Wireless Networks (2022) 28:3453–3476
https://doi.org/10.1007/s11276-022-03046-1

ORIGINAL PAPER

Benchmarking of lightweight cryptographic algorithms for wireless IoT networks

Soline Blanc[1] · Abdelkader Lahmadi[1] · Kévin Le Gouguec[2] · Marine Minier[1] · Lama Sleem[1]

Accepted: 8 June 2022 / Published online: 22 July 2022

Abstract
Cryptographic algorithms that can provide both encryption and authentication are increasingly required in modern security architectures and protocols (e.g. TLS v1.3). Many authenticated encryption systems have been proposed in the past few years, which has resulted in several cryptanalysis research work. In this same direction, the National Institute of Standards and Technology (NIST) is coordinating a large effort to find a new standard authenticated encryption algorithm to be used by resource-constrained and limited devices. In this paper, 12 algorithms of the 33 candidates of the Round 2 phase from NIST competition are being benchmarked on a real IoT test-bed. These 33 ciphers implement authenticated encryption with associated data which aims at preserving integrity, privacy and authenticity at the same time. In this work, we ported the 12 algorithms to different hardware platforms (an x86_64 PC, an AVR ATmega128, an MSP430F1611 and the IoT-LAB platform) and make a fair comparison between their performance. We adapted these algorithms to the Contiki operating system to evaluate the latency and efficiency of each algorithm on IoT applications deployed on a national experimental platform which is IoT-LAB. In addition, we used the FELICS-AE benchmark to quantify locally the RAM, execution time and code size of each algorithm. In fact, this work provides practical results of their performance in an IoT scenario which pave the way for further research on other algorithms, platforms or OS.

Keywords Lightweight cryptography, Internet of Things, Benchmarking · IoT-LAB · Felics-AE · NIST LWC round 2 candidates

1 Introduction

We live in an era where interconnected computing devices keep getting more numerous, while cyber-attacks keep getting more sophisticated and frequent. The need for new standards that can protect communications against such threats has increased. However, the available cryptography standards do not meet the requirements of the new challenges we face today where constrained devices are massively deployed in the Internet of Things. They have hardware limitations such as memory size and their battery life must be preserved. For instance, limited health sensors (e.g. heart pacemaker, brain simulator) are directly connected to a network to gather useful data. Security here plays a crucial role since unauthorized access to these critical devices can be life-threatening. Other examples are smart homes, green cities, supply chain management, etc. Lightweight cryptography in the last 10 years has resulted in more than 1400 papers which all aim at reducing resource consumption, software and hardware efficiency on different/limited platforms while remaining resilient against different kinds of attacks. To help in the process of development, evaluation and standardization of a suitable lightweight cryptographic algorithm, NIST has initiated the Lightweight Cryptography Project. Looking back

✉ Abdelkader Lahmadi
lahmadi@loria.fr

Soline Blanc
soline.blanc@loria.fr

Kévin Le Gouguec
kevin.legouguec@gmail.com

Marine Minier
marine.minier@loria.fr

Lama Sleem
lama.sleem@loria.fr

[1] Université de Lorraine, CNRS, Inria, Loria, 54000 Nancy, France

[2] Airbus CyberSecurity, Élancourt, France

at the NIST contests for the selection of new cryptographic standards [36, 37], algorithms with weak security designs were disqualified after the first evaluation phase. The evaluation and the benchmark of the proposed solutions play a major role in the evaluation of an algorithm on both hardware and software efficiency. Since benchmark frameworks allow for consistent evaluation, they are important not only in the selection process of new cryptographic standards, but also for carrying out a fair comparison of ciphers' performance in given usage scenarios.

In 2015, NIST organized a workshop on lightweight cryptography to discuss the security and resource requirements that should be available in a standard to secure IoT applications. NIST received and published 56 algorithm proposals, which include more than 200 AEAD cipher implementation variants. The final goal of this initiative is to find the best proposal that can be used in such limited devices. The proposed algorithms are based on authenticated encryption with associated data (AEAD). An Authenticated Encryption (AE) algorithm can be defined as a symmetric cryptographic algorithm that is capable of simultaneously preserving the confidentiality and authenticity of data [8].

In this paper, 12 candidate algorithms from NIST second round competition are benchmarked on different hardware platforms. Our first objective is to evaluate the ciphers by using the FELICS-AE benchamrk to measure for each algorithm three metrics: (1) RAM, (2) Execution time, and (3) Binary code size. These metrics could be computed for a PC, an AVR, an MSP430 and a 32-bit ARM processor. After that, the second objective of this work is to evaluate every algorithm on the IoT-LAB platform, which is a real test-bed that provides access to hundreds of IoT boards for experimenting and deploying across different sites. IoT-LAB is used in this study due to these advantages: (a) user-friendly interface; (b) multi-platform: offers experimentation boards; (c) multi-radio: the boards do not have the same radio chips; (d) multi-topology: different physical deployments; and finally (e) multi-OS: the different boards support one or more embedded OS.

1.1 Motivation and contribution

The goal of this work is to provide a broad overview over the ciphers performance footprints, as well as building a benchmark for the performance evaluation of the NIST round 2 candidate AEAD-algorithms[1] by using the platform IoT-LAB[2]. The benchmark code is available at this link: https://gitlab.inria.fr/anon_group/iotlabandnist/. Our benchmarking process is depicted in Fig. 1.

The four steps summarizing the motivation and the main contribution of our work are the following:

1. The NIST AEAD algorithms: The first step is to understand the NIST lightweight competition algorithms which are tested to be standardized as lightweight algorithms dedicated to authenticated encryption with associated data (AEAD). 33 public algorithms have been chosen for the round 2 competition and 12 of them are chosen in this work.
2. FELICS-AE: Adapting these algorithms to the FELICS-AE [34] platform is done by importing their NIST releases and adding their implementations to the platform, as well as one of the test vector provided in these releases. The platform then checks the implementations for their correctness on three different hardware platforms. Then, a benchmark process is carried out for the algorithms' performance in terms of cycle counts and memory usage. The current distribution of FELICS-AE includes only a few algorithms, but more could be easily added. It is based on the FELICS [18] platform[3] which is dedicated to the evaluation of stream and block ciphers that do not support authenticated encryption. This step is described in Sect. 3.2. Note that FELICS-AE and IoT-LAB support the same hardware platforms which guided our choice of using FELICS-AE in this step.
3. Contiki operating system: After evaluating the provided algorithms on FELICS-AE, we adapted their respective codes to the Contiki operating system which is supported by IoT-LAB. Contiki is dedicated to running on hardware devices that are severely constrained in memory, power, processing power, and communication bandwidth, such as embedded systems and old 8-bit hardware. We used here Contiki-NG, which is a new version of Contiki OS. It runs on a variety of platforms based on energy-efficient architectures such as the ARM Cortex-M3/M4 and the Texas Instruments MSP430. However, other operating systems could be considered such as Riot [39].
4. IoT-LAB: Finally, we used these algorithms on an IoT application that we deployed on the IoT-LAB platform to evaluate their performance from a networking perspective.

[1] https://csrc.nist.gov/Projects/lightweight-cryptography/round-2-candidates

[2] https://www.iot-lab.info/

[3] https://www.cryptolux.org/index.php/FELICS

Fig. 1 The steps of our benchmarking process

1.2 Organization

Section 2 describes the related work in the field of software performance evaluations of cryptosystems. Section 3 presents the IoT-LAB platform and details the 12 algorithms selected among the 33 NIST ligthweight candidates as well as their evaluation with FELICS-AE. In Sect. 4, we detail the way we instrument IoT-LAB to produce the seven identified metrics for the 12 algorithms. In Sect. 5 we provide the performance results while using two IoT nodes (one client and one server) for the seven identified metrics on the 12 algorithms. Finally, Sect. 6 concludes this paper.

2 Related work

In this section we present the related work regarding existing cryptographic algorithms benchmarking tools and their comparison with our methodology.

Many benchmarks have been proposed to evaluate the performance of cryptographic algorithms on both hardware and software [22, 31–33, 35]. For example, the **BLOC** [12] project is one of the first attempts to evaluate lightweight cryptographic algorithms on embedded devices. It is publicly available and contains one of the largest collection of algorithm implementations. In [13] the authors analyse the performance of lightweight cryptographic algorithms on wireless sensor nodes. The code is written in C and it targets the 16-bit MSP430F1611 device [40]. Three metrics are considered: execution time, RAM requirement and code size. However, the RAM is not computed correctly, since the unsigned int data type requires two bytes and not one byte as considered by the authors on 16-bit MSP430F1611 micro controller. Thus the identified RAM requirement is half of the actual value. Further more, the library is not flexible and does not allow the addition of new algorithms easily. Finally, some implementations of the studied ciphers do not verify the test vectors. Therefore, we avoided using this platform and studied other options.

Two years after the BLOC project, the University of Luxembourg provides the **FELICS platform** [18]. FELICS stands for Fair Evaluation of Lightweight Cryptographic Systems. This benchmarking framework is motivated by the need for a unified evaluation of lightweight block ciphers and stream ciphers performances. FELICS has a dedicated web page[4] where an open virtual machine can be downloaded and benchmarks are maintained. Designers could upload new ciphers and could get consistent and detailed feedback on how their cipher compares with the state-of-the-art. The tool can evaluate execution time, RAM footprint, and binary code size. The tool supports four microcontroller families: an 8-bit microcontroller (Atmel AVR ATmega128), a 16-bit microcontroller (Texas Instruments MSP430F1611) and finally two 32-bit microcontrollers (Arduino Due and ARM Cortex-M3). Finally, FELICS-AEAD was presented at the first NIST Lightweight workshop [21] as an extension of FELICS done by the University of Luxembourg allowing authenticated encryption. Unfortunately, we could not find the source code. In the same way, FELICS-AE, also presented at the third NIST workshop [34], is also an extension of FELICS with the additional functionality of the authenticated encryption[5].

The third platform that was also studied is the **eBACS Project** ECRYPT Benchmarking of Cryptographic Systems, which is considered as the first step to consistent evaluation of cryptographic primitives for software [9]. The web page of this platform describes how to add new implementations and how to collect the data for the existing implementations. It allows the benchmarking of algorithms implemented in C, C++ and assembly. The only metric extracted is the cycle count (speed) and the results are saved in a database in text format. The advantage of this platform is the variety of supported hardware platforms and architectures used to obtain the results while the drawback is supporting only one metric which is the execution time.

The fourth project is the **XBX Project** (eXternal Benchmarking eXtension) [41]. It allows the benchmarking of hash functions on different micro controllers. Two metrics are extracted which are the binary code size and RAM consumption. The code size is obtained through static analysis of the generated binary file. The RAM requirement is the sum of stack consumption and static RAM requirement obtained from the application binary. The framework is written in C, Perl and Bash. XBX is the first project to unify measuring the performances of software implementations of cryptographic primitives built for different embedded devices using the same evaluation methodology. The results in [42] are evaluated for eight different devices with 8-bit, 16-bit and 32-bit CPUs. However, its web page[6] is no longer maintained.

[4] https://www.cryptolux.org/index.php/FELICS

[5] the code is available on https://gitlab.inria.fr/minier/felics-ae.

[6] https://github.com/das-labor/xbx

3 Preliminaries

In this work, we relied on different platforms to build our benchmarking process. First, FELICS-AE is used to verify the test vectors of each implemented algorithm. Once the test vectors are verified and RAM size, code size and execution times have been obtained, IoT-LAB is used to analyse the performance of the algorithms on a real IoT network. Below, we describe the platform IoT-lab as well as the evaluation of the selected algorithms by using the FELIC-AE tool.

3.1 IoT-LAB: platform description

3.1.1 Overview

The IoT-LAB Platform [30] offers an easy way to deploy experiments involving IoT nodes. It is part of the FIT (Future Internet Testing) experimental facility, provided by five French institutions of higher education and research: UPMC, Institut Mines-Télécom, Inria, CNRS and University of Strasbourg. FIT is also part of OneLab [38] facility, aiming to facilitate experimentation for academic and industrial users. This facility is composed of four platforms: PlanetLab Europe, FIT CorteXlab, NITLab and FIT IoT-LAB. All these infrastructures are made available through a single entry point, providing access to the different test-beds.

The IoT-LAB Platform provides 1786 wireless sensors nodes, located on six different sites. These nodes can be selected by an authenticated user in order to be used in one (or more) experiment(s). For each of the chosen nodes, the user can then provide a firmware to be deployed on them, and select a profile defining what metrics will be measured while the experiment is running. It is done through an online dashboard, or through dedicated Python scripts (`IoT-LAB cli-tools` [27]). The collected metrics can then be accessed by the user through `ssh`.

3.1.2 Nodes, components and topologies

As previously mentioned, the IoT-LAB platform provides access to 1786 nodes. These nodes are located in six different sites: Inria Grenoble (640 nodes), Inria Lille (293 nodes), Inria Saclay (264 nodes), ICube Strasbourg (400 nodes), Institut Mines-Télécom Paris (160 nodes) and CITI Lab Lyon (29 nodes). Each site proposes a different topology. The nodes are divided into four main categories, regarding their architecture: there are 256 WSN430 nodes (of 868 MHz), 883 M3 nodes, 524 A8 nodes and 123 other "custom" nodes. Each node is identified by its site (Grenoble, Saclay, Lille, Lyon, Paris or Strasbourg), its architecture and an integer ID. For instance, the node *m3-21.lille.iot-lab.info* has *Lille* as site, *m3* as architecture and *21* as integer ID. Finally, each node has a state: it can be *Alive* if the node is available, *Busy* if it is currently used by an experiment, *Suspected* if it is not available, *Dead* if it is not working, or *Absent*.

Each node in IoT-LAB has three main components: the Open Node (ON), the Gateway (GW) and the Control Node (CN). The Open Node is flashed with the firmware provided by the user. During the experiment, it can be stopped, re-flashed, rebooted, etc. The Gateway provides a connection between the Open Node and the global infrastructure. Finally, the Control Node interacts with the Open Node in order to monitors its sensors. Thus, it handles the consumption and selects the power supply (battery or Power over Ethernet). The Gateway and the Control Node are the defined as "Host Nodes", so the user has not interactions with them.

Several network topologies for the different sites are also available. We define a topology for an experiment, which represents the way the nodes communicates with each other. It is defined by the direct communications between them. Thus, we define three types of topologies available for our experiments: the line topology, the grid topology and the star topology. Unfortunately, we only present in this paper results for the line topology as too many packets were lost in the other topologies and the obtained measurements are very noisy regarding the consumption of algorithms.

3.1.3 Profile and experiment

A profile is used to determine which metrics are collected when the experiment is running. In an experiment, a profile can be associated to a specific node. The profile creation form needs three main information: information related to the architecture, information related to the consumption and information related to the radio. Only the first one is mandatory: a profile can be set not to measure the consumption for instance. The architecture information is one of three options, related to the architecture of the associated node: "M3", "A8" or "Other". The consumption information is divided into three parts: whether or not to measure the current (in amperes), the voltage (in volts) or the power (in watts), the "period" and the "average". The "period", or "conversion times" (CT), and the "average" (AV) are used to define the periodic measure (PM) given by the formula: PM = CT * AV * 2. The periodic measure is used to configure the INA226, which is a component used to monitor the current/power. Moreover, the average value is used for the filtering of the signal. Thus, a greater number of averages leads to a noise reduction for the measurements. Finally, the information related to the radio

can be defined with two options: in the first option, the Retrieved Signal Strength Indication (RSSI) is measured, and in the other option, the traffic is sniffed. An experiment is launched by the user for a specific running time. Thus, one of its parameters is the duration. The other parameters correspond to the resources of the experiment. The resources are the nodes, their associated firmwares and their associated profiles. An experiment can be scheduled to a specific time, or can be launched as soon as possible. As soon as an experiment is defined, it has one of the following states: it can be Waiting when the experiment has not begun, Launching if it is beginning (i.e. the nodes are being set up for it), Running when the experiment is running, Terminated if the experiment is done, or Error if an error occurred (or if it has been manually stopped).

Usage. Interacting with IoT-LAB experiments and resources can be done through three main ways: using the dashboard, using command lines tools or using the Python library.

The dashboard is accessible through a web browser[7]. It allows authenticated users to monitor their experiments, their profiles and firmwares, and to get the testbed status.

The command line tools, developed in Python, are available on the IoT-LAB Github repository [29] and covered by the CeCILL v2.1 free software licence. They allow a user to manage their experiments and profiles, or to interact with running experiments. More documentation is available on the IoT-LAB CLI tools documentation [26].

The IoT-LAB client is a Python library to access the IoT-LAB API. Its source code is available on its Github repository [28], and the API is described in the IoT-LAB documentation [25].

3.2 Selected cryptographic algorithms

By lack of time, we have not implemented all the round 2 candidate algorithms of the NIST lightweight Competition. Indeed, for each algorithm, we must first test that the code correctly compiles on the dedicated hardware platform using FELICS-AE and, once done, we flash the corresponding code on nodes using the Contiki OS. Thus, we have arbitrarily decided to test the following list of 20 algorithms with all finalists included (among 33):

- Ascon-128 and Ascon-128a [20]
- Elephant [10]
- ForkAE-128 [2]
- GIFT-COFB [4]
- GRAIN-128AEAD [24]
- Isap [19]
- HyENA [14]
- Lilliput-AE [1]
- LOTUS-AEAD and LOCUS-AEAD [15]
- PHOTON-Beetle [5]
- PYJAMASK [23]
- Saturnin [11]
- SKINNY-AEAD [7]
- SPARKLE [6]
- Subterranean 2.0 [16]
- SUNDAE-GIFT [3]
- TinyJAMBU [44]
- Xoodyak [17]

In the context of the IoT-LAB evaluation, we compare 12 of those algorithms with the baseline application `no_enc` where no encryption and no authentication are used.

All the codes of these algorithms are available in the submission package of round 2 candidates[8]. The size of the key, of the nonce and of the tag are given in Table 1.

We provide in Table 2, in Table 3 and in Table 4 the evaluation results in terms of code size (Bytes), RAM (Bytes) and execution times (cycles) by using the FELICS-AE framework for the 20 algorithms on the platforms AVR ATmega128, MSP430F1611 and a classical PC. The codes are compiled with the option `-O3` and we measure the performances of encrypting 16 bytes of plaintext with 16 bytes of associated data.

In Table 2, we show the results in terms of RAM size, code size and execution time for the 20 algorithms when executing the algorithms on an AVR ATmega128. Two groups emerge regarding the execution time: one group composed of GIFT-COFB, Isap-A-128a, GRAIN-128AEAD, HyENA-128, LOCUS-AEAD-128, LOTUS-AEAD-128, Pyjamask-128, Elephant-160 and SUNDAE-GIFT-96-128 require more than 1 millions of cycles whereas the second group composed of Ascon-128, Ascon-128a, ForkAE-128, Lilliput-I-128, Lilliput-II-128, Romulus-M1-128, SKINNY-AEAD-M1-128, Saturnin-CTR-Cascade-256, Schwaemm256-128, PHOTON-Beetle-AEAD128 (with a dedicated code), Sub-terranean-SAE-128 and Xoodyak-128 require less than 550 000 cycles. The best performing algorithm in this category is Schwaemm256-128 (with a dedicated code) followed closely by PHOTON-Beetle-AEAD128 with also a dedicated code. Concerning code size, only six algorithms (PHOTO-N6Beetle-AEAD128, Ascon-128, Ascon-128a, Schwaemm256-128, Lilliput-I-128 and Lilliput-II-128) have a code size of less than or around 5000 Bytes. The other ones have higher code sizes up to 36 000 Bytes. Concerning used RAM, excepting HyENA-128, LOCUS-AEAD-128 and LOTUS-AEAD-128, all the algorithms require less than 1000 Bytes of RAM even going down to 200 or 300 Bytes.

[7] https://www.iot-lab.info/testbed/dashboard

[8] https://csrc.nist.gov/Projects/lightweight-cryptography/round-2-candidates

Table 1 Key, nonce and tag sizes for algorithms benchmarked with FELICS-AE

	Key: k	Nonce: n	tag size: τ
Ascon-128	128	128	128
Ascon-128a	128	128	128
Elephant-160	128	96	64
ForkAE-128	128	104	128
GIFT-COFB	128	128	128
Grain-128AEAD	128	96	64
Isap-A-128a	128	128	128
HyENA-128	128	96	128
LOCUS-AEAD-128	128	128	64
LOTUS-AEAD-128	128	128	64
Lilliput-I-128	128	120	128
Lilliput-II-128	128	120	128
PHOTON-Beetle-AEAD128	128	128	128
Pyjamask-128	128	96	128
Romulus-M1-128	128	128	128
Romulus-N	128	128	128
Saturnin-CTR-Cascade-256	256	128	256
Schwaemm256-128	128	256	128
SKINNY-AEAD-M1-128	128	128	128
Subterranean-SAE-128	128	128	128
SUNDAE-GIFT-96-128	128	96	128
TinyJAMBU-128	128	96	64
Xoodyak-128	128	128	128

Of course, those results highlight the better performances of a dedicated code.

In Table 3, we show the results in terms of RAM size, code size and execution time for the 20 algorithms when executing the algorithms on an MSP430F1611. The same group (GIFT-COFB, Grain-128AEAD, Isap-A-128a, HyENA-128, LOCUS-AEAD-128, LOTUS-AEAD-128, Pyjamask-128, Elephant-160 and SUNDAE-GIFT-96-128) and surprisingly PHOTON6Beetle-AEAD128 (with the reference code) have bad performances: more than 1 millions of cycles whereas the others require less than 500 000 cycles. The case of GRAIN is clearly particular and mainly depend on the code versions: with the ref code, the performances are bad but with the optimized one, GRAIN is among the best. Concerning the code size, 12 algorithms (ForkAE-128, HyENA-128, Pyjamask-128, LOCUS-AEAD-128, Romulus-N, GIFT-COFB, Isap-A-128a, Xoodyak-128, Romulus-M1-128 and LOTUS-AEAD-128) have a code size greater than 10 000 Bytes, all the other are beyond this bound with a special mention to Lilliput-II and Schwaemm256-128 where the code size does not exceed 3700 Bytes. For the RAM size, only HyENA-128, LOCUS-AEAD-128 and LOTUS-AEAD-128 have a RAM size greater than 1000 Bytes. We have to note that Grain with the optimized code has a required RAM less than 100 Bytes, closely followed by TinyJAMBU-128 adn Lilliput-II-128.

Thus, regarding the two embedded hardware platforms, the results of the algorithms have closely similar trends: they are good or bad for both.

Finally, in Table 4, we show the results in terms of RAM size, code size and execution time for the 20 algorithms when executing the algorithms on a classical PC (Intel Core^{TM} i5-3570 CPU @ 3.40GHz, 7.7GiB RAM). It seems that this evaluation is a little bit different, compared to the others. The eight algorithms (Grain-128AEAD, Elephant-160, PHOTON-Beetle-AEAD128, Grain-128AEAD, HyENA-128, LOCUS-AEAD-128, LOTUS-AEAD-128 and SUNDAE-GIFT-96-128) have execution time greater than 80 000 cycles. However, GIFT-COFB that behaves badly on small embedded platforms becomes a reasonable candidate here with an execution time equal to 18 000 cycles, the same remark holds for Pyjamask-128 that has also a lower execution time in PC compared to the other platforms. Special mention to Ascon-128 and Ascon-128a that require about 2000 cycles. Concerning code size, surprisingly, the behavior on a PC is closely the same as the behavior on the AVR: Grain-128AEAD, ForkAE-128, GIFT-COFB, HyENA-128, LOCUS-AEAD-128, LOTUS-AEAD-128, Pyjamask-128, Romulus-M1-128, Romulus-N and Subterranean-SAE-128 have a code size greater than 10 000 Bytes whereas Ascon-128 and Ascon-128a with the `ref` code have a code size of about 2 000 Bytes. Concerning RAM size, Grain-128AEAD (`ref`), HyENA-128, LOCUS-AEAD-128, LOTUS-AEAD-128 and SUNDAE-GIFT-96-128 require more than 2 000 Bytes of RAM whereas Ascon-128 (`opt64`), Ascon-128a (`opt64`) and Grain-128AEAD (`opt32`) require about 200 Bytes of RAM.

3.3 Discussion

Our results show that the algorithm Schwaemm256-128 is suitable for the AVR ATMEGA128 hardware platform since it provides well balanced performance regarding code size, RAM usage and execution time. For the MSP430F1611 hardware platform, the most suitable algorithm is GRAIN-128AEAD with optimized code (`opt32`). On PC hardware platforms, the most suitable algorithm is Ascon-128.

Thus, we find that analysing the performance of the algorithms on small embedded platforms remains important because the performance results obtained for a PC could hide different realities in terms of execution time, code size and RAM for many of them, in particular the MSP430 being the most consuming platform in terms of

Table 2 Benchmarking results using FELICS-AE on AVR ATmega128

	Version	CFLAGS	Code size (B)	RAM (B)	Execution time (cycles)
Ascon-128	ref	-O3	5838	265	193031
Ascon-128a	ref	-O3	6098	281	163965
Elephant-160	ref	-O3	5850	832	18525062
ForkAE-128	ref	-O3	36579	605	194175
GIFT-COFB	ref	-O3	17066	222	1000301
Grain-128AEAD	ref	-O3	6375	814	2886593
HyENA-128	ref	-O3	14712	1724	1199057
Isap-A-128a	ref	-O3	6790	301	2817987
LOCUS-AEAD-128	ref	-O3	19668	1462	1503685
LOTUS-AEAD-128	ref	-O3	23540	1508	1504857
Lilliput-I-128	felicsref	-O3	5258	265	111042
Lilliput-II-128	felicsref	-O3	4484	231	132532
PHOTON-Beetle-AEAD128	ref	-O3	10016	270	1114077
PHOTON-Beetle-AEAD128	avr8_speed	-O3	3504	89	59204
PHOTON-Beetle-AEAD128	avr8_lowrom	-O3	1556	89	99668
Pyjamask-128	ref	-O3	20510	700	2038726
Romulus-M1-128	ref	-O3	28314	398	416506
Romulus-N	ref	-O3	26622	377	127689
SKINNY-AEAD-M1-128	ref	-O3	11994	448	288521
SUNDAE-GIFT-96-128	ref	-O3	11674	225	1665220
Saturnin-CTR-Cascade-256	ref	-O3	19202	615	125303
Schwaemm256-128	ref	-O3	18525	313	151214
Schwaemm256-128	opt	-O3	5074	247	139847
Schwaemm256-128	avr	-O3	5776	222	41669
Subterranean-SAE-128	ref	-O3	11912	803	515615
TinyJAMBU-128	ref	-O3	5298	163	366342
TinyJAMBU-128	opt	-O3	6626	171	348636
Xoodyak-128	ref	-O3	7102	363	269422

execution time. Moreover, we also observe that dedicated codes are of course more efficient than reference codes. So, it is really important that authors provide dedicated codes for dedicated platforms. We also provide in Appendix 1 the results we obtained for the NIST finalists of an ARM Cortex-M3.

4 Our benchmarking framework with IoT-LAB

We will present in this section the way we have implemented the identified performance metrics on the 12 selected algorithms. First, those algorithms need to be compiled on Contiki before we can use them on the IoT application to deploy on IoT-LAB.

4.1 Architecture

The global architecture of our framework is presented in Fig. 2.

We describe here the different components interacting with the benchmarking tool. Two types of components can be distinguished: the external components and the internal components. The internal components are entities we developed and set up while the external components are external entities we are only using.

4.1.1 External components

Two external components with which the benchmarking tool interacts are the following: the IoT-LAB platform and the IoT-LAB tools. Their code can be retrieved from the IoT-LAB Git repository[9].

[9] https://github.com/iot-lab.

Table 3 Benchmarking results using FELICS-AE on MSP430F1611

	Version	CFLAGS	Code size (B)	RAM (B)	Execution time (cycles)
Ascon-128	ref	-O3	7638	258	475222
Ascon-128a	ref	-O3	7914	276	398238
Elephant-160	ref	-O3	6524	806	17073111
ForkAE-128	ref	-O3	24370	688	300268
GIFT-COFB	ref	-O3	12246	266	2943908
Grain-128AEAD	opt32	-O3	11032	86	244766
Grain-128AEAD	ref	-O3	6646	580	3601170
HyENA-128	ref	-O3	49064	1770	1181695
Isap-A-128a	ref	-O3	11692	300	8109430
LOCUS-AEAD-128	ref	-O3	16444	1478	1844356
LOTUS-AEAD-128	ref	-O3	19530	1520	1844030
Lilliput-I-128	felicsref	-O3	5048	352	151472
Lilliput-II-128	felicsref	-O3	4296	298	178331
PHOTON-Beetle-AEAD128	ref	-O3	7972	372	1125425
Pyjamask-128	ref	-O3	16318	732	2930109
Romulus-M1-128	ref	-O3	27096	490	471528
Romulus-N	ref	-O3	15040	526	187504
SKINNY-AEAD-M1-128	ref	-O3	9626	466	355783
SUNDAE-GIFT-96-128	ref	-O3	12122	228	4903216
Saturnin-CTR-Cascade-256	ref	-O3	11004	566	126453
Schwaemm256-128	ref	-O3	12483	292	132770
Schwaemm256-128	opt	-O3	3652	254	135686
Subterranean-SAE-128	ref	-O3	10630	816	506407
TinyJAMBU-128	opt	-O3	3806	142	299234
TinyJAMBU-128	ref	-O3	4038	170	314440
Xoodyak-128	ref	-O3	10119	316	174400

The IoT-LAB platform, described in the Sect. 3, allows an authenticated user to launch experiments on IoT assets. It then allows the user to access energy consumption data and traffic data. Thus, it handles the experiments and formats their related data.

The IoT-LAB tools are used to compile the firmwares in order to make them run on the IoT assets and to specify, retrieve and parse the requested experiments data.

4.1.2 Internal components

The benchmarking tool can be divided into three internal entities: the SQLite3 database, the local files and the Python scripts.

The SQLite3 database stores the experiments data, as well as the topologies and measured data.

The local files store the data files generated by IoT-LAB, which are retrieved by the Python script when an experiment ends. They could be considered as temporary files, as they are then parsed in order to extract metrics, which are then stored in the database. They present different formats: the consumption data is contained in OML files, the traffic data is contained in PCAP files. Log files are generated as well.

A set of Python scripts orchestrate all the benchmarking process. They parse the configuration file, create and fill the database, manipulate the local files and interact with the IoT-LAB platform and tools. A complete description of the scripts and the way they interact with all the components is given in Appendix 1. The Python scripts create the database, store and retrieve data from it by using the Database module. They also copy the files generated by IoT-LAB while running an experiment to local files. Then, they parse these files in order to sum them up in the database.

Moreover, the IoT-LAB tools propose an interface through which the Python scripts can communicate with the IoT-LAB platform. It namely includes an API to launch experiments by submitting their parameters, and a user space to retrieve the generated data.

Table 4 Benchmarking results using FELICS-AE on PC

	Version	CFLAGS	Code size (B)	RAM (B)	Execution time (cycles)
Ascon-128	opt64	-O3	9906	204	2256
Ascon-128	ref	-O3	2113	344	2282
Ascon-128a	opt64	-O3	12762	212	2529
Ascon-128a	ref	-O3	2208	344	2102
Elephant-160	ref	-O3	9284	1768	458421
ForkAE-128	ref	-O3	20534	920	28022
GIFT-COFB	ref	-O3	15456	428	18407
Grain-128AEAD	opt32	-O3	6184	168	100706
Grain-128AEAD	ref	-O3	12397	2044	151863
HyENA-128	ref	-O3	19896	1896	82328
Isap-A-128a	ref	-O3	4280	520	29535
Isap-A-128a	opt_64	-O3	21967	336	10578
LOCUS-AEAD-128	ref	-O3	28400	1960	127586
LOTUS-AEAD-128	ref	-O3	36043	2048	127831
Lilliput-I-128	felicsref	-O3	5811	608	14621
Lilliput-II-128	felicsref	-O3	5359	560	17015
PHOTON-Beetle-AEAD128	ref	-O3	8917	536	288870
PHOTON-Beetle-AEAD128	ref	-O3	8917	536	288973
Pyjamask-128	ref	-O3	17433	1128	49238
Romulus-M1-128	ref	-O3	41416	824	42228
Romulus-N	ref	-O3	33851	1044	20102
SKINNY-AEAD-M1-128	ref	-O3	7880	848	36341
SUNDAE-GIFT-96-128	ref	-O3	5774	2000	117316
Saturnin-CTR-Cascade-256	ref	-O3	10015	792	12861
Schwaemm256-128	ref	-O3	9677	456	6022
Schwaemm256-128	opt	-O3	3304	336	4343
Subterranean-SAE-128	ref	-O3	14059	1031	32584
TinyJAMBU-128	ref	-O3	2234	248	4366
TinyJAMBU-128	opt	-O3	2776	248	3503
Xoodyak-128	ref	-O3	7241	568	9549

4.2 Overall description of an experiment

We give here a general overview of generating an experiment. More details are available in Appendix 1.

4.2.1 Configuration

The parameters required by the script in order to launch experiments are given in Table 5.

4.2.2 How it works

Launching and handling an experiment is done in four main steps:

1. Defining the resources related to the given configuration file (or parameters)
2. Launching the experiment using the **iotlabcli** library
3. Launching the traffic capture when the experiment's status is "Running"
4. Retrieving the data related to the experiment when the status of the latter is "Terminated".

The resources related to an experiment are its nodes, the firmwares and profiles associated to these nodes.

The client nodes and the server node are defined from the architecture and the topology provided in the configuration file. We manually identified the nodes corresponding to a certain configuration (architecture, topology) by observing which nodes are linked together when creating a

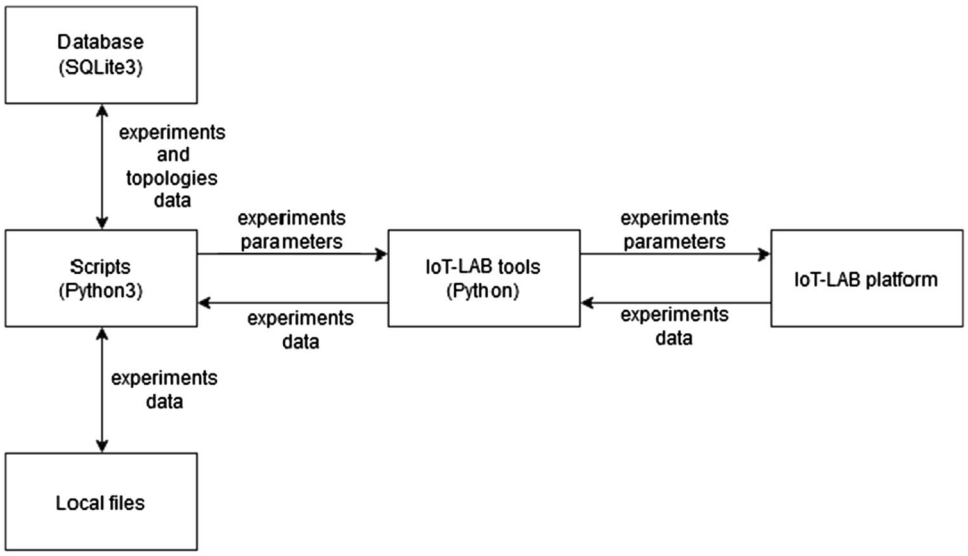

Fig. 2 Overview of the global architecture of our benchmarking framework

Table 5 Parameters of the configuration file of a benchmarking experiment

Parameter	Description
Username	Username to connect on IoT-LAB
Password	Password to connect on IoT-LAB
Ssh-key-file	SSH key file to perform a SCP on the IoT-LAB user space
Duration	Duration of the experiment in minutes
Architecture	Architecture of the nodes (for now, "m3" is implemented)
Topology	Topology of the nodes
OS	Operating system used for the firmware of the nodes (for now, "contiki-ng" is implemented)
Profile	Profile of the nodes
Operation	Type of operation the user wants to do (in this case, it is "launch-only") because we are focusing on the launching of the experiments)

RPL network[10]. Thus, the following nodes are associated to the architecture "m3" and the "line" topology:

- `m3-358.grenoble.iot-lab.info` (as server),
- `m3-318.grenoble.iot-lab.info` (as client),
- `m3-326.grenoble.iot-lab.info` (as client),
- `m3-334.grenoble.iot-lab.info` (as client),
- `m3-342.grenoble.iot-lab.info` (as client),
- `m3-350.grenoble.iot-lab.info` (as client).

For each experiment, two firmwares are used: one for the server node, and the other for the client nodes. For now, the firmwares to use are only defined by the requested Operating System because, for each chosen operating system, all the available algorithms are launched. The profile "consumption_and_sniffer" is used when traffic capture is required.

[10] As a routing algorithm is required, we decide to use the standard RPL. See [43] for more details.

Once the resources have been defined, the experiment is launched using the library *iotlabcli.experiment*. The state of the experiment is then retrieved each 30 seconds. As soon as this state is *Running*, the traffic capture is launched.

The traffic capture is launched through the following SSH command:

`sniffer_aggregator -l <site>,<architecture>, <server_id> -o <experiment_id> -server.pcap`

Thus, a PCAP file is created on the user space `<user>@<site>.iot-lab.info`. It is retrieved at the end of the experiment, along with the different measures defined by the profile.

4.2.3 Retrieving the data

Retrieving the data is the last step of the experiment deployment. It is done by remotely copying the folder generated by IoT-LAB regarding the selected profiles of

the nodes. Moreover, the PCAP file created containing the generated packets by the nodes is also retrieved. These data are stored in the local folder *data/<experiment_id>* and used by the Experiments Analysis Tool. For each node, the voltage and the power of the nodes, as well as the captured traffic are stored.

The Experiments Analysis Tool takes as inputs the files generated and retrieved by the Experiments Deployment Tool. The parameters required by the script in order to analyze the experiments data are given in Table 6.

4.3 Performance metrics

First of all, for an accurate measure of the performance metrics, we identified a transition time which corresponds to 5% of the total time of the experiment. The packets received before this transition time after the beginning of the experiment are not considered, and also the packets received after the end of the experiment, minus this time.

We identified the following metrics which are appropriate for evaluating the network performance of the algorithms.

4.3.1 Bytes per second

The number of bytes per second for an experiment is computed on the baseline experiment, where the client does not wait between sending two packets. The sum of the data packets' lengths is then computed, and divided by the time interval of the experiment which is its duration minus twice the transition time.

4.3.2 Latency

When a client sends a packet, it writes it in the log file. Likewise, when the server receives a packet, it writes it also. In these records, each message is identified by an ID. Thus, we retrieve the latency, by subtracting the reception time and the sending time of a packet having the same ID.

4.3.3 Limit of packets per second

The limit of packets per second is obtained by dividing the number of data packets received by the server, by the duration of the packets capture. In this case, the client does not wait between sending two packets. This number represents the amount of network charge a configuration can handle.

4.3.4 Packet delivery ratio (PDR)

This ratio is obtained by dividing the number received packets by the total number of packets sent. This value is then multiplied by 100 in order to get a percentage. In this case, the client does not wait between sending two packets.

4.3.5 Packets per second

The number of received packets is then computed, and divided by the duration of the experiment. The duration of the experiment is running time of the experiment minus twice the transition time. In this case, the client does not wait between sending two packets.

4.3.6 Mean of the power consumption

The experiment duration time is split in several time subintervals. For each subinterval, the mean of the power measures (in Watts) associated is computed by doing the sum of the power measures saved for all the nodes of the experiments, divided by the number of measures. These mean values are then represented in a boxplot (one boxplot for each algorithm). We also represent these mean values by a graph, where the value of the X-axis for each measure is the middle time of the associated interval. The graph can present several curves: one per algorithm.

Watts per packet. The number of watts consumed per packet is computed over the experiment duration as the total consumption divided by the number of received packets.

5 Obtained results

In this section, we present the obtained results regarding the seven chosen metrics for the 12 candidate algorithms by using two series of experiments.

Each experiment involves an IoT application deployed on 2 nodes, a client and a server[11] and lasts 20 minutes. Each node has an ARM-Cortex M3 hardware and using the OS Contiki-NG with RPL protocol as the underlying routing protocol. We run two sets of experiments: one for packets of length 128 bytes and one for packets of length 512 bytes. We have to note that the latency results of 512 bytes is missing because, we lost the corresponding data.

We provide in Figs. 3, 4, 5, 6, 7, 8, 9 and 10 the obtained results for our 7 metrics, the `no_enc` scenario is the baseline evaluation for all the experiments.

The most interesting result is the Fig. 3 where the overall watts consumption is given. First, we note that consumption values are really low (between 0.134 and 0.142 W) meaning that the consumption of all the algorithms is really low compared to the `no_enc` scenario. We

[11] We limit our study with 2 nodes because when trying more nodes, whatever the topology, the metrics become more difficult to measure.

Table 6 Parameters of the configuration file for retrieving experiments data

Parameter	Description
Architecture	Architecture of the nodes (for now, "m3" is implemented)
Topology	Topology of the nodes
Experiments	Array of the IDs identifying the experiments to compare
Operation	Type of operation the user wants to do (in this case, it is "analyse-only" because we are focusing on the experiments analysis)

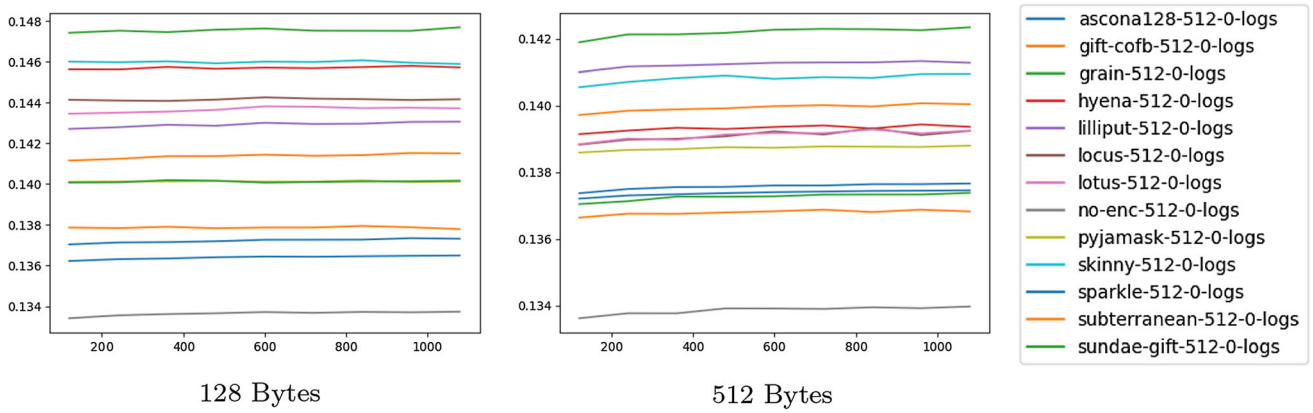

Fig. 3 Mean power consumption (watt) over time

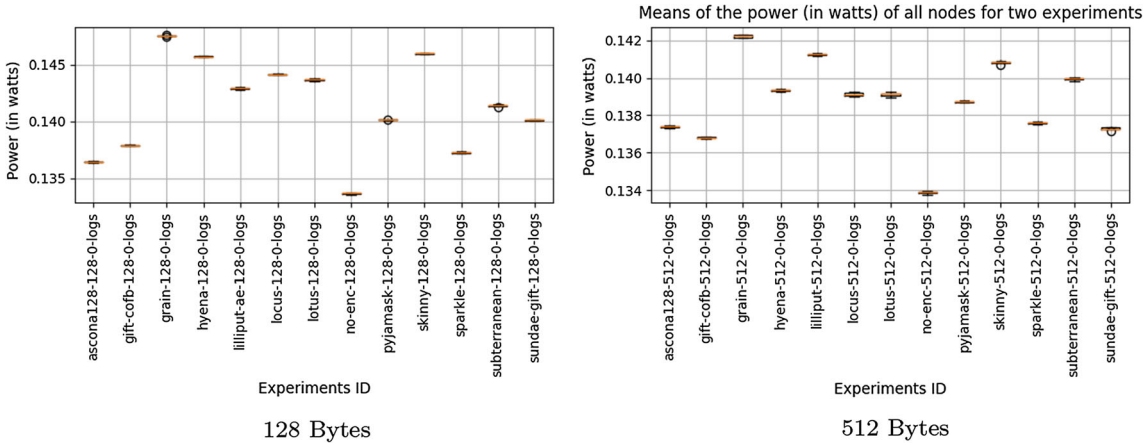

Fig. 4 Power consumption (watt) statistics

find that the cryptographic operations are not the most consuming part in IoT devices. The most efficient algorithms in terms of energy consumption are Ascon-128a, SPARKLE and GIFT-COFB for packets of size 128 bytes and SUNDAE-GIFT when considering packets of size 512 bytes. GRAIN remains the most energy consuming for both packet sizes. This first finding is confirmed by the graph presented in Fig. 4.

The interpretation of the other metrics is more difficult. However, a global view of the results show that the 512 bytes packets case seems to just flatten all these metrics except for the `no_enc` case.

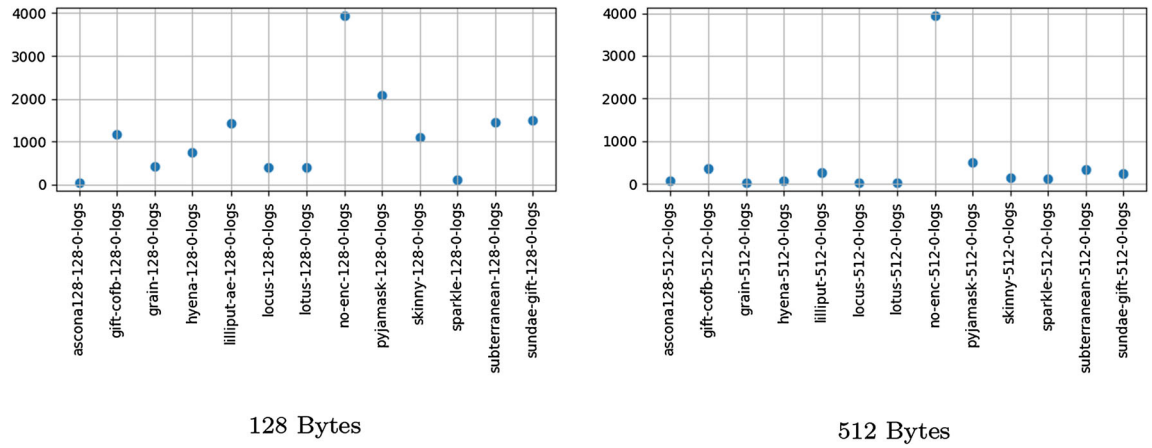

Fig. 5 The measured number of bytes per second generated by the IoT application and for each used algorithm

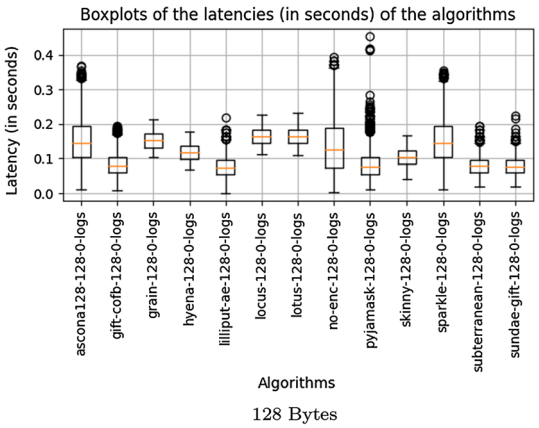

Fig. 6 Measured latency (seconds) of the generated packets by the IoT application and for each used algorithm

In terms of bytes per second as shown in Fig. 5), the most efficient algorithms in the 128 bytes packets case seems to be PYJAMASK, LILLIPUT-AE, Subterrean 2.0 and SUNDAE-GIFT. The worst algorithms in this case are ASCON-128a and Sparkle. We think that ASCON-128a is penalized because it is a stream cipher and it has thus a big warm-up step. The GRAIN algorithm seems to perform a little bit better.

In terms of latency as shown in Fig. 6, the same algorithms seem to be performing well. However, it is difficult to generalize the obtained result since the `no_enc` scenario is not representative while it has not the lowest average latency.

Regarding the limit of the number of packets per second as shown in Fig. 7, the most efficient algorithms are ASCON-128a, GIFT-COFB, LILLIPUT-AE, PYJAMASK, Sparkle, Subterrean 2.0 and SUNDAE-GIFT.

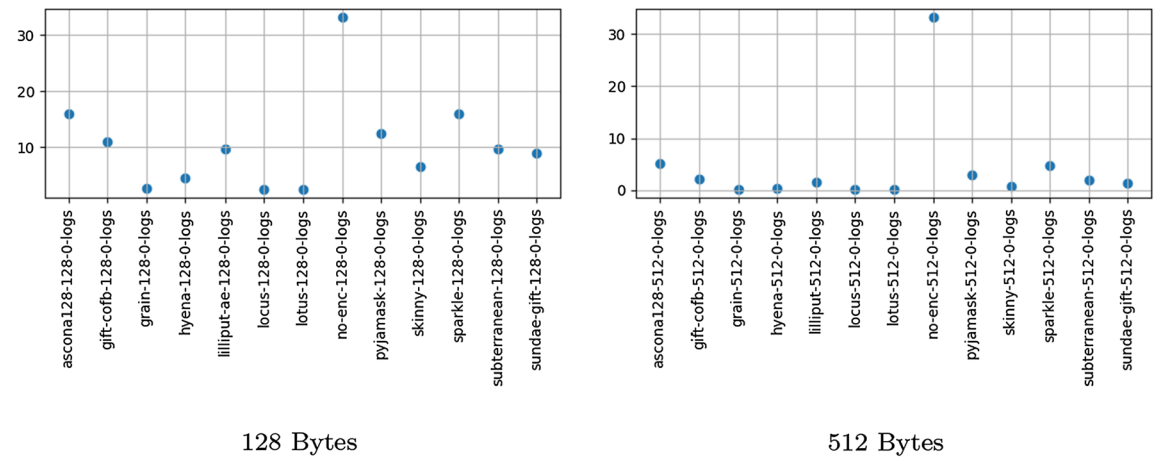

Fig. 7 The limit number of packets per second generated by the IoT application and for each algorithm

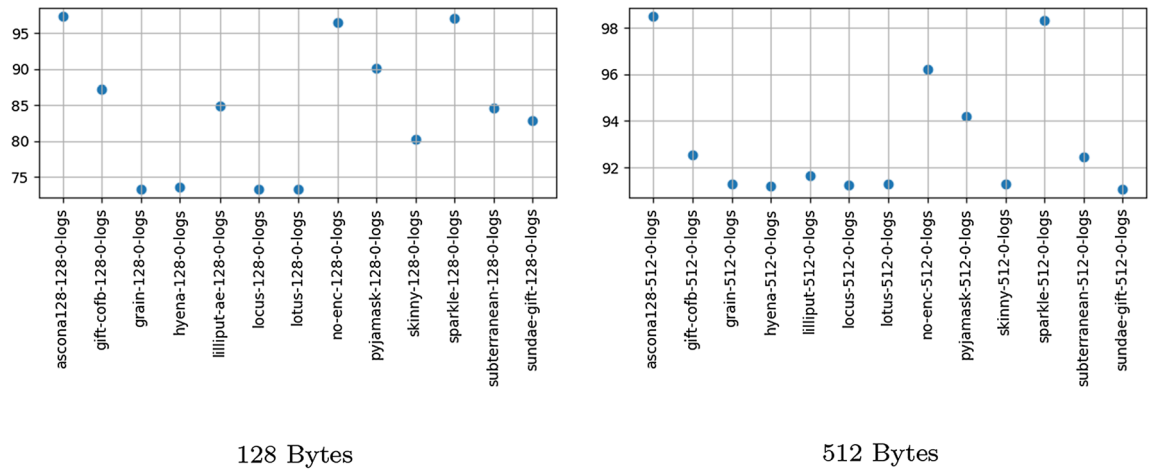

Fig. 8 The packet delivery ratio (percentage) of the IoT application and for each algorithm

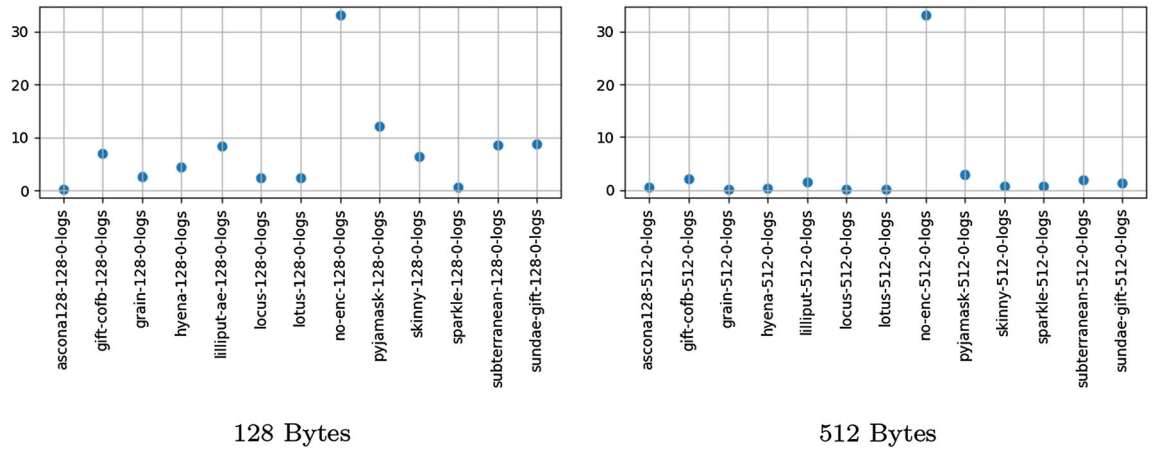

Fig. 9 Number of generated packets per second by the IoT application and for each algorithm.

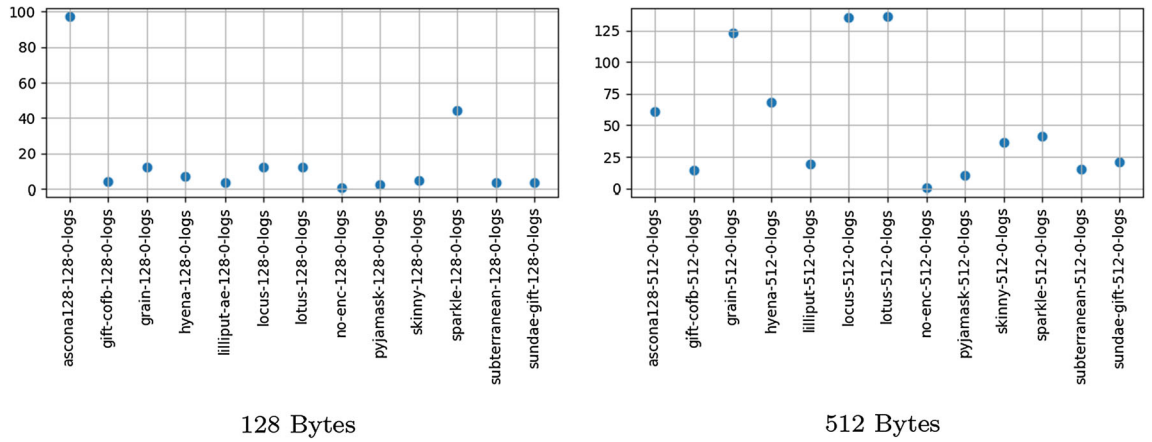

Fig. 10 The consumed watts per packet by the IoT application and for each algorithm

Concerning the Packet Delivery Ratio (PDR) as depicted in Fig. 8, for both cases (128 bytes packets or 512 bytes packets), the two algorithms Ascon-128a and SPARKLE perform similar to baseline scenario `no_enc` with a percentage close to 98 %.

Regarding the number of packets per second generated by the IoT application as shown in Fig. 9, five algorithms perform correctly: GIFT-COFB, Lilliput-AE, PYJAMASK, Subterrean 2.0 and SUNDAE-GIFT. This metric is clearly correlated to the metric of the number of bytes per second and the metric watts per packet. Thus, the best algorithms stay the best for those 3 metrics.

Finally, the metric of the consumed watts per packet is given in Fig. 10. We clearly observe thatASCON-128a and SPARKLE are disadvantaged in the 128 bytes packets case but behave better when 512 bytes packets are considered. Thus, in both cases, the algorithms that behave well are: GIFT-COFB, Lilliput-AE, PYJAMASK, Subterrean 2.0 and SUNDAE-GIFT. Note also that the bad performances of SKINNY-AEAD are mainly linked with the fact that we use SKINNY-TK3 in our implementations leading to a clear degradation of its performance due to the fact that 56 rounds of the ciphering function are required.

In fact, considering this metric is more realistic than the average consumed power. Indeed, if we study the case of Ascon-128a, that is the best algorithm in terms of average consumed power, this algorithm becomes the worst when considering watts per packet. This consumption of watts packets is correlated to its bad results in terms of bytes per second and packets per second where it they are low compared to the other algorithms. The Ascon-128a algorithm spends less energy because it ciphers fewer packets. Thus, regarding the totality of metrics, the average consumed power of the IoT application must be studied in relation with the other metrics. The algorithms that behave well for bytes per second, packets per second and watts per packet are the ones that are the most efficient regarding energy consumption. This is the case for GIFT-COFB, Lilliput-AE, PYJAMASK, Subterrean 2.0 and SUNDAE-GIFT.

6 Conclusion and future works

In this paper, we presented a new dedicated benchmarking framework based on the IoT-LAB platform to evaluate several lightweight cryptograhpic algorithms which are candidate in the round 2 of NIST competition. We ported these algortihms to the Contiki-NG OS and evaluate them by using an IoT application deployed on the physical nodes of the platform IoT-LAB while varying the packets size. The code of our benchmarking tool is publicly available. Indeed, we expect that developers and cryptographers rely on it to add more algorithms and metrics.

Our work could be extended by adding algorithms, metrics, and versions of the available algorithms, etc. However, our main perspective is that this work establishes the first step in the direction of real-world evaluation of lightweight cryptograpihic primitives to enhance IoT security.

Appendix A: performance results for the NIST competition finalists on an ARM Cortex-M3

Clearly, regarding the results of Table 7, Schwaemm256-128 is the most efficient algorithm in terms of execution time but has a dedicated implementation as the following ones: Romulus-N and TinyJAMBU-128. The importance of a dedicated implementation for the ARM Cortex-M3 is thus of primary importance because all the algorithms tested with the reference code have bad performances.

This is exactly the same behaviour concerning RAM consumption where TinyJAMBU-128 and Schwaemm256-128 have the lowest consumption, except that Romulus-N becomes worst. The same remark holds for code size.

Table 7 Performance results with `-O3` on ARM Cortex-M3

	Version	Code size (B)	RAM (B)	Execution time(cycles)
Schwaemm256-128	`armv7m_fast`	6296	248	5355
Schwaemm256-128	`armv7m_small`	2340	280	6792
Romulus-N	`arm_inline_asm`	20976	1572	10294
TinyJAMBU-128	`opt`	988	148	10342
Ascon-128	`opt32`	19744	280	13378
TinyJAMBU-128	`ref`	1212	176	13938
Schwaemm256-128	`opt`	1860	248	16035
Xoodyak-128	`ref`	5590	424	18546
Schwaemm256-128	`ref`	3070	356	20568
Ascon-128	`ref`	3256	584	35346
Isap-A-128a	`opt_32_armv67m`	13048	464	37294
GIFT-COFB	`ref`	10032	372	64143
Lilliput-II-128	`felicsref`	3744	388	90604
Romulus-N	`ref`	19756	708	100276
Isap-A-128a	`ref`	4222	656	511376
PHOTON-Beetle-AEAD128	`ref`	9758	632	1270434
PHOTON-Beetle-AEAD128	`ref`	9758	632	1270434
Elephant-160	`ref`	4606	1452	1972673
Grain-128AEAD	`ref`	3512	828	2659496

Appendix B: detailed view of the python scripts

Libraries

The Python scripts are written in Python3. They use the following libraries:

- **matplolib** and **pylab** to draw figures
- **datetime** to handle time measures and compute time intervals
- **time** to wait and get the current timestamp
- **fpdf** to generate a PDF report from a set of experiments
- **sqlite3** to interact with the database
- **path**, from **os**, to handle files
- **pyshark** to parse PCAP files
- **sys** to handle error messages
- **configargparse** to parse the configuration file
- **queue** to put the different threads in a queue
- **Thread**, from **threading**, to implement a thread
- **subprocess** to launch the threads and retrieve their output
- **SSHClient**, from **paramiko**, to handle SSH communication with the IoT-LAB user space
- **SCPClient**, from **scp**, to handle SCP operations with the IoT-LAB user space
- **json** to handle JSON data
- **iotlabcli** to interact with the IoT-LAB API
- **shutil** to copy firmwares files
- **enum** to force the Algorithm, Architecture, OperatingSystem, Operation, ProfileType, Topology and DataProcessing values to be part of a predefined set
- **Path**, from **pathlib**, to create directories.

Classes

The Python scripts relies on six main classes, with their relations are depicted in Fig. 11. The Thread class is imported from the Python library **threading**.

Enum classes Six classes do not appear in the classes diagram: they are called Algorithm, Architecture, OperatingSystem, Operation, ProfileType and Topology. They are written in the file *utils.py*. They all extend the Enum class, from the **enum** library. These classes are used while checking the content of the configuration file, in order to force the algorithms, architecture, os, operation, profile, and topology values to be part of a predefined set.

Singleton class While the data (such as the Enum classes, mentioned in the previous subsection) directly provided by the functions of the file utils.py are not retrieved from a specific instance, the data provided by the database is retrieved by calling the instance of a class. This is done in order to guarantee that only one instance accesses the database at a time, and to be able to close this instance at

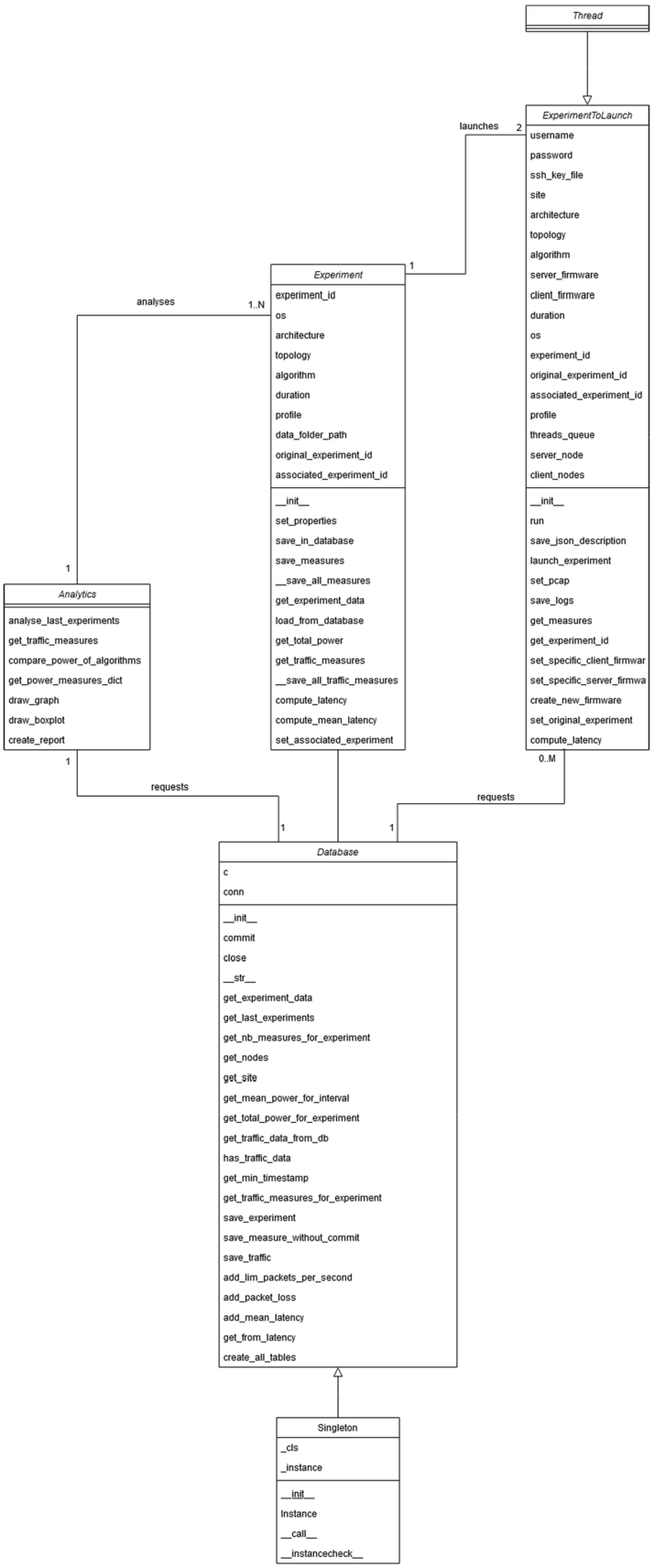

Fig. 11 Class diagram of our benchmarking tool

the end of the scripts. Thus, the Database class extends a Singleton class.

Modules

Overview of the modules

The scripts present six main modules: global, launch_experiment, analytics, experiment, database and utils. The first three modules are process-focused: they contain functions to respectively orchestrate the modules regarding to the configuration file, launch one or more experiments on the IoT-LAB platform and retrieve the generated files, and analyse these latter in order to extract measures. The last three modules handle data: the experiment module gathers data related to the experiment in order to manipulate this entity, the database module control the interaction with the database (including its creation), and the utils module contains data related to our architecture (such as a common format for the files' names and paths).

Data handling modules

Experiment The experiment module represents an experiment. It simplifies the handling of the information or the operation related to this experiment. An experiment can be instantiated using parameters defined by the user, or using the database which will provide the parameters associated to a given experiment ID.

Database The Database module provides tools to retrieve or store data from the database. It contains the Database class, which extends a Singleton in order to ensure that only one instance access to the database at a time. The Database contains six tables, depicted in the Fig. 12. Three of them are statically pre-filled by the scripts (Topology, Node and NodeTopology), while the other three (Experiment, Measure, Traffic) store data retrieved while the experiments are launched or analysed.

The Node table stores the nodes used in one or more topologies. They present an ID, specific to our database, a node ID, which is related to the ID on the IoT-LAB platform, a site which indicated where the node is located (as the IoT-LAB platform is divided into multiple sites), and the architecture of this node.

The Topology table lists the topologies: they present an ID, specific to our database and a name, for the user use.

The TopologyNode table links each node to its associated topology and defines the role it takes ('server' or 'client'). Thus, it contains an ID, specific to our database, the role the node takes in the topology, the ID of the associated topology and the ID of the associated node.

The Experiment table stores the parameters of each experiment which has been launched on IoT-LAB through the Benchmarking tool. It is linked to a topology, identified by its ID. Two tables are associated to the Experiment table: Measure and Traffic, detailed below.

The Measure table stores the power measures associated to an experiment. The experiment is identified by the ID specific to the database. Each measure presents an associated node, also identified by its ID, where the power has been measured, a timestamp indicating when the power has been measured, and the measured value in watts.

The Traffic table stores the data computed from the PCAP capture on the server side during an experimentation, identified by its ID specific to the database. It itself presents an ID specific to the database, the number of considered packets, the time interval during which the

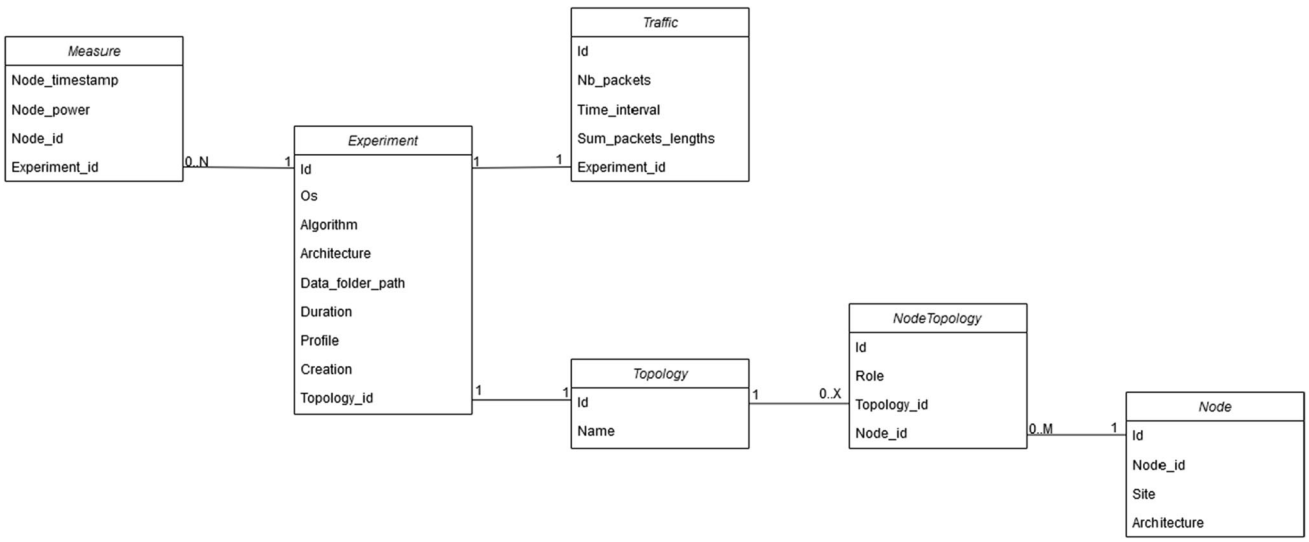

Fig. 12 Database overview

packets have been captured, and the sum of the lengths of the considered packets.

Utils The Utils module is a helper: it centralizes the static data related to the Benchmarking tool architecture of local files. Moreover, it gathers some functions used by both the Global, Launch_Experiment and Analytics modules.

Process handling modules

Global process When using the Benchmarking tool, the user can configure four distinct operations, depicted in the Fig. 13: 'launch-experiments', 'analyse-experiments', 'analyse-last' or 'launch-and-analyse-last'. We can identify two main phases of the process: 'Launching experiments' and 'Analysing experiments'. The first one when the following options are selected: 'launch-experiments' and 'launch-and-analyse-last'; the second one is used when these options are selected: 'analyse-experiments', 'analyse-last' and 'launch-and-analyse-last'. These two phases are detailed in the subsections below.

Launching experiments **Input/Output** The 'Launching' phase takes as input the experiments' parameters. To launch several experiments, a list of algorithms can be submitted in the configuration file. The launched experiments will then present the same parameters, except for this algorithm. Only experiments with the same parameters (duration, os, architecture and topology) can then be compared with each other in the 'Analysing' phase. The 'Launching' phase outputs a list of experiments IDs. It contains the IDs of the launched experiments.

Orchestrate the launching of the experiments The Fig. 14 presents the global process of the 'Launching' phase. As mentioned before, multiple experiments can be launched at once. Thus, each experiment is launched into a specific Thread. The clients of these experiments do not wait between two sent packets, so that the max number of received packets can be computed. This value is used to compile a new client, waiting between packets in order to compute the latency. A second experiment, using this client, is then launched. The second experiment is referred as the "associated experiment" of the first one, while the first experiment is referred as the "original experiment" of the second one. The process launching one experiment is described in the following paragraph.

Launch an experiment An experiment is submitted using the IoT-LAB API. This is done by providing the parameters given by the user. The nodes to be used are chosen regarding the topology data contained in the database. On IoT-LAB, an experiment can present multiple states: 'Waiting', 'Launching', 'Running', 'Finishing', 'Terminated' and 'Error'. For each experiment, its state is regularly checked in order to determine the following action of the Benchmarking tool. The first time the experiment is detected as 'Running', a traffic capture is launched on the server side. When the experiment is detected as 'Terminated', the generated files are copied locally through SCP. The experiment parameters (and ID) are then stored in the database, in the Experiment table (Fig. 15).

Analysing the experiments

Input/Output The 'Analysing' phase can take as input the experiments' parameters or their IDs. If parameters are provided in the configuration file, the analysis will be done on the last experiment launched presenting these parameters. As for the 'Launching' phase, multiple algorithms can be requested. They will be analysed separately, and all presented in the output report. The maximal number of experiments to consider for each algorithm is indicated in the configuration file. The experiments to analyse can also be identified directly by their IDs. The 'Analysing' phase

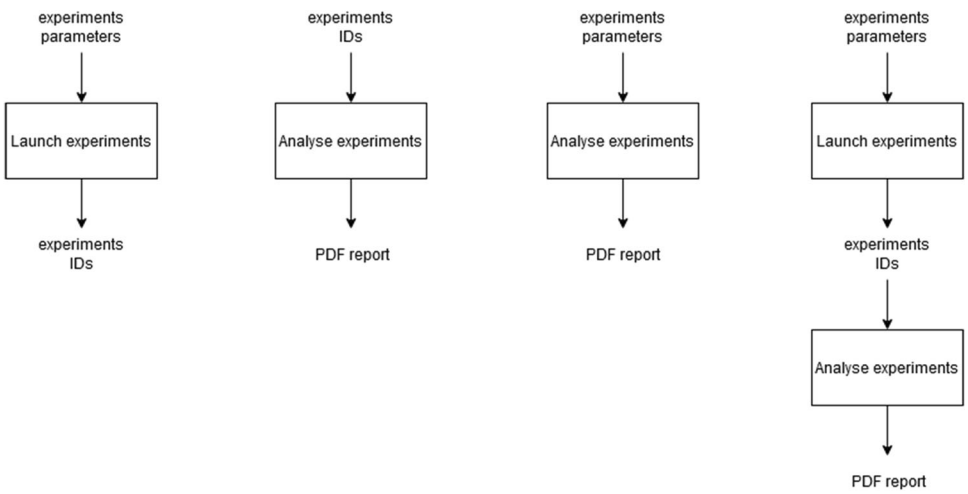

Fig. 13 Process phases overview

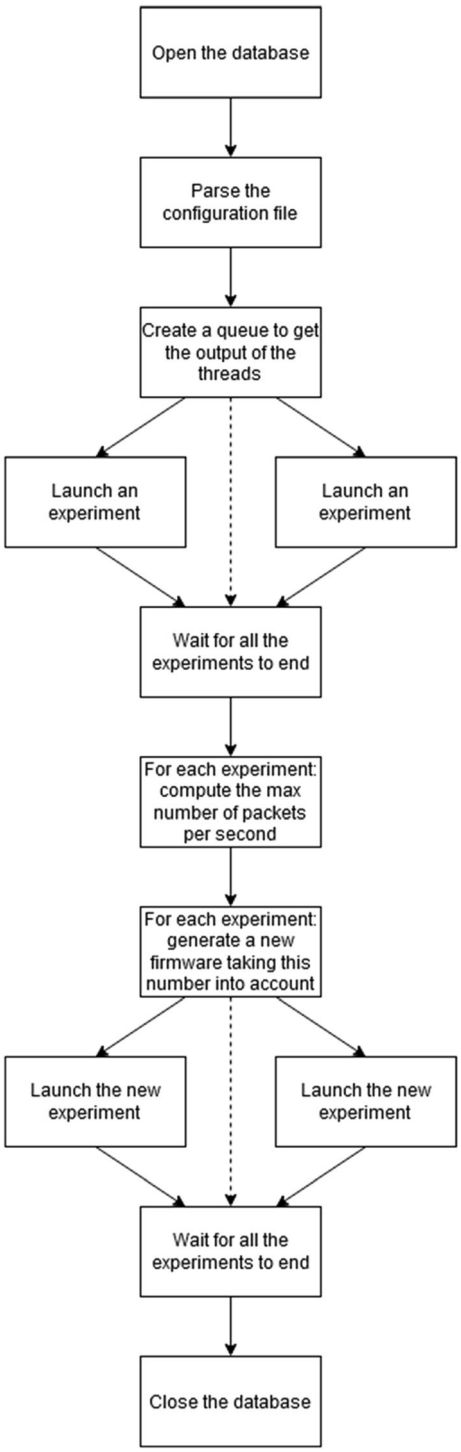

Fig. 14 Steps for launching a process

outputs a PDF report and PNG graphs (included in the PDF).

Global process The Analysing phase process is depicted in the Fig. 16. For each algorithm, the experiments are retrieved (regarding their parameters or their IDs), and, for each experiment, its power measures and its traffic data are computed.

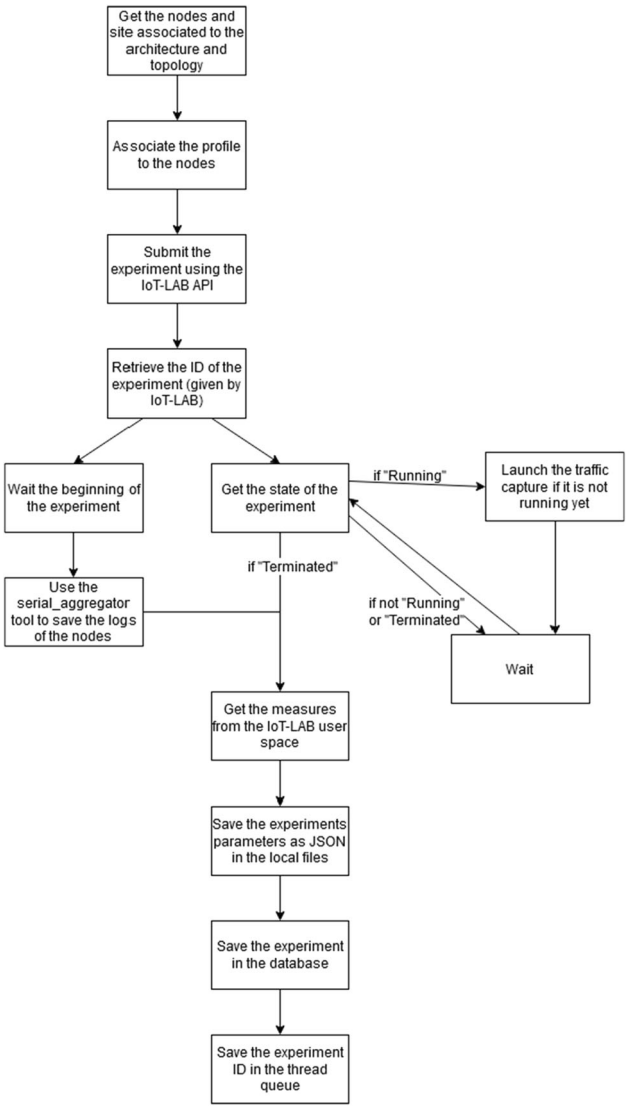

Fig. 15 Steps for launching an experiment

Power measures The power measures are retrieved from OML file generated by IoT-LAB while the experiment is running, as shown in Fig. 17. These files (one per node) are then copied to local files through SCP. For each experiment, several entries are stored in the table Measure of the database: one per power measures, namely associated to a timestamp. The generation of the power graph from these measures is detailed in the Fig. 18. When the experiments are analysed, the data of these latter is gathered when the experiments present the same encryption algorithm. As the power is measured at a specific time, the power measures are discrete values for all the experiments and nodes. Thus, in order to gather data for several experiments, their time duration is divided into several time windows. The mean power of each time window is

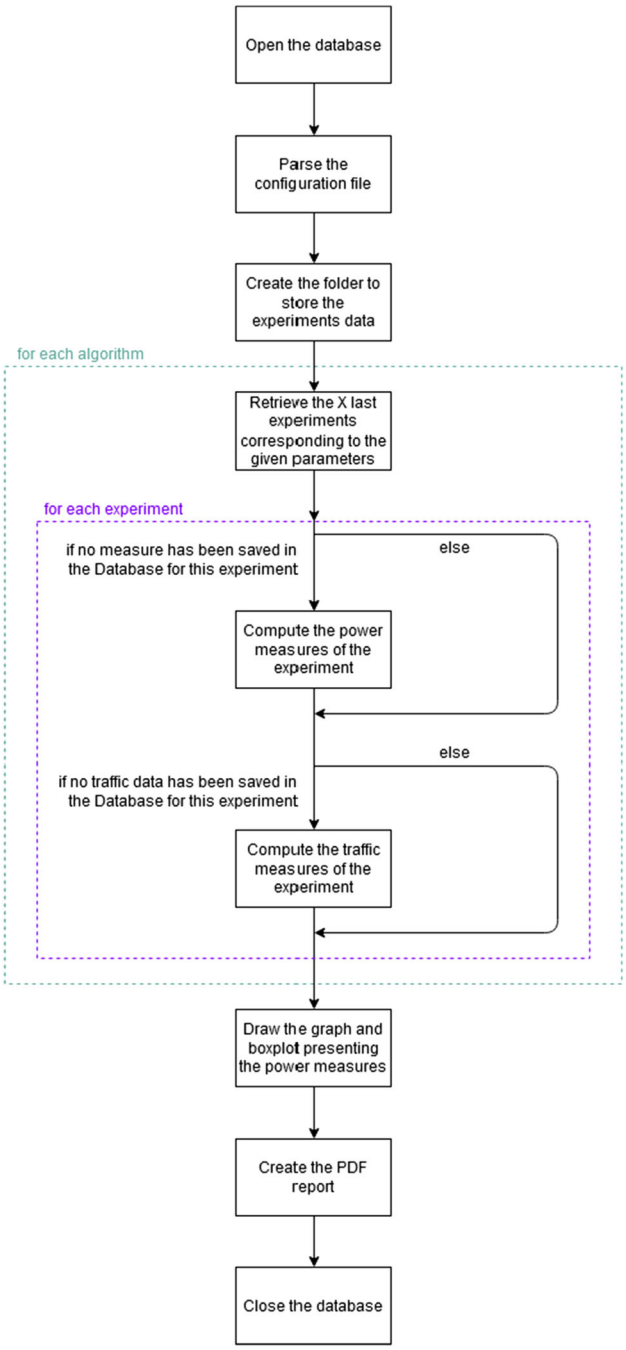

Fig. 16 Global analysis process

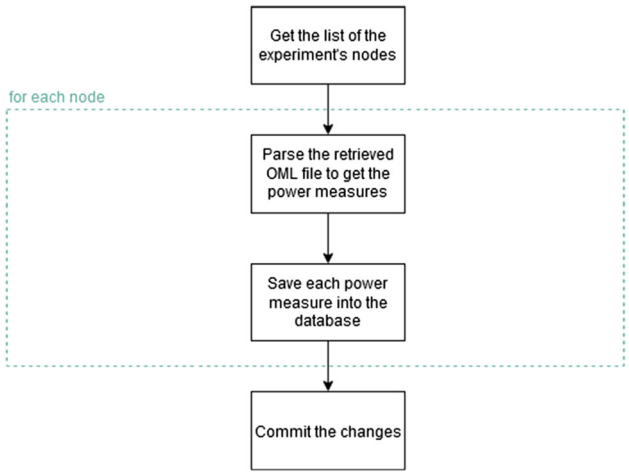

Fig. 17 Power analysis process

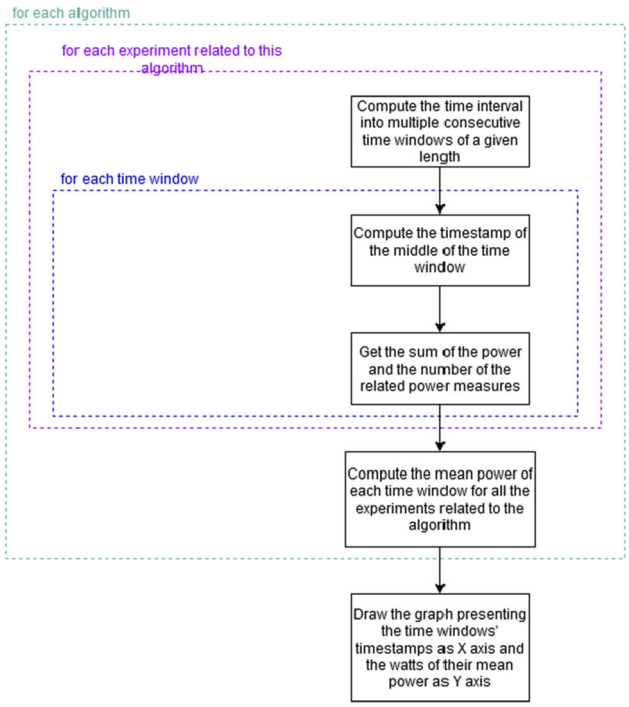

Fig. 18 Power graph generation

then computed. Thereby, a new power measure is simulated: it presents the timestamp of the middle of the time window as timestamp, and the mean of the power of the experiments on this time window as value. The power graph presents these power means. As the boxplot presents the minimum, the maximum, the median and the first and third quartile, the power measures are not to be preformatted, contrary to the graph data. Thus, the power measures of all experiments are simply retrieved from the database for each algorithm in order to build the associated boxplot.

Traffic measures The traffic data is retrieved from the PCAP file generated on the server side by IoT-LAB while the experiment is running, and from the logs of all the nodes, clients and server, as shown in Fig. 19. These files are then copied to local files through SCP. The Benchmarking tool then parses the PCAP file by using the pyshark library. The only considered packets are the packets sent from a client to the server containing an

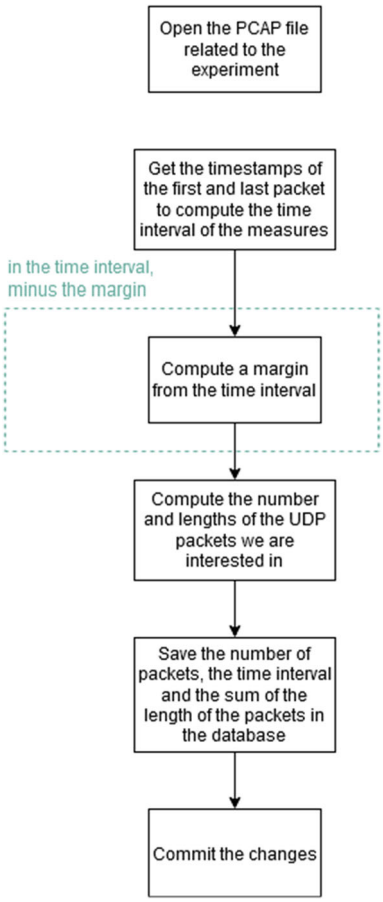

Fig. 19 Traffic analysis process

encrypted payload. A margin of 5% of the time interval of the experiment is also ignored at the beginning and the end of the capture. The traffic measures are computed from the number, timestamp and lengths of the packets, which are stored in the database as one entry of the Traffic table. The computed traffic measures are the following: packets per second, packets loss and latency.

Acknowledgements This work was partially supported by the project IMPACT DigiTrust of "Lorraine Université d'Excellence".

References

1. Adomnicai, A., Berger, T. P., Clavier, C., Francq, J., Huynh, P., Lallemand, V., Le Gouguec, K. , Minier, M., Reynaud, L., & Thomas, G. (2019). Lilliput-ae: a new lightweight tweakable block cipher for authenticated encryption with associated data. *Submitted to NIST Lightweight Project*.
2. Andreeva, E., Lallemand, V., Purnal, A., Reyhanitabar, R., Roy, A., & Vizár, D. (2019). Forkae v. *Submission to NIST Lightweight Cryptography Project*.
3. Banik, S., Bogdanov, A., Peyrin, T., Sasaki, Y., Sim, S. M., Tischhauser, E., & Todo, Y. (2019). Sundae-gift. *Submission to Round*, 1.
4. Banik, S., Chakraborti, A., Iwata, T., Minematsu, K., Nandi, M., Peyrin, T., Sasaki, Y., Sim, S. M., & Todo, Y. (2019). Gift-cofb. *Submission to Round*, 1.
5. Bao, Z., Chakraborti, A., Datta, N., Guo, J., Nandi, M., Peyrin, T., & Yasuda, K. (2019). Photon-beetle. *Submission to the NIST Lightweight Cryptography Standardization Effort*.
6. Beierle, C., Biryukov, A., dos Santos, L. C., Großschädl, J., Perrin, L., Udovenko, A., Velichkov, V., Wang, Q., & Biryukov, A. (2019). Schwaemm and esch: lightweight authenticated encryption and hashing using the sparkle permutation family. *NIST round*, 2.
7. Beierle, C., Jean, J., Kölbl, S., Leander, G., Moradi, A., Peyrin, T., Sasaki, Y., Sasdrich, P., & Sim, S. M. (2020). Skinny-aead and skinny-hash. *IACR Transactions on Symmetric Cryptology*, pages 88–131.
8. Bellare, M., Rogaway, P. (2000). Encode-then-encipher encryption: How to exploit nonces or redundancy in plaintexts for efficient cryptography. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 317–330. Springer.
9. Bernstein, D. J., & Lange, T. (2019). ebacs: Ecrypt benchmarking of cryptographic systems. http://bench.cr.yp.to, Access on line in December 2019.
10. Beyne, T., Chen, Y. L., Dobraunig, C., & Mennink, B. (2020). Status update on elephant. *NIST lightweight competition (2020)*.
11. Canteaut, A., Duval, S., Leurent, G., Naya-Plasencia, M., Perrin, L., Pornin, T., & Schrottenloher, A. (2020). Saturnin: a suite of lightweight symmetric algorithms for post-quantum security. *Transactions on Symmetric Cryptology, 2020*(S1), 160–207.
12. Cazorla, M., Gourgeon, S., Marquet, K., & Minier, M. (2015). Implementations of lightweight block ciphers on a wsn430 sensor.
13. Cazorla, M., Marquet, K., & Minier, M. (2013). Survey and benchmark of lightweight block ciphers for wireless sensor networks. In *2013 international conference on security and cryptography (SECRYPT)*, pages 1–6. IEEE.
14. Chakraborti, A., Datta, N., Jha, A., & Nandi, M. (2019). Hyena. *Submission to Round*, 1.
15. Chakraborti, A., Datta, N., Jha, A. L., Cuauhtemoc, M., Nandi, M., & Sasaki, Y. (2019). Lotus-aead and locus-aead. *Submission to Round*, 1.
16. Daemen, J., Massolino, P. M. C., Mehrdad, A., & Rotella, Y. (2020). The subterranean 2.0 cipher suite. *IACR Transactions on Symmetric Cryptology*, pages 262–294.
17. Daemen, J., Hoffert, S., Peeters, M., Van Assche, G., & Van Keer, R. (2020). Xoodyak, a lightweight cryptographic scheme. *Transactions on Symmetric Cryptology, 2020*(S1), 60–87.
18. Dinu, D., Le Corre, Y., Khovratovich, D., Perrin, L., Großschädl, J., & Biryukov, A. (2019). Triathlon of lightweight block ciphers for the internet of things. *Journal of Cryptographic Engineering, 9*(3), 283–302.
19. Dobraunig, C.E., Eichlseder, M., Mangard, S., Mendel, F., Mennink, B., Primas, R., & Unterluggauer, T. (2020). Isap v2. 0.
20. Dobraunig, C., Eichlseder, M., Mendel, F., & Schläffer, M. (2014). Ascon. *Submission to the NIST LWC competition: http://ascon. iaik. tugraz. at*.
21. Dos Santos, L. C., Großschädl, J., & Biryukov, A. (2019). Felics-aead: benchmarking of lightweight authenticated encryption

algorithms. In *International Conference on Smart Card Research and Advanced Applications*, pages 216–233. Springer.
22. Eisenbarth, T., Kumar, S., Paar, C., Poschmann, A., & Uhsadel, L. (2007). A survey of lightweight-cryptography implementations. *IEEE Design and Test of Computers, 24*(6), 522–533.
23. Goudarzi, D., Jean, J., Kölbl, S., Peyrin, T., Rivain, M., Sasaki, Y., & Sim, S. M. (2020). Pyjamask: Block cipher and authenticated encryption with highly efficient masked implementation. *IACR Transactions on Symmetric Cryptology*, pages 31–59.
24. Hell, M., Johansson, T., Meier, W., Sönnerup, J., Yoshida, H. (2019). Grain-128aead-a lightweight aead stream cipher. *NIST Lightweight Cryptography, Round*, 1.
25. Iot-lab api documentation. https://github.com/iot-lab/iot-lab-client/blob/master/iotlabclient/client_README.md.
26. Iot-lab cli tools documentation. https://github.com/iot-lab/iot-lab/wiki/CLI-Tools.
27. Iot-lab cli tools. https://pypi.org/project/iotlabcli/.
28. Iot-lab client github. https://github.com/iot-lab/iot-lab-client.
29. Iot-lab github. https://github.com/iot-lab.
30. Iot-lab website. https://www.iot-lab.info.
31. Kerckhof, S., Durvaux, F., Hocquet, C., Bol, D., & Standaert, F.-X. (2012). Towards green cryptography: a comparison of lightweight ciphers from the energy viewpoint. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 390–407. Springer.
32. Knežević, M., Nikov, V., & Rombouts, P. (2012). Low-latency encryption–is "lightweight= light+ wait"? In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 426–446. Springer.
33. Law, Y. W., Doumen, J., & Hartel, P. (2006). Survey and benchmark of block ciphers for wireless sensor networks. *ACM Transactions on Sensor Networks (TOSN), 2*(1), 65–93.
34. Le Gouguec, K., & Huynh, P. (2019). Felics-ae: a framework to benchmark lightweight autheticated block ciphers. In *Proceedings of the 2019 NIST Lightweight Cryptography Workshop*.
35. Matsui, M., & Murakami, Y. (2013). Minimalism of software implementation. In *International Workshop on Fast Software Encryption*, pages 393–409. Springer,.
36. National Institute of Standards and Technology (NIST). sha-3-project. [Online; 2007].
37. NIST Fips Pub. (2001). 197: Advanced encryption standard (aes). *Federal information processing standards publication,197*(441), 0311.
38. Onelab website. https://onelab.eu.
39. RIOT. The friendly operating system for the internet of things.
40. Texas Instruments. Msp430f1611 datasheet. https://www.ti.com/lit/ds/symlink/msp430f1611.pdf.
41. Wenzel-Benner, C., & Gräf, J. (2010). Xbx: external benchmarking extension for the supercop crypto benchmarking framework. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 294–305. Springer.
42. Wenzel-Benner, C., Gräf, J., Pham, J., & Kaps, J.-P. (2012). Xbx benchmarking results january 2012. In *Third SHA-3 Candidate Conference (2012)*, http://xbx.das-labor.org/trac/wiki.
43. Winter, T., Thubert, P., Brandt, A., Hui, J. W., Kelsey, R., Levis, P., et al. (2012). Rpl: Ipv6 routing protocol for low-power and lossy networks. *RFC, 6550,* 1–157.
44. Wu, H., & Huang, T. (2019). Tinyjambu: A family of lightweight authenticated encryption algorithms. *Submission to the NIST Lightweight Cryptography Competition*, https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/round-1/spec-doc/TinyJAMBU-spec. pdf.

Soline Blanc received the master's degree in computer science from TELECOM Nancy Engineering School, France, in 2015. She was a research engineer in the RESIST team of Inria Nancy and the CARBONE team of Loria, France; and is currently part of the Innovation, Development and Technologies team of Mila, Canada, where she contributes to the development of tools for machine learning researchers.

Abdelkader Lahmadi is Associate Professor of Computer Science at University of Lorraine, France. He is a permanent member of the RESIST research team at LORIA - INRIA Nancy Grand Est, working on security monitoring and management. He received a PhD degree in Computer Science (2007). He published in all major conferences in Network and Service Management. His research interests are mainly focusing on the securitin monitoring of networked systems including Software Defined Networks, IoT and SCADA systems by applying machine learning techniques.

Kévin Le Gouguec is software engineer previously employed by Airbus CyberSecurity, formerly specialized in network security and cryptography implementation. Now employed by AdaCore and focused on cross-environment toolchains & debugging environments.

Marine Minier received the Ph.D. degree in 2002, from the Université de Limoges and the French Habilitation from the Université de Lyon in 2012. In 2005, she joined the INSA de Lyon and the CITI Laboratory, as an Assistant Professor. Since 2016, she is professor at Université de Lorraine and at the LORIA Lab. Her research interests include Symmetric Key Cryptography and Security in WSNs.

Lama Sleem is a Doctor in cyber security. She works as a post-doctoral researcher at the University of Lorraine with the RESIST team. She graduated from University of Franche-Comté in 2020. Her research is mainly about lightweight cryptography and developing cyber content for security training.