISA extensions for the SDR. The authors of [30] suggested adding DSP extensions to the RISC-V ISA. These extensions are implemented on a 32-bit processor with RV32IM for integer multiplication and division. This setup provides significant flexibility and ultra-low-power consumption. Demonstrations showed a potential power consumption of 380 μW for Bluetooth LE demodulation and 225 μW for LoRa preamble detection.

Another study [31] focused on an ultra-low-power SoC architecture. It uses an ARM Cortex-M4 processor with SIMD or DSP extensions for protocol-specific computations. It also utilizes a hardware digital front end for generic signal processing. The proposed architecture was prototyped using 28 nm FDSOI. The PHY layers of LoRa and Sigfox protocols were implemented in the software, resulting in sub-milliwatt power consumption (32 – 332 μW).

Table 2.3 presents a comparison of the wireless connectivity of IoT SoCs using SDR baseband processor architectures and their features. Architectures are compared based on their capability to run more than one protocol at the same time and their programmability based on the difficulty required to develop waveforms. Orders of magnitude for the dynamic power for each type of architecture were also proposed. More details can be found in [28–32].

Utilizing generic CPU architectures for baseband processors in an SDR is promising. This is because of their flexibility, programmability, and power efficiency. However, during

Table 2.3 – Comparison of Network Processor Architectures in Research

Architecture	Hybrid FPGA [32]	CPU (dedicated) [28, 29]	CPU (extended) [30, 31]
Multi Protocol	✓	✗	✗
Programmability	+	+	+++
Flexibility	+++	++	++
Dynamic power	$\sim 300\ mW$	$\sim 2\ mW$	$\sim 0.5\ mW$
Prototyping	ASIC	FPGA	FPGA [30], ASIC [31]

the design phase, challenges, such as ensuring security against traditional vulnerabilities for wireless connectivity, need to be addressed. The use of SDR in wireless connectivity makes the software part of an IoT stack more extensive. The expansion of software enhances its flexibility and supports multiple protocols. However, this can increase the number of SoC attack surfaces, as illustrated in Figure 2.3. Consequently, it may increase the vulnerability to security threats. In the next section, security threats related to wireless connectivity are explored, including their points of entry, potential vulnerabilities and attacks, and the associated countermeasures.

2.3 Security of SoC Wireless Connectivity

IoT end-devices, in which wireless capabilities are potentially vulnerable, face a significant attack. This section details the common vulnerabilities and attacks on low-data-rate, low-power waveforms/protocols. This also includes an assessment of the attack model. In addition, it explores the security mechanisms discovered during the literature review. This section concludes with security recommendations for IoT devices.

2.3.1 Attack Entry Points

SoCs in IoT devices can have multiple points that are susceptible to attack. In Figure 2.4, we highlight only two potential entry points. One entry point could stem from the application processor executing a malicious process that attempts to carry out illegitimate actions, such as launching a DDoS attack [4]. In such attacks, numerous IoT devices are exploited to overwhelm servers collectively. The second potential entry point relates to the vulnerabilities present in the lower layers of the protocol stacks. Our research primarily focuses on this entry point, specifically, the wireless connectivity of SoCs.

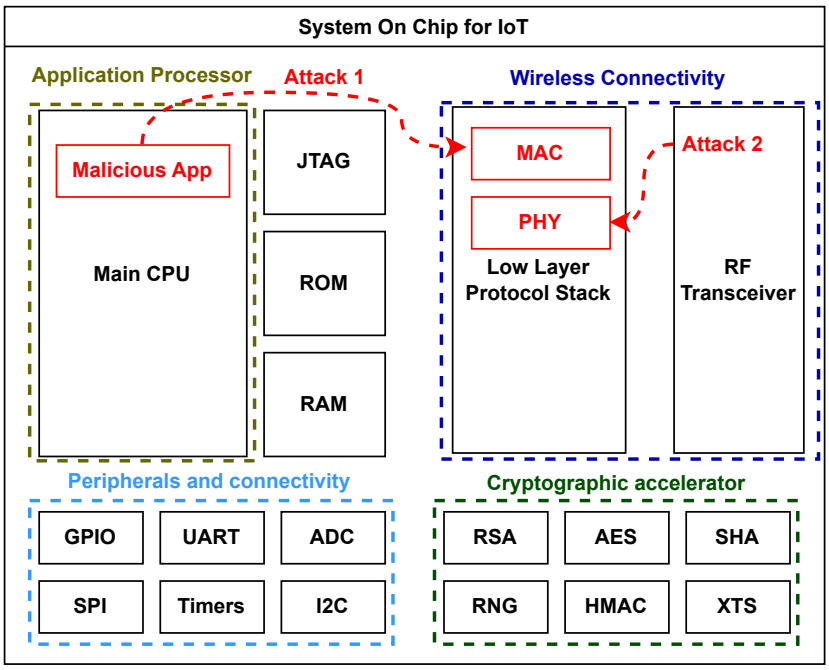


Figure 2.4 – IoT SoC Attacks Entry Points

2.3.2 Vulnerabilities

Several vulnerabilities in IoT devices have recently been identified. Vulnerabilities are found in the communication elements of the devices. They affect several protocols. These include TCP/IP, LoRaWAN, ZigBee, and BLE,.... Numerous vulnerabilities are often associated with the implementation of the protocol standards. However, others are specific to the standard itself.

AMNESIA:33 The Forescout Research Labs conducted a security study on the TCP/IP stack [7]. They published a report analyzing seven open-source TCP/IP stacks. They found 33 new critical vulnerabilities in four stacks: uIP, FNET, picoTCP, and Nut/Net. These stacks are heavily used in millions of IoT devices and by more than 150 vendors. Many of these vulnerabilities are traced back to poor software development practices in the operating systems. These practices include neglecting the basic input validation in SoCs, embedded devices, networking equipment, and various IoT devices for enterprise and consumer use. The exploitation of these vulnerabilities, referred to as AMNESIA33, can result in remote control by attackers. This can occur through memory corruption (buffer overflow) in IoT devices. These risks include compromised devices, execution of

malicious code, DoS attacks, and sensitive information theft. The authors of AMNESIA:33 acknowledge that identifying and rectifying these vulnerabilities pose a significant challenge to the security community. In response, they proposed mitigation measures for AMNESIA33. These include implementing solutions that provide granular visibility of devices through network communication monitoring. They also recommend isolating devices or network segments that are vulnerable to these threats to manage potential risks.

BLEEDINGBIT is a significant vulnerability of BLE chips [33]. It impacts access points from the Cisco, Meraki, and Aruba solutions. These access points use a Texas Instruments (TI) BLE chip. This vulnerability allows an unauthenticated attacker to execute code remotely on targeted chips. This issue, referred to as CVE-2018-16986, originates from a masking error. This error can lead to remote code execution in the BLE stack of the TI chip. This is caused by a memory corruption bug during the parsing of BLE advertising packets. Bluetooth's specifications changed from version 4.2 to 5.0. This change allows larger advertising packets. The size limit ranged from 37 bytes in version 4.2 to 255 bytes in version 5.0. Upgrading a BLE stack from version 4.2 to 5.0 can cause bugs due to this change. A specific problem can occur if a developer does not hide the Reserved for Future Use (RFU) bits. They must do this when writing the code to parse the packet header length field. The RFU bits were 2 in version 4.2 and decreased to 1 in version 5.0. This vulnerability was identified in the BLE stack of the Texas Instruments CC26xx chip family.

Another vulnerability, CVE-2018-7080, was found in the Texas Instruments BLE stack SDK. This vulnerability involves OTA firmware download (OAD), on the affected device, provided its BLE is turned on, without any other prerequisites or knowledge about the device. First, the attacker sends multiple benign BLE broadcast messages, called "advertising packets," which will be stored in the memory of the vulnerable BLE chip in targeted device. While the packets are not harmful, they contain code that will be invoked by the attacker later on. This activity will be undetected by traditional security mechanisms of TI's BLE stack SDK. The Aruba 300 Series access point uses TI's OAD and CC2540 chips to update firmware. However, this allows a nearby attacker to access the device and install malware.

LoRaDawn The Tencent Blade team discovered two vulnerabilities in LoRaWAN, CVE-2020-11068 and CVE-2020-4060 [34]. These vulnerabilities can cause a remote DoS on LoRaWAN end-devices and potentially enable Remote Code Execution (RCE) on the LoRaWAN gateway under certain conditions. CVE-2020-11068 was found in the LoRaWAN end-device stack implementation, specifically in versions of the LoRaMac-node below V4.4.4. This issue arises during the OTA Activation (OTAA) process and involves a reception buffer overflow condition. This can occur when the size of the received data remains unchecked.

In contrast, CVE-2020-4060 is a Use-after-free (UAF) vulnerability. This leads to memory corruption on 32-bit machines and was found in the LoRaWAN Gateway implementation, specifically the LoRa Basics™ Station. This station uses a Configuration and Update Server (CUPS) protocol to check for updates. The UAF issue occurs when the signature length of a message from a CUPS server exceeds 2 GBytes.

2.3.3 Attacks

Many attacks exploit vulnerabilities in IoT network protocol stacks [35, 36]. These can affect service availability, data integrity, and confidentiality. Attackers can exploit vulnerabilities in the lower layers of a protocol stack This includes the MAC layer or the physical layer. Attackers can introduce malicious network packets to target victims or jam their communication channel, resulting in exploits as DoS, Man in the Midlle (MITM), and RCE attacks.

Table 2.4 provides a comparison of reported attacks on various protocols. It focuses on LoRaWAN, a sub-GHz protocol, and 2.4 GHz protocols like Bluetooth, BLE, and ZigBee. The comparison is based on the targeted protocols, exploited and targeted layers in the IoT protocol stack, as well as the used vulnerabilities and their exploits.

In the following paragraphs, we provide detailed explanations of packet injection and jamming wireless attacks.

Packet Injection: An attacker, typically an unauthorized entity, introduces malicious packets into the target device or its communication pathway. Through packet injection attacks, the attacker aims to inject false data, take over legitimate device roles, and disrupt the network availability. Such attacks can pave the way for more advanced exploits including RCE and MITM attacks. Within the LoRaWAN structure, an attacker might position himself between a gateway and an IoT device, impersonating either the end-device

Table 2.4 – Security State-of-the-Art IoT Low Data-Rate Protocols (LoRaWAN, ZigBee, BLE)

Attack	Protocol	PHY	MAC	UL	Vulnerability	Exploit
Wazabee [37]	ZigBee	E	E/T	T	BLE API	DoS, injection
Selective Jamming [35]	LoRaWAN	E	E/T	T	Header plain-text	DoS, Wormhole
Spoofing [38]	LoRaWAN	E	E/T	-	Authentication	DoS
Rescuing [39]	LoRaWAN	-	T	T	Protocol weakness	Replay, DoS
InjectBLE [40]	BLE	E	E/T	T	Pairing	MITM, Sniffing
Downgrade [36]	BLE	-	-	T	Design flaws	DoS, MITM
Injection-free [41]	BLE	-	-	E/T	Limited Bounding	DoS, MITM
Downgrade [42]	BT/BLE	-	E/T	E/T	Insecure BLE clock	MITM

T (targeted layer), E (Exploited layer), UL (Upper layers).

or presenting a false gateway. Flooding the LoRaWAN environment with excessive data can trigger several exploits; for example deny service to legitimate devices, and because LoRaWAN operates in a low-power context, constant receiving or transmitting can lead to rapid battery reduction.

Prior studies have highlighted the viability of various exploits associated with packet injection attacks in IoT protocols. The chirpOTLE framework, designed for evaluating LoRaWAN security, is detailed in [38]. The tool provided allows the reproduction of various LoRaWAN attacks that affect its availability and integrity, including wormhole and replay attacks. This involves capturing and retransmitting data packets using two LoRa transceivers. The first transceiver, acting as an entry node, is positioned near the target End Device (ED), while the second transceiver, functioning as an exit node, is situated close to the gateway (GW). When the entry node captures a complete LoRa frame, it sends it to the exit node via an out-of-band channel, which then replays the message for the GW to process. In this setup, the attacker can modify the message metadata, especially by introducing incorrect data regarding the timing, location, SNR, and RSSI values. The security protocol of LoRaWAN 1.0, as thoroughly analyzed in [39], exhibits multiple vulnerabilities. The authors delved into attacks on LoRaWAN that compromise data integrity, confidentiality, and network availability. Highlighted attacks capture LoRaWAN's join procedure frames and subsequently execute exploits, including desynchronization of legitimate end-devices, replay attacks, and frame decryption.

Attacks similar to those found in LoRaWAN also plague Bluetooth, BLE, and ZigBee

stacks. The authors of [40] introduced InjectBLE, an attack that can insert arbitrary frames into a pre-established BLE connection. This can lead to various exploits, such as the hijacking of master-slave roles and MITM attacks. InjectBLE leverages a feature in the BLE specification that allows devices to adjust their reception windows to account for clock inaccuracies. By exploiting this, InjectBLE carries out a race-condition attack, permitting an attacker to inject a frame at the start of the reception window. This vulnerability exists inherently in the BLE specification, regardless of how the stack is implemented. In [37], the same authors introduce the "wazabee" attack, which manipulates the radio device within a BLE chip to transmit and receive 802.15.4 frames, specifically ZigBee frames. This exploitation hinges on the similarities between the physical layers, notably the GFSK and O-QPSK modulations utilized by both the BLE and ZigBee protocols. In [41], researchers pointed out a vulnerability in the BLE stack implementation linked to its bonding list—a storage mechanism for cryptographic keys from prior bonded devices. When this list is full and a new bond request is made, the existing key is replaced, affecting its associated device. Attackers can exploit this by injecting packets from a new device and filling up the list, prompting the BLE device to discard all legitimate keys and forcing genuine devices to rebond. Some BLE versions, when faced with a full list, may decline new bonds or permit insecure connections, opening up potential security risks, including DoS attacks.

Packet injection attacks often exploit vulnerabilities found in the lower layers of IoT protocol stacks. These vulnerabilities may arise from the protocol's specifications or implementation.

Jamming: Jamming is a significant security concern for IoT networks. Even systems with robust high-level security can be compromised. Essentially, jamming disrupts wireless signals, either intentionally, as with radio frequency interference, or unintentionally, from noise and receiver collisions. The aim of the jammer is to dominate the channel and block legitimate nodes. There are two main types of jamming: continuous and triggered [43]. During continuous jamming, the attacker persistently interferes with the channel. In triggered jamming, interference occurs only upon meeting specific conditions such as preamble detection.

Recent studies, such as those referenced in [44, 45], have highlighted the vulnerability of IoT protocols such as BLE, ZigBee, and LoRaWAN to jamming. The rise of affordable SDR platforms, coupled with open-source software, such as [8], facilitates access to

frequency ranges used by BLE and LoRaWAN. A refined jamming technique, known as selective jamming, has emerged. Here, the jammer disrupts communication after decoding the MAC header and end-device address. This precise attack can isolate and block a specific device in an IoT environment without affecting the other network devices.

The study in [35] showcases selective jamming attacks that target LoRa and LoRaWAN transmissions. The vulnerability arises from the protocol's extended packet air time, giving attackers ample opportunities to detect specific messages and emit jamming signals during the broadcast of the original message. A similar threat exists in BLE stacks. As noted in [46], researchers have created a selective jammer for BLE advertising using affordable, readily available hardware. The jammer can target specific beacons with unique device addresses. Given that BLE beacons operate on various advertisement channels, the designed jammer includes a discovery component that scans all channels and identifies those in use using targeted beacon sources.

2.3.4 Security Mechanisms

Many academic and industrial studies have been conducted on IoT security mechanisms. They focused on detecting various types of attack and intrusion. Security mechanisms typically address one or more elements of the CIA Triad. This included confidentiality, integrity, and availability. The proposed mechanisms may be software- or hardware-based, or software that utilizes hardware acceleration.

In [17], the authors implemented a hardware dynamic information flow tracking (DIFT) architecture for RISC-V processor cores. This DIFT aims to detect memory-corruption attacks, such as buffer overflows and format strings. These mechanisms require modifications at the compiler level. Architectural changes, particularly inside the pipelines, are also necessary. Other countermeasures include memory protection with a safe programming language, such as RUST [47]. The authors of [48] also performed code instrumentation. They added tags to the memory locations for each memory allocation. They used additional tag-checking instructions to find illegal accesses for all memory accesses. Such mechanisms usually require memory layout changes, leading to memory overheads. The tools for static analysis can detect bugs during the compilation stage.

Intrusion and anomaly detection approaches [49, 50] have been proposed for IoT environments. They detect attacks using a signature list or by learning the legitimate behavior of a system. This solution comprises three main modules: acquisition, analysis, and alertness. Probes, in hardware or software, collect the system metrics in the acqui-

sition part. This information is then analyzed to identify ongoing attacks. If malicious action is detected, an alert warns the user. These mechanisms are highly accurate in detecting attacks. However, the detection rate performance and overhead, such as area, code size, and power consumption, pose challenges. Lightweight detection algorithms can help resource-constrained IoT devices overcome this overhead. In addition, moving remote analysis and detection algorithms to a server or gateway could be helpful. The end-device then only collects the metrics.

IoT SoC chip manufacturers typically include capabilities, such as cryptographic hardware acceleration for IoT stack protocols. In addition, they include true random number generators, memory encryption, and the ability to perform secure boot and firmware updates. They also have the ability to encrypt data stored in flash memory. These security mechanisms were designed to authenticate the integrity of the firmware. This provides important protection against unauthorized software.

Table 2.5 summarizes the categories of security mechanisms used and unused by the industrial IoT SoC. SoC chip manufacturers for the IoT commonly provide protection and update mechanisms. However, monitoring and detection mechanisms such as intrusion detection systems are usually not included.

Table 2.5 – Industrial IoT SoCs Security Features Comparison

Security Mechanisms		[10, CC1352R]	[9, STM32WL55]	[11, ESP32-H2]
Protection ✓	Cryptography	✓	✓	✓
	Code Authentification	✗	✓	✗
	Secure Boot	✓	✓	✓
	Memory Encryption	✓	✓	✓
Update ✓	Firmware Update	✓	✓	✓
Detection ✗	Flow Tracking	✗	✗	✗
	Intrusion Detection	✗	✗	✗
	Anomaly Detection	✗	✗	✗

Having explored and emphasized security threats associated with wireless connectivity, along with recommended countermeasures, the next section shifts its focus to the field of IDS in this research. Subsequently, we delve into discussions regarding intrusion and anomaly detection approach categories, both established and proposed in research, while also addressing the challenges identified in the process of proposing an IDS solution for IoT devices.

2.4 Intrusion Detection System - IDS

2.4.1 IDS Overview and Taxonomy

An IDS is a widely used security mechanism. It monitors and analyzes network or system activities for potential vulnerabilities and attacks. An IDS comprises three main components: acquisition, analysis, and warning. The acquisition component, in the form of hardware or software probes, collects the system information. This can target the network, software, or hardware. The analysis component processes the collected information to identify ongoing attacks. Finally, if there is ongoing malicious activity, the warning component alerts the user.

IDSs often target availability-related attacks, such as DoSs and DDoSs. They can also detect other types of attacks, such as spoofing, packet injection, and MITM attacks.

Implementation and Placement Strategy: An IDS can be placed in an IoT environment such as a smart building as an IoT Node. It can also be embedded in an IoT device or a gateway. Typically, an IDS is implemented as software on these devices. There are two types of IDS: Network-Based Intrusion Detection System (NIDS) and HIDS. An NIDS uses radio probes within an IoT environment to monitor network activities. These activities include the traffic flow and packet headers. Some probes implemented with SDR platforms target multiple frequency bands. In terms of IDS placement, an NIDS is placed in an environment to detect ongoing attacks. An HIDS, on the other hand, can be placed in an IoT device, gateway, or both. It uses embedded probes on each device or gateway to collect and analyze the system data. These data include system calls, memory access, running processes, and MAC and PHY layer features, such as packet headers and RSSI.

Detection Methodology: There are four IDS detection techniques:

— Signature-based: Detects attacks using a matching system or network behavior to compare with an attack signature in the IDS database. An alert is raised if a matching activity is found.

— Anomaly-based: Identifies attacks by comparing legitimate and illegitimate activities. If a deviation over a certain threshold is found, the alert is raised.

— Rule-based: Discovers attacks based on deviations from manually defined specifications and thresholds for normal network component behavior.

— Hybrid-based: Combines signature-based, anomaly-based, and rule-based detection to leverage their advantages and minimize their weaknesses.

Metrics Layer: The metrics used by an IDS come from various layers of an IoT Device:

— Network: this includes the protocol stack with different layers (PHY, MAC, and upper layers)

— Hardware: this refers to the processor microarchitecture (pipeline, memory,...)

— Software: this encompasses the run-time or operating system (process, memory,...)

Figure 2.5 provides a summary of IDS taxonomy for an IoT environment. The proposed IDS taxonomy is based on several aspects. These include the placement and detection methodologies, metric layers, and targeted attacks.

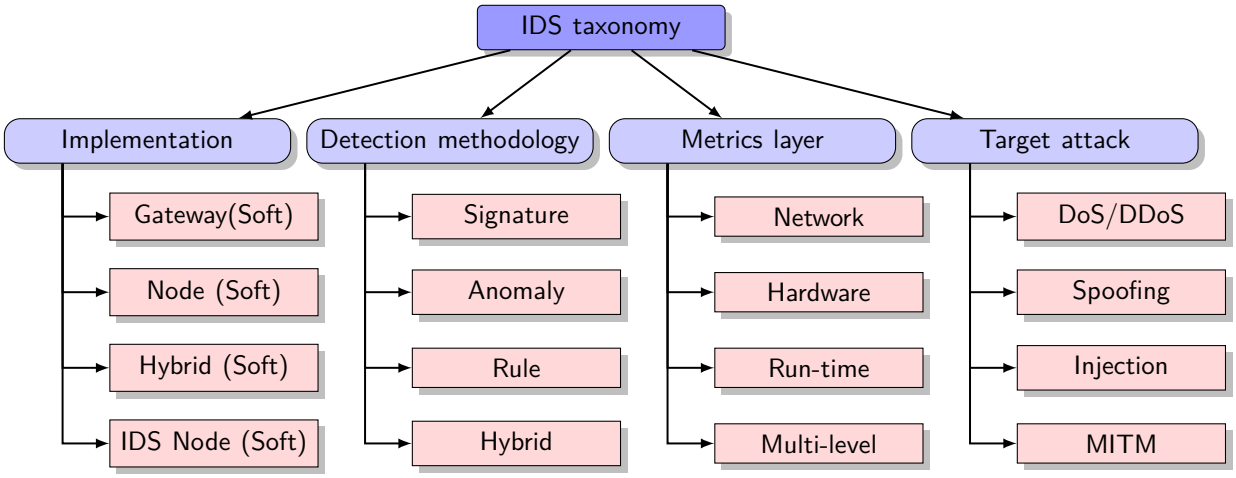


Figure 2.5 – IDS Taxonomy for IoT Environment

2.4.2 Network-Based IDS

The authors of [51] proposed a DoS detection architecture for IoT devices. This architecture captures and examines network packets using probes specific to a 6LoWPAN network. An IDS detects attacks based on signatures, primarily focusing on flooding attacks. However, the detection rate was relatively low when a single attack probe was used. The detection performance improves when more probes are involved in the same attack.

In [52], the authors leveraged the flexibility of the SDR to target multiple protocols across different ranges. They introduced a demodulation-free Radio-IDS (RIDS) that is

anomaly-based. It uses PHY layer network metrics to detect availability attacks, such as jamming and DoS, as well as integrity attacks, such as MITM, in IoT environments. They constructed a model of legitimate behavior using machine learning, considering factors such as the RSSI and frequency patterns captured from smart home devices using distributed SDR probes.

MedMon, as presented in [53], employs a multilayered anomaly detection system. This system was specifically designed to detect malicious transactions in implantable and wearable medical devices (IWMDs). SDR probes were used to intercept packets and measure the RSSI. Besides these metrics, MedMon incorporates time-series metrics from the MAC and application layers, such as the Time of Arrival (TOA), Differential Time of Arrival (DTOA), and Angle of Arrival (AOA). However, MedMon's focus is strictly on device integrity and does not address availability issues such as jamming attacks.

Table 2.6 provides a comparative analysis of the NIDS discussed in this section. The comparison is structured around several key aspects: the placement strategy, network metrics used, detection methodology, and types of attacks that can be detected.

Table 2.6 – Network-based IDS for IoT

NIDS features	[52]	[51]	[53]
PHY Metrics	RSSI, Frequency	✗	RSSI
MAC Metrics	✗	Packet	✗
Application Metrics	✗	✗	Transmission schedule
Attacks	DoS, Jamming, MITM	DoS	Injection
SDR Probes	✓	✗	✓
Detection	Behavior	Behavior	Behavior
Placement	IDS Device	Gateway	IDS Device

An NIDS with probes is effective for monitoring traffic flow and packet exchange, particularly in small environments such as smart homes. These IDSs can detect ongoing attacks, especially those that affect network availability, based on normal behavior patterns and attack signatures. However, monitoring radio activities alone is insufficient for accurately assessing the IoT security levels. Metrics at the host level, coupled with network metrics, are required to verify intrusions and minimize false positives. An NIDS may struggle to monitor nodes during an attack that compromises a portion of its network. Thus, it is necessary to add more probes to monitor the network activity across the entire environment. However, this leads to significantly increased system costs and

management complexity.

2.4.3 Host-Based IDS

An HIDS [50, 54, 55] incorporates software and/or hardware probes embedded within the system. The HIDS monitors not only network activity but also additional hardware and runtime metrics on IoT devices.

Physical layer IDS (PHY-IDS) [56], an RSSI-based IDS framework, was presented to identify body movement spoofing attacks on wearable devices. First, the system builds a legitimate behavior model using the RSSI time-series data features. This model is then used to spot frames that deviate from the regular wireless signal patterns of legitimate wearable devices. The PHY-IDS can be located in a hub, such as a smartwatch or a smartphone. The experimental results demonstrate that PHY-IDS has an average detection accuracy of approximately 99.8% for naive attacks. However, comprehensive knowledge and advanced learning capabilities are still required to counter the most sophisticated spoofing attacks. Other research works utilized MAC layer metrics, such as packet headers, to detect DoS and jamming attacks. In their work, the authors of [57] proposed Demo, an IDS framework tailored for a 6LoWPAN-supported IoT environment. However, Demo can only be integrated into the gateway and requires additional probes to sniff packets from the IoT devices. [58] is another research that focused on analyzing packet headers to detect spoofing attacks in 6LoWPAN networks. This was achieved using SVELTE, an HIDS integrated with a mini-firewall for IoT devices. SVELTE employs a hybrid detection method based on both signatures and anomalies as well as a hybrid placement strategy. With eight nodes per 6LoWPAN network, SVELTE achieved a high detection rate of approximately 90%. However, as the number of nodes increases, detection accuracy is affected. For example, with 32 nodes, the detection rate dropped to 70%.

The authors of [50] utilized metrics from both the MAC and physical layers. For example, Passban IDS uses anomaly-based detection with RSSI and packet headers to identify malicious packet injections in Linux-based IoT gateway devices. It is capable of detecting various types of malicious traffic, such as port scanning, HTTP and SSH brute-force attacks, and SYN flooding. However, the assessment of Passban IDS has a considerable impact on the IoT gateway: it incurs a 6% memory overhead, increases CPU usage by 30%, and reduces network speed by 7%. This suggests that it is unsuitable for embedding in smaller devices. Certain hardware metrics can be utilized to implement an

HIDS that can detect attacks, particularly those that compromise integrity.

Bourdon et al., in [59], propose a Hardware Performance Counter (HPC) based anomaly detection system to detect packet injection and DoS attacks. They selected seven HPC registers that reflect the CPU state (memory cache, instructions, exceptions, prediction branches, bus access) to create an application-independent mechanism for detecting malware. A model of legitimate device behavior was generated using machine-learning algorithms and data from HPC registers. The IDS employs a hybrid placement strategy. A kernel module, acting as a tracer, was installed on each device. This tracer records time-series data for each HPC register in a local file. This local file is then transmitted to the server using FTP communication to be further analyzed using machine-learning algorithms. The performance of the HPC based IDS was evaluated using various machine-learning detection algorithms. The detection rate accuracy was measured using two metrics: true positive rate (TPR) and false positive rate (FPR). Overall, with a maximum of 1% compromised devices, the TPR was approximately 80% and the FPR was less than 1%. However, the detection rate declined when 5% of the total number of devices were compromised, indicating that identifying intrusions in the case of widespread attacks poses a considerable challenge.

In [60], the authors introduced a framework for creating embedded detection software for devices using a BLE protocol. This detection module is built on the instrumentation of the MAC layer (Link layer) within the BLE stack. The detection method relies on specific rules applied to network metrics. These rules involve setting threshold values for the Cyclic Redundancy Check (CRC) validity number and the advertising interval. The CRC thresholds are particularly useful for identifying jamming attacks, whereas the advertising interval time helps detect other types of attacks related to the BLE protocol, including MITM attacks. Several experiments have demonstrated that this method can be used to identify existing BLE attacks. Using the instrumentation of stack code combined with a rule-based approach shows promise as a technique for integrating detection methods directly into the existing stacks. However, this approach presents several challenges. First, rule-based detection can sometimes produce higher values for false positives and negatives. Attackers may even exploit the known rules to bypass them during more complex attacks. Second, adding extra code for stack instrumentation can lead to performance issues, including increased execution times and memory costs. These limitations can be particularly problematic for IoT devices with constrained resources. Finally, implementing this approach requires a thorough understanding of the wireless communication protocol

stack. Specific knowledge is required regarding where to insert additional functions for the monitoring and analysis modules. Overall, this study introduces an innovative but complex method to enhance the security within the BLE protocol.

Table 2.7 summarizes state-of-the-art HIDS features. For each HIDS, we identify its type and placement, the context including protocols and attacks it can detect, and finally, the metrics (network and hardware) utilized by each solution. The key features of our proposed approach, *Diwall*, are listed for comparison, more details are provided in the next chapter.

Table 2.7 – Diwall and Related Works for HIDS in IoT

HIDS			**Context**		**Metrics**	
Ref.	**Detection**	**Place**	**Protocols**	**Attacks**	**Network**	**HW**
[56]	Anomaly	Gateway	IEEE 802.15.4	Spoofing	RSSI	-
[57]	Signature	Gateway	6LoWPAN	Flooding, Jamming	Packet	-
[50]	Signature	Gateway	BLE, WiFI	HTTP/SSH brute force, ...	Packet	-
[58]	Hyprid	Hybrid	6LoWPAN	Routing, Spoofing,...	Packet	-
[59]	Anomaly	Hybrid	TCP/IP	SSH brute force, DDoS, ...	-	HPC
[60]	Rule	Node	BLE	Jamming, MITM	advInterval, CRC	-
Our	**Anomaly**	**Node**	**LoRaWAN**	**Jamming, Injection**	**RSSI**	**HPC**

RSSI (Received Signal Strength Indicator), advInterval (Advertising Interval),
MITM (Man-in-the-middle), CRC (Cyclic Redundancy Check), HW (Hardware), HPC (Hardware Performance Counter).

Software-based IDS solutions are commonly implemented in wireless network environments. Their popularity stems from their flexibility and ability to be reconfigured, making them a preferred choice in many situations. However, this approach is challenging. Implementing additional software can lead to performance overheads. It may also introduce new security risks owing to the potential software vulnerabilities. However, hardware-based IDS solutions are implemented infrequently. These systems offer advantages such as lower performance overhead and reduced potential for additional security risks. Despite these advantages, hardware-based IDS solutions are not as commonly used as their software-based counterparts.

2.4.4 Identified Challenges

This study identified several challenges in IDSs for an IoT environment. Many solutions are focused on availability attacks (DoS, jamming), with a few addressing both

availability and integrity (packet injection). The placement strategy is often restricted to the gateways or servers. This leaves resource-constrained IoT devices without built-in analysis and detection mechanisms. Therefore, placement strategies often lean towards hybrid or gateway solutions, particularly because IDSs use machine learning. Such systems require large areas, high-performance computing, and considerable energy consumption. Hybrid implementation is due to the limited resources on the IoT device, including energy, memory, and computing. Additionally, they require memory storage, as most IDS solutions are software-based and resource-constrained IoT SoCs that only have hundreds of kilobytes of memory. Therefore, lightweight machine learning detection methods are required. IoT devices with limited resources typically have only tracers for event monitoring and data transmission. With the analysis and detection modules placed on a server, real-time detection of IoT device intrusions becomes a challenge.

Some IDS mechanisms provide solutions customized for a single IoT network protocol. This excludes the wide range of IoT devices with multi-protocol wireless connectivity. We have noted that most solutions also use single-level metrics, such as networks, hardware, or runtime. Only few studies have used a multi-level approach. An IDS based on a single-level metric can enhance detection efficiency and relevance. However, an attacker may exploit unmonitored levels, and the system can produce high false-positive rates. A deviation from a legitimate signature or behavior can be interpreted as an intrusion. Deploying a multi-level IDS for resource-constrained IoT devices is an interesting challenge. Such a system should detect both availability attacks, such as DoS, and integrity attacks, such as injection. They should also be suited to devices with limited processing capacity and low power consumption, especially those running on batteries.

In designing HIDS security mechanisms for IoT SoCs, where resources are particularly limited, it is essential to meet three fundamental requirements:

Req1 - Lightweight & Local Analysis: An HIDS should integrate its monitoring and detection modules directly into the IoT SoC, eliminating the need for remote analysis and detection. In addition, this integration should ensure minimal overhead in terms of the energy, area, memory, and execution time.

Req2 - Multi-level Monitoring: Given the expansive attack surface of the IoT environment, it is essential for an HIDS to monitor data across multiple layers of an IoT SoC. This encompasses the network, software, and hardware components.

Req3 - Reconfigurability: An HIDS must be flexible with an evolving threat landscape. Its parameters should be easily adjusted to counteract emerging attack vec-

tors and should accommodate a range of protocols, making it ideal for multi-protocol IoT SoCs.

2.5 Summary

This chapter offers a comprehensive view of resource-constrained IoT SoCs in industry and research, with a specific focus on wireless connectivity unit architecture, its security challenges, and potential countermeasures. Wireless connectivity is managed by a network processor, and we discuss the flexibility, programmability, and power consumption of various designs employing different technologies, such as FPGA, hybrid FPGA, dedicated CPU, generic CPU, and dedicated ASIC modules. We also explored the security aspects of wireless connectivity, highlighting existing vulnerabilities and attacks found in various IoT stacks, such as LoRaWAN, Bluetooth, BLE, and ZigBee. It was noted that many IoT protocol stacks widely used by vendors and researchers are insecure, with vulnerabilities primarily linked to the implementation of lower IoT protocol stack layers, specifically the MAC and physical layers. Some of these vulnerabilities are directly related to IoT protocol standards. To address these vulnerabilities, several security mechanisms have been proposed by both the research community and chip manufacturers. Industrial solutions primarily offer two types of security services: protection mechanisms and secure update mechanisms. However, detection and monitoring mechanisms are absent.

Our discussion then turned to IDSs, a promising monitoring and detection approach for IoT SoCs suggested in the literature. We introduced a taxonomy for classifying IDS in IoT architectures based on the detection methodology, placement strategy, metric layers, and targeted attacks. Using this taxonomy, we classified state-of-the-art IDSs for both host- and network-based systems. We further discussed the challenges of using an IDS in resource-constrained IoT devices, including the placement strategy, detection performance, area overhead, and power consumption.

We conclude that future resource-constrained IoT end-devices are expected to incorporate built-in wireless connectivity. For further flexibility, reconfigurability, and programmability, this unit is expected to include an extended version of the existing generic CPU, a network processor capable of handling multiple IoT protocols under low-power conditions. Industrial solutions now incorporate various security modules within wireless connectivity, mostly related to protection and update mechanisms. However, there remains a clear need for monitoring and detection mechanisms that can provide real-time

data tracing and attack identification within IoT SoCs.

The following chapter introduces a Host-based IDS (*Diwall*) approach designed to protect against jamming and packet injection wireless attacks. In-depth coverage is provided to the proposed approach, including the methodologies used for studying and detecting wireless attacks within our defined threat model.

Chapter 3

Diwall: Host-based Intrusion Detection System

Introduction

IoT SoCs are often subjected to various forms of wireless attacks that pose significant threats to their availability, integrity, and confidentiality. Among the major attacks they face, packet injection and jamming are the most prevalent. In this chapter, we discuss *Diwall* specifically designed to detect these attacks. This chapter is organized into four sections. The first section provides details about our threat model. The second section details and discusses our approach with respect to related works mentioned in the previous chapter. Sections 3 and 4 explore our methodology and experimental framework for studying packet injection and jamming attacks. They offer a comprehensive look at our experimental setup, spanning both simulated and real-world environments, outline the process of generating our datasets, and specify the detection method used. The experimental results are presented in detail in the next chapter.

3.1 Threat Model

The vulnerabilities inherent in the lower layers of numerous IoT protocol stacks can be exploited by attackers to launch a range of attacks such as jamming and packet injection. These basic attacks can serve as the starting point for more sophisticated exploits. It includes DoS, MITM, RCE and Privilege Escalation.

The victim IoT SoC has a wireless connectivity subsystem that leverages one or several protocol stacks handled by the network processor. In addition, it has a CPU that is in charge of executing the user applications and the upper protocol stack layers. The attack surface of the IoT protocol stack mainly includes its PHY and MAC layers because these layers provide the first entry point for the attacker and are most sensitive to vulnerabil-

ities. The remote attacker can perform the attack on the victim's IoT system by using either an SDR platform or a protocol-specific dongle. An attack vector of the attacker includes packet injection and jamming, targeting the victim's wireless connectivity network processor. When conducting such attacks, the attacker attempts to compromise the system.

Our research work focuses on the wireless IoT attacker model as illustrated in Figure 3.1. In this figure, an IoT end-point victim initiates communication with the IoT gateway. The victim's wireless connectivity possesses vulnerabilities on its attack surface encompassing the MAC and PHY layers. Positioned between the communication, the attacker can inject a packet or jam the victim's communication channel. *Diwall* addresses this threat model by monitoring metrics at both the network and microarchitecture levels. This monitoring provides insights into the activity of the network processor and the traffic surrounding the victim.

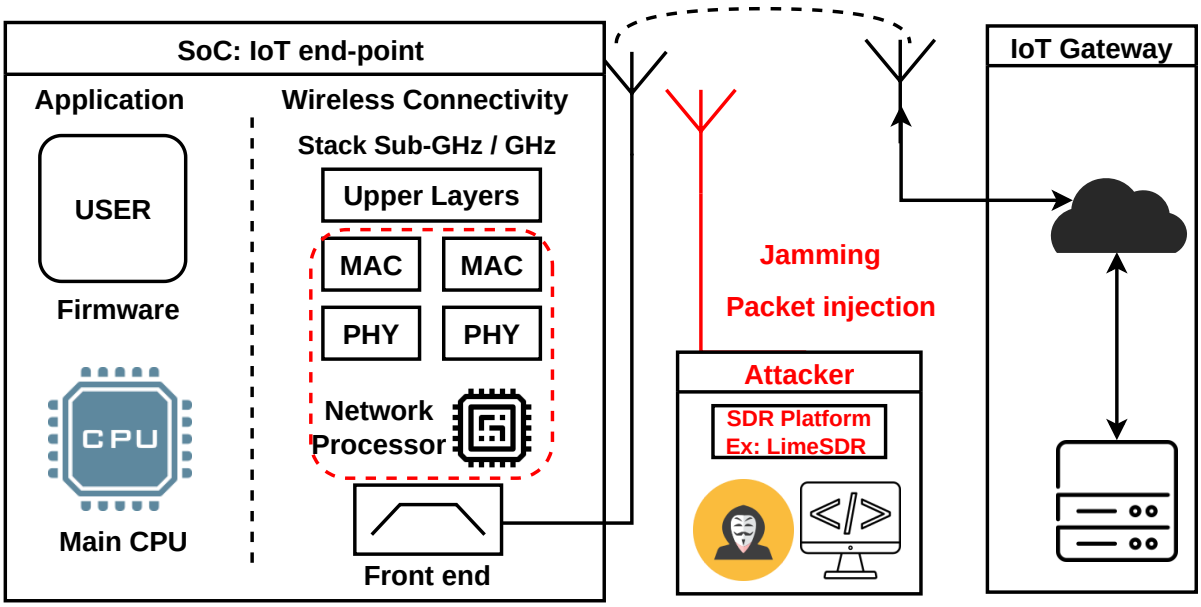


Figure 3.1 – IoT Wireless Connectivity with considered Threat Model

The next section provides a details about the proposed *Diwall* approach.

3.2 Proposed HIDS: Diwall

3.2.1 Key Features

In this study, we present *Diwall* that leverages existing HPCs on a network processor as hardware probes. A comparative examination of the *Diwall* approach with several notable methodologies found in the literature is presented in Table 2.7.

Diwall approach sets itself apart from the existing systems in several essential aspects. *Diwall* is entirely hardware-implemented and designed expressly for IoT SoCs with restricted resources, eliminating the need for remote detection through a gateway or server. Instead of relying heavily on software-based solutions, such as traditional strategies, our system is primarily implemented in hardware, with only a minimal number of control and configuration instructions in software. Our approach adopts both hardware and network metrics to determine the normal operation of the IoT SoC's wireless connectivity, diverging from the common practice of utilizing a single metric in most related studies. We utilized built-in CPU HPCs as monitoring probes, thereby avoiding extra area and performance overhead. We propose an extension to the network processor architecture by incorporating dedicated HPCs designed for the specific purpose of monitoring network metrics. Finally, *Diwall* can detect remote packet injection and jamming attacks that target the network processor of an SoC.

Diwall characteristics can be described as follows:

Detection methodology: Anomaly;

Placement strategy: IoT Node;

Protocols: LoRa PHY and LoRaWAN MAC layers;

Attacks: Jamming and Packet injection;

Metrics: Hardware (Microarchitectural) and Network (RSSI).

3.2.2 Overview

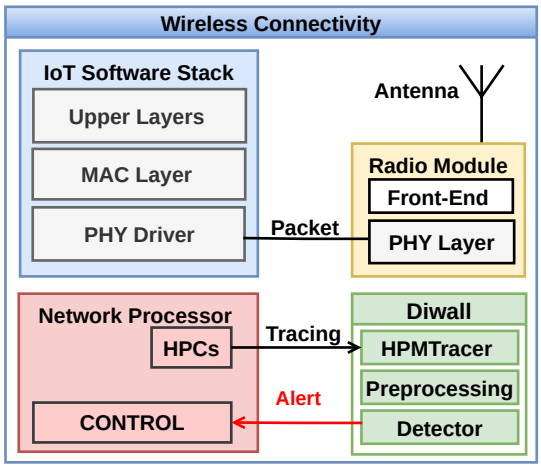
The wireless connectivity of an IoT SoC device can be depicted using the highlighted block diagram in Figure 3.2a. This representation was divided into hardware and software components. The hardware wireless connectivity encompasses the radio module, which includes the PHY layer, Front-end, and Antenna, along with a network processor responsible for managing the radio module and IoT software stack. On the other hand,

the software wireless connectivity comprises layers of the IoT software stack that operate on the network processor. It involves a PHY driver for handling the PHY layer, a MAC layer responsible for processing incoming packets from the PHY layer, and upper layers specific to individual applications for each IoT protocol stack. In the present study, we focus solely on software implementations of the MAC layer, while recognizing that MAC layers can also be implemented using dedicated hardware.

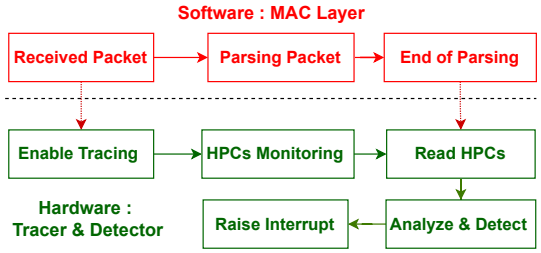
We propose to secure wireless connectivity with *Diwall* for attack detection. *Diwall* directly performs monitoring from the network processor, data analysis, and raising alerts. It has three main hardware units coupled with wireless connectivity. Hardware Performance Monitoring Tracer (HPMtracer) is a hardware tracer used to monitor the HPC data of the network processor. A preprocessing unit is used to prepare data before the detection. The detector, is a hardware unit used for data analysis, and decision-making based on a legitimate model behavior. *Diwall* targets multi-level metrics, and both network and microarchitecture levels are tracked. The network metrics are stored in a new dedicated HPC that we propose to add to the network processor. Microarchitectural events occur when a network packet is parsed by the MAC layer, and they are monitored by HPCs.

Figure 3.2a presents a simplified view of *Diwall* coupled with the block diagram of wireless connectivity. *Diwall* connects to the network processor via two signal blocks: Tracing and Alert. Tracing signals establish a link between HPCs in the network processor and HPMtracer in *Diwall*. This signal includes essential control commands for enabling or disabling *Diwall* from software. Alert signals are used to convey the decision-making by *Diwall*. An alert is generated by *Diwall* through its detector after analyzing the HPC data. This alert is closely tied to network processor control and managing interrupts, according to a user-defined security policy. This policy determines the actions to be taken following attack detection.

The processing flow for *Diwall* is shown in Figure 3.2b. The network processor software runs the MAC layer of the IoT protocol stack and analyzes the incoming packets from the PHY layer. Simultaneously, *Diwall* hardware monitors the metrics using the HPMtracer. Then, the metrics are processed using a dedicated detector model for identifying specific attacks. The cumulative values of microarchitectural events provide an indicator of the network processor's behavior during packet parsing. The value of network metrics, such as RSSI, provides an indicator of the network traffic received. The output value is then compared with a threshold, and an alert can be raised.



(a) Wireless Connectivity and Diwall Block Diagram



(b) Flow diagram of Network Packet processing, HPC Monitoring and Detection by Diwall

Figure 3.2 – Diwall Approach

Figure 3.3 illustrates a chronology example of the IoT stack activities during network packet handling and *Diwall* operations. A network packet undergoes several stages within an IoT stack. Upon successful reception in the PHY layer, the packet was transferred to the MAC Layer. Here, the packet undergoes MAC processing, which can include cryptographic security and preprocessing of the upper layers. After MAC processing, the application layer exploits the packet, as indicated in the chronogram. *Diwall* approach monitors the microarchitectural metrics and RSSI network metadata for a window from MAC processing. Its activation is confined to this window, as highlighted in chronogram in Figure 3.3. For both packets N and N+1, *Diwall* is exclusively enabled during the monitored window. *Diwall* remains inactive for the other segments of the IoT stack.

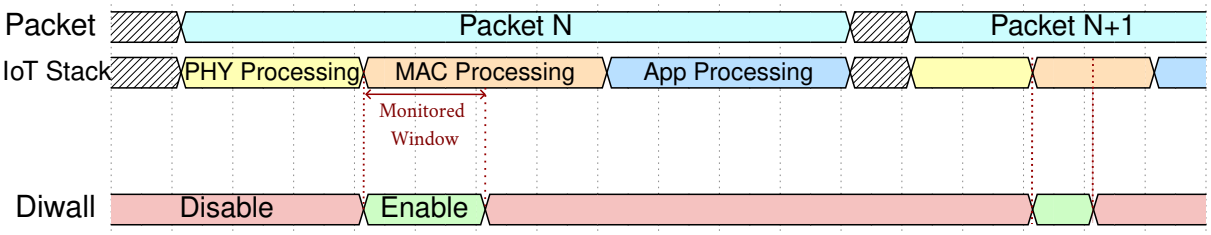


Figure 3.3 – Example : Diwall Chronogram with IoT Stack

3.2.3 Target CPU

The CV32E40P (formerly known as RI5CY), a compact and efficient 32-bit, in-order RISC-V core with a 4-stage pipeline, serves as the CPU core [61]. The Open Hardware Group (OpenHW Group) implemented the RV32IM[F,Zfinx]C[Zce] 32-bit RISC-V instruction-set architecture. The core features a robust implementation of hardware performance events and counters, compliant with the RISC-V Privileged Specification [62]. Notably, the CV32E40P has been comprehensively verified, attaining 100% code coverage and has been optimized for high performance and energy efficiency, making it a common choice for low-power applications [19, 20]. Furthermore, the core demonstrates competitive performance on several benchmarks, such as Embench and CoreMark, when compared with 21 other existing RISC-V cores [63, 64]. The Embench individual test results fall within the range of 24% to 65% relative to Cortex-M4. Figure 3.4 (image source extracted from OpenHW Group documentation in [61]) provides a block diagram of the top level, showing the core and the FPU.

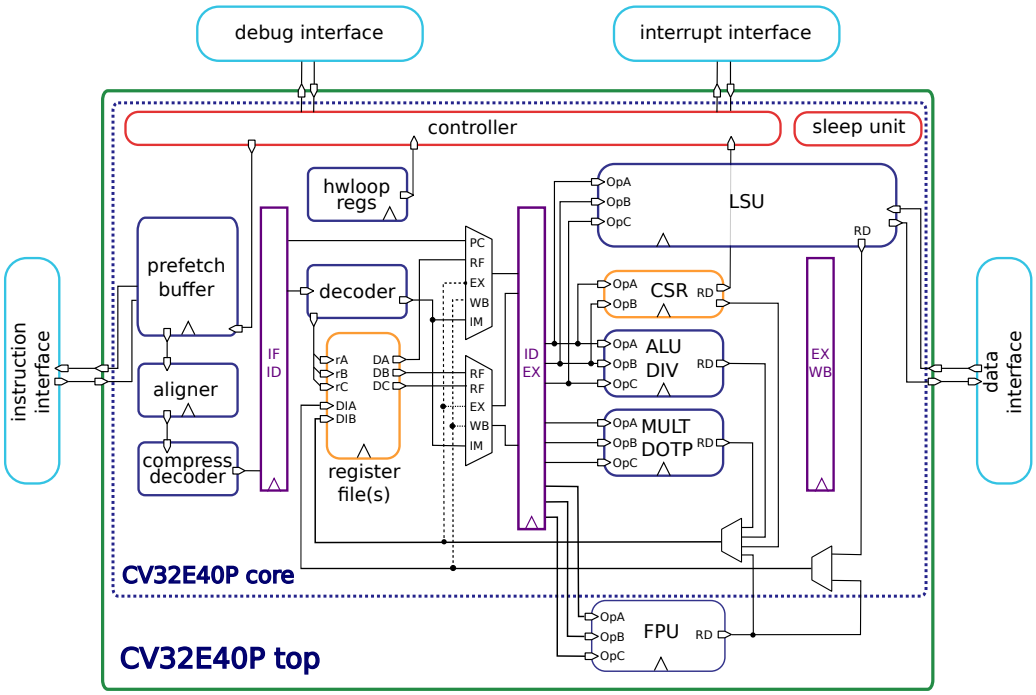


Figure 3.4 – Block Diagram of CV32E40P 32 bits RISC-V Software Core

Our choice of RISC-V, specifically the CV32E40P core, provides flexibility for our research, enhanced by the comprehensive toolset and rich documentation offered by the

RISC-V community. However, employing the RISC-V ISA or CV32E40P is not a prerequisite for *Diwall*. In fact, *Diwall* methodology can feasibly be applied to other modern processors, as various ISAs implement HPCs and share similarities in microarchitectural events.

3.2.4 Metrics

Microarchitectural Events

HPCs offer developers access to detailed low-level information about the processor's microarchitecture and memory access. These resources can serve a variety of uses, including benchmarking, debugging, and security applications. Several studies have used the capabilities of HPCs to develop robust security mechanisms, particularly for IDSs [65,66]. Authors in [67] have identified various challenges, pitfalls and risks associated with the use of HPCs in security defence. These challenges cover issues such as noise measurements, the non-deterministic nature of microarchitectural events and the sampling method used. The identified issues are particularly present in complex systems that are based on operating systems. In these systems, multiple applications and processes run concurrently, which can impact HPC event monitoring and lead to over- or under-counting of cumulative values. In addition, adversaries can exploit a vulnerable process to intentionally manipulate HPCs. In our work, we adopt a strategy of employing HPCs within the framework of a bare-metal application. While noise may still originate from peripherals like hardware interrupts or timers in this context, the advantageous aspect is that the running application wields significant control over the hardware. This level of control serves to mitigate issues arising from noise sources during the execution process. Furthermore, instead of the sampling method, we employ the polling method. This alternative method involves instrumenting the code within the software and reading the HPCs at the end of the monitored window.

The CV32E40P includes a set of 64-bit HPCs within Control Status Register (CSR) unit. It implements a clock cycle counter and a retired instruction counter, which are always activated, and 29 configurable event counters. Microarchitectural events are assigned to an event counter using an event selector CSR, by setting its value to the event's ID. The 29 configurable HPCs are disabled by default. The number of available HPCs can be controlled using the NUM_MHPMCOUNTERS parameter. By default, NUM_MHPCOUNTERS is set to 1. An increment of 1 to NUM_MHPCOUNTERS

results in the addition of 64 flip-flops (FFs). The RISC-V ISA provides a number of microarchitectural events, but the CV32E40P only implements a short list of these. the CV32E40P tracks using HPCs a set of microarchitectural events listed in Table 3.1. The ID column represents the selection bit used for assigning events to counters using the CSR event selector.

Table 3.1 – List of Hardware Microarchitectural Events Monitored by the CV32E40P HPCs

Microarchitectural Events		**ID**
CYCLES	Number of cycles	0
INSTR	Number of instructions retired	1
LD_STALL	Number of load use hazards	2
IMISS	Cycles waiting for instruction fetches	3
LD	Number of load instructions	4
ST	Number of store instructions	5
JUMP	Number of jumps (unconditional)	6
BRANCH	Number of branches (conditional)	7
BRANCH_TAKEN	Number of branches taken (conditional)	8
COMP_INSTR	Number of compressed instructions retired	9

Network Metrics

Several metadatas are available on the lower layers of IoT protocol stacks. Mainly at the PHY layer and others at the MAC Layer. These metadatas have been used for performance monitoring and wireless communication management. Various PHY layer metadatas have also been used for security mechanisms [52]. In *Diwall*, we chose to study RSSI and signal-to-noise ratio (SNR) metrics, both of which are frequently employed in wireless communications. These metrics provide information regarding the strength and quality of a signal during transmission or reception. Our decision to incorporate the RSSI and SNR metrics aims to ensure that *Diwall* remains independent of any specific IoT protocol stack. Given the ubiquity of these metrics across the most common IoT protocols, their use enhances the scalability and flexibility of *Diwall*, enabling compatibility with various protocols. Within the targeted CPU of our network processor, we configure a dedicated network HPC, Network Hardware Performance Counter (NwHPC), specifically for RSSI and SNR monitoring. This 64-bit network HPC is used as a register to store the RSSI values in the lower 32 bits and the SNR values in the upper 32 bits for each

received packet. This register can then be accessed by *Diwall* for further analysis and decision-making.

In the next section, we present a study on packet injection attacks considered in this threat model and its detection methodology.

3.3 Study of Packet Injection Attacks

Our research primarily focuses on the detection of packet injection attacks. An attack mainly consists of sending undesired data into a wireless network. It exploits memory vulnerabilities such as buffer overflows on memory locations in order to erase the return address of a function. This will result in a deviation of the program workflow and lead to DoS or pointing to the next return address to a malicious action. We aimed to develop a reliable behavioral model using classification machine learning algorithms, which were trained by processing aggregated microarchitectural metrics. The application of these algorithms focused on the parsing of network packets by the network processor, a common operation in the MAC layer of many IoT protocol stacks. This operation, which serves as the initial stage of data receipt from the PHY layer, is vulnerable to several forms of attacks, including memory corruption and packet injection exploits.

3.3.1 Simulation Experimental Setup

Our goal in using a simulation testbed is to produce a large dataset for training machine learning models. We simulated diverse scenarios and conditions. Simulations can help cut down the costs and complexities of getting real-world data for training and testing. The testbed subsystems is illustrated in Figure 3.5 and explained in the following paragraphs.

SoC: Hardware We used a LiteX SoC builder to construct the SoC, as described in [12]. The network processor is a RISC-V soft-core CV32E40P, which also features RAM and a UART. All elements were interconnected via a Wishbone bus. The HPMtracer part of the *Diwall* block diagram in Figure 3.2a is directly coupled to the CPU core through the CSRs signals. It records the process behavior of parsing packets on a network processor. This is accomplished by accumulating the selected microarchitectural metric values during network packet parsing. The HPMtracer in our demonstrator is accessible from the MAC layer software. This enables monitoring based on specific security policies.

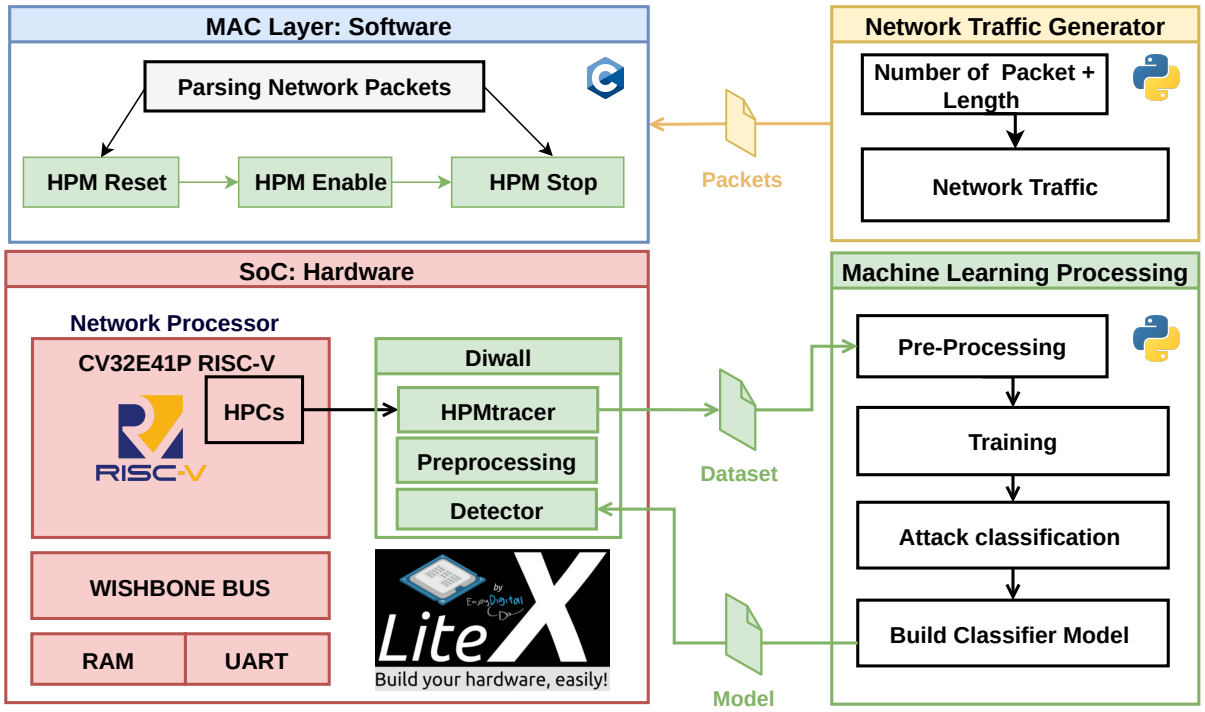


Figure 3.5 – SoC of Wireless Connectivity and Network Processor Testbed for Training

The following signals were set to control the HPMtracer from the software:

— HPM_Reset: Reset the HPCs;

— HPM_Enable: Assign events to counters and start the monitoring;

— HPM_Stop: Stop the monitoring.

Software The software component includes the firmware executed by the network processor. This software runs specific parts of a full IoT protocol stack, including the MAC layer, as illustrated in Figure 3.2a. We developed a MAC layer packet parser in C language. This represents a simplified MAC layer for the IoT protocol stack. To monitor network packet parsing, we instrumented the Reception Function as represented in Algorithm 1. The HPMtracer is controlled using three signals: HPM_Reset, HPM_Enable, and HPM_Stop. With this instrumentation, HPMtracer tracks microarchitectural events while storing the payload in the MAC reception buffer.

Network Traffic Generator is a Python script used to create network traffic in the simulation. It is also employed to generate malicious packets based on various packet

Algorithm 1 Reception Function in simplified MAC Layer

```
1: procedure ReceptionFunction(payload)  ▷ Triggered by radio module interrupt
2:     ...
3:     Call HPM_Reset()                                  ▷ Reset HPCs to 0
4:     Call HPM_Enable()                  ▷ Select/Enable HPCs and enable Diwall
5:     Recepetion_Buffer ← payload                       ▷ Getting MAC payload
6:     Metadata ← metadata                ▷ Getting Metadata value such as RSSI
7:     Call HPM_Stop()                    ▷ Stop HPCs, read and analyze by Diwall
8:     ...
9: end procedure
```

injection scenarios, as described in Section 3.3.2. The generated network data are stored directly in the network processor memory for subsequent parsing using a simplified MAC layer.

Machine Learning Processing This part of the testbed was operated offline for training. Python was used for post-processing the dataset. The main objective was to label the dataset and distinguish between legitimate and malicious packets. We then trained the machine learning algorithms for classification. The comparison of these algorithms was based on detection accuracy, precision, and recall.

After the algorithm training phase, we built a machine learning model. A dedicated testbed, as highlighted in Figure 3.5, is used for runtime and model tests. Subsequently, the model parameters were added to *Diwall*. *Diwall* will use the model to analyze data during runtime.

3.3.2 Packet Injection Reproduction Attacks

Our study on packet injection attacks focuses on those that exploit software vulnerabilities. These attacks primarily target memory corruption, causing a buffer overflow in the stack or heap. We replicated legitimate behaviors and overflow exploits as summarized in Table 3.2.

S0: Legitimate Packets This mirrors the typical behavior of a packet parser in the MAC layer of an IoT protocol stack. The size of the generated network traffic precisely matches the reception buffer of the MAC layer packet parser. The allocated memory is fully utilized but is not exceeded during network packet parsing. The traffic size of each

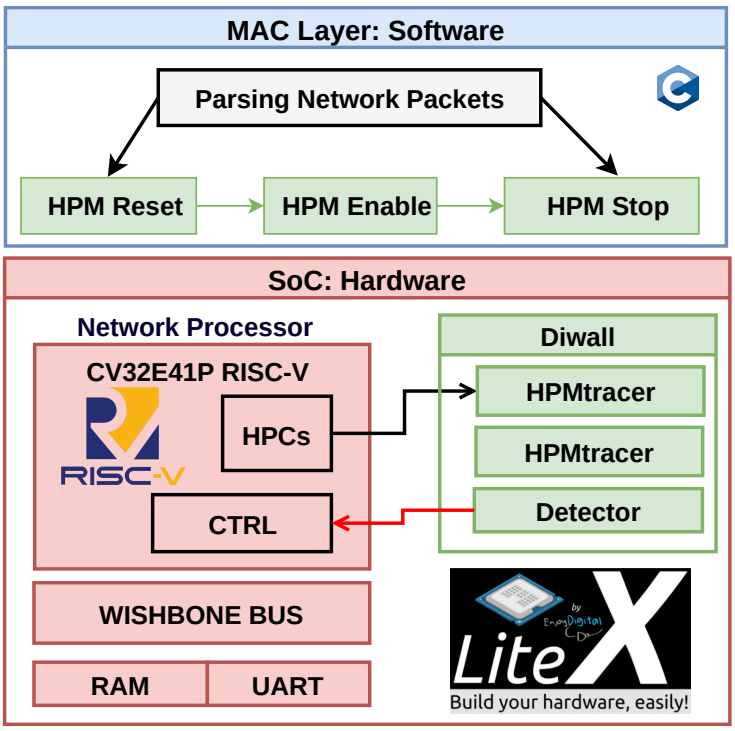


Figure 3.6 – SoC of Wireless Connectivity and Network Processor Testbed for Diwall Test and Validation

network packet ranges from 5 to 10 bytes, and the reception buffers on the heap and stack locations can accommodate up to 10 bytes. Utilizing the Network Traffic Generator module, we produced $1,000,000$ network packets that varied in size from 5 bytes to 10 bytes. Given that these packet sizes were within the capacity of the 10-byte reception buffer we allocated, these packets were able to fit perfectly into the buffer without causing any overflow issues.

S1: Stack Overflow Exploit The vulnerability in question, known as LoRaDawn [34] CVE-2020-11068, involves buffer overflow in the LoRaWAN protocol stack. This is because of insufficient checks on buffer sizes, and exploiting it could lead to a DoS on a LoRaWAN node. This weakness was studied by designating a 10- or 23-byte receiving buffer in the packet parser software, which is a software module that processes data encapsulated in network packets. In our experiment, we reproduced this vulnerability by allocating a reception buffer of only 10 bytes on the stack. We then generated $1,000,000$ network packets using a Network Traffic Generator module, with packet sizes varying between 13 bytes and 23 bytes. These packets, which are larger than the allocated buffer,

cause buffer overflow instances in the 10-byte reception buffer, demonstrating the potential risks of this vulnerability.

S2: Heap Overflow Exploit A heap overflow vulnerability, specifically CVE-2022-0204 [68] identified in the open-source Bluetooth stack, Bluez, was carefully studied. This vulnerability manifests itself in GATT protocol implementation. The vulnerability was reproduced by declaring a reception buffer in the heap through a *malloc()* function in the packet parser software. The methodology is similar to the previous stack overflow scenario, employing packet sizes, as outlined in Table 3.2.

Table 3.2 – Attacks Scenarios: The buffer size is 10 or 23 bytes. Larger Packets result in a Buffer Overflow.

Attack Scenarios		**Buffer Size**	
Packet Type	**Traffic Size**	**Stack**	**Heap**
S0: Legitimate	$5 - 10$ *bytes*	10 *bytes*	10 *bytes*
S1: Stack Overflow	$13 - 23$ *bytes*	10 *bytes*	23 *bytes*
S2: Heap Overflow	$13 - 23$ *bytes*	23 *bytes*	10 *bytes*

3.3.3 Dataset Generation

In this study, a dataset of $3,000,000$ network packets is generated, as shown in Figure 3.7. These samples were collected from the testbed, as shown in Figure 3.5 and were categorized according to the scenarios detailed in Table 3.2. The dataset was structured using 10 different features defined by the microarchitectural metrics listed in Table 3.1. These features, found in a RISC-V-based network processor, are essentially a collection of accumulated values from the HPCs. These values were subsequently utilized for machine learning classifier training.

The frequency histogram of the cumulative microarchitectural metrics for the 10 hardware events is shown in Figure 3.7. These HPCs were monitored using the HPMtracer after the parsing process of the network packets. Our generated dataset was categorized into three distinct groups: benign packets and malicious packets for heap and stack overflows. Figure 3.7 provides an overview of the features of the $3,000,000$ network packet samples in the context of stack overflow, heap overflow, and legitimate packet processing. The green results represent the normal behavior of the network processor when legitimate

packets are handled by the software in the MAC layer without any attack. The red and blue results, on the other hand, illustrate the network processor's behavior during stack overflow and heap overflow attacks. During buffer overflow conditions, the magnitudes of the HPCs increase compared with situations generated by the processing of legitimate packets.

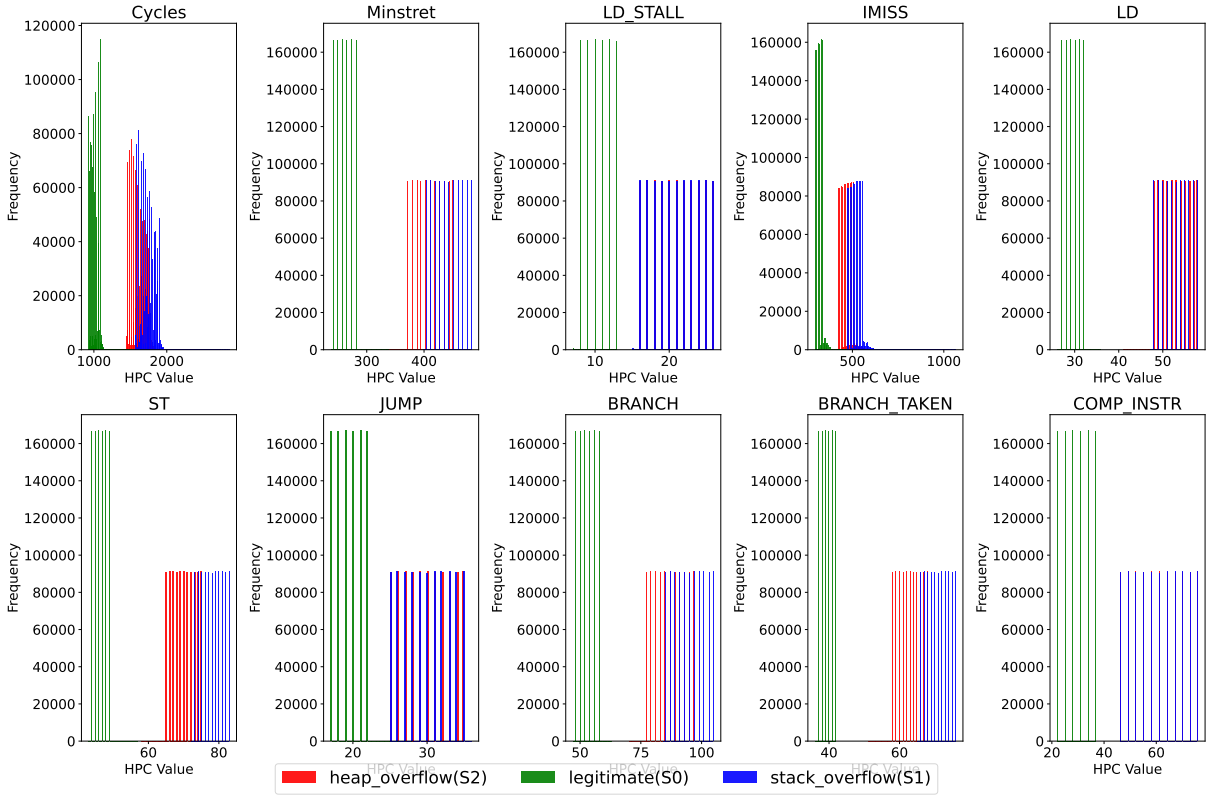


Figure 3.7 – Frequency Histogram of Microarchitectural Metrics monitored with HPM-tracer

The highest HPC values, shown in Figure 3.7, are the results of the software execution in Algorithm 1 on the network processor following the flow diagram and the chronogram illustrated in Figure 3.2b and Figure 3.3. The processing mechanism used in this study involved the use of two reception buffers. The first buffer is allocated to the stack, whereas the second buffer is dynamically allocated to the heap. On arrival, each network packet was stored in both buffers. However, if the size of a network packet exceeds the capacity of either buffer, buffer overflow occurs in either the stack or the heap. This overflow causes an increase in the event counter values, effectively leading to an increased number of cycles and instructions within the packet-parsing window. Consequently, the processor

experiences unexpected branches and jumps.

In Figure 3.7, we observe distinct patterns in the HPCs during buffer overflows. For the BRANCH event, the HPC count ranged from approximately 30 to 60 during the processing of the legitimate network packets. However, during heap and stack overflows, the counter increased to a range of $75 - 100$ branches. Similarly, for the JUMP event, the monitored HPC values range from around 5 to 22 under normal conditions but can reach up to 35 during buffer overflows. This behavior also extends to other microarchitectural events. Additionally, CPU memory access metrics affected by buffer overflow result in an elevated number of load and save operations. Consequently, this increase leads to longer delays when loading data from the memory. This is represented by the LD_STALL HPC event: a count between 5 and 13 during normal behavior, whereas in the presence of heap and stack overflows, the associated HPC counter increases to as high as 26.

As discussed in [67] the non-deterministic nature is a significant challenge associated with microarchitectural events. In this experiment, we used over a million packets in the legitimate class, with sizes ranging from 5 to 10 *bytes*. The cumulative HPC values for the entire million packets remained consistent without any measurement noise being introduced during monitoring. In Figure 3.7, LD_STALL HPC is shown, with legitimate traffic represented by green bars corresponding to different packet sizes randomly selected between 5 and 10 bytes. The figure shows six packet types, with no significant variation observed other than minor fluctuations due to the random selection process. This observation confirms the absence of a non-deterministic nature in our case.

A quick examination of the entire dataset revealed that the behavior of microarchitectural metrics during memory buffer overflows can help identify remote packet injection attacks that exploit memory vulnerabilities. Various strategies can be considered for detecting malicious behavior. One straightforward approach is applying predefined thresholds to selected HPCs. In our research, we adopted a more comprehensive approach, regardless of the distinctive dataset. We aim to explore machine learning classification algorithms with lightweight hardware implementations. Machine learning strategies offer valuable tools for classifying normal and malicious behavior patterns.

3.3.4 Machine Learning Classification

For a better understanding of the relationship between the microarchitectural features and the behavior of network processor during parsing network packet we use supervised machine learning classification algorithms. In this study, microarchitectural features are

represented into three categories of network packet (legitimate, heap overflow or stack overflow). This dataset is labeled with previously-mentioned categories, which implies that it is a supervised dataset. Supervised datasets are used for supervised learning, where the goal is to learn a function that maps input to output based on example input-output pairs. In this case, the output is the network packet category (benign, malicious for heap overflow, or malicious for stack overflow), and the inputs are the features derived from the microarchitectural events. The machine learning classification algorithms are useful for better mapping the microarchitectural features to the class of network packet.

We used Scikit-Learn, an open source data analysis library, a gold standard for machine learning in the Python ecosystem. The Scikit-Learn provides a range of classification machine learning algorithms. We have trained various machine learning classifiers using our labeled dataset. We evaluated Accuracy, Precision, Recall and F1-Score performance metrics for each classification algorithm.

Accuracy measures the correctness of predictions by calculating the ratio of correct predictions to total predictions.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.1}$$

Where TP, TN, FP, and FN are defined as follows.

— **TP**: True positives, the number of truly malicious packets detected;

— **TN**: True negatives, the number of truly legitimate packets detected;

— **FP**: False positives, the number of false alerts or legitimate packets considered as malicious.

— **FN**: False negatives, the number of alerts not raised or malicious packets considered as legitimate;

Precision measures the correctness of positive predictions and indicates the proportion of true positive predictions among all positive predictions.

$$\text{Precision} = \frac{TP}{TP + FP} \tag{3.2}$$

Recall mesures the sensitivity or true positive rates(TPR).

$$\text{Recall} = \frac{TP}{TP + FN} \tag{3.3}$$

The F1-Score measures the harmonic mean of the precision and recall.

$$F1\text{-score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{3.4}$$

The histograms in Figure 3.8 show the evaluation results of a comparison of several classification algorithms using the generated dataset. The highest values of Accuracy, Precision, Recall and F1-Score, which is 100%, represent perfect performance of the algorithms, and lower values indicate poorer performance.

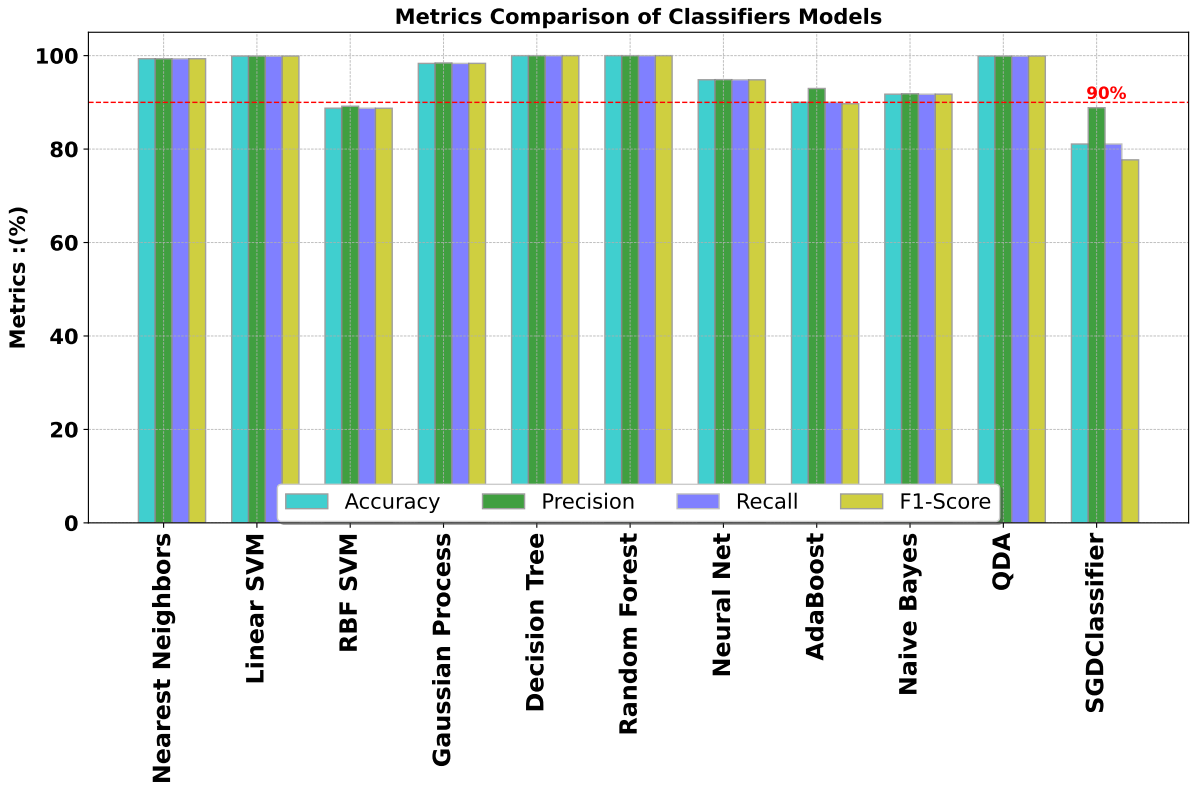


Figure 3.8 – Machine Learning Classifiers Comparison Accuracy

Preliminary results demonstrate the potential of using various machine learning algorithms to solve this classification problem. While the most commonly used algorithms provided superior results in terms of accuracy and precision, three of the 11 used only offered modest results. In the majority of the machine learning classifiers used, high accuracy and precision scores ranging from 91% to 99% were achieved, as highlighted in Figure 3.8. Only three classifiers–the SGDClassifier, AdaBoost, and RBF SVM–showed relatively low performance, with scores between 82% and 90%. The achieved performance

was owing to the distinctive nature of the previously gathered dataset. These datasets simplify the task of machine-learning classifiers to distinguish between benign and malicious network packets.

However, achieving high scores with machine learning classifiers is not the only criterion for selection. Because the target end-devices are resource-constrained, several challenges and requirements related to power consumption, code size, and hardware area overhead need to be considered. To this end, we target lightweight implementation strategies and requirements for machine learning classifiers suitable for IoT end-devices. Given that our target system is a resource-constrained IoT end-device, the feasibility of hardware implementation in terms of performance and overhead is a key selection criterion for the detection algorithm. Other parameters such as execution time, memory cost, and power consumption should also be considered.

Based on the results in Figure 3.8, Linear SVM, Decision Tree, Random Forest, and QDA constitute a shortlist of machine learning classifiers that consistently achieve near-perfect classification scores of approximately 99% in terms of detection Accuracy, Precision, F1-Score, and Recall. Considering the constraints and requirements of the targeted IoT end-devices, the chosen machine learning classifier must quickly profile and detect network packets using hardware events. Among the shortlisted machine learning classification algorithms, tree-based algorithms, such as Decision Tree, Random Forest, and AdaBoost, due to their simplicity, are suitable for a hardware-only approach. However, in a software implementation approach, they may pose an overhead in terms of execution time and speedup, as they entail more branches and require more time to classify data. Ultimately, we decided to proceed with the Decision Tree classifier because of its relative simplicity and rapid classification capabilities in a hardware-implementation setting.

3.3.5 Decision Tree Classifier Model

Our primary research objective was to identify ongoing packet injection attacks using microarchitectural metrics. Certain microarchitectural features are more pertinent for detection using machine learning classifiers. For this purpose, a supervised learning method that employs training samples to construct decisions within a tree model was considered. To reduce the area overhead within the network processor, only two HPCs from the microarchitecture were used. We performed feature selection using the Decision Tree on 10 microarchitectural metrics, as listed in Table 3.1. We trained the Decision Tree classifier using the dataset described in Figure 3.7. Figure 3.9 illustrates the comparative behavior

of the two selected features, BRANCH_TAKEN and LD_STALL by the Decision Tree. These HPCs measure the number of branch instructions and the number of delayed load instructions.

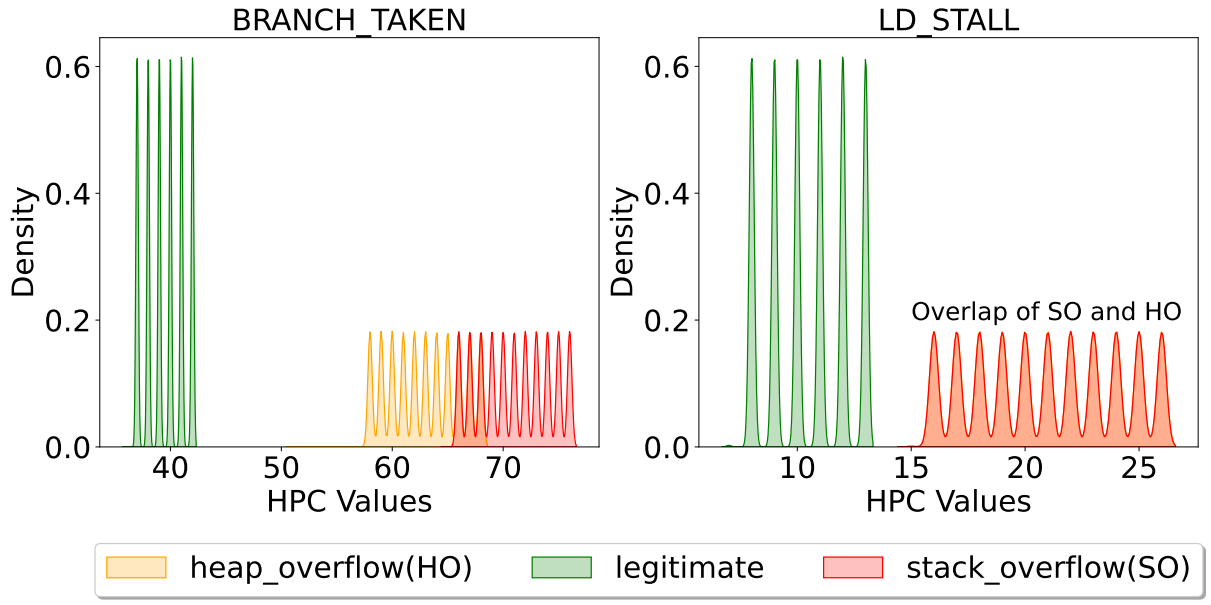


Figure 3.9 – Selected Microarchitectural Metrics by Decision Tree Model

Buffer overflow results in the alteration of CPU behavior to include unexpected branches and large delays in data retrieval from memory. A decision tree selects the increased BRANCH_TAKEN and LD_STALL counter values from the 10 microarchitectural metrics to identify potential attacks against the MAC layer of a protocol stack. Figure 3.10 illustrates the Decision Tree model block diagram produced using our dataset. The model splits the generated dataset from Figure 3.9 into three classes: (legitimate, stack_overflow, and heap_overflow). Indeed, during the learning phase, the BRANCH_TAKEN and LD_STALL HPC values demonstrated a high capacity to detect attacks. These values are used for decision-making based on thresholds $K1$ and $K2$ determined from the training data: BRANCH_TAKEN < 65.5 and LD_STALL < 14. The values are directly related to the code used to parse the received network packets and store the received data in the buffers.

There are several options for decision implementation for IoT end-devices. This includes hardware, software, and a combination of both. Hardware design can be developed using a Finite State Machine (FSM) and hardware accelerators in hardware description language. However, software implementations can be crafted using the C programming

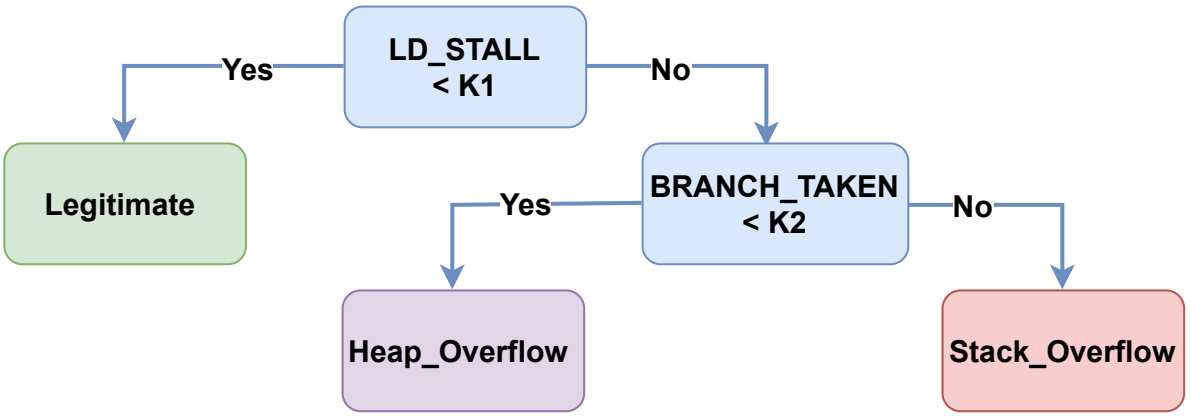


Figure 3.10 – Generated Decision Tree Classifier Model

language. This is because of the minimal hardware overhead and the satisfactory classification speed, decision tree models are particularly suitable for FPGA implementations. As a result, we decided to proceed with hardware-based implementation.

3.3.6 Conclusion

We explored packet injection by reproducing buffer overflows on heap and stack locations within a simulated minimal SoC that includes a network processor, memory, and UART. We observed the behavior of microarchitectural metrics on the network CPU while parsing received network packets and generated a dataset consisting of 11 microarchitectural features under both attack and legitimate scenarios. We trained and evaluated several machine learning classification algorithms with the generated dataset, achieving high detection rates with several classifiers. Owing to its low implementation complexity and fast classification speed in hardware, a Decision Tree classifier was selected. The model, generated as a tree structure, could distinguish between heap overflow, stack overflow, and legitimate packet processing behavior using only two microarchitectural features (BRANCH_TAKEN and LD_STALL).

In this study, we fulfilled requirement **Req1 - Lightweight & Local Analysis** by opting for a decision-tree-based machine learning classifier. This choice, which is known for its minimal overhead and rapid classification speed in FPGA setups, is advantageous. Implementing this model in *Diwall* incurs minimal area overhead, and the operation of the model is limited to just a few clock cycles, resulting in reduced energy consumption of *Diwall*.

In the next section, we address the second requirement **Req2 - Multi-level Monitoring:** by examining the detection of jamming attacks through *Diwall*, using network metrics.

3.4 Study of Jamming Attacks

This study primarily aimed to recreate and detect jamming attacks with a focus on constructing a model for legitimate network traffic around an IoT device. To achieve this, we monitored the PHY layer metadata, specifically, the RSSI and SNR. These metrics, which are prevalent in various IoT protocols, have been selected for their potential in identifying wireless attacks. We utilized an FPGA demonstrator with a LoRa modem for dataset generation, comprising RSSI and SNR measurements obtained during both legitimate network behavior and jamming attacks. We employed the Exponentially Weighted Moving Average (EWMA), a statistical technique used to preprocess and analyze the RSSI and SNR values in the LoRa PHY layer, thereby tracking the evolution of RSSI and SNR relative to their historical data. Previous research supports the potential of the RSSI's moving average in jamming attack detection [69, 70]. In this study, we implement a hardware-based approach to realize this potential. Network metrics are monitored using a dedicated HPC on the network processor, tracked by the HPMtracer, and subsequently analyzed by the detector implementing the EWMA.

3.4.1 FPGA Experimental Testbed

To obtain real datasets for the PHY layer metadata features, we implemented an SoC on an FPGA platform with a LoRa modem. This implementation enable the successful reception and transmission of LoRa packets using an Arty A7 100T FPGA board. The SoC is equipped with various components including a network processor (CV32E40P), RAM, TIMER, UART, SPI, IRQs (DIOs), and a embedded logic analyzer from LiteX (Litescope). All the peripherals were interconnected using a Wishbone bus. The LoRaMAC node stack, developed by Semtech, was adapted and integrated into the SoC. This implementation utilized the SX1276 LoRa transceiver. Figure 3.11 shows the experimental LoRa-based FPGA wireless connectivity subsystem.

The software running on the network processor incorporates the necessary drivers and a board support package (BSP) to support the LoRa protocol stack. To study jamming

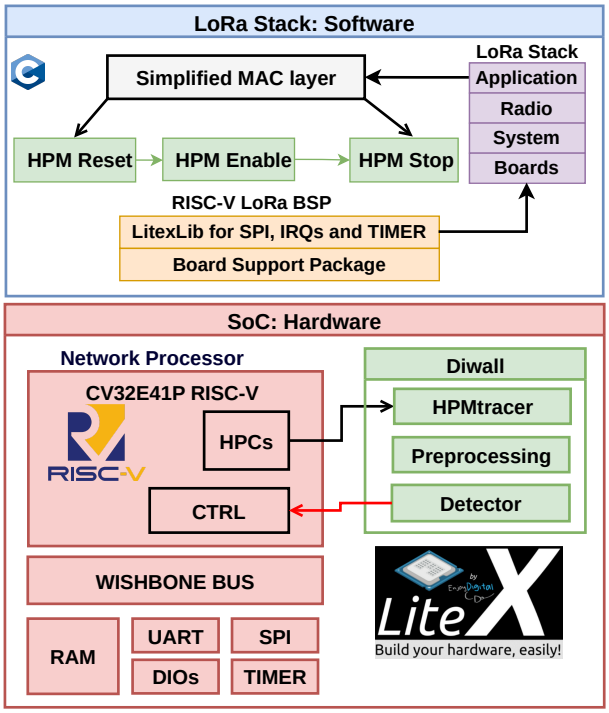


Figure 3.11 – SoC Architecture with LoRaMACnode stack

attacks, we focused entirely on the LoRa PHY layer. In addition, we developed a simplified MAC layer for wireless communication within the sub-GHz frequency band (868 MHz). NwHPC, a 64-bit HPC on the CV32E40P network processor, is dedicated as a register to track the PHY layer metadata, including the RSSI and SNR. The RSSI feature is stored in the lower 32 bits $(0-31)$, whereas the SNR feature is stored in the upper 32 bits $(32-63)$. The NwHPC was monitored using the HPMtracer and was utilized for further analysis.

3.4.2 Jamming Reproduction Attacks

In this section, the focus is on replicating and analyzing two types of jamming attacks: trigger-based and continuous. The objective is to reproduce these attacks to gain insights into the characteristics and behaviors of the RSSI and SNR during such scenarios. Figure 3.12 provides a summary of the two jamming categories employed in this study.

S1: Jamming Continuous In this scenario, a jammer deliberately transmits interference signals on a targeted frequency channel regardless of the current state of the victim's

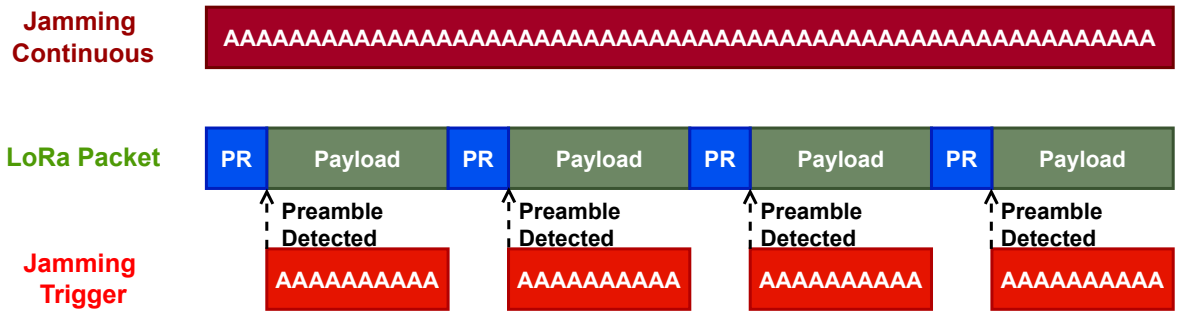


Figure 3.12 – Trigger and Continuous Jamming Categories

channel. To accomplish this, the jammer employs a strategy of generating random LoRa packets, disguising himself as a legitimate end node.

S2: Jamming Trigger First, the jammer scans the current channel activity while patiently waiting for a preamble to be transmitted by another LoRa node. Once the preamble is detected, the jammer transmits an interfered legitimate packet.

3.4.3 Dataset Generation

The experimental setup illustrated in Figure 3.13 was used to generate the RSSI and SNR datasets. It involves three LoRa devices consisting of one transmitter (TX victim) and one receiver (RX victim) positioned within a distance of less than 10 $meters$. The jammer device (attacker) is located in close proximity to the RX device within a range of less than $1 - 3$ $meters$. Throughout the jamming phase and legitimate communication between the TX and RX, the HPMtracer records the RSSI and SNR values for each network packet received. We used in this setup LoRa spreading factor of $SF = 12$, and a LoRa channel on 868.1 MHz.

Figure 3.14 illustrates the behavior of SNR and RSSI features during both trigger-based and continuous jamming scenarios. We monitored approximately 5000 network packets, where the x-axis represents the index of each processed packet. For continuous jamming, the jamming window spans from packet index 3000 to 4000, while for trigger-based jamming, it ranges from index 3300 to 3800. During jamming attacks, a clear pattern emerges where the behavior of the RSSI shows a notable increase, whereas the behavior of the SNR does not exhibit significant changes. The presence of jamming signals

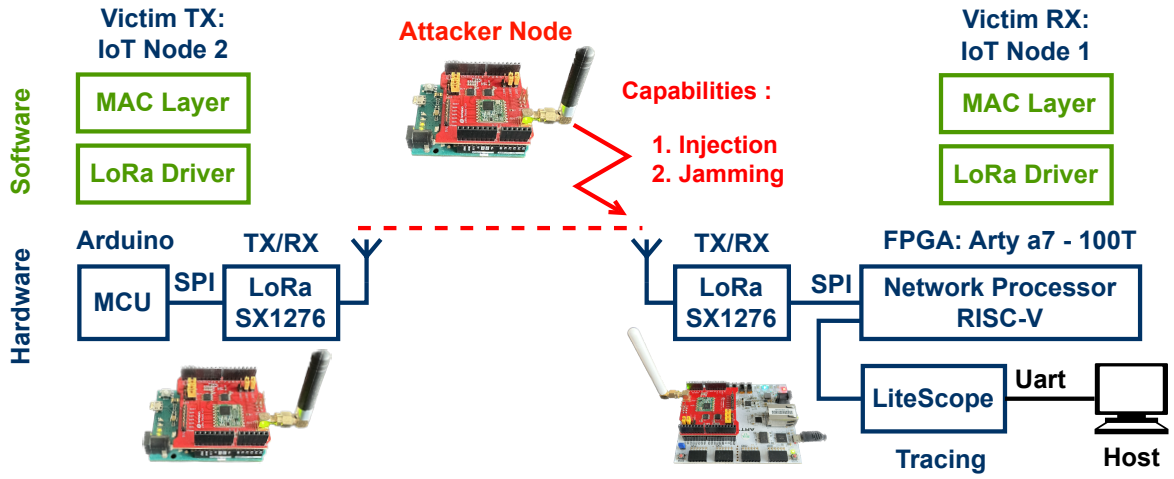


Figure 3.13 – LoRa Testbed with Diwall implemented on FPGA Arty A7 100T Board

introduces interference and noise into the communication channel, leading to a substantial increase in RSSI values. This increase reflects the overwhelming effect of the jamming signal on the received signal strength. Jamming signals introduce additional noise and do not necessarily affect the desired signal power to the same degree. As a result, the SNR remains relatively stable during trigger-based and continuous jamming scenarios compared to legitimate range.

LoRaWAN, a wireless communication protocol optimized for long-range connectivity between gateways and IoT devices spanning distances of several hundred meters, is susceptible to short-range jamming interference. During jamming attacks, when the jammer is in proximity to the victim device, the RSSI values experience a substantial increase, reaching exceptionally high levels. In this situation, the jamming signal plays a dominant role within the received signal, overpowering the background noise. Consequently, the SNR may exhibit a limited decrease or even appear higher than the standard, owing to the influence of the powerful jamming signal. During the analysis of the dataset, it became evident that monitoring the RSSI levels is crucial for detecting jamming attacks, as it reveals significant deviations from the expected range. Specifically, a notable increase in the RSSI beyond normal behavior serves as an important indicator of a nearby jamming attack. Leveraging this feature can enhance the effectiveness of the trigger and continuous jamming attack detection. Although monitoring the SNR remains valuable, relying solely on it for jamming detection represents a challenge. Consequently, our focus is directed towards analyzing and utilizing RSSI features to achieve enhanced jamming

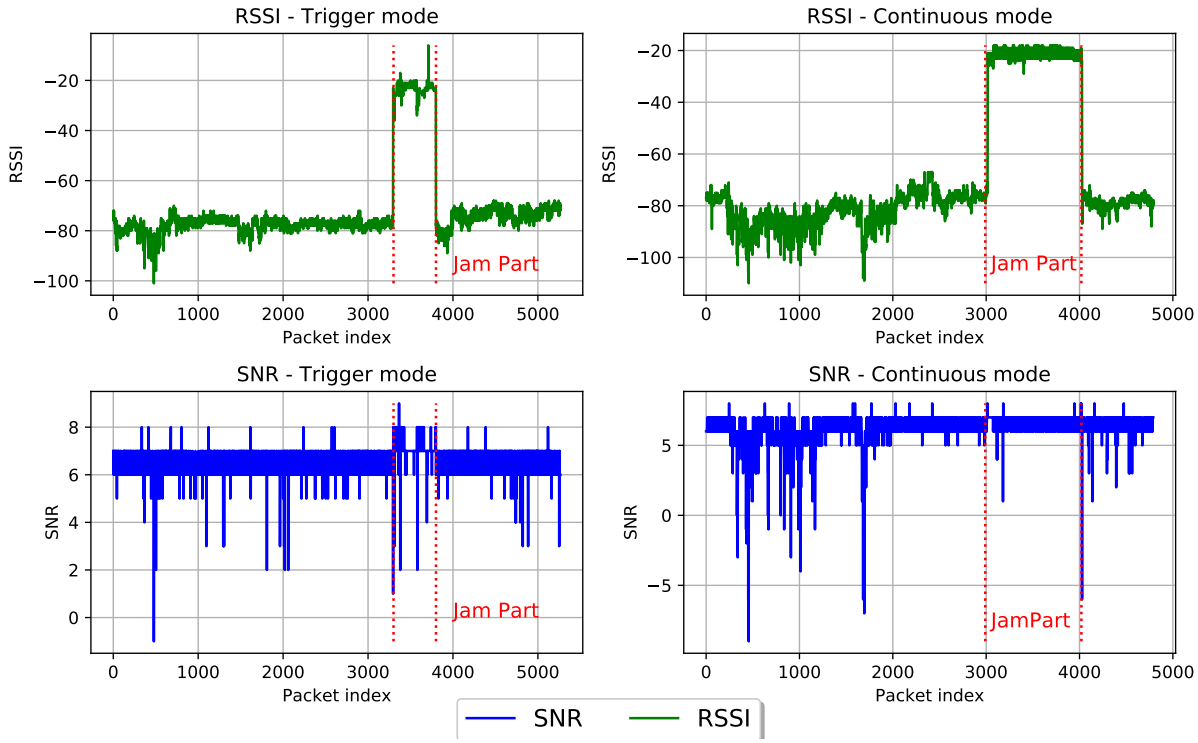


Figure 3.14 – RSSI and SNR Metrics during Legitimate Traffic and Jamming

detection capabilities.

3.4.4 Jamming Detection Methodology

Several strategies for countering jamming attacks have been proposed in the literature, demonstrating their effectiveness in detecting such attacks in wireless sensor networks and IoT architectures [71,72]. Many of these strategies use metadata from the PHY and MAC layers as features for jamming countermeasures [73,74]. Ruotsalainen et al. [72] highlighted numerous such mechanisms. These include radio fingerprinting, an authentication technique that identifies devices by analyzing the characteristics of the received signals, and a novel technique of key generation based on RSSI values, which extracts secret keys from these values. Machine learning and deep learning techniques are frequently incorporated into these mechanisms, owing to their superior performance in recognition and classification tasks. In [71], the authors used a neural-network-based machine learning approach to propose a novel jamming detection and classification algorithm. This approach employs features such as Packet Delivery Rate (PDR), SNR mean and variance, SNR Power Spectral Density (PSD), and cross-correlation. However, although the use of deep learning

techniques, such as neural networks, in jamming detection provides high accuracy and effective feature performance, it also requires significant computational resources. This approach is more suited to the upper levels of IoT architecture, such as gateways or servers, where voluminous resources are available, and all features are accessible.

EWMA, a statistical method, is a valuable tool for detecting small deviations in statistical data, and it has a relatively low computational complexity. Its effectiveness in identifying jamming attacks while requiring a minimal number of metadata features has been proven. Compared to neural-network-based methods, this methodology requires significantly fewer computational resources. In their work, the authors of [70] proposed deploying EWMA on the cluster head to identify attacks on member nodes and on the base stations to detect attacks on the cluster heads. The implemented EWMA is capable of detecting anomalous changes in the intensity of a jamming attack event by utilizing the packet inter-arrival feature of the packets received from the sensor nodes. In another study [69], the authors proposed the use of EWMA for jamming detection in LoRaWAN networks. They employed RSSI and packet inter-arrival time (IAT) as datasets for model training and evaluation using both large-scale simulation datasets and small-scale real-world datasets. They compared their approach with a Recurrent Neural Network (RNN) machine learning model, demonstrating higher detection rates. The authors reported a True Positive Rate of approximately 90% for the statistical model and 98% for the neural network machine learning model. This demonstrates that although the EWMA method has a slightly lower accuracy, it offers a competitive alternative with less computational complexity.

The EWMA formula is given in Equation 3.5 with λ the smoothing constant determining the depth of the EWMA.

$$\mathrm{EWMA}_0 = \mathrm{RSSI}_0 \qquad \mathrm{EWMA}_n = \lambda \mathrm{RSSI}_n + (1-\lambda)\mathrm{EWMA}_{n-1} \tag{3.5}$$

Our goal was to focus on resource-limited IoT devices. Unlike previous works that have implemented EWMA in software for gateways [70, 74], we chose a hardware-based implementation of EWMA. This strategy aims to identify acceptable RSSI values for the network metric, with any values beyond this indicating a statistical anomaly. During periods of triggered and continuous jamming, the EWMA of RSSI value increased significantly, as illustrated in Figure 3.15. This increase indicated a jamming attack.

We customized our network processor (CV32E40P) to include an NwHPC that mon-

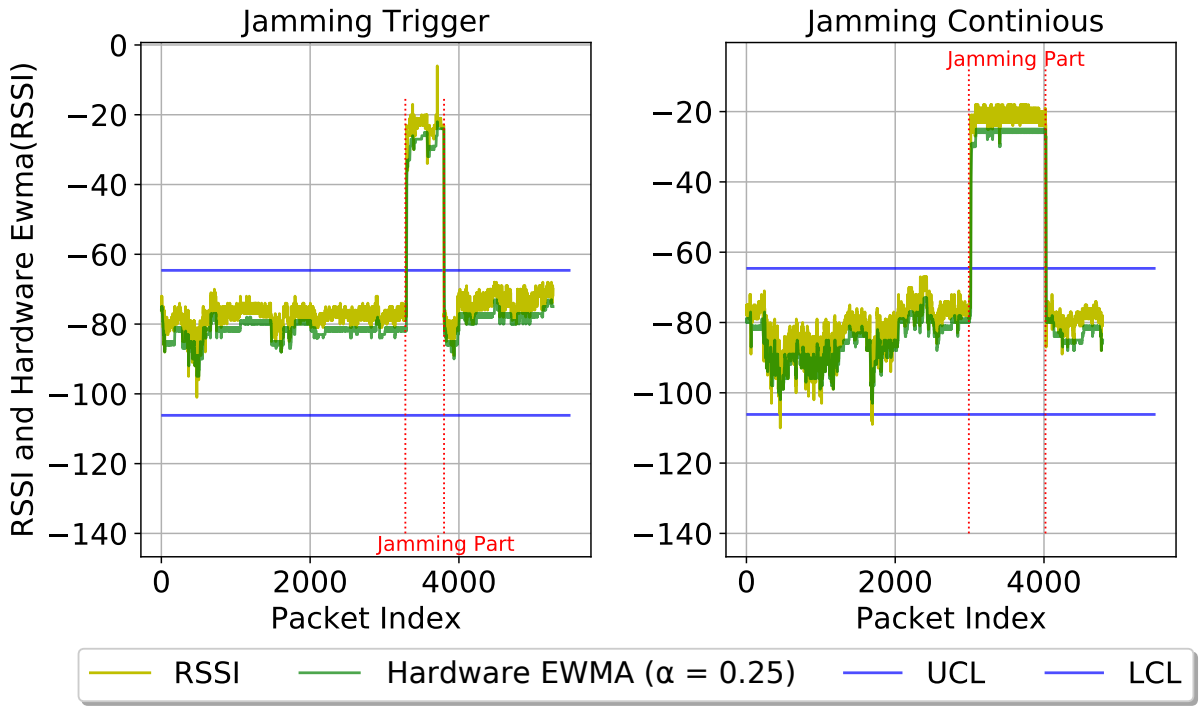


Figure 3.15 – RSSI and Hardware based EWMA during Legitimate Traffic and Jamming Attacks

itored the RSSI of each incoming packet. Within *Diwall*, we implemented the EWMA algorithm in a hardware block to analyze the RSSI values and make decisions. For the hardware EWMA, we assigned a value of $\lambda = 0.25$. Previous research [70] has suggested that this value should be between 0.2 and 0.5. To detect jamming attacks, we used the Upper Control Limit (UCL) ($UCL = -65$) and Lower Control Limit (LCL) ($LCL = -106$) thresholds in our hardware. These thresholds represent the legitimate upper and lower limits of the EWMA values. The LCL and UCL values for the EWMA control chart were calculated as follows:

$$\text{UCL} = EWMA_0 + f \cdot \sigma_{ewma} \qquad \text{LCL} = EWMA_0 - f \cdot \sigma_{ewma} \tag{3.6}$$

— UCL: Upper Control Limit

— LCL: Lower Control Limit

— EWMA_0: Target Value

— f: Multiplier

— σ_{ewma}: Standard Deviation

$$\sigma^2_{ewma} = \sigma^2_{rssi} \cdot \left(\frac{\lambda}{2-\lambda}\right) \tag{3.7}$$

— σ_{ewma}: Variance of the EWMA of RSSI

— σ_{rssi}: Variance of the RSSI Values

— λ: Smoothing Factor

EWMA, a statistical technique, efficiently detects small shifts in time-series data owing to its low complexity and requires updates only for newly observed data. It effectively combines current and historical data, facilitating the quick detection of small shifts.

Setting the value $\lambda = 0.25$ in Equation 3.5 simplifies the implementation of EWMA. This is because multiplying by 0.25 is equivalent to dividing the value by 4, which can be approximated by right-shifting the binary representation of the RSSI by 2 bits. The same strategy can be applied to previous $(1-\lambda)\text{EWMA}_{n-1}$ values. For instance, multiplying by 0.75 can be represented by adding 0.25 and 0.5, which can be approximated by right-shifting by 2 and 4 bits, respectively.

Using this approach, we respect the requirement **Req1 - Lightweight & Local Analysis**. This hardware implementation approach enhances the computational efficiency and reduces the need for complex multiplication circuits, ultimately reducing the overall area overhead in *Diwall*.

3.4.5 Conclusion

We studied jamming attacks by replicating both the triggered and continuous jamming on a LoRaWAN FPGA-based testbed. This testbed, used in conjunction with two other LoRa end-devices, helped generate real-world datasets of the PHY layer metadata features. We examined the behavior of RSSI and SNR during jamming windows and periods of legitimate network traffic. Notably, the RSSI feature displayed significant changes during the jamming windows compared to the SNR. We also discuss several anti-jamming strategies, including deep learning, machine learning such as neural networks, and statistical approaches such as EWMA. We chose to use EWMA because of its ability to detect

jamming attacks in wireless networks, including LoRaWAN and other protocols, with low computational resources. This approach tracks the PHY statistical metadata over time, providing a smoothed average that can identify abrupt changes or trends deviating from the expected behavior. The expected behavior is controlled by predefined UCL and LCL thresholds, where deviations from these thresholds indicate the presence of a jamming attack.

In this section, we address requirements **Req1 - Lightweight & Local Analysis** and **Req2 - Multi-level Monitoring**. For the first requirement, we used EWMA, a lightweight methodology that effectively identifies jamming attacks and requires minimal computational resources. The second requirement is addressed by the integration of RSSI network metrics with monitored microarchitectural events. *Diwall* analyzes these combined metrics to increase the number of target attack vectors.

3.5 Summary

In this chapter, we introduce a novel approach for detecting wireless attacks against resource-constrained IoT end-devices. First, we explain *Diwall* methodology and subsequently compare it with previous research. Our approach aims to monitor multi-level data related to the wireless connectivity of an IoT end-device. *Diwall* comprises three hardware units: an HPMtracer, Preprocessing and Detector units. The HPMtracer tracks the HPCs of a network processor, a 32-bit RISC-V CPU. These HPCs monitor microarchitectural features and dedicated network HPCs for PHY metadata. We extended the network processor to include NwHPC to monitor the RSSI metadata from the PHY layer. In our threat model, we considered wireless attacks, including memory vulnerability-based packet injection and jamming attacks. Various scenarios were studied for the same class of attacks using both simulation and FPGA-based testbeds.

Our proposed approach offers a distinctive alternative to the previous methods. Many of these strategies have been dedicated primarily to IoT gateways because of their greater availability of performance compared to IoT end-devices. These strategies often employ software-based solutions and implement them directly on the gateways. However, the proposed approach used a different route. Our primary focus is on resource-constrained nodes. We aim to overcome these limitations by employing a lightweight hardware implementation of *Diwall* and leveraging existing HPCs on the network processor as probes. Additionally, we maintained monitoring control using the software with only a few essen-

tial instructions. This strategy promotes the efficient utilization of resources and delivers improved performance regardless of the restrictions presented by these IoT end-devices.

Proposed *Diwall* addresses the mentionned requirements previously as follow:

Req1 - Lightweight & Local Analysis: *Diwall* uses HPCs as probes for monitoring, which are available in several modern CPUs that are used as network processors. This reduces the cost of implementing extra probes in wireless connectivity for monitoring, making *Diwall* lightweight. Monitored data from HPCs are analyzed locally using lightweight detection methodologies and implemented in hardware with minimal overhead. *Diwall* is fully implemented in hardware with a few instructions in software for control, which does not directly impact the execution time and memory usage of the network processor.

Req2 - Multi-level Monitoring: *Diwall* leverages multi-level monitoring using RSSI and microarchitectural events. These metrics are based on network and hardware levels and are tracked by *Diwall* for potential attacks. *Diwall* extension for extra software events and other network metadata in the IoT stack can be easily monitored by HPCs.

Req3 - Reconfigurability: *Diwall* choice of metrics was independent of the waveforms considered and the targeted ISA in this study. *Diwall* can be reconfigured to detect the same attacks on another protocol stack using the same methodologies. Because RSSI metadata are maintained by various IoT stacks, microarchitectural events are present in most modern processors. *Diwall* parameters in the FPGA can be modified and changed for use in another context.

In this chapter, we have presented the details of our framework, which encompass both simulation and emulation techniques. This framework used to create *Diwall* IDS, can simulate attacks in both simulation and real-world scenarios. It is used to generate representative datasets and train algorithms to establish machine learning and statistical models for *Diwall* IDS. The provided framework can be leveraged by developers to design, configure, and enhance IDS systems for IoT devices.

In the next chapter, we present experimental results evaluating *Diwall* approach using a real FPGA testbed with LoRa and LoRaWAN IoT stacks. We implemented the elements of *Diwall* entirely in the FPGA, including existing HPCs in the network processor and NwHPC for network metric monitoring. Our system includes an HPMtracer

for tracking the montioned HPCs and a detector module that implements the generated models of the Decision Tree and EWMA. This module can detect both packet and jamming attacks. This evaluation includes a discussion of the obtained results on detection rates, performance, and area overhead.

Chapter 4

Diwall: Implementation and Experimental Evaluation

Introduction

Following the insights from the previous chapter on packet injection and jamming attacks. *Diwall* parameters model were generated for detecting the considered attacks. This chapter reviews the implementation and evaluation of *Diwall*. First, it details the *Diwall* FPGA-based architecture. Then, it explains the methods used to assess wireless attacks, particularly jamming and packet injection. Second, the experimental results are detailed: detection accuracy, FPGA resources use, and performance relative to a baseline network processor. Finally, we evaluate *Diwall*'s effectiveness in a full IoT setup. For the demonstration, an IoT node with *Diwall* is integrated in a LoRaWAN network.

4.1 Diwall FPGA Implementation

4.1.1 Diwall Architecture Implementation

Diwall was implemented on an Arty A7 100T FPGA board, along with a RISC-V CPU network processor. We extended the RISC-V CV32E40P pipeline, described in the previous chapter, as shown in Figure 4.1. We modified this figure provided by OpenHW Group documentation in [61] to add *Diwall*. This figure represents a new variant of the CV32E40P, now serving as a network processor with *Diwall* integration. The CV32E40P connects to *Diwall* through a CSR located in the execute stage. In this architecture, *Diwall*'s signals and data are connected to the CV32E40P's CSR to monitor microarchitectural events and network metrics. Once *Diwall* analyzes data metrics, it raises an alert signal, which is then connected to the CV32E40P's interrupt controller.

The proposed *Diwall* for detecting wireless attacks could be integrated as a small

hardware component within the CPU architecture, like the FPU or TRNG. This choice offers quicker access to the HPCs and reduces the performance impact, including the maximum clock frequency reduction.

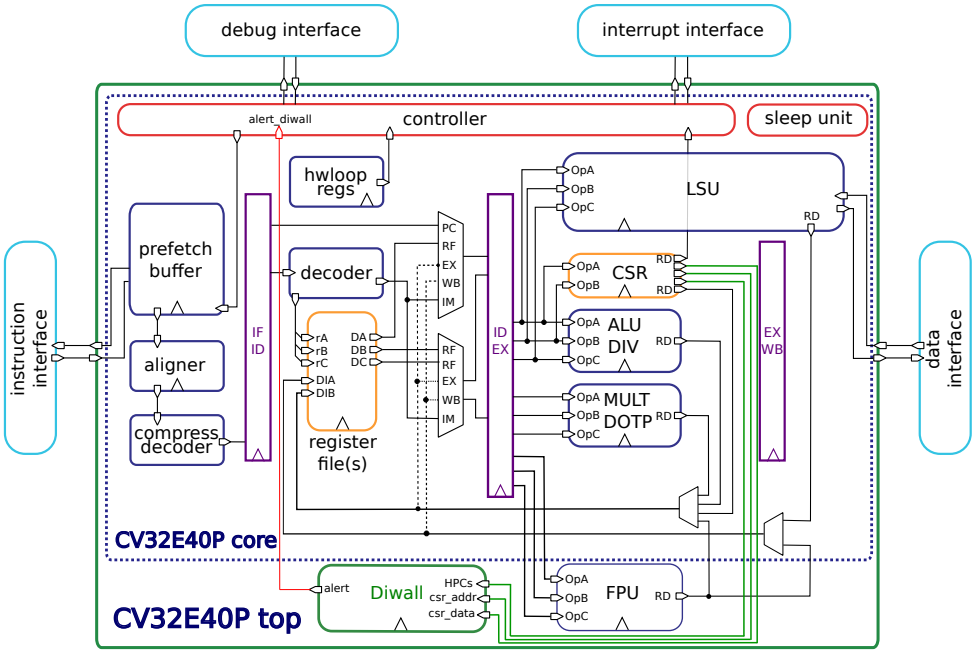


Figure 4.1 – CV32E40P Architecture Integrating *Diwall*

Figure 4.2 shows the architectural details of *Diwall* components. *Diwall* comprises hardware components, organized as follows: HPMtracer, Preprocessing, and Detector units.

All highlighted units were designed using an FSM. Each unit in *Diwall* has independent FSM states and controls. Only enable, disable, and data signals are propagated between the units.

The HPMtracer tracks the HPCs and sends them to preprocessing, where the moving average EWMA of the RSSI network metric is calculated. The microarchitectural event data are directly transferred without any preprocessing. These data are then sent to the detector unit for comparison with the thresholds. Once the Detector finishes its analysis, it activates the end signal to reset the values stored in the HPMtracer and puts *Diwall* in an idle state.

More information about each hardware unit and *Diwall* configuration is provided in the following subsections.

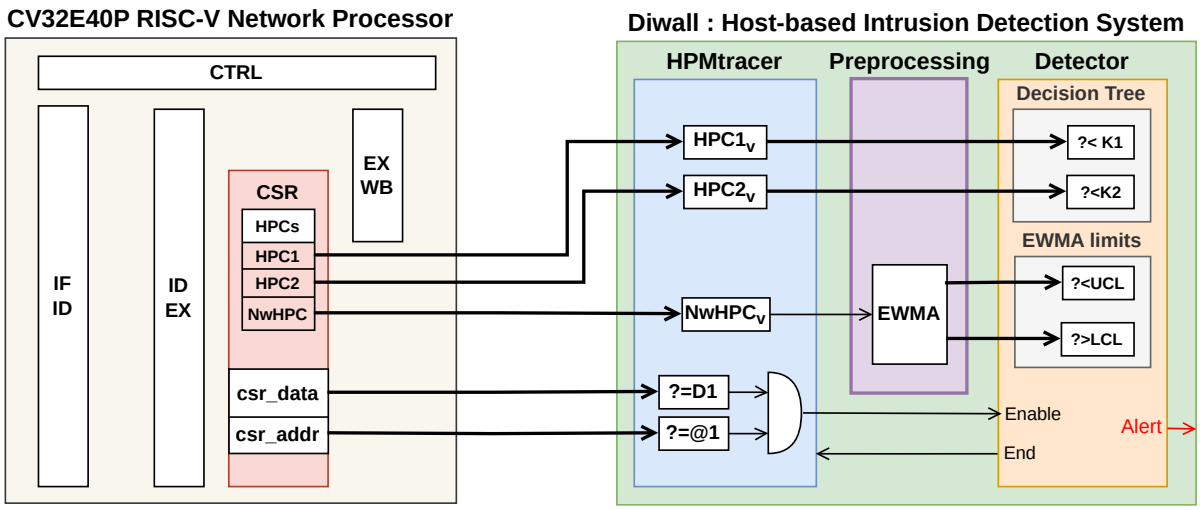


Figure 4.2 – Block diagram illustrating the architecture of Diwall incorporating CV32E40P Processor

4.1.2 HPMtracer: Hardware Tracer

The primary hardware component of *Diwall* is the HPMtracer. This hardware tracer monitors data from the network processor, and its operations are managed by software running on the processor. The network processor is equipped with HPCs tailored to monitor the microarchitecture and network data. The CV32E40P incorporates CSR into its execution stage pipeline. This CSR houses HPCs such as HPC1 and HPC2, which track selected microarchitectural events, as determined by our Decision Tree model. These events can be configured and adjusted by users using the software. The details of *Diwall* configuration are provided in Subsection 4.1.5. In addition, we configure a dedicated Network HPC named NwHPC, a 64-bit register, where each 32-bit half can serve as a register for monitoring IoT stack metadata.

CSR also features signals essential for controlling the HPMtracer, represented as *csr_data* and *csr_addr*. *Diwall* is activated and deactivated when *csr_addr* is set to $0x320$ and *csr_data* has values of $0x0$ and $0xF$. This action corresponds to writing into a dedicated register called *mcountinhibit*, which controls the enabling or disabling of HPCs. If the enabling condition is met, the HPMtracer forwards the HPCs' current values to the subsequent stage for analysis and decision making. In this architecture, the present values of the HPCs are denoted as $HPC1_v$, $HPC2_v$ and $NwHPC_v$ with v corresponding to value.

4.1.3 Preprocessing

This hardware block prepares the data for further analysis, particularly focusing on preprocessing several network metrics such as SNR and RSSI owing to noise in the data. In our research, we specifically analyzed and implemented EWMA preprocessing for RSSI metadata in the LoRa and LoRaWAN protocol stack. The goal here is to preprocess the data before it is analyzed by the detector. In this scenario, preprocessing receives three pieces of data from the HPMtracer: $HPC1_v$, $HPC2_v$, and $NwHPC_v$. $NwHPC_v$ represents the RSSI value coded in 32 bits, which requires preprocessing. On the other hand, $HPC1_v$ and $HPC2_v$ are the cumulative values of the selected microarchitectural events, they are used directly in our detector without requiring further preprocessing. The EWMA block processes the data sourced from $NwHPC_v$. It begins by calculating the EWMA for the incoming $NwHPC_v$ data.

In *Diwall*, the RSSI is preprocessed by calculating the EWMA in the hardware before the detector analysis. The EWMA is implemented in this preprocessing block with several considerations to reduce the overhead. The EWMA equation involves multiplications, and the direct implementation of multiplication in hardware can result in a higher area overhead for *Diwall*. As previously explained in 3.4.4, we simplify EWMA by using the right shift of the binary representation instead of complex multiplication. For instance, when multiplying the RSSI by a parameter such as 0.25, this is roughly equivalent to dividing by 4, which can be implemented in hardware by right-shifting the RSSI binary representation by two bits.

The EWMA hardware is implemented with an FSM consisting of three states: an IDLE state, a state for calculating the necessary right-shifting values for the equation, updating the output (the EWMA of the RSSI), and a state to send data to the detector. Further preprocessing methods for the metrics can also be incorporated into this block.

4.1.4 Detector: Decision Tree and EWMA Control Limits

The second module, the detector, is responsible for data analysis and decision making based on the data monitored by the HPMtracer and handled data by the Preprocessing unit. The Detector comprises two modules, a Decision Tree model for detecting packet injection attacks and EWMA control limits for jamming attack detection.

For this purpose, we use a previously generated Decision Tree model tailored to detect memory corruption based on packet injection attacks. This model utilizes two thresholds,

$K1$ and $K2$, and the monitored values from HPC1 and HPC2 are compared. These thresholds are the maximum legitimate cumulative microarchitectural events used by a monitored window in the software of the IoT protocol stack. In our study on LoRa and the LoRaWAN stack, the monitored window corresponding to a frame is received and parsed. If a frame is received, *Diwall* monitors these microarchitectural events and compares them with these thresholds.

The Detector module issues an alert when the data exceeds the thresholds set by the Decision Tree model. If the network processor exerts significant effort during frame parsing, it is reflected in the values of HPC1 and HPC2. In our findings, buffer overflow in the stack and heap leads to higher $HPC1_v$ and $HPC2_v$ values, deviating from the model defined by $K1$ and $K2$.

Another module within our detector, designed to detect jamming attacks, employs EWMA control limits thresholds. The calculated EWMA of $NwHPC_v$ value is compared with the predefined EWMA control limits, denoted as UCL and LCL. If the network metadata drifts away from these predefined EWMA limits, the detector module raises an alert.

The *Diwall* architecture, combining both the HPMtracer and Detector modules, efficiently detects both jamming and packet injection attacks. This is achieved within a brief time frame, taking less than 10 clock cycles to operate the entire architecture. This efficiency stems from our method of reading HPCs only at the end of the monitoring phase. Moreover, *Diwall* does not utilize local counters. Instead, it relies on registers to temporarily store data for a few cycles.

4.1.5 Diwall Configuration

To enhance the flexibility of the *Diwall* approach, we utilize configuration parameters from the software. HPC1, HPC2, and NwHPC are configured using software running on a network processor. The configuration of HPC1 and HPC2 is achieved using *Diwall* software functions, which are based on the assembly code of RISC-V. These functions write *Diwall* parameters directly to the mapped CSR registers. In the CV32E40P architecture, HPC1 and HPC2 are represented by $mhpmcounterX$, where X represents the counter ID, ranging from 3 to 31. These counters are mapped to the CSR addresses from $0xB03$ to $0xB1F$. Specifically, we use $mhpmcounter3$ and $mhpmcounter4$ to monitor HPC1 and HPC2, with addresses $0xB03$ and $0xB04$.

Assigning microarchitectural events to $mhpmcounterX$ is accomplished through soft-

ware using the event selector CSR, which is denoted as $mhpmeventX$. Each event selector corresponds to a counter ID, with X ranging from 3 to 31. These event selectors are located within the address range of $0x323$ to $0x33F$. Microarchitectural events are identified with an ID from 0 to 15, and we associate this ID with the desired $mhpmeventX$, which, in turn, assigns the event to $mhpmcounterX$. For example, if $mhpmevent3$ is linked to the ID of LD_STALL (represented by ID 2), setting the second bit in $mhpmevent3$ to 1 results in $mhpmevent3$ being set to 0×4, which will affect LD_STALL to $mhpmcounter3$. The relationship between $mhpmeventX$ and $mhpmcounterX$ is important. If $mhpmevent3$ is set to 1, it implies that it will be counted by HPC $mhpmcounter3$. Both $mhpmeventX$ and $mhpmcounterX$ are mapped in the CSR of the RISC-V architecture and can be accessed for read and write operations using the CSR instructions provided: *csrr* for read and *csrw* for write. We utilize the *csrw* instruction to select the desired counter ID by setting the address of the event selector to the microarchitectural event ID. This instruction also helps us reset the counters by setting their data to 0. During the training phase, we used the *csrr* instruction to read the HPC values for the dataset generation.

Following the HPCs configuration, a dedicated register for enabling and disabling the HPCs is *mcountinhibit*, located at CSR address $0x320$. Each HPC can be individually enabled or disabled by modifying the corresponding bit in the *mcountinhibit* address. For instance, to enable $mhpmcounter3$, set bit 3 to 0; to disable it, set bit 3 to 1. To monitor the network metadata using NwHPC, we used $mhpmcounter5$. The configuration is specified for the RSSI and declared in the software as a 32-bit value at address $0xB05$. We reset the counter using the *csrw* write instruction to set this register to 0, and then write the received RSSI value to the same address to track it using *Diwall*.

K1, K2, LCL, and UCL are the parameters of *Diwall*'s detector, representing the generated model of the Decision tree and the EWMA control limit thresholds. In this version of *Diwall*, these parameters are configured directly in the HDL code, which requires new bitstream generation. To increase flexibility, several approaches are expected to allow software configurations for K1, K2, UCL, and LCL. The first configuration uses SoC-CSR-dedicated registers. The SoC includes its own external CSR, which follows the same methodology as the RISC-V network processor for configurable HPCs. Each register in the SoC has its own address and data accessible through the provided software instructions. Memory-mapped registers on the SoC's CSR can be associated with each *Diwall* parameter, enabling user configuration and updates to the detector's models. In the LiteX framework, functions such as $csr\_write\_simple(reg\_data, reg\_@)$ and

$reg\_data = csr\_read\_simple(reg\_@)$ are provided to manage memory-mapped hardware registers. These functions can be used to configure and update *Diwall* parameters.

However, an alternative, more efficient option for *Diwall* configuration is to use dedicated registers implemented within the network processor's CSR. The RISC-V CSR includes additional registers, specifically for *Diwall*, containing its parameters. Although not all CSR registers are implemented in the CV32E40P, they can be extended to include the necessary registers for configuration. The RISC-V ISA provides dedicated instructions for CSR handling using the software. These added *Diwall* dedicated configuration registers will not require ISA extensions; they will use the provided CSR instructions for configuration, specifically, *csrr* and *csrw*.

Configuring *Diwall* using the CSRs of the network processor provides faster access, lower overhead, and minimal performance impact.

4.1.6 FPGA Resource and Performance Overhead

In this section we delve into the implementation cost obtained on the FPGA board. In Table 4.1, we present the area metrics of *Diwall* and network processor in terms of lookup tables (LUTs), FFs, and the maximum frequency for an FPGA (XC7A100TICSG324-1) deployed on Arty-A7 100T board. These results are obtained using the Xilinx Vivado v2020.2 tool. We evaluate three variants of the network processor:

— **V1:** This is the RISC-V baseline version, devoid of any *Diwall* component. This serves as a comparison benchmark for V2 and V3.

— **V1':** V1 integrated with the HPCs: (HPC1, HPC2) This version illustrates the effect of activating 2 HPCs on the RISC-V CPU.

— **V1":** V1 integrated with the HPCs: (HPC1, HPC2), and NwHPC. This version illustrates the effect of activating three HPCs on the RISC-V CPU.

— **V2:** It builds on V1' by incorporating an HPMtracer, Detector for detecting packet injection attacks. This highlights the overhead introduced by previous *Diwall* [75] in contrast to the V1 baseline.

— **V3:** It builds on V1" by incorporating an HPMtracer, EWMA preprocessing, and Detector for detecting jamming and packet injection attacks. This highlights the overhead introduced by *Diwall* in contrast to the V1 baseline.

Table 4.1 – Resource Utilization and Maximum Frequency of Implementation for 5 Versions of the Network Processor with and without Diwall

Network Processor			**Overhead**		**Freq**
CV32E41P	**HPCs**	***Diwall***	**LUT**	**FF**	**MHz**
V1 (Base)	1 (Default)	✗	4676 (+00%)	2136 (+00%)	65.69
V1'	2	✗	4777 (+2.16%)	2217 (+3.79%)	65.60
V1"	3	✗	4897 (+4.73%)	2298 (+7, 58%)	65.62
V2 ([75])	2	✓	5105 (+9.17%)	2352 (+10.11%)	65.50
V3 (This article)	3	✓	5345 (+14.30%)	2625 (+22.89%)	65.07

For the *Diwall* architecture in V3, the CV32E41P employs three HPCs. Two counters tracks microarchitectural events (HPC1 and HPC2), whereas a dedicated register-based HPC NwHPC measures the network metric RSSI.

Comparison between V3 and V1" with V1:

— The integration of the three HPCs incurs an overhead of 7.58% in FFs and 4.73% in LUTs.

— *Diwall* itself contributes to an area overhead of approximately +22.89% in FFs and 14.30% in LUTs.

— The additional FFs and LUTs in *Diwall* are predominantly associated with the three counters.

When examining the performance implications of *Diwall* design units V3 and V2 with respect to V1:

— There is no substantial impact on design performance.

— The maximum frequency consistently hovers at approximately 65 MHz.

After describing the implementation details of *Diwall*, in the next section, we highlight the experimental evaluation testbed and results.

4.2 Experimental Evaluation Framework

In this section, we describe the experimental testbed used to assess the effectiveness of the *Diwall* implementation. We reproduce and launch packet injection and jamming scenarios in real-world settings using LoRa communication. We then calculate the detection rates to demonstrate the effectiveness of *Diwall*.

4.2.1 LoRa and Simplified MAC Layer

Our testbed comprises three LoRa devices, as illustrated in Figure 4.3: IoT nodes 1 and 2 for establishing a LoRa network and an attacker responsible for executing wireless attacks. This configuration ensures a comprehensive assessment of the proficiency of *Diwall* in detecting wireless attacks in a representative communication setting.

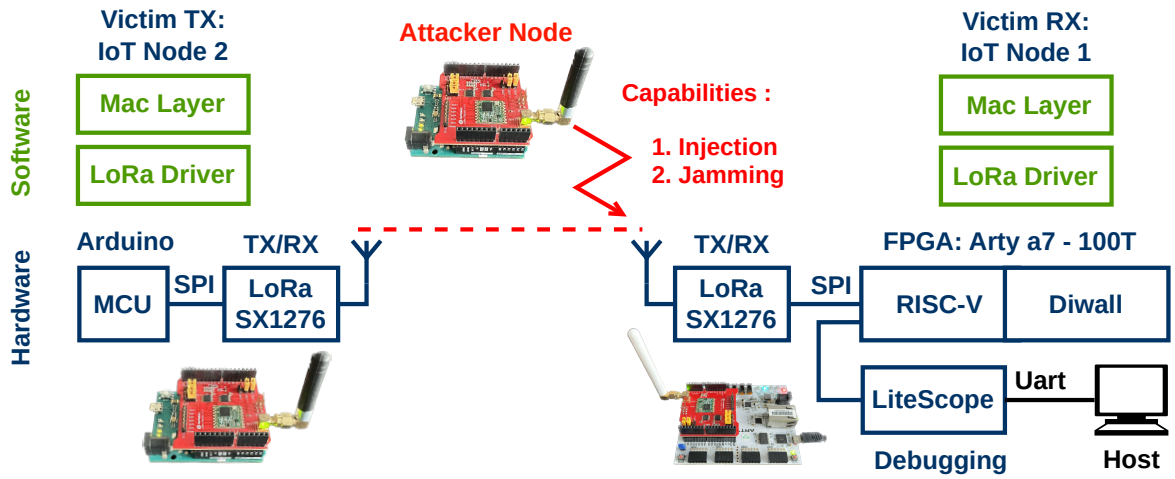


Figure 4.3 – Evaluation of Diwall on FPGA Arty A7 100T Board within LoRa Testbed

IoT node 1 (Victim): This custom-built IoT end-device acts as a potential victim. It integrates the necessary hardware for both the LoRa PHY layer and simplified MAC layer, which is further tailored for compatibility with the proposed *Diwall*. The simplified MAC layer parses LoRa frames and stores them in a reception buffer. It is instrumented with *Diwall* instructions for configuration and control. The configuration includes selecting microarchitectural events LD_STALL and BRANCH_TAKEN, assigned to HPC1 and HPC2, and RSSI to NwHPC. A Diwall-enabled instruction is placed at the beginning of parsing LoRa frames, and a disable instruction is placed at the end of parsing. For every received packet, *Diwall* verifies the data value of the microarchitectural events and the RSSI value in the hardware. These values are compared with the parameter thresholds established by an embedded Decision Tree model and EWMA control limits. An alert is issued if the frame contradicts the established security policy. To collect results on detection rates, a debugger called *LiteScope* observes details within registers and counters inside *Diwall* and RISC-V. A dedicated counter is placed inside the *Diwall* detector unit,

counting the triggered alerts. *LiteScope* facilitates reading this counter, which logs the total number of alerts triggered. During this evaluation, extra IoT node 2 is used to establish a LoRa communication link with the victim (IoT node 1).

Attacker Located between two IoT nodes (IoT nodes 1 and 2), it has the ability to inject and jam signals. We use an Arduino MCU with a Dragino LoRa Shield, which utilizes an SX1276 LoRa radio module. To manage the LoRa radio module, we utilize RadioLib, a versatile wireless communication library for Arduino. The programs provided by the Radiolib library have been customized to inject substantial network traffic, including a specified range of packet sizes, and to activate both jamming modes: triggered jamming and continuous jamming.

Evaluation of Packet Injection Attack In the packet injection scenario, IoT node 1 establishes a typical communication link by sending legitimate traffic to IoT node 2. Meanwhile, an attacker positioned between the two nodes competes with IoT node 2 and injects oversized malicious packets into IoT node 1. While IoT node 1 is equipped with a 10-byte reception buffer, it lacks a software-based size check for the incoming frames. The attacker capitalizes on this by sending packets that, although in line with the MAC layer protocol, exceed this 10-byte buffer, aiming to disrupt node 1's operation. To detect this, *Diwall* system continuously monitors microarchitectural data for incoming frames, raising an alert for any packet that breaches the buffer limit. For this evaluation, we utilized $400,000$ packets. The LoRa PHY layer parameters are set with a Spreading Factor (SF) of SF7 and an interval of approximately $100\ ms$ between the packets. The operating frequency adheres to the 868 MHz, specifically using the 868.1 MHz LoRa channel.

Evaluation of Jamming Attack In this study, an attacker transmits random frames, potentially aligning them with the buffer size of the victim's receiver. The primary goal of a jammer is to occupy the victim's channel.

Our testbed consists of three LoRa devices:

1. A transmitter (TX victim) and receiver (RX victim) are positioned within 10 $meters$ of each other.
2. A jammer (attacker) located close to the RX device is approximately $1-2$ $meters$ distant.

We investigate two jamming methodologies:

1. **Continuous Jamming**: Here, the attacker persistently transmits random frames, aiming to disrupt IoT node 2 during the communication with the victim.
2. **Trigger Jamming**: In this approach, the attacker eavesdrops on the channel, springing into action upon the detection of a preamble from IoT node 2. Upon such detection, the channel is interfered with.

Interestingly, even with channel interference, the victim remains capable of receiving frames and discerning the RSSI value of the frame. *Diwall* verifies the RSSI value of each acquired network packet. It calculates the EWMA and compares it with established EWMA control limits. If a packet's RSSI diverges from the EWMA limits, an alert is triggered, signifying a potential jammer proximity. For both jamming strategies, *Diwall* detection efficacy was gauged over 4000 frames. Our LoRa trials use a channel frequency of about 868.3 MHz, and employ an SF12. LoRa devices use various SF values to balance the data rate and communication range. These values typically ranged from 7 to 12. Higher values extend the range but reduce data rate, whereas lower values offer higher data rate but a shorter range. The SF in the LoRa PHY layer impacts the Time on Air (ToA). Higher SF values lead to a longer ToA, whereas lower SF values result in a shorter ToA. Choosing SF12 provides an ideal ToA for jammers to succeed in their attack.

In this section, we outlined the details of the evaluation testbed. We now discuss the detection results.

4.2.2 Results

Packet Injection Detection Rates Table 4.2 presents metrics including TP, TN, FP, FN, FPR, FNR, and detection accuracy during memory corruption attacks via packet injection. During packet injection attack, the network traffic comprises $200,000$ benign packets and an equal number of packets subjected to stack and heap buffer overflow. During the experiment $400,000$ packets were sent, $389,097$ were successfully received, resulting in a Packet Loss Rate (PLR) of approximately 2.72%. This PLR was deemed acceptable for LoRa networks. The reduced PLR can be attributed to the indoor location of the testbed. The detection accuracy for packet injection was significant at 99.98%. These results are obtained with numbers of FP of approximately 53 (0.027%) and FN of approximately 13 (0.007%) , both of which are notably low. This high accuracy is backed by the distinct behavior of microarchitectural HPCs, enabling effective differentiation

between legitimate packets and those affected by buffer overflow. Complete 100% accuracy was not realized with our Decision Tree model. This was owing to our testing under the most challenging scenario, where the sizes of benign and malicious packets were very similar, being 10 and 13 Bytes, respectively.

Table 4.2 – Evaluation of Diwall Detection Rates for Packet Injection Attacks

Attack	FP	FN	TP	TN	FNR	FPR	ACC	PLR
Packet Injection	53	13	193,327	195,704	0.007%	0.027%	99.98%	2.72%

F (False), N (Negative), T (True), P (Positive), R (Rate), ACC (Accuracy), PLR (Packet Loss Rate).

Jamming Detection Rates For the jamming attack assessment, we examined both the triggered and continuous detection rates. To create an ideal environment for the jammer, we used a long interval of approximately 1 s between transmitted packets. The LoRa transmitter is positioned 8 meters from the victim, with the jammer situated close to the victim. In the experiment, the LoRa transmitter dispatches approximately 2000 legitimate packets to a victim. Concurrently, the jammer sends an equivalent traffic volume split between the two jamming types. Table 4.3 summarizes results about detection rates metrics for jamming attacks. The observed PLR is 59%, which is typical for jamming scenarios. This high PLR occurs because most data packets are lost or corrupted owing to interference caused by the jamming signal.

The combined detection rate achieved by *Diwall* for both jamming methods is approximately 99.92%. This accuracy is accompanied by the absence of FP and negligible FN of approximately 1 (0.24%). While our evaluation is based on received packets, implementing a security policy on *Diwall* to verify the RSSI value even if there are no received packets could potentially elevate the detection rate to 100%.

Table 4.3 – Evaluation of Diwall Detection Rates for Jamming Attacks

Attacks	FP	FN	TP	TN	FNR	FPR	ACC	PLR
Jamming Trigger	0	1	402	1000	0.25%	0%	99.92%	59.7%
Jamming Continuous	0	1	413	1000	0.24%	0%	99.92%	58.7%

F (False), N (Negative), T (True), P (Positive), R (Rate), ACC (Accuracy), PLR (Packet Loss Rate).

The outcomes highlighted in this section emphasize that *Diwall* is a resource-efficient hardware solution, particularly for IoT end-devices with limited resources. It effectively detects both packet injection and jamming attacks at the network processor level. The next section is dedicated to a real use case. It details the porting of *Diwall* and the detection rates on a complex MAC layer. A LoRa PHY layer is maintained, and the simplified MAC layer is replaced by our SoC software using LoRaWAN.

4.2.3 Discussion

In this study, we assessed the LoRa PHY layer using a simplified MAC layer equipped with the necessary instructions for *Diwall* control. In our evaluation, we tested jamming and packet injection attacks and *Diwall* achieved an overall detection accuracy of approximately 99.94%. *Diwall*'s accuracy results in very few false alerts and negligible undetected attacks. We opted for a simplified MAC layer to represent a portion of the IoT protocol's MAC layer, specifically focusing on packet reception from the PHY Layer. This choice allows the rapid reproduction of packet injection attacks and simulations to generate a large dataset of microarchitectural events. However, using a more complex MAC layer can provide a more representative evaluation in some scenarios.

The parameters generated for *Diwall*'s detector are not universally applicable, and may vary under different conditions. K1 and K2, and the selection of microarchitectural events may depend on the monitored window set by the IoT stack software. Using a simplified MAC layer in training determines K1 and K2 and selects specific microarchitectural events, such as LD_STALL and BRANCH_TAKEN. This choice can be changed if a different MAC layer-monitored window is used. The generation of UCL and LCL can be influenced by changes in legitimate RSSI behavior from the gateway's legitimate node, and the RSSI values themselves depend on the location. Usually, jammers provide a greater RSSI value because of their proximity to the victim node, which will deviate up to the UCL. To overcome the limitations of *Diwall*, we consider the software configuration of *Diwall* parameters. New *Diwall* parameters can be generated after training and an update of *Diwall* configuration can be performed from the software using the provided framework.

Several IoT protocol developers utilize the LoRa PHY layer as a foundation and develop custom MAC layers to suit their requirements. Developers can employ a *Diwall* framework to enhance the security of their MAC layer implementations. To illustrate *Diwall*'s use cases, we integrated it with LoRaWAN, a widely used MAC layer for LoRa

in IoT end-devices. The following section delves into the integration of *Diwall* and the detection of attacks on LoRaWAN.

4.3 Use Case: LoRaWAN with *Diwall*

In this section, we study the integration of *Diwall* into the LoRaWAN end-node running the LoRaMAC-node stack developed by Semtech [76]. Our solution was tested using a real LoRaWAN network. This encompasses the LoRa PHY layer discussed earlier as well as the LoRaWAN MAC layer. This integration enables the detection of jamming and packet-injection attacks.

4.3.1 LoRaWAN Testbed

Integration of *Diwall* with LoRaMac-node Stack After successfully studying a simplified MAC layer and achieving promising detection rates, we extend *Diwall* methodology to the LoRaWAN protocol. Our initial implementation focused on the full LoRaWAN stack utilizing the RISC-V board support package, as illustrated in Figure 4.4. We introduced LitexLib, a C library, into the first layer of the LoRa driver, called Boards. This library facilitates the management of the SoC peripherals required by LoRaWAN, including handling interrupts with LibDIOs, managing timers with LibTimer, and interfacing with peripherals such as SPI and UART. We provide LibDiwall as part of LitexLib, which includes the necessary functions for configuring *Diwall* from software. Within the LoRaWAN protocol, the LoRaMAC layer is responsible for parsing LoRa frames. In our study, we integrate *Diwall* software instructions into this layer to enhance security.

The LoRaMac-node stack offers diverse application possibilities; however, for our research, we concentrate on a periodic uplink scenario. Uplink frames are transmitted from end-devices to gateways, whereas downlink frames are transmitted from gateways to end-devices. In periodic uplink applications, end-devices associated with Class A send uplink frames to gateways on a regular basis. Initially, these frames are in the form of join request frames as the end-devices seek to join the LoRaWAN network. Upon successful registration, the end-device receives downlink join acceptance. The objective of our study is to verify whether jammers are in proximity to the victim during the reception of downlink frames. Additionally, we aim to identify any attempts to inject fake join accept frames with larger packets, potentially leading to disruption of the LoRaWAN end-device.

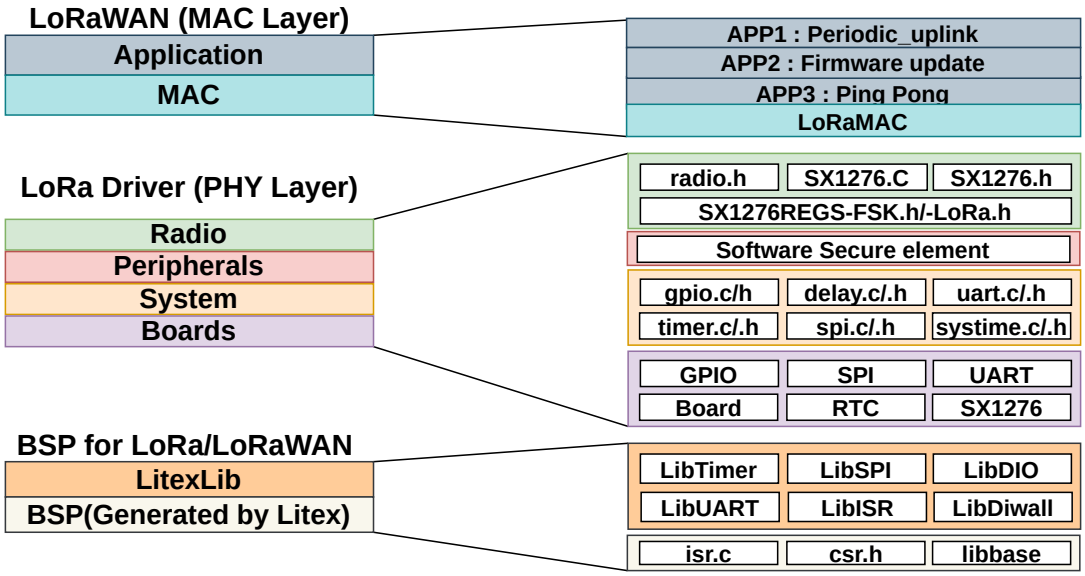


Figure 4.4 – LoRaMac-node Stack and LoRa Driver with RISC-V BSP

Within the LoRaWAN MAC layer, the frames that arrive from the radio and are managed by the LoRa PHY layer undergo parsing through the $OnRadioRxDone$ function located in LoRaMAC. Various functions in the LoRaWAN/LoRa context, including this particular function, are associated with hardware interrupts generated by the radio interface. The $OnRadioRxDone$ function, as depicted in Algorithm 2, was connected to the $DIO0$ interrupt. It signifies that LoRaWAN completes a received frame or the packet is successfully transmitted. The $OnRadioRxDone$ function facilitates the transfer of the message and its metadata (such as the RSSI, SNR, and size) from the LoRa PHY layer to the LoRaWAN MAC layer. Algorithm 2 presents the $OnRadioRxDone$ procedure. It has been instrumented with instructions to manage *Diwall* and utilize HPCs. We enhanced the $OnRadioRxDone$ function, as outlined in Algorithm 2, by incorporating instrumentation to initiate monitoring with Hardware Performance Monitoring ($HPM\_$) instructions whenever a frame is received. This instrumentation process involves utilizing the instructions provided by the RISC-V processor to control *Diwall* using CSR RISC-V. We include commands for *Diwall* configuration and control with HPM_reset, HPM_enable, and HPM_stop, functions provided by Lib*Diwall*. In conjunction with this instrumentation, we also ensure that the RSSI metadata associated with each received packet is sent to *Diwall* using $HPM\_ReadRSSI(rssi)$.

LoRa radio modules possess a FIFO buffer capable of holding up to 256 bytes, which

Algorithm 2 *OnRadioRxDone* reception function in LoRaMAC

```
1:  procedure OnRadioRxDone(payload, size, rssi, snr)   ▷ Triggered by interrupt
2:      ...
3:      Call HPM_Reset()                                ▷ Reset HPC1, HPC2 and NwHPC
4:      Call HPM_Enable()                               ▷ Select/Enable HPCs and Diwall(IDLE State)
5:      Recepetion_Buffer ← payload                     ▷ Getting MAC payload for LoRaMAC
6:      Size ← size                                     ▷ Getting size of payload
7:      Rssi ← rssi                                     ▷ Getting rssi metadata
8:      Snr ← snr                                       ▷ Getting snr metadata
9:      Call HPM_ReadRSSI(rssi)                         ▷ Send RSSI to NwHPC
10:     Call HPM_Stop()                                 ▷ Stop HPCs monitoring and analyze by Diwall
11:     ...
12: end procedure
```

matches the capacity of the frames of the LoRa PHY layer. However, the classes in this LoRaWAN MAC layer have their own reception buffers. The network packets exchanged between gateways and end-devices do not typically reach the maximum 256-byte size allowed by the LoRa PHY layer's standard.

We conducted a study on memory-corruption vulnerabilities within the reception buffers. This involved implementing buffer overflows for both the stack and the heap. We introduce packet injection scenarios similar to those explored in the previous LoRa section. Subsequently, we curated a new dataset and retrained the Decision Tree classifier. This effort led to the creation of a dedicated Decision Tree model designed specifically for detecting packet injection attacks for LoRaWANs. Although the new model retained the same structure as its predecessor, modifications were made to $K1$, $K2$, and HPC2 because of the usage of larger network packets and a new monitoring window designed for the LoRaWAN MAC layer.

The selection of the reception buffer size was based on the standard specifications for Class A as well as the frequency used by the radio center. Analyzing the frames exchanged between our end-device and LoRaWAN gateway revealed that the frame size for uplink and join accept frames typically ranged from 18 to 35 bytes. In alignment with the structure of the LoRaMAC payload and its equivalence to 51 bytes in Class A, we chose a reception buffer size of 51 bytes. The provided architecture featuring the updated *Diwall* now incorporates new parameters that were generated with the novel Decision Tree model for deployment on hardware.

The chosen parameter values are as follows:

— $K1$ is set to 55, with HPC1 representing the count of LD_STALL event.

— $K2$ is adjusted to 595.

— HPC2 refers to the number of instructions executed (INSTR) by the CPU during the period of parsing the MAC payload.

In this context, the INSTR value is a more important feature in Decision Tree classification for monitoring the LoRaMAC-node software window. Previously, BRANCH_TAKEN was used as the simplified MAC layer. This shift in relevance is owing to a new approach for parsing network packets, specifically, the LoRaMAC $OnRadioRxDone$ process.

The UCL and LCL parameters were software independent, and we retained the same values as before. We assumed that gateways tend to remain stationary for extended periods, resulting in relatively consistent RSSI values within a stable distance. Gateways will not provide RSSI values that deviate from EWMA control limits. Throughout this evaluation, we did not generate new parameters for the EWMA control limit. They remained unchanged and stable, as we maintained the same positions as previously used between the indoor gateway and end-device.

LoRaWAN Evaluation framework We assessed the performance of *Diwall* in detecting memory corruption based packet injection and jamming attacks on LoRaWAN end-devices. For that we employed the testbed highlighted in Figure 4.5. Our evaluation primarily focused on assessing the detection rates achieved by *Diwall*. For packet injection attacks, we crafted a LoRaWAN fake join to accept messages that adhered to the LoRaWAN MAC layer standard to achieve success in injection. To assess jamming attacks, we utilized random LoRa packets to disrupt the victim's communication in its channel. We introduced several modifications to the LoRa PHY layer of the victim device to ensure a successful injection and jamming. During the experiments, we configured the LoRa reception period to be continuously open to constantly receive frames. Because the LoRaWAN stack dynamically switches between three frequency channels (868.1, 868.3, and 868.5 MHz), we specifically operated on the first channel, 868.1 MHz, to concentrate on the jamming efforts and increase the likelihood of successfull attacks. To accelerate the evaluation process, we used LoRa with SF7, leading to extremely short intervals between transmitted frames. We examined over $23,000$ frames including legitimate packets, packet injection and jamming across two defined categories: Trigger and Continuous.

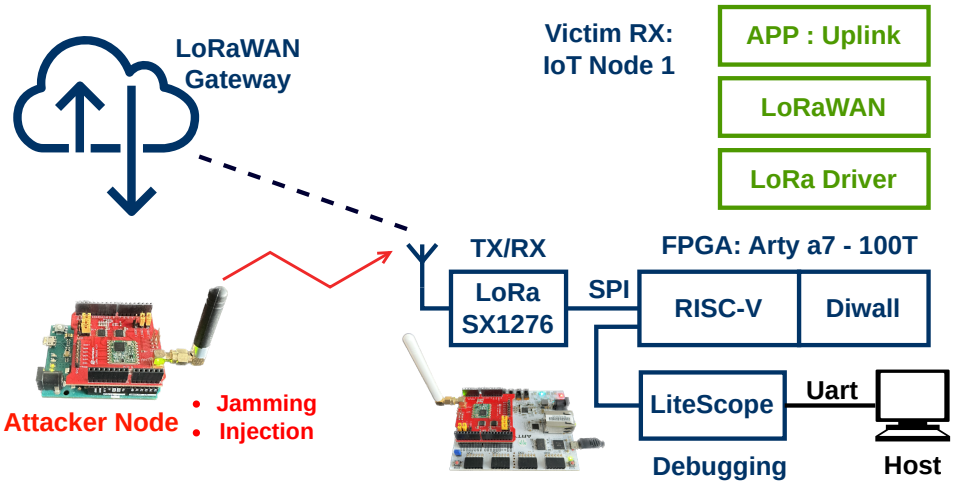


Figure 4.5 – Testbed Evaluation of Detection Rates for Diwall in LoRaWAN Networks

4.3.2 Results

Table 4.4 provides a summary of key metrics, including TP, TN, FP, FN, and accuracy rates, obtained during packet injection and jamming attacks on LoRaWAN network. Our observations for both packet injection and jamming attacks revealed detection accuracy rates that approached 99.98%. This high accuracy is notable because of the absence of FP and a minimal FN of approximately 1 (0.017%) to 2 (0.031%); every legitimate frame is recognized as a true negative, and the majority of alerts are raised. This demonstrates that *Diwall* does not generate false alerts during normal end-device operations or adherence to its policy. Of the 23,000 frames analyzed during the experiment, only three malicious network packets were incorrectly identified as false negatives. This result further highlights the efficiency of the proposed approach. This indicates that *Diwall* successfully identifies jamming and packet injection attacks in LoRaWAN networks, while maintaining a minimal rate of false negatives.

Table 4.4 – Diwall Detection Rates in LoRaWAN Network

Attacks	FP	FN	TP	TN	FNR	FPR	ACC
Packet Injection	0	1	5589	5630	0.017%	0%	99.99%
Jamming Trigger & Continuous	0	2	6335	5520	0.031%	0%	99.98%

F (False), N (Negative), T (True), P (Positive), R (Rate), ACC (Accuracy).

Figure 4.6 compares the code size percentages in two scenarios: one with and one

without *Diwall* integration overhead. It visually demonstrates how integrating *Diwall* impacts the code size of a LoRaMAC-node.

Integrating LibDiwall into an IoT protocol stack for *Diwall* configuration and control adds only 74 extra bytes to the code size. In the context of a LoRaMAC-node implementation in the RISC-V BSP, this represents 0.07% increase in code size.

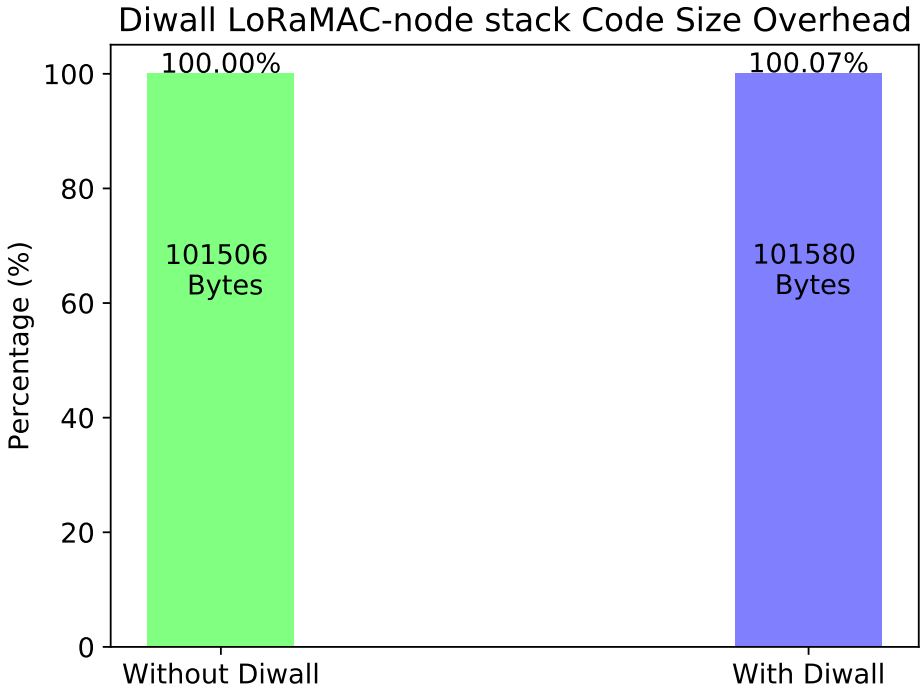


Figure 4.6 – Code Size Percentage Comparison with and without *Diwall* Integration Overhead

4.3.3 Discussion

In this study, we demonstrate the integration of *Diwall* into LoRaWAN networks. We equipped the LoRaMAC-node with functions provided by LibDiwall to enable *Diwall* to detect packet injection and jamming attacks. Within the LoRaMAC-node stack, various software parts can be monitored using the *Diwall* approach. We specifically selected a window within the LoRaMAC-node software that corresponds to the reception function of LoRa frames from the PHY layer. During the execution of this software window, *Diwall* monitors microarchitectural events and RSSI values. We introduced modifications to the monitored window to create vulnerabilities for packet injection attacks. After training, both K1 and K2 values were adjusted, and we chose a new microarchitectural event using HPC2 for the INSTR. These parameter changes were necessary because of the new software monitored window compared to the previous one used for a simplified MAC layer. In addition, we updated the Decision Tree model in *Diwall*. UCL and LCL values remained stable without any changes. The overall detection accuracy achieved in this experiment was approximately 99.98%. Instead of relying solely on software patches to address buffer overflow vulnerabilities, *Diwall* can monitor the network processor behavior and distinguish between legitimate and malicious activities. This integration of *Diwall* into the LoRaMAC-node stack provided a higher detection accuracy.

Certainly, *Diwall* has some limitations. It is vital to carefully choose the monitored window because buffer overflow can occur in different parts of the LoRaMAC-node. Notably, buffer overflows can exhibit similar behavior highlighted by excessive values of microarchitectural events. Another limitation is that introducing new instructions to the monitored window may impact the decision parameters, necessitating retraining. To address these challenges, one approach is to use *Diwall* on a larger monitored window that encompasses most of the MAC layer during PHY layer frame parsing. Alternatively, *Diwall* can be utilized with multiple monitored windows within the LoRaMAC-node software. This allows for the precise tuning of *Diwall* parameters based on the selected software window, which can be updated as needed. Users have the flexibility to define a *Diwall* monitored window anywhere in the IoT software. The software-based configuration of *Diwall* provides reconfigurability and enhances its overall flexibility.

4.4 Conclusion

This chapter outlines the implementation architecture and experimental evaluation of the proposed *Diwall* mechanism. The security strategy employed includes microarchitecture and network metric monitoring at the network processor level, with a focus on wireless attack detection. The designed *Diwall* structure comprises three core hardware elements: an HPMtracer responsible for HPC monitoring, EWMA preprocessing of RSSI values and a detector module. The detector module includes an EWMA control that utilizes calculated EWMA value of RSSI analysis to detect jamming attacks. A dedicated HPC is introduced to the RISC-V processor, which is assigned to monitor the RSSI metadata within the hardware. Furthermore, a Decision Tree model exploits a microarchitectural HPC analysis to identify packet injection using memory vulnerabilities. *Diwall* was implemented on an FPGA and integrated with a RISC-V processor. The initial evaluation involved a simplified MAC layer responsible for parsing network packets received from the LoRa PHY Layer. The MAC layer was equipped with the necessary instructions to manage *Diwall* from the software upon frame reception. In addition, this chapter highlights the crucial experimental assessment setup employed in real-world attack scenarios, and presents the achieved outcomes for LoRa/LoRaWAN communication.

Diwall achieves an impressive overall average detection rate of approximately 99.94% for the considered attacks. This highlights the exceptional capability of *Diwall* to identify and counter jamming and packet injection attempts. In the subsequent phase of our work, we implemented and assessed *Diwall* across the entire LoRaWAN stack. To achieve this, we updated our SoC using the requisite board support package incorporating the LoRaWAN MAC layer. During the implementation, *Diwall* effectively detected designated jamming and packet injection attacks within the LoRaWAN Network. The detection rates remained consistently high at 99.98%. *Diwall* demonstrates the ability to promptly detect packet injection by exploiting memory vulnerabilities, as well as real-time recognition of jamming attacks. This was achieved through low-complexity FPGA implementation. The FPGA implementation offers the advantage of easy and efficient updates through reconfigurable hardware. This eliminates the need for a complete system reprogramming when new attack scenarios emerge. Furthermore, the FPGA implementation introduces an area overhead of approximately 14.30%, consuming 22.89% of LUTs and FFs. Despite these additions, the maximum clock frequency of 65 MHz remained unaffected. A negligible increase in code size of 0.07% with respect to the LoRaMAC-node software, corresponding

to 74 bytes.

The results obtained in this chapter fulfilled the three fundamental requirements detailed previously.

Req1 - Lightweight & Local Analysis: *Diwall* monitoring and detection, implemented locally on an IoT end-device, incurs minimal overhead in terms of FPGA resource utilization, which remains within 13%. This has no discernible impact on the design performance. Only a few instructions were added to the software stack, ensuring no direct influence on memory usage or execution time. *Diwall* operation requires only a small number of clock cycles, and the energy cost within this timeframe is negligible compared to the overall SoC energy consumption.

Req2 - Multi-level Monitoring: *Diwall* uses HPCs as probes for monitoring, and are available locally in a processor network. Metrics from the microarchitecture and IoT stack metadata are used in *Diwall*, which proves the scalability of the multi-level approach at the hardware and network levels. *Diwall* can also track other metrics in the software or on other layers of an IoT protocol stack.

Req3 - Reconfigurability: *Diwall* structural model was initially applied to the LoRa PHY layer and a simplified MAC layer, demonstrating our approach. Upon achieving promising results with this approach, the parameters were readily reconfigured for seamless adaptation to the LoRaWAN MAC layer. The utilization of RSSI network metrics and microarchitectural HPCs is not tied to the specific LoRaWAN stack or RISC-V processor. This monitoring and analysis capability can be easily extended to alternative protocol stacks and different ISA microarchitectures. To address the reconfigurability requirement, we are also considering a full software-based configuration of *Diwall* parameters.

Conclusion & Future Perspectives

Conclusion

The IoT environment faces numerous challenges, and security is a significant concern. IoT devices attempt to address these issues by offering protective measures and updating the security protocols. However, the attack surface of IoT devices is expanding rapidly owing to the integration of built-in wireless connectivity in SoCs that handle multiple protocols. In this thesis, we argue that IoT devices require an additional security mechanism for monitoring and detection in addition to existing protection and update mechanisms. These mechanisms are crucial for tracking and analyzing hardware, networks, and software metrics at different levels of IoT SoCs. These metrics help to establish legitimate IoT SoC behavior and identify unusual activities. However, constrained IoT devices with limited resources in terms of computing capacity, memory, and power consumption pose a challenge to their implementation. Monitoring and detection mechanisms such as Intrusion Detection Systems require computing capacity and are software-based solutions that may not be suitable for IoT devices. They are often deployed on gateways and servers, where power and performance resources are larger than those of IoT devices. Given the growing attack surface and resource limitations of IoT devices, we conclude that any implementation of monitoring and detection should meet specific requirements: it should be lightweight, operate at multiple levels, and reconfigurable.

In this thesis, we introduce *Diwall*, a lightweight methodology for an HIDS that serves as a monitoring and detection security mechanism. Our approach focuses on monitoring and analyzing microarchitectural events and network metric data on resource-constrained IoT devices. *Diwall* was implemented within a RISC-V CPU, which functions as a network processor within a wireless connectivity system using the LoRa PHY layer. We utilize existing HPCs as probes to monitor metrics at both the network and microarchitecture levels. Specifically, microarchitectural events were counted by HPCs during the execution of a targeted monitoring window within the software. This monitoring helps to identify vulnerabilities related to memory corruption. Network metrics are monitored using a dedicated HPC on hardware and occur when a packet is received. Our

approach incorporates the Decision Tree machine learning classifier technique to detect packet injection, which is based on memory corruption and EWMA statistics for detecting jamming attacks. We conducted studies on attacks and metrics using the environmental framework methodology outlined previously. Initially, a simplified MAC layer was used to create datasets of microarchitectural events and network metadata, including the RSSI and SNR, in both simulated and real-world attack scenarios. The Decision Tree classifier selects features from the dataset of microarchitectural events and generates a model for distinguishing legitimate packets from those resulting from the packet injection. In addition, EWMA analyzes the RSSI metadata behavior to establish control limits and thresholds for identifying jamming attacks.

We evaluated the *Diwall* solution in real-world attack scenarios, covering the entire IoT protocol stack. Specifically, we replicated the jamming and packet injection attacks on the LoRa PHY and simplified MAC layer, achieving an impressive detection accuracy of 99.98%. Furthermore, we modified and evaluated *Diwall* to operate with the LoRaWAN MAC layer. When the same attacks were assessed using both the LoRa PHY and LoRaWAN MAC layers, we observed even higher detection rates. Additionally, in terms of FPGA implementation, there was an area overhead of approximately 14.30% and 22.89% of LUTs and FFs. It is worth noting that, despite these additions, the maximum clock frequency remained unaffected at 65 MHz. Regarding the software configuration and control of *Diwall*, there was a minor overhead in terms of code size, amounting to 0.07%. This corresponds to a mere 74 *bytes* of additional memory usage, which can be considered negligible.

In comparison to related works, *Diwall* introduces several key features that bring significant value:

i) **Local Analysis and Detection**: *Diwall* implements both monitoring and detection mechanisms directly on IoT devices, offering a local approach. This sets it apart from related works that often rely on hybrid placement, which may involve external components or cloud-based processing.

ii) **Multi-Level Approach**: The *Diwall* adopts a multi-level approach, tracking metrics at both the hardware and network levels. This comprehensive analysis enhances the understanding of IoT device behavior, in contrast to existing studies that typically focus on a single-level metric.

iii) **Flexibility and Reconfigurability**: *Diwall* provides customizable software setup for its metrics. We anticipate employing the same approach to update *Diwall* pa-

rameters, thereby enhancing its flexibility and reconfigurability to various scenarios and requirements.

iv) **Hardware-Based Implementation**: A novel feature of *Diwall* is its hardware-based implementation of monitoring and detection, controlled by software running on the network processor. This approach minimizes area overhead, reduces memory impact, and has a minimal impact on the execution time of the IoT protocol stack, whereas related works tend to be more software-based.

v) **Built-in CPU HPCs**: *Diwall* utilizes built-in CPU HPCs as monitoring probes, eliminating the need for additional area and performance overhead for adding probes at the system level. *Diwall* also extends the network processor architecture by incorporating dedicated HPCs designed specifically for monitoring network metrics.

vi) **ISA and IoT Stack Adaptability**: *Diwall*'s choice of the RISC-V ISA and IoT protocol stack, LoRaWAN, is not fixed. The concept could be integrated in other ISA-based CPU network processors, such as ARM or Xtensa. These are typically provided with HPCs for benchmarks and debugging. This ability extends their applicability to a wide range of devices. While *Diwall* was initially designed for LoRaWAN, it can be adapted to work with other protocol stacks, such as ZigBee and BLE, which are also susceptible to the considered attacks. Protocol developers using the LoRa PHY layer with proprietary MAC layers can benefit from *Diwall*'s approach and the proposed methodology with the associated experimental framework for generating new parameters and implementations of HIDS.

These key features collectively contribute to *Diwall*'s innovation and effectiveness in addressing the security concerns in IoT devices.

Future Work & Perspectives

In this thesis, we introduced the *Diwall* approach, which has proven effective in detecting wireless attacks using minimal FPGA resources and has minimal impact on performance and memory usage. *Diwall* successfully tackled the challenges and requirements posed by IoT devices with limited resources. However, future work will focus on further improving existing requirements and introducing new key features.

4.4.1 Improvements to Requirements Covered

Lightweight & Local Analysis Using additional HPCs (HPC1, HPC2 and NwHPC) for *Diwall* resulted in an FPGA area increase of +4.73% in LUTs and +7.58% in FFs. To reduce this overhead, we can move the network metadata metrics out of the HPCs. Instead of employing extra HPCs for each metric, we can place dedicated, low-overhead registers directly within *Diwall*. These registers can be configured and updated using software. Alternative methodologies can be explored to improve the area overhead caused by the Decision Tree and EWMA. In future work, we will examine and compare the overheads of these alternative algorithms when implemented in hardware. We plan to investigate other machine learning algorithms, such as KNN and Random Forest, because of their relevance compared with the Decision Tree. Additionally, we will explore statistical approaches for preprocessing and decision-making by comparing them with the currently used EWMA for RSSI. Methods such as statistical process control (SPC) will also be explored in future research.

Multi-Level Monitoring *Diwall* security mechanism addresses multi-level monitoring by tracking microarchitectural events from the CPU and RSSI network metadata from the PHY layer. To ensure behavior tracing across IoT devices, SoC-level monitoring (software, hardware, and network) is required. Integrating new monitored data at these levels is required to enhance the efficiency of *Diwall*. Metrics on the MAC layer can improve the detection rates for complex jamming and packet injection scenarios. These metrics include invalid CRC, the number of rejected frames, frames received with errors, and significant increases in these metrics can help detect attack attempts. The Packet Delivery Rate (PDR), calculated as the ratio of valid CRC packets to received preamble packets, combined with the RSSI metrics, aids in detecting selective jamming. A lower PDR over time indicates the presence of jamming. Another metric from the application layer is the inter-arrival time (IAT), which is the time between the received packets. Unusual IAT values may indicate the presence of an attack.

At the software level, metrics are worth exploring in security mechanisms, especially those responsible for transitions between user-level and low-level hardware. Several metrics can be used in *Diwall*. These include system calls, their frequencies, and unexpected system-call patterns. Monitoring transitions between User and Privilege Levels helps to detect unauthorized privilege escalation and resource access attempts. Timestamp metrics can be used in *Diwall* to capture value timings, measure time intervals between actions,

and identify malicious or out-of-order events. An extension of monitored microarchitectural events to other parts of the IoT stack helps detect buffer overflow vulnerabilities and exploits them in these areas.

Reconfigurability To enhance the reconfigurability, we plan to fully configure *Diwall* parameters from the software. The thresholds for each detection model used in decision-making are updated via instructions for writing data into dedicated registers. Another reconfigurability improvement involves enabling dedicated network packets for *Diwall* updates and reconfigurations. These can be transmitted from the IoT gateway to an IoT device.

To address the retraining limitations, in case of software modifications included to the IoT protocol stack or due to the IoT device mobility, we aim to implement a periodic update mechanism in which *Diwall*'s model can be regenerated. For instance, EWMA involves calculating the UCL and LCL parameters from RSSI values over a period of time. We could add an extra module to *Diwall*'s design that can receive packets dedicated to update UCL and LCL parameters or calculate new parameters during a period of time of reception of legitimate packets. This module must align with the conclusions drawn in this thesis regarding IoT devices and have a shorter time period to prevent attacks during *Diwall* update.

4.4.2 Flexibility and Security of Diwall

In addition to discussing improvements that could be made to the requirements, *Diwall* has potential security and flexibility enhancements:

i) Investigating and evaluating more complex jamming and packet injection attack scenarios with *Diwall*.

ii) Implementing *Diwall* on different IoT protocols (e.g., BLE or ZigBee) allows the same approach and metrics for attack detection to be tested.

iii) Integration with the IoT device OS (e.g., Zephyr and FreeRTOS) should be explored, especially for network processors handling multiple protocols.

iv) Noise on HPCs microarchitectural events for OS based implementation should be studied in future work.

v) Standardizing the interface between the CPU and *Diwall* is essential for portability on different CPUs and ISAs.

vi) Enhancing *Diwall* flexibility by providing a fully software re-configurable accelerator with algorithm and technique options for each monitored feature.

vii) Optimizing *Diwall*'s detection capabilities by implementing a single module for multiple attacks reduces the need for individual detection modules.

After the detection of an attack by *Diwall*, several countermeasures and actions could be launched with the alert signal :

i) After detecting attacks, IoT devices can take countermeasures, such as resetting the CPU or rebooting with new firmware to remove exploits.

ii) Communication with the gateway can be established to secure alert transmission, isolate affected devices, and ensure uninterrupted services.

iii) During jamming attacks, devices can be placed in sleep mode to conserve battery power or switch to another IoT protocol, if supported.

Diwall is proposed as a monitoring and detection mechanism in the wireless connectivity of an IoT device. However, the security considerations of *Diwall* should be addressed in future work as follows:

i) Protecting *Diwall* from unauthorized access and securing software configurations to prevent tampering are crucial.

ii) Evaluating security in the context of potential threats involving an attacker's perception of *Diwall* presence is crucial. This should include scenarios where attackers try to manipulate packets or adjust signal levels to evade detection by *Diwall*

iii) Addressing potential physical attacks (e.g., fault injection) that are not considered in the threat model is essential for *Diwall*'s security.

PUBLICATIONS AND PRESENTATIONS

International Conferences

i) **M. E. Bouazzati**, R. Tessier, P. Tanguy and G. Gogniat, "A Lightweight Intrusion Detection System against IoT Memory Corruption Attacks," 26th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS), Tallinn, Estonia, 2023, pp. 118-123, doi: 10.1109/DDECS57882.2023.10139718 ***Best Regular Paper Award*** [75]

Journal Publications

i) **M. E. Bouazzati**, P. Tanguy, G. Gogniat and R. Tessier, "Diwall : A Lightweight Host-based Intrusion Detection System against Wireless Attacks," To be submitted.

International Workshop

i) **Mohamed El-Bouazzati**, Philippe Tanguy, Guy Gogniat. Towards Low-Power and Low Data-Rate Software-Defined Radio Baseband with RISC-V Processor for Flexibility and Security. Workshop CryptArchi 2022, May 2022, Porquerolles, France. ⟨hal-04164363⟩ [77]

Posters

i) **Mohamed El-Bouazzati**, Philippe Tanguy, Guy Gogniat. Towards Low-Power and Low Data-Rate Software-Defined Radio Baseband with RISC-V Processor for Flexibility and Security. 15ème Colloque National du GDR SOC2, Jun 2021, Rennes, France. ⟨hal-04164388⟩ [78]

ii) **Mohamed El-Bouazzati**, Philippe Tanguy, Guy Gogniat. Towards Low-Power and Low Data-Rate Software-Defined Radio Baseband with RISC-V Processor for

Flexibility and Security. RISC-V Spring Week 2022, May 2022, Paris, France. ⟨hal-04164321⟩ [79]

Invited Talks

i) **Mohamed El-Bouazzati**, Towards Low-Power and Low Data-Rate Software-Defined Radio Baseband with RISC-V Processor for Flexibility and Security at Taltech University, Tallinn, Estonia, 2021.

ii) **Mohamed El-Bouazzati**, Processor with HIDS Using HPCs Against Wireless IoT Protocol Remote Attacks at University of Massachussetts (Umass), Amherst, MA, USA, 2022.

iii) **Mohamed El-Bouazzati**, A Lightweight HIDS against IoT Memory Corruption Attacks, Meeting with Creach'lab participant, Lab-STICC, Lorient, France, 2022.

Source Code and Datasets

i) Diwall framework with BSP software for LoRa, LoRaMAC-node and simplified MAC layer **https://github.com/mohamedElbouazzat/Diwall-framework.git**.

ii) CV32E40P extension with Diwall **https://github.com/mohamedElbouazzati/CV32E40P-Diwall.git**.

iii) Diwall post processing and generated datasets **https://github.com/mohamedElbouazzati/Diwall-Post-Processing.git**

Bibliography

[1] Eurostat, "Eurostat - internet of things (iot) usage in households and by individuals." https://ec.europa.eu/eurostat/databrowser/view/isoc_iiot_use/default/table?lang=en, 2020.

[2] I. Analytics, "Number of connected iot devices." https://iot-analytics.com/number-connected-iot-devices/, 2023.

[3] Statista, "Iot revenue worldwide by vertical." https://www.statista.com/statistics/1183471/iot-revenue-worldwide-by-vertical/, 2021.

[4] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou, "Understanding the mirai botnet," in *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, (USA), p. 1093–1110, USENIX Association, 2017.

[5] Statista, "Worldwide annual internet of things attacks." https://www.statista.com/statistics/1377569/worldwide-annual-internet-of-things-attacks/, 2023.

[6] Y. A. N. H. dos Santos, "How embedded tcp/ip stacks breed critical vulnerabilities," tech. rep., Black Hat Europe, 2020.

[7] F. R. Labs, "Amnesia:33, how tcp/ip stacks breed critical vulnerabilities in iot, ot and it devices." https://www.forescout.com/company/resources/amnesia33-how-tcp-ip-stacks-breed-critical-vulnerabilities-in-iot-ot-and-it-devices, 2020.

[8] L. Microsystems, "Limesdr mini rx & tx 10mhz - 3.5ghz full-duplex." https://www.passion-radio.fr/emetteur-sdr/limesdr-mini-667.html, 2017. CROWD-LIME-MINI-667.

[9] STMicroelectronics, "STM32WL55CC: Wireless System-on-Chip (SoC)." https://www.st.com/en/microcontrollers-microprocessors/stm32wl55cc.html, 2023. STMicroelectronics website.

[10] Texas Instruments, "CC1352R: Multi-Band Wireless SoC." https://www.ti.com/product/CC1352R, 2023. Texas Instruments website.

[11] Espressif Systems, "ESP32-H2: Wi-Fi and Bluetooth LE SoC." `https://www.espressif.com/en/products/socs/esp32-h2`, 2023. Espressif Systems website.

[12] EnjoyDigital, "LiteX: Open-source SoC builder and integration framework." `https://github.com/enjoy-digital/litex`, 2012. GitHub repository.

[13] A. Pullini, D. Rossi, I. Loi, G. Tagliavini, and L. Benini, "Mr.wolf: An energy-precision scalable parallel ultra low power soc for iot edge processing," *IEEE Journal of Solid-State Circuits*, vol. 54, no. 7, pp. 1970–1981, 2019.

[14] P. D. Schiavone, D. Rossi, A. Di Mauro, F. K. Gürkaynak, T. Saxe, M. Wang, K. C. Yap, and L. Benini, "Arnold: An efpga-augmented risc-v soc for flexible and low-power iot end nodes," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 4, pp. 677–690, 2021.

[15] A. Traber, F. Zaruba, S. Stucki, A. Pullini, G. Haugou, E. Flamand, F. K. Gurkaynak, and L. Benini, "Pulpino: A small single-core risc-v soc," in *3rd RISCV Workshop*, 2016.

[16] L. Delshadtehrani, S. Canakci, B. Zhou, S. Eldridge, A. Joshi, and M. Egele, "PHMon: A programmable hardware monitor and its security use cases," in *29th USENIX Security Symposium (USENIX Security 20)*, pp. 807–824, USENIX Association, Aug. 2020.

[17] C. Palmiero, G. Di Guglielmo, L. Lavagno, and L. P. Carloni, "Design and implementation of a dynamic information flow tracking architecture to secure a risc-v core for iot applications," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pp. 1–7, 2018.

[18] E. Cui, T. Li, and Q. Wei, "Risc-v instruction set architecture extensions: A survey," *IEEE Access*, vol. PP, pp. 1–1, 01 2023.

[19] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini, "Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2700–2713, 2017.

[20] P. Davide Schiavone, F. Conti, D. Rossi, M. Gautschi, A. Pullini, E. Flamand, and L. Benini, "Slow and steady wins the race? a comparison of ultra-low-power risc-v cores for internet-of-things applications," in *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pp. 1–8, 2017.

[21] P. D. Schiavone, D. Rossi, A. Pullini, A. Di Mauro, F. Conti, and L. Benini, "Quentin: an ultra-low-power pulpissimo soc in 22nm fdx," in *2018 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*, pp. 1–3, 2018.

[22] T. T. Network, "Lorawan documentation." `https://www.thethingsnetwork.org/docs/lorawan/`, 2023.

[23] A. Augustin, J. Yi, T. Clausen, and W. M. Townsley, "A study of lora: Long range & amp; low power networks for the internet of things," *Sensors*, vol. 16, no. 9, 2016.

[24] L. Alliance, "Lorawan specification v1.0.3." `https://resources.lora-alliance.org/technical-specifications/lorawan-specification-v1-0-3`, 2023.

[25] B. SIG, "Bluetooth core specification." `https://www.bluetooth.com/specifications/bluetooth-core-specification/`, 2019.

[26] Z. Alliance, "Zigbee specification." `https://csa-iot.org/developer-resource/specifications-download-request/`, 2015.

[27] Z. Alliance, "Ieee standard for low-rate wireless networks," *IEEE Std 802.15.4-2020 (Revision of IEEE Std 802.15.4-2015)*, pp. 1–800, 2020.

[28] Y. Chen, S. Lu, H.-S. Kim, D. Blaauw, R. G. Dreslinski, and T. Mudge, "A low power software-defined-radio baseband processor for the internet of things," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 40–51, 2016.

[29] S. Wu, S. Kang, C. Chakrabarti, and H. Lee, "Low power baseband processor for iot terminals with long range wireless communications," in *2016 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pp. 728–732, 2016.

[30] H. B. Amor, C. Bernier, and Z. Přikryl, "A risc-v isa extension for ultra-low power iot wireless signal processing," *IEEE Transactions on Computers*, vol. 71, no. 4, pp. 766–778, 2022.

[31] M. Xhonneux, J. Louveaux, and D. Bol, "A sub-mw cortex-m4 microcontroller design for iot software-defined radios," *IEEE Open Journal of Circuits and Systems*, vol. 4, pp. 165–175, 2023.

[32] M. Hessar, A. Najafi, V. Iyer, and S. Gollakota, "TinySDR: Low-Power SDR platform for Over-the-Air programmable IoT testbeds," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, (Santa Clara, CA), pp. 1031–1046, USENIX Association, Feb. 2020.

[33] Armis, "Bleedingbit vulnerability analysis." https://www.armis.com/research/bleedingbit/, 2019.

[34] T. B. team, "Loradawn." https://github.com/Lora-net/LoRaMac-node/security, 2020.

[35] E. Aras, N. Small, G. S. Ramachandran, S. Delbruel, W. Joosen, and D. Hughes, "Selective jamming of lorawan using commodity hardware," in *Proceedings of the 14th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, MobiQuitous 2017, (New York, NY, USA), p. 363–372, Association for Computing Machinery, 2017.

[36] Y. Zhang, J. Weng, R. Dey, Y. Jin, Z. Lin, and X. Fu, "Breaking secure pairing of bluetooth low energy using downgrade attacks," in *Proceedings of the 29th USENIX Conference on Security Symposium*, SEC'20, (USA), USENIX Association, 2020.

[37] R. Cayre, F. Galtier, G. Auriol, V. Nicomette, M. Kaâniche, and G. Marconato, "Wazabee: attacking zigbee networks by diverting bluetooth low energy chips," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 376–387, 2021.

[38] F. Hessel, L. Almon, and F. Álvarez, "Chirpotle: A framework for practical lorawan security evaluation," in *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '20, (New York, NY, USA), p. 306–316, Association for Computing Machinery, 2020.

[39] G. Avoine and L. Ferreira, "Rescuing lorawan 1.0," in *Financial Cryptography and Data Security: 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26 – March 2, 2018, Revised Selected Papers*, (Berlin, Heidelberg), p. 253–271, Springer-Verlag, 2018.

[40] R. Cayre, F. Galtier, G. Auriol, V. Nicomette, M. Kaâniche, and G. Marconato, "Injectable: Injecting malicious traffic into established bluetooth low energy connections," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 388–399, 2021.

[41] A. C. Santos, J. L. S. Filho, Á. Í. Silva, V. Nigam, and I. E. Fonseca, "Ble injection-free attack: a novel attack on bluetooth low energy devices," *Journal of Ambient Intelligence and Humanized Computing*, pp. 1–11, 2019.

[42] D. Antonioli, N. O. Tippenhauer, and K. Rasmussen, "Key negotiation downgrade attacks on bluetooth and bluetooth low energy," *ACM Trans. Priv. Secur.*, vol. 23, jul 2020.

[43] A. Mpitziopoulos, D. Gavalas, C. Konstantopoulos, and G. Pantziou, "A survey on jamming attacks and countermeasures in wsns," *IEEE Communications Surveys & Tutorials*, vol. 11, no. 4, pp. 42–56, 2009.

[44] H. Pirayesh and H. Zeng, "Jamming attacks and anti-jamming strategies in wireless networks: A comprehensive survey," *IEEE Communications Surveys & Tutorials*, vol. 24, no. 2, pp. 767–809, 2022.

[45] C.-Y. Huang, C.-W. Lin, R.-G. Cheng, S. J. Yang, and S.-T. Sheu, "Experimental evaluation of jamming threat in lorawan," in *2019 IEEE 89th Vehicular Technology Conference (VTC2019-Spring)*, pp. 1–6, 2019.

[46] S. Bräuer, A. Zubow, S. Zehl, M. Roshandel, and S. Mashhadi-Sohi, "On practical selective jamming of bluetooth low energy advertising," in *2016 IEEE Conference on Standards for Communications and Networking (CSCN)*, pp. 1–6, 2016.

[47] N. D. Matsakis and F. S. Klock, "The rust language," *Ada Lett.*, vol. 34, p. 103–104, oct 2014.

[48] A. Saeed, A. Ahmadinia, A. Javed, and H. Larijani, "Intelligent intrusion detection in low-power iots," *ACM Trans. Internet Technol.*, vol. 16, dec 2016.

[49] A. Tabassum, A. Erbad, and M. Guizani, "A survey on recent approaches in intrusion detection system in iots," in *2019 15th International Wireless Communications & Mobile Computing Conference (IWCMC)*, pp. 1190–1197, 2019.

[50] M. Eskandari, Z. H. Janjua, M. Vecchio, and F. Antonelli, "Passban ids: An intelligent anomaly-based intrusion detection system for iot edge devices," *IEEE Internet of Things Journal*, vol. 7, no. 8, pp. 6882–6897, 2020.

[51] P. Kasinathan, C. Pastrone, M. A. Spirito, and M. Vinkovits, "Denial-of-service detection in 6lowpan based internet of things," in *2013 IEEE 9th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pp. 600–607, 2013.

[52] P.-F. Gimenez, J. Roux, E. Alata, G. Auriol, M. Kaaniche, and V. Nicomette, "Rids: Radio intrusion detection and diagnosis system for wireless communications in smart environment," *ACM Trans. Cyber-Phys. Syst.*, vol. 5, apr 2021.

[53] M. Zhang, A. Raghunathan, and N. K. Jha, "Medmon: Securing medical devices through wireless monitoring and anomaly detection," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 7, no. 6, pp. 871–881, 2013.

[54] B. B. Zarpelão, R. S. Miani, C. T. Kawakani, and S. C. de Alvarenga, "A survey of intrusion detection in internet of things," *Journal of Network and Computer Applications*, vol. 84, pp. 25–37, 2017.

[55] I. Butun, S. D. Morgera, and R. Sankar, "A survey of intrusion detection systems in wireless sensor networks," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 1, pp. 266–282, 2014.

[56] W. Yan, S. Hylamia, T. Voigt, and C. Rohner, "Phy-ids: A physical-layer spoofing attack detection system for wearable devices," in *Proceedings of the 6th ACM Workshop on Wearable Systems and Applications*, WearSys '20, (New York, NY, USA), p. 1–6, Association for Computing Machinery, 2020.

[57] P. Kasinathan, G. Costamagna, H. Khaleel, C. Pastrone, and M. A. Spirito, "Demo: An ids framework for internet of things empowered by 6lowpan," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, (New York, NY, USA), p. 1337–1340, Association for Computing Machinery, 2013.

[58] S. Raza, L. Wallgren, and T. Voigt, "Svelte: Real-time intrusion detection in the internet of things," *Ad Hoc Networks*, vol. 11, no. 8, pp. 2661–2674, 2013.

[59] M. Bourdon, P.-F. Gimenez, E. Alata, M. Kaaniche, V. Migliore, V. Nicomette, and Y. Laarouchi, "Hardware-performance-counters-based anomaly detection in massively deployed smart industrial devices," in *2020 IEEE 19th International Symposium on Network Computing and Applications (NCA)*, pp. 1–8, 2020.

[60] R. Cayre, *Offensive and defensive approaches for wireless communication protocols security in IoT. (Approches offensives et défensives pour la sécurité des protocoles de communication sans fil de l'IoT).* PhD thesis, INSA Toulouse, France, 2022.

[61] OpenHWGroup, "Cv32e40p user manual." `https://docs.openhwgroup.org/projects/cv32e40p-user-manual/en/latest/`, 7 2023.

[62] A. Waterman, K. Asanović, and J. Hauser, *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*. SiFive Inc. and CS Division, EECS Department, University of California, Berkeley: RISC-V International, 20111203 ed., 2021.

[63] Embench, "About the embench results repository, benchmark results collection." https://github.com/embench/embench-iot-results/tree/8fab201, 2023.

[64] Antmicro, "Embench tester." https://antmicro.github.io/embench-tester/, 2023.

[65] S. P. Kadiyala, P. Jadhav, S.-K. Lam, and T. Srikanthan, "Hardware performance counter-based fine-grained malware detection," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 19, no. 5, pp. 1–17, 2020.

[66] M. Mushtaq, A. Akram, M. K. Bhatti, M. Chaudhry, V. Lapotre, and G. Gogniat, "Nights-watch: A cache-based side-channel intrusion detector using hardware performance counters," in *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, pp. 1–8, 2018.

[67] S. Das, J. Werner, M. Antonakakis, M. Polychronakis, and F. Monrose, "Sok: The challenges, pitfalls, and perils of using hardware performance counters for security," in *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 20–38, 2019.

[68] C. Details, "Cve-2022-0204 bluez: heap-based buffer overflow in the implementation of the gatt protocol." https://www.cvedetails.com/cve/CVE-2022-0204/, 2022.

[69] I. M. Martinez Bolivar, *Jamming on LoRaWAN Networks : from modelling to detection.* Theses, Institut National des Sciences Appliquées de Rennes, Jan. 2021.

[70] O. Osanaiye, A. Alfa, and G. Hancke, "A statistical approach to detect jamming attacks in wireless sensor networks," *Sensors*, vol. 18, p. 1691, May 2018.

[71] E. Testi, L. Arcangeloni, and A. Giorgetti, "Machine learning-based jamming detection and classification in wireless networks," in *Proceedings of the 2023 ACM Workshop on Wireless Security and Machine Learning*, WiseML'23, (New York, NY, USA), p. 39–44, Association for Computing Machinery, 2023.

[72] H. Ruotsalainen, G. Shen, J. Zhang, and R. Fujdiak, "Lorawan physical layer-based attacks and countermeasures, a review," *Sensors*, vol. 22, p. 3127, Apr 2022.

[73] M. Strasser, B. Danev, and S. Čapkun, "Detection of reactive jamming in sensor networks," *ACM Trans. Sen. Netw.*, vol. 7, sep 2010.

[74] H. Ruotsalainen, "Reactive jamming detection for lorawan based on meta-data differencing," in *Proceedings of the 17th International Conference on Availability, Reliability and Security*, ARES '22, (New York, NY, USA), Association for Computing Machinery, 2022.

[75] M. E. Bouazzati, R. Tessier, P. Tanguy, and G. Gogniat, "A lightweight intrusion detection system against iot memory corruption attacks," in *2023 26th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, pp. 118–123, 2023.

[76] Semtech, "LoRaMac-Node Repository." `https://github.com/Lora-net/LoRaMac-node`, 2013. Accessed: August 30, 2023.

[77] M. El-Bouazzati, P. Tanguy, and G. Gogniat, "Towards Low-Power and Low Data-Rate Software-Defined Radio Baseband with RISC-V Processor for Flexibility and Security," in *Workshop CryptArchi 2022*, (Porquerolles, France), May 2022.

[78] M. El-Bouazzati, P. Tanguy, and G. Gogniat, "Towards Low-Power and Low Data-Rate Software-Defined Radio Baseband with RISC-V Processor for Flexibility and Security." 15ème Colloque National du GDR SOC2, June 2021. Poster.

[79] M. El-Bouazzati, P. Tanguy, and G. Gogniat, "Towards Low-Power and Low Data-Rate Software-Defined Radio Baseband with RISC-V Processor for Flexibility and Security." RISC-V Spring Week 2022, May 2022. Poster.

COLLEGE MATHSTIC
DOCTORAL BRETAGNE
BRETAGNE OCEANE

Université Bretagne Sud

Titre : Un Système de Détection d'Intrusion Léger Basé sur l'Hôte Utilisant un Moniteur Assisté par Matériel pour Détecter les Attaques sans Fil Ciblant les Dispositifs IoT Contraints

Mot clés : HIDS, RISC-V, LoRaWAN, Brouillage, Injection de Paquets, HPCs, EWMA, FPGA

Résumé : La croissance rapide des applications de l'Internet des objets (IoT) dans divers secteurs a entraîné une augmentation significative du nombre d'appareils IoT. Cela a conduit au déploiement de nombreux protocoles IoT pour offrir une connectivité accrue. Cependant, cette adoption extensive les a également rendus vulnérables aux attaques. En particulier, les attaques visant les capacités de communication sans fil représentent une menace importante. De telles attaques exploitent diverses vulnérabilités dans l'unité de connectivité sans fil, compromettant ainsi leur sécurité. Pour contrer cette menace, nous proposons un Système de Détection d'Intrusion Hôte (HIDS) contre les attaques sans fil. Ses composants sont personnalisés pour prendre en charge les terminaux IoT utilisant des protocoles de débit de données basse fréquence dans les domaines gigahertz (GHz) et sous gigahertz (sub-GHz). Le HIDS déploie un traceur matériel pour surveiller la microarchitecture et les métriques réseau en utilisant des compteurs de performances matériels (HPCs). Il effectue une trace des données pour une unité de connectivité sans fil basée sur une architecture RISC-V 32 bits. Nous évaluons l'efficacité du HIDS dans la détection d'attaques par injection de paquets et de brouillage. Notre mise en œuvre FPGA du HIDS présente un surcoût en logique d'environ 14.30% et une pénalité de fréquence de fonctionnement et de taille de code de moins de 1% pour un processeur RISC-V.

Title: A Lightweight Host-based Intrusion Detection System using a Hardware-Assisted Monitor to detect Wireless Attacks Targeting Constrained IoT Devices

Keywords: HIDS, RISC-V, LoRaWAN, Jamming, Packet Injection, HPCs, EWMA, FPGA

Abstract: The rapid growth of Internet of Things (IoT) applications in various sectors has led to a significant increase in the number of IoT devices. This has led to the deployment of numerous IoT protocols to provide greater connectivity. However, this extensive adoption has also left them vulnerable to attack. In particular, attacks targeting the wireless communication capabilities are a significant threat. Such attacks exploit various vulnerabilities in the wireless connectivity unit, compromising its security. To counter this threat, we provide a Host Intrusion Detection System (HIDS) against wireless attacks. Its components are customized to support IoT end-devices using low-GHz and sub-GHz data rate protocols. The HIDS deploys a hardware tracer to monitor microarchitecture and network metrics using hardware performance counters (HPCs). It performs data tracing for a 32-bit RISC-V based wireless connectivity unit. We evaluate the effectiveness of the HIDS in detecting packet injection and jamming attacks. Our FPGA implementation of HIDS has a logic overhead of about 14.30% and a design frequency and code size penalty of less than 1% for a RISC-V processor.