FELICS-AE: a framework to benchmark lightweight authenticated block ciphers

Kévin Le Gouguec*

* Airbus CyberSecurity - ZA Clef Saint-Pierre, 1 Bd Jean Moulin, CS 40001, MetaPole, 78996 ÉLANCOURT Cedex - France - kevin.legouguec@airbus.com

October 18, 2019

1 Introduction

The CAESAR competition [4] and the NIST Lightweight Cryptography Standardization Process [12] have brought to light several new Authenticated Encryption with Associated Data (AEAD) algorithms dedicated to "lightweight" use-cases. In these use-cases, target devices are strongly constrained in terms of computing resources: they have limited volatile memory (RAM) and non-volatile memory (ROM), their processors operate at low frequencies and feature few registers, they may only be able to draw power from a battery that can neither be recharged nor replaced, etc.

These devices thus have very few resources to spare on security. This implies that the aforementioned algorithms must be selected not only for their robustness, but also according to their efficiency. Given two encryption schemes with equivalent security, the scheme which leaves the target device more resources to perform its designed function will be preferred.

Thus measuring the performance of these algorithms is an integral part of the selection process carried out in [12]. In this paper, we present FELICS-AE, an adaptation of the FELICS framework [5] dedicated to AEAD schemes, which we use to assess the performance of our candidate Lilliput-AE [1]. We have released this framework on a public Git repository[6].

First, in section 2, we will present the original FELICS framework. We will then present FELICS-AE in section 3, going over our work to adapt the framework and explaining how to use it. We will present the results we obtained in section 4. To conclude, we will mention possible improvements for FELICS-AE in section 5.

2 Background: the FELICS framework

The FELICS framework [5] includes a collection of implementations of encryption algorithms in C and assembly, as well as a set of shell scripts which measure the performance of these algorithms on various microcontrollers representative of "Internet of Things" (IoT) devices.

2.1 Supported devices

FELICS supports the following microcontrollers (**bold** words denote the codenames used within the framework):

- 8-bit **AVR** ATmega128,
- 16-bit **MSP**430F1611,
- 32-bit **ARM** Cortex-M3.

The AVR and MSP platforms are entirely simulated, which allows one to measure algorithm performance on these microcontrollers without physically owning them. To measure performance on ARM, however, FELICS requires an Arduino Due board, as well as a J-Link probe. Algorithms can also be benchmarked on the implementer's x86 platform, codenamed **PC**.

2.2 Metrics

Code size: FELICS adds up the `text` and `data` sections of an implementation's compiled object code, as reported by the GNU `size` program, to measure the algorithm's footprint on non-volatile memory.

RAM: to measure the working memory needed by an algorithm, the framework runs the implementation through a debugger, spraying a known pattern on the stack before execution and counting how many bytes were modified after execution. This figure is added to the object code's `data` section.

Execution time: for simulated devices, FELICS relies on the simulator to keep track of the number of clock cycles spent on encryption. For other devices, FELICS uses specialized assembly instructions to get this information.

2.3 Distribution

The CryptoLUX wiki [8] hosts an archive containing FELICS's source code (algorithm implementations and benchmarking scripts). The wiki also provides detailed instructions to install the dependencies FELICS needs to compile implementations and measure their performance on every platform.

The wiki also hosts a virtual machine (32-bit Ubuntu 14.04) where all dependencies are pre-installed. This makes it easier to use the framework since one then does not need to track down all of its dependencies.

2.4 Algorithm instrumentation

Algorithm implementations must comply with a number of requirements in order to work with FELICS. This section presents some of these constraints.

Algorithm entry points must conform to a specific API. FELICS features multiple *scenarios*, implemented as C files which define the `main` function that will call the cipher implementation. Each scenario calls the cipher with different parameter sizes, which allows observing the evolution of the algorithm's performance as its input grows.

Each `main` function is generic with respect to the algorithm under test: it is expected that each implementation defines high-level encryption and decryption functions with specific signatures, so that scenarios can be applied to all algorithms included in FELICS.

Encryption and decryption code must be split across distinct files. When measuring an algorithm's code size, FELICS outputs three distinct tallies: encryption code size, decryption code size, and total code size. To achieve this, FELICS requires integrators to fill in metadata files, spelling out which object files are used for encryption, and which are used for decryption.

This means that if an implementation originally had one file featuring both encryption and decryption functions, an integrator must split it into two files and tell FELICS which file serves which purpose. If the original file contained code used by both encryption and decryption functions, the integrator must further create a third file to move the common code to.

Array declarations must be annotated. FELICS defines a set of macros that annotate integer types for two purposes:

- They specify optimal memory alignment for integer arrays: this ensures that implementations aliasing byte arrays as e.g. 32-bit integer arrays do not accidentally access an array member at an address which is not aligned for a 32-bit variable, which can degrade performance or cause undefined behaviour.
- They add the platform-dependent keywords necessary to tell the compiler whether arrays should be stored in ROM or RAM: e.g. for AVR `ROM_DATA_BYTE` expands to `const uint8_t PROGMEM aligned`, where `PROGMEM` instructs `gcc` to move the variable to Program Memory; `READ_ROM_DATA_BYTE` expands to `pgm_read_byte`, which performs the operations needed to read from this specific memory region.

An integrator must therefore go over each array declaration in an implementation and change its type using the correct macro.

3 FELICS for Authenticated Encryption

FELICS was initially developed to measure the performance of block ciphers and stream ciphers. In this section, we describe the changes we made to adapt the framework to AEAD algorithms; we then describe how we use it.

3.1 Changes from FELICS

We began our work with release 1.1.0 of the FELICS framework. Our goals were to support AEAD algorithms, simplify the algorithm integration process, and improve the feedback given to implementers while they optimize their code.

3.1.1 AEAD support

Authenticated encryption primitives have specific signatures: their inputs include the associated data to authenticate, a nonce to achieve semantic security [16]; encryption produces an authentication tag which is consumed by decryption.

The common C API we chose for cipher implementations is inspired by the *crypto_aead* API described in the call for submissions of the CAESAR competition [3], which has been re-used in the NIST standardization process [13]. Our API differs in minor ways:

- It does not include the `nsec` parameter, which is unused in the context of the NIST standardization process. Removing this unused variable reduces the number of compiler warnings, which helps implementers spot actual errors in their code.
- Arrays are passed as pointers to `uint8_t` rather `unsigned char`. Array sizes are passed as `size_t` rather than `unsigned long long`. The latter is unnecessarily large on some platforms: e.g. on AVR, an `unsigned long long` variable takes 8 bytes, while `size_t` is only 2 bytes.

3.1.2 Tools for implementation optimization

The original FELICS framework offers multiple entry points:

- one makefile for each implementation, which compiles a scenario for any platform, or runs a scenario on the development PC,
- one script per metric, which runs the relevant tools (e.g. simulators, debuggers) for a given device and algorithm, and either displays the results in a human-readable table or serializes them in an unspecified format,
- `collect_ciphers_metrics.sh`, a more complex script which iterates over algorithms, architectures, platforms, scenarios, and compiler options, and calls on the aforementioned scripts and makefiles to run a comprehensive measurement campaign.

While developing optimized implementations of Lilliput-AE, we found that our main tasks were:

- running a full measurement campaign for a set of algorithms,
- comparing a set of results against a previous set,
- comparing two implementations of the same algorithm (e.g. reference vs. threshold, reference vs. optimized for a specific architecture),
- exporting a set or a subset of results into various formats.

We developed a new set of scripts to perform these tasks, which we present in section 3.2. We chose to implement these tools in Python, which we found more convenient than Bash for multiple reasons: e.g. a rich library ecosystem, reduced boilerplate (the `argparse` library, for example, produces detailed usage messages automatically, whereas the usage messages for the original shell scripts must be maintained manually).

While adapting FELICS to AEAD algorithms and integrating Lilliput-AE, Ascon and ACORN, we also stumbled on several issues related to the poor error-reporting capabilities of shell scripts:

- None of the framework's scripts uses the `errexit` shell option: command errors are only detected when they are followed by ad hoc checks.
- These checks do not necessarily stop the script, which means that some failures cannot be detected unless the user either carefully watches the framework's output, or surveys the results closely enough to notice suspicious patterns (e.g. metrics set to zero).
- Simply setting the `errexit` option only marginally improves the situation, since it does not produce a backtrace when an error happens: tracking down a failed command which produces no output involves a certain amount of manual work. The situation is aggravated by the framework's error-checking convention, where commands writing to the standard error stream are assumed to have failed: naively setting the `xtrace` shell option in sub-scripts then causes spurious failures in the parent scripts.

Python exceptions interrupt the program flow immediately, include precise backtraces, and may be enriched with arbitrary information by the developer. We found these properties to make error-handling more ergonomic.

3.1.3 Distribution

In order to make it easier to setup FELICS-AE, we wrote scripts to fetch and install the framework's dependencies. Building on those scripts, we produced a ready-to-use Docker image bundling FELICS-AE with its dependencies; the generation of this image is itself fully scripted.

3.1.4 Miscellaneous

We made several changes that aimed at simplifying the framework, removing degrees of freedom which we did not need. For example, we removed the "scenario" parameter; measurement campaigns now always begin by checking the implementation against a test vector, then measure the performance of encrypting 16 bytes of plaintext with 16 bytes of associated data.

We also removed `collect_ciphers_metrics.sh`'s support for multiple output formats: we settled on JSON as a unique format for the results of a measurement campaign. All tools presented in section 3.2 use this format as their input, and a specific script is used to convert them to other formats.

Beyond encryption and decryption, FELICS further distinguished metrics for key schedule (for block ciphers) and setup phase (for stream ciphers). Some AEAD algorithms call the underlying block cipher and its key schedule repeatedly for every block of input; there was no obvious way to adapt the framework to preserve this distinction, so FELICS-AE only provides metrics for the whole encryption process.

While adapting the cycle-counting assembly code for x86_64 architectures, we observed considerable variance in our figures for execution time. A lot of CPU features of modern workstations contribute to this variance: to mitigate these factors, we implemented several countermeasures, following Intel's guidelines for benchmarking on IA-64 architectures [14]:

- Use the `cpuid` instruction to ensure that all instructions are serialized correctly, otherwise out-of-order execution may move part of the code we want to benchmark out of the scope of the timestamp-measuring instructions, or even move unrelated code inside this scope.
- Use the `taskset` command to pin the benchmark program to a single core, otherwise the code may be moved to other cores while it runs, and the timestamp counters of these cores may not be synchronized.
- If the user is privileged enough, set the CPU frequency scaling governor for this core to "performance" to ensure a fixed frequency.
- Run the scenario multiple times and take the median cycle count, to account for the remaining variance.

3.2 Usage

In this section, we will present some of the tools we developed while adding support for AEAD algorithms to FELICS.

3.2.1 `felics-run`

This script expects a list of algorithms (each element of the list can include wildcards to designate multiple algorithms), an optional set of architectures (all

supported platforms are selected by default), and an optional list of compiler flags (only `-O3` is used by default).

For every combination of the requested parameters, this script checks the implementation's test vector, then measures the implementation's code size, RAM usage and cycle count when encrypting 16 bytes of plaintext with 16 bytes of associated data.

The metrics are serialized in a JSON file, along with some metadata to identify the framework revision.

3.2.2 `felics-publish`

This script either prints every setup (i.e. a given set of algorithm, architecture, and compiler options) from a results file to the user's console, or exports these setups into a new output format. The supported formats are HTML table, LaTeX table, OOXML spreadsheet or ODF spreadsheet.

Several options allow the user to control the output:

`--sort-by:` how setups are ordered,

`--filter:` which setups are included,

`--info:` which metadata and metrics are displayed,

`--table-label:` anchor for documents supporting cross-references,

`--table-caption:` additional text to describe the data set.

Sample console output:

```
On AVR
Lilliput-I-128 (felicsref, -O3):  6100 266 129093
Lilliput-II-128 (felicsref, -O3): 6062 243 132650

On MSP
Lilliput-I-128 (felicsref, -O3):  5760 300 121646
Lilliput-II-128 (felicsref, -O3): 4932 272 144399

On ARM
Lilliput-I-128 (felicsref, -O3):  4656 444 86293
Lilliput-II-128 (felicsref, -O3): 4684 420 89390

On PC
Lilliput-I-128 (felicsref, -O3):  6880 528 10030
Lilliput-II-128 (felicsref, -O3): 6783 528 11816
```

3.2.3 `felics-compare`

This script iterates over every setup found in one results file, computes the performance ratio with respect to the same setups in a second results file, and highlights these ratios so that the implementer can judge at a glance whether a code change had a positive or negative impact on performance.

Sample output:

```
Comparing
        foo.json
        (master) 1234567 Commit summary foo
against
        bar.json
        (master) 89abcde Commit summary bar

Lilliput-I-128 on AVR (vfelicsref with -Os)
        code_size: -12.19% (3166 ↘ 2780)
        code_ram: -49.22% (514 ↘ 261)
        code_time: +32.05% (189818 ↗ 250657)

Lilliput-I-192 on AVR (vfelicsref with -Os)
        code_size: -10.47% (3268 ↘ 2926)
        code_ram: -50.71% (562 ↘ 277)
        code_time: +36.73% (230309 ↗ 314893)

Lilliput-I-256 on AVR (vfelicsref with -Os)
        code_size: -8.25% (3392 ↘ 3112)
        code_ram: -53.19% (626 ↘ 293)
        code_time: +40.83% (290363 ↗ 408907)
```

The user can choose to ignore minor differences by providing a `--threshold` argument; ratios lower than this threshold will be hidden.

Other scripts perform similar comparisons: if the FELICS-AE directory is version-controlled, **`felics-compare-revisions`** automatically checks out two revisions, runs the requested benchmarks, and compares their results files. **`felics-compare-implementations`** pairs comparable setups (architecture, compiler options) for two implementations in the same results file, and compares their metrics.

4 Results

This section comments on our measurements of the performance of Lilliput-AE and the members of the final portfolio of the CAESAR competition for the "lightweight" use-case [4], Ascon and ACORN.

4.1 Setup

The measurements were performed on an Ubuntu 16.04 64-bit desktop with 4 3.5 GHz CPUs and 8 GB RAM. The software versions for platform-specific compilers, debuggers and other such utilities correspond to those distributed by Ubuntu, with the exception of software listed in table 1.

Platform	**Software**	**Version**	**Origin**
AVR	simavr	`v1.6`	Developer release [15]
	Avrora	`1.7.117-patched`	Cf. FELICS documentation [7]
MSP	MSP430-GCC	`7.3.2.154`	Texas Instruments [11]
	MSPDebug	`v0.25`	Developer release [2]
ARM	J-Link Software	`V6.42f`	SEGGER [9]

Table 1: Software versions for the FELICS-AE framework.

We considered two compilation options:

- `-O3`, to minimize computation time and decrease power consumption,
- `-Os`, to reduce code size and thus optimize for low memory footprint.

4.2 Implementations

The source code for the CAESAR algorithms was adapted from the SUPERCOP [17] framework. In order to provide a fair assessment of each algorithm's performance, we looked for implementations that performed well (i.e. better than the reference version) for each FELICS platform. Table 2 sums up which implementations were considered for each platform.

Algorithm	**Platform**	**Implementations**
Ascon	AVR	`ref`
	MSP	`ref`
	ARM	`ref`, `opt32`
	PC	`ref`, `opt64`
ACORN	AVR	`8bitfast`
	MSP	`8bitfast`
	ARM	`opt1`
	PC	`opt1`

Table 2: Algorithm implementations for each platform.

For Lilliput-AE, we used the same implementation on all platforms. This implementation, called `felicsref`, closely resembles the reference implementation, except for a few tweaks documented in each source file's header comments.

4.3 Discussion

We published our measurements in our submission to the first round of the LWC standardization process [1]. We also published these figures on the Lilliput-AE website [10], where we intend to add results for optimized versions of Lilliput-AE, as well as for other candidates to the standardization process.

First, we compare the performance of Ascon and ACORN to the 128-bit key variants of Lilliput-AE:

On 8-bit AVR, when compiled for speed, both variants of Lilliput-AE compare favorably to Ascon and ACORN, even in terms of code size and RAM usage: the only unfavorable comparison is ACORN's lower code size. When compiled with `-Os`, Lilliput-AE variants achieve the lowest code size, though Ascon and ACORN become faster.

On 16-bit MSP, when compiled for speed, Lilliput-AE variants come well ahead in terms of cycle count, though again ACORN has a lower code size. When compiled with `-Os`, Lilliput-AE variants are the lightest and the fastest, though ACORN has a slightly smaller RAM footprint.

On 32-bit ARM and 64-bit PC, whichever compilation option we consider, Lilliput-AE variants are the slowest algorithms; in terms of code size, they compare unfavorably to the reference version of Ascon.

Next, we study the impact of the choice of key length (128, 192 or 256) and AE mode (ΘCB3 for Lilliput-I, SCT-2 for Lilliput-II) on the performance of Lilliput-AE:

- Longer keys imply more tweakey *lanes* (64-bit words). The key schedule updates each lane with a different matrix multiplication, therefore longer keys lead to a higher memory footprint, as well as more cycles. This may explain why SCT-2 variants have the smallest memory footprint: they use 128-bit tweaks, whereas ΘCB3 variants use 192-bit tweaks.
- ΘCB3 variants, on the other hand, are consistently faster than SCT-2 variants. This may be due to ΘCB3 making fewer calls to the underlying block cipher E_K: for an l-block plaintext, ΘCB3 calls E_K l times, while SCT-2 calls E_K $2l$ times.

5 Future work

In this section, we identify ways in which FELICS-AE could be improved: new benchmarking features, simplifications to make extensions easier, structural cleanups, etc.

Simplify the addition of new devices. The plumbing required to support a microcontroller architecture is scattered across several files: e.g. shell and debugger scripts to measure metrics, makefiles to handle compilation, C code to provide device-specific functions. Adding a new platform is very much a trial-and-error process; re-organizing this infrastructure code would go a long way toward smoothing the steps out.

Simplify the addition of new algorithms. For example, no source file can contain both encryption-specific and decryption-specific code: since FELICS-AE only measures the size of whole object files, the size of the decryption-specific code will be added to the tally for encryption, and vice versa.

To remove this restriction, user-supplied metadata could be enriched to specify symbols (functions or data) that are specific to one operation; the size of these symbols would be subtracted from the irrelevant tallies. Alternatively, such symbols could be annotated with the compiler attribute `section`: code-size measurement scripts could then assume that symbols in a specific ELF section are only used for one operation.

Allow multiple scenarios. The original FELICS framework allowed metrics to be collected for various use-cases. Adding this capability back could be as simple as adding `--plaintext-size` and `--adata-size` parameters to `felics-run`, and recording the value of these parameters in the results file.

Allow multiple test vectors. AEAD algorithms may contain branching code paths, e.g. to handle padding. A single test vector cannot cover both sides of a branch; implementers may therefore have no way to realize that their optimizations break the code for some inputs.

Add profiling The original FELICS framework allowed the user to profile an algorithm in some limited way, by hooking into specific functions such as a block cipher's key schedule. It may be possible to generalize this feature using traditional profiling tools such as `gprof`.

References

[1] Alexandre Adomnicai, Thierry P. Berger, Christophe Clavier, Julien Francq, Paul Huynh, Virginie Lallemand, Kévin Le Gouguec, Marine Minier, Léo Reynaud, and Gaël Thomas. LILLIPUT-AE: a New Lightweight Tweakable Block Cipher for Authenticated Encryption with Associated Data. *NIST LWC Standardization Process Round 1 Candidates*.

[2] Daniel Beer. mspdebug. `https://github.com/dlbeer/mspdebug`, 2019. Accessed: 2019-03-07.

[3] Crypto competitions: CAESAR call for submissions. https://competitions.cr.yp.to/caesar-call.html. Accessed: 2019-07-24.

[4] Crypto competitions: CAESAR submissions. https://competitions.cr.yp.to/caesar-submissions.html. Accessed: 2019-07-23.

[5] Dumitru-Daniel Dinu, Alex Biryukov, Johann Groszschaedl, Dmitry Khovratovich, Yann Le Corre, and Léo Paul Perrin. FELICS - Fair Evaluation of LIghtweight Cryptographic Systems, 2015.

[6] FELICS-AE · GitLab. https://gitlab.inria.fr/minier/felics-ae. Accessed: 2019-10-04.

[7] CryptoLUX > FELICS Avrora patch. https://www.cryptolux.org/index.php/FELICS_Avrora_patch, 2019. Accessed: 2019-03-07.

[8] Fair Evaluation of LIghtweight Cryptographic Systems. https://www.cryptolux.org/index.php/FELICS. Accessed: 2019-07-24.

[9] J-Link Software and Documentation Pack. https://www.segger.com/downloads/jlink/#J-LinkSoftwareAndDocumentationPack, 2019. Accessed: 2019-02-26.

[10] Lilliput-AE implementations. https://paclido.fr/lilliput-ae/implementation/. Accessed: 2019-07-30.

[11] MSP430-GCC-OPENSOURCE GCC - Open Source Compiler fro MSP Microcontrollers. http://www.ti.com/tool/msp430-gcc-opensource, 2019. Accessed: 2019-03-07.

[12] Lightweight Cryptography | CSRC. https://csrc.nist.gov/Projects/Lightweight-Cryptography. Accessed: 2019-07-23.

[13] Submission requirements and evaluation criteria for the lightweight cryptography standardization process. https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/final-lwc-submission-requirements-august2018.pdf. Accessed: 2019-07-24.

[14] Gabriele Paoloni. How to benchmark code execution times on Intel® IA-32 and IA-64 instruction set architectures. Technical report, Intel Corporation, 2010.

[15] Michel Pollet. simavr. https://github.com/buserror/simavr, 2019. Accessed: 2019-03-07.

[16] Phillip Rogaway. Authenticated-encryption with associated-data. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, CCS '02, pages 98–107, New York, NY, USA, 2002. ACM.

[17] SUPERCOP. https://bench.cr.yp.to/supercop.html. Accessed: 2019-02-21.