A Hardware in the Loop Benchmark Suite to Evaluate NIST LWC Ciphers on Microcontrollers

Sebastian Renner[1,2], Enrico Pozzobon[1,3], and Jürgen Mottok[1]

[1] OTH Regensburg, Germany
{sebastian1.renner, enrico.pozzobon, juergen.mottok}@othr.de
[2] Technical University of Munich, Germany
[3] University of West Bohemia, Czech Republic

Abstract. The National Institute of Standards and Technology (NIST) started the standardization process for lightweight cryptography algorithms in 2018. By the end of the first round, 32 submissions have been selected as 2nd round candidates. NIST allowed designers of 2nd round submissions to provide small updates on both their specifications and implementation packages. In this work, we introduce a benchmarking framework for evaluating the performance of NIST Lightweight Cryptography (LWC) candidates on embedded platforms. We show the features and application of the framework and explain its design rationale. Moreover, we provide information on how we aim to present up-to-date performance figures throughout the NIST LWC competition. In this paper, we present an excerpt of our software benchmarking results regarding speed and memory requirements of selected ciphers. All up-to-date results, including benchmarking different test cases for multiple variants of each 2nd round algorithm on five different microcontrollers, are periodically published to a public website. While initially only the reference implementations were available, the ability of automatically testing the performance of the candidate algorithms on multiple platforms becomes especially relevant as more optimized implementations are developed. Finally, we show how the framework can be extended in different directions: support for more target platforms can be easily added, different kinds of algorithms can be tested, and other test metrics can be acquired. The focus of this paper should rather lay on the framework design and testing methodology than on the current results, especially for reference code.

Keywords: Lightweight Cryptography · Benchmarking · Embedded Systems · RISC-V

1 Introduction

In the era of rising numbers of interconnected computing devices and frequent cyber attacks, an increased need for secure communication exists. Standard cryptosystems often cannot be applied in areas like sensor networks, since the devices used here typically consist of low-performance hardware components. To aid in the process of development, evaluation and standardization of suitable lightweight cryptography algorithms, the NIST has initiated the Lightweight Cryptography Project with the final goal to standardize lightweight hash functions and cryptosystems which support authenticated encryption with associated data (AEAD). NIST received 57 and accepted 56 algorithm proposals, from which 32 primitives have been announced as 2nd round candidates in August 2019.

In this paper, we introduce a Hardware in the Loop (HIL) benchmarking setup for the evaluation of software implementations of the submitted LWC ciphers' performance. We explain the architecture and design of the framework, its core hardware and software components and how they interact with each other. By dissecting the compilation, testing process and result acquisition, we want to make the framework as transparent as possible.

We started the development of the framework already shortly after the beginning of the NIST LWC competition. First proof-of-concept testing results had already been acquired during the 1st selection round. Since then, our tests have been performed periodically on all implementations available for 2nd round candidates. Of course, results for the speed, code size or RAM utilization of reference software implementations provide little value for an actual comparison since the performance of a cipher here depends highly on its implementation – which will be optimized over time and therefore its performance figures will change. That's why we established a submission system tied to our framework, which allows designers and developers to hand in their optimized implementations for testing on a variety of architectures commonly found on embedded hardware.

Contribution The main contribution of this work is the introduction and publication of a HIL performance benchmarking framework for authenticated cipher software implementations of NIST LWC candidates. Our setup integrates actual hardware test devices, which allows for real world and fair performance evaluation in a HIL setting in contrast to a simulated environment. We provide an in-depth description of the software architecture, its implementation and the communication between the different software and hardware parts. Moreover, we explain how we designed the testing process and how we perform the measurements for each test case (speed, ROM size and RAM utilization). We also show how we designed a basic implementation submission system, which allows developers to get their latest code evaluated on regular basis and how we present the up-to-date data to the public. Furthermore, we discuss the framework's capability to extend the support of embedded platforms and how it could be tweaked to allow different kinds of tests, both within the context of the NIST LWC competition and also regarding various other use cases in the domain of algorithm performance testing.

As a proof-of-concept, we also provide an excerpt of preliminary benchmarking results for 2nd round candidates. As of now, highly optimized software implementations, especially for embedded devices, are not yet available for all of the 32 remaining candidates. That is why a reliable comparison of the implementation between candidates is hard and can lead to false conclusions. However, a comparison of different implementations of the same cipher can sometimes be of value when analyzing how special tweaking of (a part) of the algorithm alters its performance. Furthermore, with the advancement of the NIST LWC competition, more optimized variants of 2nd or the upcoming 3rd round candidates are expected, so benchmarks of those more tailored implementations will likely result in a more meaningful comparison of performance figures in between the candidates.

Outline The rest of this paper is structured as follows: The next section will describe related work in the field of benchmarking cryptographic algorithms. In section 3, we present our custom HIL benchmarking framework for the NIST LWC candidates and its features. Furthermore, the test setup, test cases, database backend and the evaluated microcontroller units (MCUs) are described. Section 4 introduces some preliminary exemplary performance results, before we conclude our work in section 5. The last section discusses various possible future research paths.

2 Related Work

This work is about software performance analysis of the NIST LWC project candidates. Ankele et al. published software benchmarks of 2nd round submissions of the CAESAR AEAD competition on Intel desktop processors [1][2]. Cazorla et al. compared implementations of 17 block ciphers on a 16 bit MCU from Texas Instruments [4]. Similar research was conducted by Hyncica et al. in 2011. They evaluate 15 symmetric cryptographic primitives regarding throughput, code size and storage utilization on three different embedded platforms [8]. Tschofenig et al. analyzed the performance of cryptographic algorithms, also on MCUs. Their work focuses on asymmetric elliptic curve ciphers executed on ARM Cortex-M cores [10]. An evaluation of 19 block and stream ciphers was published by Dinu et al. in 2015. A previous paper written by the same authors, introduces a benchmark framework for cryptographic ciphers, which focuses on fair performance testing [6] [5]. The frameworks eBacs and SUPERCOP are additional examples for popular software written for evaluating implementations of cryptographic algorithms [3]. Built to extend SUPERCOP, XBX and XXBX enhance the testing framework to support the evaluation of hash functions and AEAD ciphers on embedded devices [11] [9].

The research presented in this paper focuses on the evaluation of 2nd round candidates of the NIST LWC project. The software implementations are benchmarked using a custom HIL setup featuring multiple different MCU platforms and architectures. The framework is currently capable of evaluating the performance (speed), RAM and ROM utilization of the AEAD algorithms proposed to the NIST LWC competition on five different MCUs. Due to its modular structure, adding support for more platforms or altering the processed test vectors to focus on specific use cases is trivial.

3 Methodology

The NIST stated the delivery of a software implementation to be mandatory for each submitted AEAD cipher in its call for submissions. Besides requirements concerning the cryptographic primitive itself, the set of guidelines included some formal regulations. For example, the static directory structure within submissions and the use of a predefined software Application Programming Interface (API) for cryptographic functions are mentioned. Before developing the methodology and test procedures for the software benchmarks, an analysis of these formal requirements was conducted. The goal was to extract the basic guidelines for the creation of a test setup, which is completely compliant to the defines of NIST and yet flexible in terms of expandability.

3.1 Framework

After reviewing existing performance benchmark frameworks for AEAD ciphers, a decision was made towards the development of a custom test tool. That was because our focus regarding the hardware architecture was set on various instruction sets, typically found on microcontrollers. Since an intensive study of an existing framework and probably programming a manual extension would have been necessary to execute our test cases on the selected MCUs, the decision to built test routines from scratch was considered to be more suitable in our case.

Our framework consists of a couple of C, Python and Bash scripts, which are communicating with each other in a mostly automated manner. Moreover, we use JavaScript, PHP and HTML for the presentation of the results on the web and an SQL database to store all relevant information. The `compile_all.py` script is responsible for compiling each submitted cipher implementation for each of the target platforms. Note, that our routine always tries to compile each submitted cipher (variant) as it was provided in the ZIP file; no changes are made to the received implementation. `compile_all.py` fetches the source files of the `crypto_aead` directories and adds them into the target template structure one after the other. The MCU-specific template implements a basic runtime environment and utilizes the NIST API when calling the encryption/decryption functions. Templates are written in C / C++ depending on the development kit of the target MCU, and are responsible for providing a standardized communication protocol between the MCU and the rest of the test setup. For each combination between cipher implementation and template, `compile_all.py` attempts to produce a binary firmware ready to be flashed on the target MCU.

After the compilation has terminated, the performance benchmarks can be started for each successfully compiled implementation by using the `test.py` script included in each template. Each `test.py` script implements the flashing and communication routines specific to an MCU, while all the common testing functions are inherited from the imported file `test_common.py`, thus providing a standardized public interface to the test scheduler.

The test scheduler is another Python script responsible for distributing the compiled firmware binaries across the available MCU development boards and starting the correct `test.py` script, making sure that only one test is executed on a given piece of hardware but allowing multiple tests to be executed in parallel on different boards. The test scheduler also provides a web GUI that shows the results and the error logs of the tests, and allows to repeat failed tests or to upload the result data to the results database.

Once one of the `test.py` scripts flashes the binary onto an MCU, it starts sending one test vector at a time. The tested MCU, upon receiving the test vector, will toggle the logic value of one of its General-purpose input/output (GPIO) pins before and after executing the tested cryptographic function, which allows a logic analyzer attached to the GPIO pin to measure the execution time precisely. The logic analyzer used is a Saleae Logic Pro 16, driven using the sigrok library in streaming mode and a custom C program to allow multiplexing the single logic analyzer to multiple tests that could be running in parallel. The logic analyzer "multiplexer" software communicates to the individual `test.py` instances using UNIX domain sockets (or alternatively TCP sockets).

If the tested MCU allows debugging over JTAG and a suitable JTAG interface is connected, `test.py` will also capture the contents of the entire Random Access Memory (RAM) of the MCU before and after performing a cryptographic operation. This, combined with filling the RAM with a random pattern before starting the test procedure, allows to evaluate the memory usage of each algorithm.

The architecture of the performance evaluation framework allows testing all compiled cipher variants in a completely automated manner. The integration of new target devices requires little effort and no generic test routines need to be reconfigured – only a specific `test.py` and the runtime environment for calling the encrypt/decrypt functions from the NIST API on the MCU have to be provided. The software design of the framework satisfies some common requirements regarding test automation. Test data is provided and collected through a standard interface, which communicates with exchangeable and modular scripts. Once the performance test has been started, no user intervention is necessary until all suitable cipher variants have been evaluated. Moreover, a basic logging functionality is included, and continuous checks of the transmitted data ensure the recognition and reporting of communication errors.

To conclude the introduction to the test framework, Figure 1 visualizes its communication model and its previously described parts.

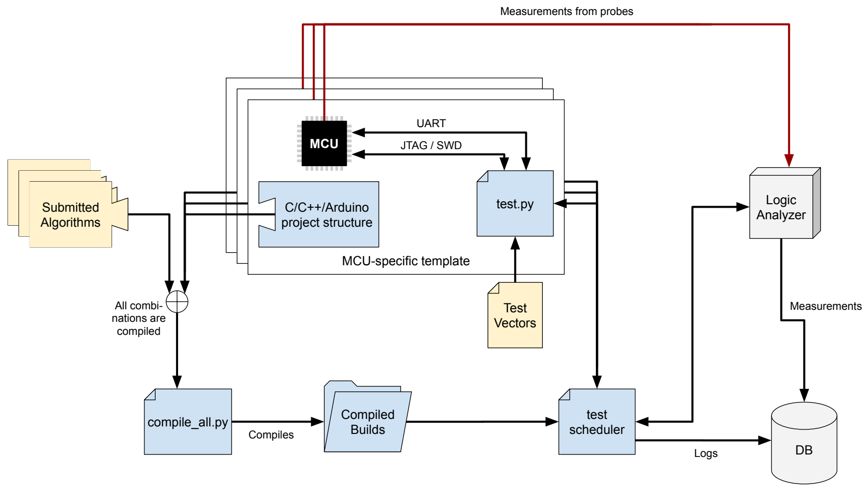

Fig. 1: Core components and data flow of the test framework

3.2 Test Setup

The physical hardware setup necessary for performing the tests consists in a single laptop computer, a Saleae Logic Pro 16 logic analyzer and one development board for every tested MCU. For boards that don't have an integrated Universal Serial Bus (USB)-to-Universal asynchronous receiver/transmitter (UART) interface or a programmer, external interfaces need to be provided as well. For this purpose, FT2232H Mini Modules were used since each of them can provide JTAG and UART connectivity over USB at the same time. The power for each tested MCU is also provided over USB from the computer.

Since our test framework makes use of the sigrok library, any supported logic analyzer could be used alternatively. We decided to use a Saleae Logic Pro 16 because it is capable of keeping a fast sampling rate of 100 MHz when using 5 channels. One GPIO pin from each tested MCU is connected to the logic analyzer to precisely measure the duration of each cryptographic operation as described previously.

The appropriate software to compile and run the performance tests, including its underlying functions, concludes the test environment. Table 1 briefly shows the tools which have been deployed. We used the platform packages provided by the most recent versions of PlatformIO and CubeMX, which include a complete toolchain for each of the tested boards. The makefiles for the building of the MCU firmwares specify the recommended compiler flags from NIST, if applicable on the MCU.

The presented testing framework is not limited to use of any of the described software or hardware elements. Support for any additional compiler could easily

Software type	Tool	Version
Compiler (Uno)	gcc	5.4.0
Compiler (F1)	gcc	9.2.1
Compiler (ESP)	gcc	5.2.0
Compiler (F7)	gcc	7.3.1
Compiler (R5)	gcc	8.2.0
Framework	PlatformIO	4.3.3
Framework	STM32CubeMX	5.4.0
Interpreter	Python	3.7.3
Logic Analyzer Library	libsigrok	0.5.1
Debugger Software	openocd	0.10.0

Table 1: Overview of used software tools

be added, the logic analyzer hardware and software can be replaced as long as the replacements allow for scripting of the logic captures, and protocols other than UART can be used to communicate with the MCU.

3.3 Results Storage

Each successful test produces the following results:

– Time duration of each cryptographic operation.
– Size of the compiled binary.
– Memory utilization, if possible on the tested MCU.

All these results are stored in a MariaDB SQL Database, together with information regarding the family, variant, implementation and revision of the tested cipher, version of the template that was used to compile the test, and timestamp of the execution of the test. This allows tracking the change in performance of each algorithm when any of the parts of the setup are changed (like compiler updates or bugfixes in the templates).

3.4 Test Cases

In this work, we introduce three different basic test cases, which are of relevance when assessing how lightweight a software implementation of a cipher is, the performance (speed), the size of the binary and the utilization of RAM. Of course, the test results of each cipher variant can be compared to its competitors within the NIST LWC project. However, we decided to include two more algorithms in the tests: a what we call *nocrypt* algorithm, which simply copies the plaintext from input to output without performing any encryption, and an implementation of one of the current state-of-the-art AEAD algorithms, AES-GCM. The results of the nocrypt benchmarks give an estimate for the overheads introduced by the framework for execution time, memory requirement and code size for all the tested platforms. AES-GCM implementations represent the state-of-the-art in the field of symmetric AEAD ciphers.

It is a well-tested and standardized cipher. Comparing optimized implementations of ciphers from the NIST LWC project to AES-GCM can later show how they perform against the actual standard in the different test cases. We modified the AES-GCM implementation found in mbed TLS to respect the NIST submission guidelines in order to make it testable with our framework. mbed TLS (formally known as PolarSSL) is part of the popular IoT operating system mbed OS and is compliant to NIST SP800-38D [7]. The flags MBEDTLS_AES_ROM_TABLES and MBEDTLS_AES_FEWER_TABLES were added to the configuration of mbed TLS since they are commonly used flags on embedded devices with a small amount of RAM and Read Only Memory (ROM). MBEDTLS_AES_ROM_TABLES places the SBOX and RCON tables and their inverses in the ROM instead of initializing them in the RAM on the first utilization of the AES algorithm. The MBEDTLS_AES_FEWER_TABLES reduces the binary size by avoiding the inclusion of some optimizations, bringing it closer to the one from other LWC entries.

We conduct benchmarks for all officially submitted software implementations of 2nd round candidates. These include reference implementations, as well as various optimizations. Results for reference implementations might often not be very representative. However, they have been included in our early proof-of-concept tests to verify the correct behavior of the benchmark framework. For a competitive comparsion of different ciphers, always the latest and best optimizations have to be taken into account. It is also important that different candidates are on a similar level of optimization to get meaningful results out of a performance comparison. For example, it is fair to compare two cipher designs implemented fully in ARM assembly. Besides all official 2nd round implementations available from the NIST web page, we are continuously testing new and optimized implementations received through our online submission form or mail. We do not change any of the implementations, in order to support a neutral evaluation. The tests include processing the test vectors available in the submitted ZIP archive. The vectors for AES-GCM have been created using the `genkat_aead.c` file to ensure a fair evaluation. However, in terms of the benchmark framework, different or more test vectors can be included in the test by simply providing them in the same format that `genkat_aead.c` produces. For the speed test case, each cipher runs an encryption and decryption of 1089 NIST test vectors stored in a text file provided in the submission package. After selecting and publishing the 2nd round candidates, NIST allowed reasonable updates on implementations to fix possible bugs. Since the deadline for these modifications was set to the 27th of September 2019, our retesting of the 2nd round candidates is based on the most recent version of the official LWC code repository.

The speed benchmark measures the time for the encryption and decryption of the message per test vector. If the vector contains associated data, its signing and verification is also taken into consideration. The time measurement is taken directly at the target and does not include the transmission time, e.g. on the serial line. The logic analyzer gathers each encryption/decryption cycle from the GPIO pin toggle and saves the captured data to a text file upon the processing of the last test vector. The correct behavior of the cipher is checked by comparing the calculated plain- and ciphertext to the values in the test vector file. All measurement results are later processed and stored into a SQL database. The test data is then exposed to the public through a website. In that way everyone can inspect it and also see which test has been conducted at which time. Furthermore, we provide additional plots to visualize e.g. the encryption/decryption time for each vector of each speed evaluation.

To compare the code size of the cipher variants, the AES-GCM implementation and the nocrypt routine are also included in the ROM usage test case. We integrate each implementation into the template sources and compile a flashable binary for each cipher and test platform. The size of the nocrypt image can be seen as the minimal code size, when the template projects are applied. The compilation of each algorithm includes the use of NIST's provided flags. After the `compile_all.py` script finishes, the code size of the binaries is determined with a small bash script utilizing the `du` system command on Linux. The binary size can then be compared to the size of the binary produced using no encryption to remove the overhead of the test framework.

To measure the RAM usage, the memory of the chip is filled with a known pseudo-random pattern, the test vectors are run, and the memory is dumped afterwards. By checking the differences between the memory dumps before and after the algorithm has been executed, it is possible to determine how many memory locations have been written during the execution of the encryption and decryption algorithms. The largest number of consecutive untouched memory locations between the end of the BSS segment and the beginning of the stack is considered the "unused memory". The number of additional bytes used by each algorithm when compared to the nocrypt implementation is seen as the memory utilization of the examined algorithm.

3.5 Tested Platforms

The benchmarking framework currently supports five different platforms, featuring one 8 bit-, three 32 bit- and one 64 bit MCU and four different architectures. By choosing this set of supported boards, we aim to cover a wide range of microcontrollers, which are frequently used in IoT development. Also, the afterwards described platforms are real-world low-cost-targets for the NIST LWC candidates. With the recent rise of the open-source RISC-V architecture, we decided to extend our initial selection of platforms with a device, which uses a chip based on RISC-V. Providing templates for different architectures should show the simple expansion of the framework on the one hand. On the other hand, the diversity of the test platforms amplifies a fair evaluation of various cipher optimizations for low-, mid- and high-performance MCUs. The following paragraphs introduce the key features of each test platform briefly.

Arduino Uno R3 The Arduino Uno features an 8 bit ATmega328P MCU from Atmel/Microchip. The AVR-based controller has a clock speed of 16 MHz and provides 32 KB flash. The ATmega chip represents a simple low-end/low-cost processor, which is very popular in the community.

STM32F1 “bluepill” The “bluepill” or “blackpill” boards are cheap 32 bit evaluation platforms based on a STM32F103C8T6 MCU. The ARM Cortex-M3 core provides a clock frequency of 72 MHz and 64 KB of flash memory.

STM32 NUCLEO-F746ZG The F746ZG NUCLEO board is considered a high-power 32 bit device. It features 1 MB of flash memory and an ARM Cortex-M7 core which clocks at a frequency of up to 216 MHz. In contrast to the “bluepill”, this chip is already better suited for more resource-intensive IoT products.

Espressif ESP32 WROOM The Espressif ESP32 WROOM evaluation kit is based on a dual-core 32 bit Xtensa LX6 MCU. With a maximum clock frequency of 240 MHz and a flash memory size of 4 MB, it is currently the second most powerful platform supported by the test framework. The ESP32 and its predecessor ESP8266 are widely used for various IoT and automation projects.

Sipeed Maixduino RISC-V 64 The Sipeed Maixduino development board is including a Kendryte K210 64-bit MCU clocked at a maximum of 400 MHz and 8 MB on-chip SRAM. The Maixduino also features a MAIX AI module and an ESP32 MCU used for wireless communication. The module is advertised as a development platform for AI and IoT applications.

4 Results

In this section, we provide an excerpt of some preliminary results obtained with our test setup. As stated beforehand, the result dataset is continuously extended since we receive and also start to contribute optimized implementations of various ciphers which then get evaluated. All test data is publicly available on lwc.las3.de. In the following, we show inner-family comparisons of some tested ciphers as an example. We include the test result for the reference implementation for completion purposes. However, competitive performance evaluations should always take into account the maturity and the optimization level of an implementation.

Figure 2 shows a comparison of the speed benchmark results of the *RomulusN1v12* variant on the STM32F7 MCU for two optimized implementations (we do not consider the reference implementation – *ref* –to be optimized). *rhys* refers to an optimized C implementation developed by Rhys Weatherley[4]. Weatherley

[4] https://github.com/rweather/lightweight-crypto

provided implementations optimized for 32-bit MCUs for all 2nd round candidates. Moreover, some performance figures were also obtained and published[5]. Every implementation called *rhys* in the upcoming plots refers to the work from Weatherley.

The *armsc* result in figure 2 corresponds to an implementation from Alexandre Adomnicai optimized for ARM architecture. It can be observed that both optimizations easily outperform the reference implementation which supports the claim that reference implementations should not be used for a competitive comparison. Moreover, the tailored version for the ARM instruction is roughly 69% faster than *rhys*. This might be linked to the *rhys* implementation being optimized for generic 32-bit MCUs, while *armsc* is specifically built to perform well on an ARM chip.

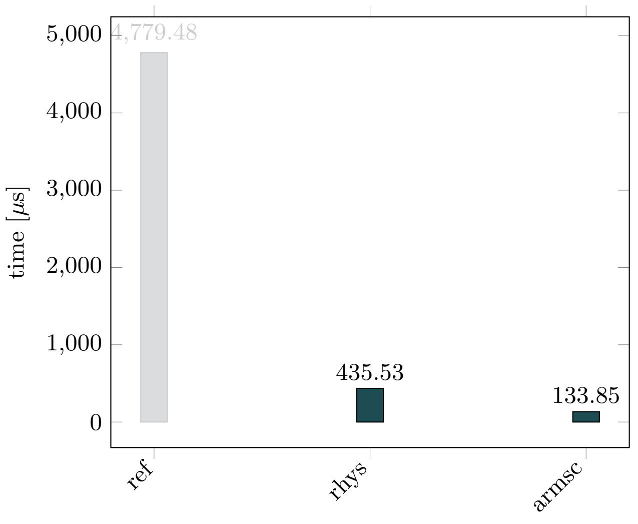

Fig. 2: Speed measurements of RomulusN1v12 on the STM32F7

Figure 3 depicts preliminary speed results for the *Xoodyak* cipher on the STM32F103. Besides the reference implementation, we again include the *rhys* optimization, as well as implementations from the cipher designers, which have been extracted from the eXtended Keccak Code Package (XKCP)[6]. Here, it is specifically interesting to see how the performance differs between the optimizations for ARMv6M and ARMv7M. As the STM32F103 features a Cortex-M3 core with ARMv7M architecture, it is reasonable that the *xkcp-armv7m* variant outperforms version *xkcp-armv6m*. The more generic *rhys* implementation ranks between the two.

[5] https://rweather.github.io/lightweight-crypto/index.html
[6] https://github.com/XKCP/XKCP

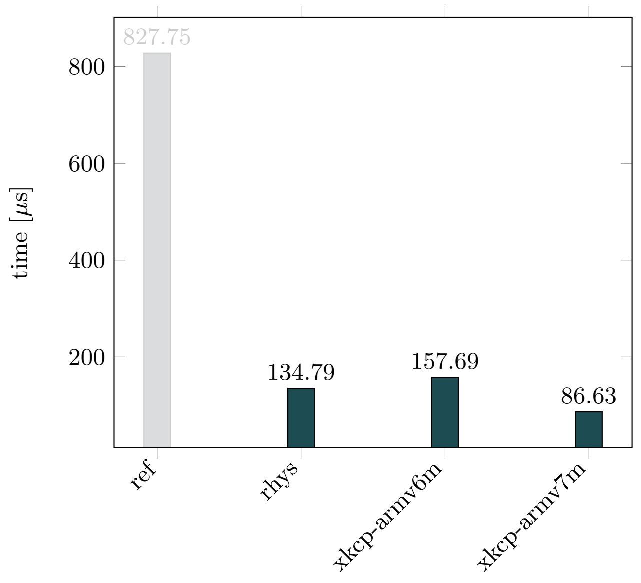

Fig. 3: Speed measurements of Xoodyak on the STM32F103

Figure 4 shows the results of the speed benchmark of the *GIFT-COFB* candidate for multiple implementations. The *opt32* variant represents an optimization for 32-bit platforms, which has originally been submitted within the NIST package. The *arm-** implementations are different optimizations mostly written in ARM assembly. Again, these have been provided by Alexandre Adomnicai. The *rhys* submission performs very similar to the *opt32* version, likely because both have been programmed with optimization strategies for more generic 32-bit architectures in mind. *arm-fast* leads on the performance chart, while *arm-compact* ranks last. In between these two, *arm-balanced* is placed. Since *arm-fast* obviously targets high-speed use cases and *arm-compact* seems to mainly aim for a small ROM footprint, these results reflect this intent.

In Figure 5, we compare the ROM size of the *GIFT-COFB* implementations mentioned beforehand on the STM32F7 platform. We want to especially emphasize the results for the *arm-** optimizations. In contrast to the speed test case, *arm-fast* now ranks last, while *arm-compact* produces the smallest ROM footprint. Again, *arm-balanced* is located in between the other two ARM variants. This supports the claim that these implementations suit their use case. Depending if either ROM size and/or speed are a priority, one can choose either implementation.

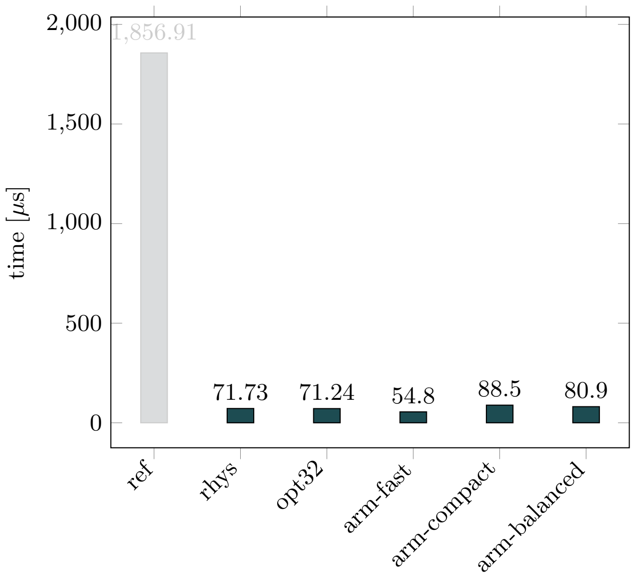

Fig. 4: Speed measurements of GIFT-COFB128v1 on the STM32F7

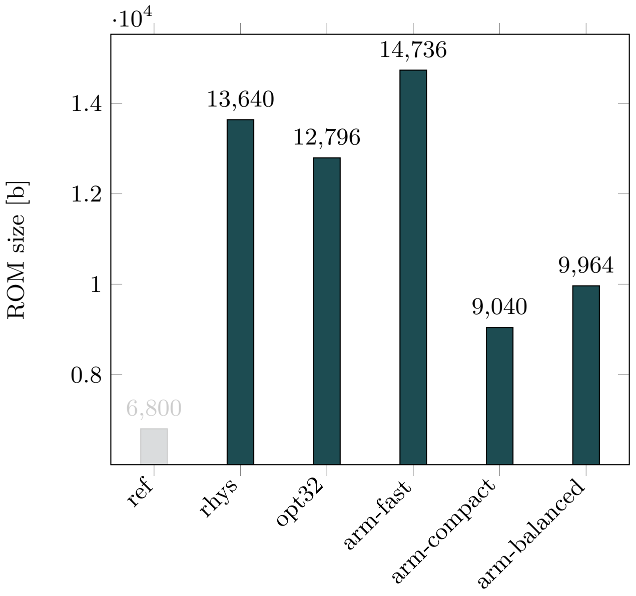

Fig. 5: ROM size measurements of GIFT-COFB128v1 on the STM32F7

5 Conclusion

In this paper, we introduced a framework for benchmarking cipher software implementations of the NIST LWC project on various MCUs. We gave an overview over its architecture, the core components and the communication channels. It was described how the compilation, the test procedure and the results acquisition are conducted. We explained which performance tests can be carried out at the moment and showed how the test setup can be extended to support e.g. more hardware platforms or different test inputs. Additionally, we introduced an online submission system, job scheduling and database backend to allow developers to hand in their most recent implementations and receive the benchmark results upon test completion. With exposing and continuously updating the source code and all performance figures on a public website, we aim for maximum transparency and easy reproducibility of our results.

We also showed an excerpt of preliminary benchmark data for some cipher implementations in this paper. Due to the high dynamics in the development of new and more optimized implementations, we decided to not include a full set of results in a static publication. All test data for all reference and known optimized implementations will be periodically updated on the public website. Moreover, as mentioned earlier, tailored software implementations do not currently exist for every cipher (variant) and therefore a comparison in between the candidates could sometimes be unfair or lead to wrong conclusions. Again, we believe the best strategy is to index and test all available upcoming implementations, so that we will reach a competitive and more comparable data set with the advancement of the NIST LWC competition.

6 Future Work

Apart from the already provided tests, different real-world test cases e.g. in the context of a TLS connection could be integrated into the framework. Furthermore, adding support for other MCU platforms could be considered. By integrating the RISC-V-based chip, we have already proven the possibility of an easy integration of novel devices. Extending the portfolio especially on the lower-performance end will be a future project. Another area of research in this context involves side-channel analysis. We could try to add a feature to gather e.g. power traces during the execution of the ciphers. When capturing these traces in a predefined and fixed manner, their release could facilitate an investigation of the ciphers' resistance against basic side-channel attacks like Correlation Power Analysis (CPA) and Differential Power Analysis (DPA). Since NIST also specified resistance against such attacks as a nice-to-have feature in their call for algorithms, this could help in the evaluation of the candidates.

7 Acknowledgements

This work is supported by the Bavarian State Ministry of Science and the Arts in the framework of the Bavarian Research Institute of Digital Transformation (bidt), the PTJ and the German Federal Ministry of Economic Affairs and Energy on the basis of a decision by the German Bundestag (grant 0350042A).

References

1. Ankele, R., Ankele, R.: Software benchmarking of the 2nd round caesar candidates (09 2016). https://doi.org/10.13140/RG.2.2.28074.26566
2. Bernstein, D.J.: Caesar: competition for authenticated encryption: Security, applicability, and robustness (2014), https://competitions.cr.yp.to/caesar.html (accessed 2019-07-28)
3. Bernstein, D.J., Lange, T.: eBACS: ECRYPT benchmarking of cryptographic systems, http://bench.cr.yp.to (accessed 2019-07-28)
4. Cazorla, M., Gourgeon, S., Marquet, K., Minier, M.: Survey and benchmark of lightweight block ciphers for msp430 16-bit microcontroller. Sec. and Commun. Netw. **8**(18), 3564–3579 (Dec 2015). https://doi.org/10.1002/sec.1281, http://dx.doi.org/10.1002/sec.1281
5. Dinu, D., Biryukov, A., Großschädl, J., Khovratovich, D., Corre, Y.L., Perrin, L.: FELICS - Fair Evaluation of Lightweight Cryptographic Systems. NIST Workshop on Lightweight Cryptography (2015)
6. Dinu, D., Le Corre, Y., Khovratovich, D., Perrin, L., Großschädl, J., Biryukov, A.: Triathlon of lightweight block ciphers for the internet of things. Journal of Cryptographic Engineering (07 2015). https://doi.org/10.1007/s13389-018-0193-x
7. Dworkin, M.J.: Nist. no. special publication (nist sp)-800-38d: Recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac (2007)
8. Hyncica, O., Kucera, P., Honzik, P., Fiedler, P.: Performance evaluation of symmetric cryptography in embedded systems. In: Proceedings of the 6th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems. vol. 1, pp. 277–282 (Sep 2011). https://doi.org/10.1109/IDAACS.2011.6072756
9. Kaps, J.P.: eXtended eXternal Benchmarking eXtension (XXBX). SPEED-B - Software performance enhancement for encryption and decryption, and benchmarking (Oct 2016), utrecht, Netherlands, invited talk
10. Tschofenig, H., Pegourie-Gonnard, M.: Performance of state-of-the-art cryptography on arm-based microprocessors. NIST Workshop on Lightweight Cryptography (2015)
11. Wenzel-Benner, C., Gräf, J.: Xbx: external benchmarking extension for the supercop crypto benchmarking framework. In: Mangard, S., Standaert, F.X. (eds.) Cryptographic Hardware and Embedded Systems, CHES 2010. pp. 294–305. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)