EasyChair Preprint

№ 15742

Comparison Between in-Core Hardware IDS, off-Core Hardware IDS and Software IDS

Tianxu Li, Mohamed El Bouazzati, Camille Monière, Philippe Tanguy and Guy Gogniat

January 20, 2025

Comparison Between In-core Hardware IDS, Off-core Hardware IDS and Software IDS

Tianxu Li[1 [0000−0002−0677−6999]], Mohamed El-Bouazzati[1 [0000−0001−5830−6756]], Camille Monière[1 [0000−0002−4934−4969]], Philippe Tanguy[1 [0000−0003−4686−6435]], and Guy Gogniat[1 [0000−0002−9528−5277]]

Univ. Bretagne-Sud, UMR 6285, Lab-STICC, F-56100 Lorient, France

Keywords: Embedded System Security, Intrusion Detection System, Internet of Things, Wireless Security

Abstract Wireless attacks targeting the Internet of Things (IoT) pose challenges to its security. To counter this threat, in-depth security mechanisms such as Intrusion Detection Systems (IDSs) are used. The implementation of IDSs in edge devices is challenging, considering the inherent constrained nature of IoT devices. In this paper, three Intrusion Detection System (IDS) implementation approaches, software, in-core hardware, and off-core hardware are defined and compared, using an IoT-context representative case study. Advantages and disadvantages of each approach are assessed and discussed, comparing design time, ease of maintenance, detection performance and SoC resource consumption. Our results, relative to the SoC baseline, show that the software approach used 17.92% more energy consumption per packet ($+0.19mJ/p$) than the hardware approach. Conversely, the hardware approach incurs a higher FPGA resource overhead, requiring up to 12.06% more LUT and 7.75% more FF.

1 Introduction

The Internet of Things (IoT) refers to a network of connected physical devices gathering and sharing data to ease everyday tasks and increase productivity. However, as the number of devices continuously grows, security matters become more pressing, especially in wireless communication. Threats like jamming, replay attacks, and memory corruption threaten communication confidentiality, device security and data integrity. Therefore, protecting IoT devices from these threats is crucial to ensure the reliability of the IoT [16].

Detecting threats or attacks is a task commonly assured by an Intrusion Detection System (IDS). It is a security mechanism designed to monitor networks or systems for malicious activities. It analyses data using techniques like signature-matching and anomaly-based detection, gathering information at various levels of the IoT device: hardware, software and network. When a potential threat is detected, the IDS sends an alert to network administrators, or to automated systems (Security Operations Center), enabling a rapid response to mitigate or even prevent the attacks [18].

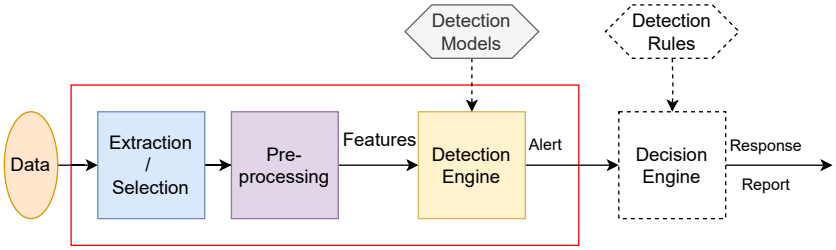

Figure 1: Functional Model of an IoT-targeting IDS

This paper aims to compare several implementation variations of an on-edge IDS targeting resource-constrained IoT devices, which can detect various types of wireless attacks on low-rate wireless communication protocols. We evaluate the IDS performances in the context of the LoRa physical layer, using a system composed of COTS devices and a RISC-V soft-core processor. The IDS can be implemented in software or in hardware. Each variant has its strengths and limitations, suitable for different network environments and security requirements. To better understand how these implementations perform, this paper quantitively compares several key performance indicators such as detection performance, SoC resource consumption and energy-efficiency. This comparison will help in evaluating the efficiency and practicality of the different implementations in real-world situations, thereby providing a scientific basis for designing more efficient embedded IDSs and thus, more reliable and secure IoT systems.

In Section 2, the context and current studies on IDS architecture and implementation are reviewed. In Section 3, the threat model, the IoT-targeting IDS chosen as a case study are summarized, and the three implementation approaches explored are detailed. Section 4 presents the experimental setup and the results, and discusses the design choices based on key performance indicators. Finally, Section 5 provides conclusions and future perspectives.

2 Background and Related Works

Securing network communications, particularly through IDSs, is addressed in the literature [9] indeed. Figure 1 describes the flowchart and the main components of an IDS.

This work focuses on the design choices for implementing three key functions: *Extraction/Selection* (E/S), *Preprocessing* (PPC) and the *Detection Engine* (DE). Indeed, most studies focus on optimizing detection models and algorithms implemented in the detection engine or selecting the best features to enhance performance [12,7]. In addition to this, resource-constrained IoT devices often lack the processing power for efficient IDS implementation, necessitating hybrid strategies that offload computations to the cloud [15,13]. This approach can degrade throughput and network performance, especially in systems that prioritize flexibility to support various IoT protocols, as well as widening the attack surface. IDS implementations can be broadly separated into two categories, software and hardware.

Software implemented IDSs process the monitored data and make decisions in software executed by programmable circuits, leveraging the flexibility of microprocessors. They are widely adopted due to their configurability, short implementation time, and lack of need for specialized hardware [3]. However, they can put pressure on the CPU and consume more memory resources than hardware implementations, especially with heavy data traffic or complex attacks, which is challenging for resource-constrained IoT nodes. Nevertheless, Cayre et al. [1] proposed OASIS, a framework to implement an IDS for Bluetooth Low Energy (BLE) devices. The firmware of the BLE controller is instrumented to get features and detection modules are implemented to detect five low-level attacks. It is also extensible to support other attacks. However, the detection modules are based on heuristics, with each module targeting a specific attack. Therefore, it requires more resources to detect additional attacks.

In this paper, the term "hardware implemented IDSs" is used to describe two distinct types of design: (a) a standalone unit that integrates all the functionalities of an IDS, and (b) a unit that performs some tasks of an IDS. Considering this definition, a hardware unit commonly comes in two forms described thereafter, depending on where it is implemented in a System-on-chip (SoC). One is implemented outside the CPU core as an independent peripheral (therefore referred to as "Off-core"), connected to the SoC via Direct Memory Access (DMA) or through the on-chip interconnect bus. It can operate either autonomously or as an accelerator assisting the processor. For instance, hardware accelerators for neural networks assist processors in calculations [11], while IDSs such as [14] enforce security policies independently. Another approach consists in integrating IDSs directly into the processor [5] (therefore referred to as "In-core"). Since they process and analyse data directly at the hardware level, hardware IDSs offer faster response time, lower latency, and minimal utilization of the IoT system computing resources. Thus, existing hardware implementations are designed to allow high-speed network connections without compromising energy efficiency. However, hardware IDSs have higher implementation costs, longer development time, and are harder to modify or upgrade compared to software solutions. Zareen et al. [20] proposed an in-core solution for botnet detection that uses static and unique feature selection and extraction. They evaluated it for both classification accuracy and hardware implementation. El-Bouazzati et al. [2] proposed an IDS for LoRa devices that can detect ongoing remote attacks in real time, and that can be embedded in resource-constrained devices. It uses Hardware Performance Counters (HPCs) to collect data at the processor level and can detect memory corruption attacks. The feature extraction and selection module supports only one data source, i.e., the HPCs, but they can count different events. The implementation has been deployed on an FPGA and evaluated for detection performances and resource usage on a RISC-V-based SoC. However, the implementation supports only one attack type and lacks extensive evaluation in execution time and energy consumption.

There is currently a lack of research on the comparison of the different IDS implementation approaches, especially for detecting wireless attacks in constrained

devices. To the best of our knowledge, no studies specifically discuss hardware, software and hybrid implementations of IDS in this context. Therefore, a quantitative analysis is highly needed to highlight challenges like energy consumption, computation, and memory usage. Additionally, the ability to update the IDS to counter new attacks is an important metric. Regarding IDS selection, we revisited a previously mentioned work [2]. This IDS employs a hybrid signature- and anomaly-based algorithm using a decision tree, a method also utilized in other IDSs. Therefore, this research holds significant representativeness, while being open-source and not complex to re-use. For a fair comparison, we reimplemented and extended the in-core IDS from [2] and applied the same algorithm using software and off-core methods. Our main evaluation criteria include resource consumption, response time, flexibility, energy consumption, and adaptability, with a particular focus on comparing in-core and off-core approaches.

3 Case Study Description

In this section, we introduce our case study by defining the threat model. Then, we succinctly present the selected IDS, its analysing and detection methods, and how it responds to the threat model. Finally, we detail the three considered implementation approaches.

3.1 Threat Model

In this study, the attacker is assumed capable of conducting wireless attacks on embedded devices participating in a LoRa Wide Area Network (LoRaWAN®). The attacker can use dedicated COTS devices to launch attacks remotely, but also have access to versatile Software-defined Radio (SdR) devices. We assume the attacker cannot have physical access to the target device. However, the attacker can manipulate any layer of a communication stack, from the physical layer to the application layer.

Hessel et al.[6] provided an extensive review of LoRaWAN® vulnerabilities and the attacks exploiting them. They found that jamming attacks are common, either disrupting wireless signals directly or enabling more complex attacks like device impersonation. Such attacks can lead to man-in-the-middle scenarios, data leakage, or even complete network takeover. In their paper, the authors categorize jamming attacks based on the attacker's network knowledge and ability to gather nodes information. "Triggered jamming", where the jammer activates only when a preamble is detected, is particularly complex and stealthy. Due to the long transmission time of LoRaWAN® packets, attackers have ample opportunity to detect specific preambles and emit jamming signals during a single transmission. This type of attack is the focus of our study.

Given the increasing number of vulnerabilities in IoT protocol implementations [17], we also consider memory corruption attacks [4] for completeness. These attacks target the implementation of network protocol stacks. They consist in exploiting a memory weakness or vulnerability (such as a buffer overflow on a memory location) to erase a function's return address. This can destabilize programs, leading to denial of service or even to a Remote Code Execution

(RCE). In our case, we consider two types of memory corruption attacks, those involving stack overflow and those involving heap overflow.

The IDS takling those threats is described thereafter.

3.2 IDS Architecture and Detection methods

The IDS considered in this work integrates the one developed in [2], which can be assimilated to a Machine-Learning (ML)-based hybrid signature-based and anomaly-based IDS, leveraging HPCs as data source. It has been proven capable of detecting memory corruption attacks, and distinguishing between stack and heap overflow. In the generic model in Figure 1, the data would be the events generated by the core. Event selection is done offline, and extraction is performed by storing and retrieving data from HPCs. The decision engine is a decision tree. This approach may be less efficient with a context-switching software platform, like a real-time OS, but is sufficient for bare-metal software running on limited-resource devices.

A HPC is intended to enable developers to monitor processor events, offering insights into the system's performances during runtime. However, several studies have taken advantage of HPCs capabilities to develop robust security mechanisms, particularly for intrusion detection. The processor used in this study includes a clock cycle counter, a "instruction retired" counter, and 29 configurable counters capable of monitoring various events. Previous studies tested ten metrics under memory corruption attacks using a machine learning classifier and identified two key metrics: `BRANCH-TAKEN`, which counts branch instructions, and `LD-STALL`, which counts delayed load instructions. A decision tree model has been built for these two metrics, for detection and classification purposes. In our re-implementation, we built a new dataset and retrained the model to get decision tree parameters that fit our research environment.

The IDS can be reduced to two main components: the metric collection module and the decision-making module. The metric collection module configures the HPCs to monitor the two selected events, activating them during the processing of LoRaWAN® data packets and stopping them afterward. The decision-making module receives data from the HPCs and uses a decision tree to detect anomalies. In our upcoming comparative studies, these two modules will be deployed in various ways depending on the IDS implementation, described in Subsection 3.3.

To improve the representativeness of the IDS, a jamming detector based on statistical analysis of Received Signal Strength Indicator (RSSI) of received LoRaWAN® packets has been added. This extension allows detecting jamming attacks [19]. In the generic model in Figure 1, the data are the LoRa packets. Extraction consists in retrieving the RSSI from the COTS device, and the value is preprocessed through an Exponentially-Weighted Moving Average (EWMA) filter. The decision engine checks that the resulting value is between upper and lower thresholds. The EWMA filter allows reducing spurious values impact, following the formula: $y_n = \frac{1}{4}x_n - \frac{3}{4}y_{n-1}$, with y_n the EWMA value for the nth received packet, and x_n the RSSI value of the nth received packet. The thresholds are obtained by statistical analysis of the values for legitimate packets over a trustworthy channel. Similar to the IDS designed for memory corruption at-

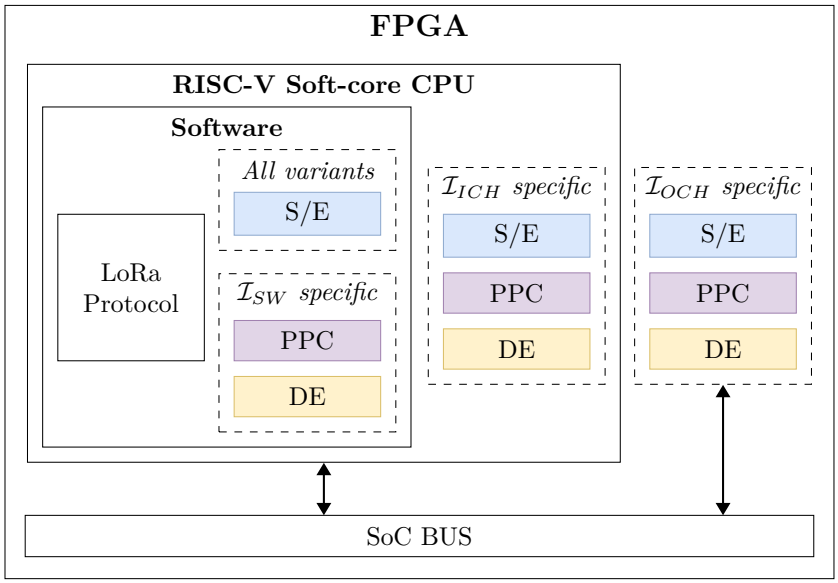

Figure 2: Selection / extraction (S/E), preprocessing (PPC), and decision engine (DE) function implementation sites (software, CPU logic and peripheral logic) depending on the implementation variant.

tacks, this extended IDS for jamming attacks also consists of three key modules: RSSI collection, EWMA preprocessing, and a threshold-based decision module. As previously, these modules will be implemented differently based on IDS implementation variants, described in Subsection 3.3

The IDS described in this section, thereafter simply referred to as "the IDS", should have low power and low compute resource requirements, while still addressing the challenges of the considered threat model. The next section is dedicated to the description of the three implementation variants of the IDS.

3.3 Considered Implementation Approaches

In this paper, we propose to study three kinds of implementations, (1) a software implementation $\mathcal{I}_{SW}$, (2) an in-core hardware one $\mathcal{I}_{ICH}$, and (3) an off-core hardware implementation $\mathcal{I}_{OCH}$. This section describes how the focused blocks in Figure 1 are dispatched between the software, the CPU circuit, and eventual peripherals as shown in Figure 2. Data are not represented to lighten the figure, since all approaches use the same, i.e., RSSI values reported by LoRa COTS devices retrieved in software, and HPC values retrieved directly from the CPU.

In the software approach $\mathcal{I}_{SW}$, the extraction (S/E), the preprocessing (PPC), and the decision engine (DE) are run within a program executed by the CPU. In our case, this is achieved using a bare-metal program written in C language. As implied in Figure 2, it is the simplest approach from a design point-of-view, since all components live in the same domain. However, latency and throughput should be strongly impacted since any access to the hardware is constrained by API calls. Moreover, this approach is subject to common software weaknesses,

whether they come from coding malpractice, or result from CPU vulnerabilities [10].

The $\mathcal{I}_{ICH}$ variant can be considered the conceptual opposite. All the IDS functions are implemented as hardware components of the CPU itself, only S/E is partially present in software, due to the requirement of the data source. This requires intimate knowledge of the CPU ISA, and to have access to its hardware description. However, this approach should have a low resource-access overhead, since all components can be tightly coupled, as well as the highest performance and energy efficiency.

In the same fashion, the $\mathcal{I}_{OCH}$ variant is also completely implemented in hardware, except for the software required by the data source. It takes the form of a peripheral connected to the CPU using available SoC interconnections. While still requiring hardware development, it is more independent of the core. It still requires some components to be available in the core, like HPCs in our case, but it is a common requirement of mainstream ISAs like RISC-V, ARM, or x86_64. It should be less efficient than $\mathcal{I}_{ICH}$ due to extra latency introduced by the bus, but still more than $\mathcal{I}_{SW}$ as it is not overly reliant on API calls.

To quantify their respective strength, a test platform and protocol has been defined and is described in the next section.

4 Results and Discussion

This section outlines the experimental setup, then it presents the results that enabled us to compare each approach. Before concluding, we provide insights into the advantages and disadvantages of each method.

4.1 Test bench

To fairly compare the three IDS implementations, we ensured identical experimental conditions for each implementation, using the protocol shown in Figure 3. Test data are replayed by the SdR through shielded cables to an IDS-enabled device, with adjusted gain to ensure consistent Signal-to-Noise Ratio (SNR). The IDS-enabled device is composed of a LoRa shield COTS device connected to an Artix 7 Xilinx FPGA (xc7a100t) using SPI. In the FPGA, a RISC-V RV32IMC processor (CV32E41P) is deployed. RISC-V allows modifying freely the core description. The hardware is described in System Verilog, and the complete SoC is generated using LiteX [8], an open-source Python library that streamlines hardware integration.

Exclusively used for evaluation, the test data consist of I/Q samples captured by mimicking full-scale scenarios, insuring that the IDS's behaviour closely mirrors actual deployment conditions. For each attack type considered, we recorded 400 packets using a SdR device and a LoRa Shield at a distance of 33 meters with direct line-of-sight. This included 200 genuine packets and 200 packets attacked by an independent LoRa COTS device.

This test bed allows us to verify the detection performances for each IDS implementation. Since we used the same IDS model across all three architectures, performance indicators like precision, accuracy, and F1 score are consistent. The packet loss rate is approximately 2% for legitimate packets and memory corrup-

tion attack packets, and about 50% for jamming attack packets, thus resulting in 26% in average. Only successfully decoded packets were accounted, since the IDS cannot detect packets that are not received, as stated in Section 3.2. Nevertheless, for jamming attacks, the performance metrics of the three IDS architectures are equivalent. For the memory corruption attack, the result also showed an identical detection performance, with all attacks successfully detected and no false positives. This outcome aligns with our expectations: IDSs using the same model yield the same (or nearly identical) results across different architectures.

4.2 Results

This section compares the resource consumption of different IDS architectures to provide quantitative reference results for selecting the appropriate deployment strategy.

First, we focus on the hardware resource consumption required by the IDS, reported in Table 1, and obtained after place-and-route and bitstream generation using the synthesis tool (namely, Xilinx Vivado 2022). For the software variant $\mathcal{I}_{SW}$, the only additional hardware resources consumed compared to the basic SoC correspond to the overhead induced by adding HPCs to the RISC-V core, a common requirement of all IDS implementation approaches. Unsurprisingly, hardware IDS variants consume significantly more hardware resources than the software one. The $\mathcal{I}_{OCH}$ consumes slightly more hardware resources than the $\mathcal{I}_{ICH}$ because it requires extra logic links on the interconnect bus to transmit monitored data. However, this increase is minimal - only 0.41% more FF and 2.00% more LUT compared to the $\mathcal{I}_{ICH}$. This is because even though the $\mathcal{I}_{ICH}$ is integrated within the CPU core, it still requires additional connections to transmit monitored indicators and receive results, consuming resources similar to those used by a lightweight interconnect bus. The difference in hardware resources is insignificant for the entire SoC. However, modifying the core to add new IDS modules and new connections supposes access to the hardware description, rarely provided with closed-source ISA like ARM.

We also consider the maximum operating frequency of the IDS. For $\mathcal{I}_{SW}$, its maximum frequency theoretically matches that of the CPU (CPU Maximum Frequency, CMF). From the perspective of the overall system's maximum operating frequency, there isn't much difference. This is because the hardware IDS we selected and built is lightweight and high-speed, with the $\mathcal{I}_{ICH}$ operating maximum at 218.05 MHz and the $\mathcal{I}_{OCH}$ maximum at 220.60 MHz. The SoC's timing constraints primarily stem from other components, like the core and peripherals.

The usage of software resources will also vary depending on the IDS architecture. Even for hardware IDS, the in-core and off-core architectures will have different resource consumption due to the varying software support needed

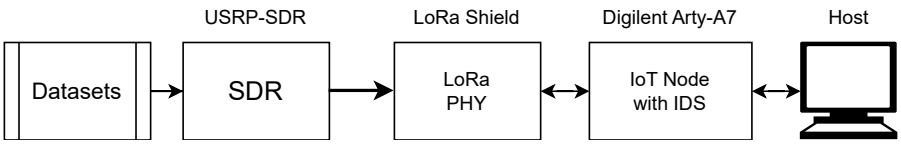

Figure 3: Dataset replay setup to test the IDS

Table 1: Resource consumption and maximum achievable CPU frequency (CMF) without IDS ($\mathcal{I}_{base}$) with the software IDS ($\mathcal{I}_{SW}$), the in-core IDS ($\mathcal{I}_{ICH}$), or the off-core IDS ($\mathcal{I}_{OCH}$) on an Artix 7 FPGA (xc7a100t).

Implem.	**Soft-core CPU's FPGA resources**					**SoC** (*Freq:* 50 *MHz*)			
	LUTs		**FFs**		**CMF**	**LUTs**		**FFs**	
	Value	*Cost (%)*	*Value*	*Cost (%)*	*MHz*	*Value*	*Cost (%)*	*Value*	*Cost (%)*
$\mathcal{I}_{base}$	4676	N/A	2136	N/A	65.69	4800	N/A	7887	N/A
$\mathcal{I}_{SW}$	4777	2.16	2217	3.79	65.60	4883	1.72	8044	1.99
$\mathcal{I}_{ICH}$	5345	14.30	2625	22.89	65.07	5283	10.06	8466	7.34
$\mathcal{I}_{OCH}$	4777	2.16	2217	3.79	65.60	5379	12.06	8498	7.75

for collecting metrics, transmitting metrics, and obtaining results. Since we used bare-metal programming without a Real-Time Operating System (RTOS) in the resource-constrained IoT nodes, we measured software resource consumption by the size of the compiled binary files. As shown in Table 2, the code size increases significantly with the addition of the $\mathcal{I}_{SW}$, resulting in a 7.64% increase for the resource-constrained IoT node. However, the specific increase of 1566 bytes is acceptable for our lightweight IDS, but may be more impactful with complex IDS implementations. $\mathcal{I}_{ICH}$ has the smallest increase in code size, only 0.58%, while $\mathcal{I}_{OCH}$ requires an additional link to interconnect bus, slightly increasing the code size by 342 bytes. However, it still offers advantages in memory space compared to $\mathcal{I}_{SW}$.

Next, we analysed the required clock cycles, including the IDS detection time (the number of cycles from receiving metrics to deciding) and the complete time (the total cycles from packet reception to processing completion). The number of clock cycles for all complete times and $\mathcal{I}_{SW}$ processing times were obtained using the core's built-in cycle counter. This counter starts when a packet is received or detection begins and stops at the end of processing. To ensure stability, we began measurements after processing 10 packets, then recorded the complete times for the next 30 packets and calculated the average. For the processing times of the two hardware IDS architectures, we used Vivado simulation software, building a test platform in System Verilog to obtain exact processing cycles through simulation and waveform observation.

In terms of the clock cycles required for data processing, both hardware IDS architectures are much faster than the software IDS. Regarding the complete cycles, the total processing time does not significantly increase for hardware IDS because they can handle metrics for jamming attacks and memory corruption attacks in parallel. On the other hand, the software IDS needs to execute sequentially, and with the added overhead of CPU control, the number of cycles of complete time increases significantly. $\mathcal{I}_{ICH}$ has an advantage over $\mathcal{I}_{OCH}$ in terms of metric transmission time, but this does not significantly impact the total time.

In practice, IoT nodes will not reach saturation due to the duty cycle regulations. Even the slower software IDS can complete processing before the next

Table 2: IDS implementation effect on firmware size and CPU clock cycles

IDS Implem.	Size *(bytes)*	Cost		Processing Cycles		
		Direct *(bytes)*	Relative *(%)*	Jamming Detection	Mem. Corr. Detection	Complete
$\mathcal{I}_{base}$	20,500	∅	∅	∅	∅	112,028
$\mathcal{I}_{SW}$	22,066	1,566	7.64	1,034	227	133,215
$\mathcal{I}_{ICH}$	20,618	118	0.58	3	1	112,457
$\mathcal{I}_{OCH}$	20,842	342	1.67	4	1	113,099

packet arrives. However, it is important to note that, for some time-sensitive applications, prolonged processing times may lead to response delays, causing the response packets to either be missed or rejected.

4.3 Discussions

Considering the previous results, several conclusions can be drawn. Both $\mathcal{I}_{ICH}$ and $\mathcal{I}_{OCH}$ variants require, on average, around +7.54% (+595) of FFs and +11.06% (+531) of LUTs compared to the SoC baseline. Activating the HPCs adds an extra overhead of +1.99% (+157) of FFs and +1.72% (+83) of LUTs, consistent across all IDS versions. Despite this, hardware IDS implementation eliminates the need for additional memory usage in an IoT device. In contrast, the software IDS requires an additional +7.63% (1566 bytes) of memory compared to the original firmware. It's also important to note that hardware solutions generally have longer design times than software ones.

The CPU Maximum Frequency (CMF) remains stable at around 65.60 MHz across all versions, unaffected by the implementation approach. The in-core and off-core can reach higher maximum frequencies (up to 218.05 MHz and 220.60 MHz) when operating independently. While the SoC reached 50.00 MHz due to other timing constraints, the use of another clock domain would allow benefiting from the in-core and off-core IDS maximum frequency. However, the software IDS operates at the standard SoC frequency, which means it will be slower, especially when handling more complex IDS tasks.

The processing time and, therefore, the response time, varies between approaches. The software IDS requires more time (+1261 clock cycles in total) to detect memory corruption attacks and jamming attacks, compared to the others, which require no more than 5 clock cycles to detect these attacks. In Table 2, the last column includes the number of clock cycles required to completely process a packet, which also accounts for the control required by each approach. We observe that this time is significantly higher in the software IDS, which is +18.91% (+21, 187 clock cycles) relative to the baseline processing time, corresponding to 423.74 μs of response time. The response time reaches 21.42 μs for the software version, offering more opportunities for a successful attack compared to hardware solutions.

Finally, energy consumption estimates using the Xilinx Vivado power estimation tool show an average power consumption of 470 mW for the SoC in

all approaches. As a comparison point, we calculated the coarse-grained energy consumption per packet (mJ/p) by multiplying the processing cycles of each approach by the package power consumption and dividing by the SoC clock frequency (50 MHz). The baseline energy consumption per packet is 1.06 mJ/p for both hardware implementations. However, it reaches 1.25 mJ/p for the software IDS due to additional CPU usage and processing cycles. Since the hardware resources consumed by our hardware-based IDS are negligible compared to the overall complexity of the SoC, which includes the processor and other peripherals, the primary factor influencing energy consumption is the number of clock cycles utilized. Thus, the results highlight the higher energy efficiency of hardware approaches compared to the software one, which can directly impact the overall battery lifetime of an IoT device.

5 Conclusion and Perspectives

Through a detailed comparison of different implementation approaches for an IoT IDS, in terms of hardware and software resource consumption, processing time, detection performance, and estimation of energy efficiency, we have identified distinct advantages and disadvantages for each approach. Hardware approach (both in-core and off-core) indeed demonstrates significantly faster processing speeds, thus reducing energy consumption and response time. Despite their higher resource consumption, requiring more LUTs and FFs, for lightweight IDSs, the benefits outweigh the costs. The in-core IDS has the lowest overall resource requirements, while the off-core IDS requires slightly more resources but is way more portable, as it is not entangled with the CPU microarchitecture. Software IDS, however, excels in flexibility and portability, allowing easy updates and configurations to counter new threats.

Among hardware options, the off-core approach thus stands out compared to in-core IDS due to its unique advantages. This dual capability to maintain efficient performance while offering flexibility and scalability makes it the suitable choice for the integration and expansion of current systems. Future work will focus on the study of more complex IDS for advanced cores such as CVA6 with embedded operating systems such as Zephyr RTOS. Portable IP-based IDS libraries with partial reconfiguration support are also targeted, to help developers improve security measures efficiently.

References

1. Cayre, R., Nicomette, V., Auriol, G., Kaâniche, M., Francillon, A.: OASIS: An Intrusion Detection System Embedded in Bluetooth Low Energy Controllers. In: Proceedings of the 19th ACM Asia Conference on Computer and Communications Security (ASIA CCS). ACM. https://doi.org/10.1145/3634737.3645004
2. El Bouazzati, M., Tessier, R., Tanguy, P., Gogniat, G.: A Lightweight Intrusion Detection System against IoT Memory Corruption Attacks. In: Proceedings of the IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS). IEEE. https://doi.org/10.1109/DDECS57882.2023.10139718

3. Eskandari, M., Janjua, Z.H., Vecchio, M., Antonelli, F.: Passban IDS: An Intelligent Anomaly-Based Intrusion Detection System for IoT Edge Devices . https://doi.org/10.1109/JIOT.2020.2970501
4. Github, Inc.: NVD - CVE-2022-39274, `https://nvd.nist.gov/vuln/detail/CVE-2022-39274`
5. Harris, A., Verma, T., Wei, S., Biernacki, L., Kisil, A., Aga, M.T., Bertacco, V., Kasikci, B., Tiwari, M., Austin, T.: Morpheus II: A RISC-V Security Extension for Protecting Vulnerable Software and Hardware. In: Proceedings of the IEEE International Symposium on Hardware Oriented Security and Trust (HOST). https://doi.org/10.1109/HOST49136.2021.9702275
6. Hessel, F., Almon, L., Hollick, M.: LoRaWAN Security: An Evolvable Survey on Vulnerabilities, Attacks and their Systematic Mitigation . https://doi.org/10.1145/3561973
7. Jan, S.U., Ahmed, S., Shakhov, V., Koo, I.: Toward a Lightweight Intrusion Detection System for the Internet of Things . https://doi.org/10.1109/ACCESS.2019.2907965
8. Kermarrec, F., Bourdeauducq, S., Lann, J.C.L., Badier, H.: LiteX: An open-source SoC builder and library based on Migen Python DSL. https://doi.org/10.48550/arXiv.2005.02506
9. Khraisat, A., Alazab, A.: A critical review of intrusion detection systems in the internet of things: Techniques, deployment strategy, validation strategy, attacks, public datasets and challenges . https://doi.org/10.1186/s42400-021-00077-7
10. Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre attacks: Exploiting speculative execution . https://doi.org/10.1145/3399742
11. Ngo, D.M., Temko, A., Murphy, C.C., Popovici, E.: FPGA Hardware Acceleration Framework for Anomaly-based Intrusion Detection System in IoT. In: Proceedings of the 31st International Conference on Field-Programmable Logic and Applications (FPL). https://doi.org/10.1109/FPL53798.2021.00020
12. Oh, D., Kim, D., Ro, W.W.: A Malicious Pattern Detection Engine for Embedded Security Systems in the Internet of Things . https://doi.org/10.3390/s141224188
13. Pongle, P., Chavan, G.: Real Time Intrusion and Wormhole Attack Detection in Internet of Things . https://doi.org/10.5120/21565-4589
14. Pontarelli, S., Bianchi, G., Teofili, S.: Traffic-Aware Design of a High-Speed FPGA Network Intrusion Detection System . https://doi.org/10.1109/TC.2012.105
15. Raza, S., Wallgren, L., Voigt, T.: SVELTE: Real-time intrusion detection in the Internet of Things . https://doi.org/10.1016/j.adhoc.2013.04.014
16. Schiller, E., Aidoo, A., Fuhrer, J., Stahl, J., Ziörjen, M., Stiller, B.: Landscape of IoT security . https://doi.org/10.1016/j.cosrev.2022.100467
17. Siwakoti, Y.R., Bhurtel, M., Rawat, D.B., Oest, A., Johnson, R.C.: Advances in IoT Security: Vulnerabilities, Enabled Criminal Services, Attacks, and Countermeasures . https://doi.org/10.1109/JIOT.2023.3252594
18. Soniya, S.S., Vigila, S.M.C.: Intrusion detection system: Classification and techniques. In: Proceedings of the International Conference on Circuit, Power and Computing Technologies (ICCPCT). https://doi.org/10.1109/ICCPCT.2016.7530231
19. Zahra, F.T., Bostanci, Y.S., Soyturk, M.: Real-Time Jamming Detection in Wireless IoT Networks . https://doi.org/10.1109/ACCESS.2023.3293404
20. Zareen, F., Fernandes Amador, M.A., Karam, R.: Hardware Immune System for Embedded IoT . https://doi.org/10.1109/TCSII.2022.3187312