Available online at www.sciencedirect.com

ScienceDirect

IFAC PapersOnLine 55-4 (2022) 266–271

Benchmarking of Symmetric Cryptographic Algorithms on a Deeply Embedded System

Heiko Bühler [*], **Andreas Walz** [*], **Axel Sikora** [*]

[*] *Institute of Reliable Embedded Systems and Communication Electronics (ivESK) Offenburg University of Applied Sciences, Badstrasse 24, 77652 Offenburg, Germany*

{*heiko.buehler, andreas.walz, axel.sikora*}*@hs-offenburg.de*

Abstract: In this paper, we study the runtime performance of symmetric cryptographic algorithms on an embedded ARM Cortex-M4 platform. Symmetric cryptographic algorithms can serve to protect the integrity and optionally, if supported by the algorithm, the confidentiality of data. A broad range of well-established algorithms exists, where the different algorithms typically have different properties and come with different computational complexity. On deeply embedded systems, the overhead imposed by cryptographic operations may be significant. We execute the algorithms AES-GCM, ChaCha20-Poly1305, HMAC-SHA256, KMAC, and SipHash on an STM32 embedded microcontroller and benchmark the execution times of the algorithms as a function of the input lengths.

Copyright © 2022 The Authors. This is an open access article under the CC BY-NC-ND license (https://creativecommons.org/licenses/by-nc-nd/4.0/)

Keywords: cybersecurity, benchmarking, cryptography

1. INTRODUCTION

The ever-growing pervasion of interconnected smart computer systems and Cyber Physical Systems (CPSs) in all areas of modern life requires secure data communication and storage. Strong cryptographic algorithms are among the major means to achieve security goals like confidentiality, integrity, and authenticity (Ferguson et al. (2011)).

However, security does not come for free. In addition to processing overhead of key and credential management, strong cryptographic algorithms tend to be computationally complex and expensive. Implementing and using them is particularly challenging on deeply embedded systems with severely restricted computational and memory resources, as the execution of such algorithms may easily be in conflict with the required real-time responsiveness of the system (Walz et al. (2016)).

For the selection of a cryptographic algorithm for a particular use case, the performance (i.e., execution time) of the algorithm on the target system should, of course, not be the only criterion, but it often is among the most important. A pragmatic approach to determine the performance of cryptographic algorithms is to actually execute them on a real system and benchmark the required runtime, preferably as a function of the lengths of algorithm inputs.

For obvious reasons, the performance of algorithms in general, not only of cryptographic algorithms, may strongly depend on the platform they are executed on. For cryptographic algorithms in particular, this holds true not only because different CPUs may run at different clock frequencies, but also because these may feature different architectures and degrees of hardware-based support for particular cryptographic algorithms (Skuballa et al. (2021)).

In this paper, we present an experimental analysis of the performance of

- *AES-GCM* (FIP (2001); NIS (2007)), which is common cipher with widespread hardware acceleration support,
- *ChaCha20-Poly1305* (Nir and Langley (2015)), which is a cipher that is considerably faster than AES in software-only implementations,
- *HMAC-SHA256* (Krawczyk et al. (1997); Hansen and Eastlake (2011)),
- *KMAC* (NIS (2016)), which uses the novel SHA-3 / Keccak algorithms and
- *SipHash* (Aumasson and Bernstein (2012)), which claims faster runtimes than existing message authentication code (MAC) algorithms, determined on a single platform.

For our measurements, we use an STM32 evaluation board, featuring an ARM Cortex-M4 microcontroller clocked at 100 MHz, which is a very widespread architecture in the field of deeply embedded systems. Our study intends to illustrate the *relative* behaviour of these algorithms as a function of the input lengths. For the implementations, we rely on well-known open source libraries like mbedTLS [1], CycloneCRYPTO [2], and for SipHash on the author's C reference implementation [3].

The paper is structured as follows. Section 2 provides some background on the topic, in particular with regard to

[1] https://tls.mbed.org/
[2] https://www.oryx-embedded.com/products/CycloneCRYPTO.html
[3] https://github.com/veorq/SipHash

2405-8963 Copyright © 2022 The Authors. This is an open access article under the CC BY-NC-ND license.
Peer review under responsibility of International Federation of Automatic Control.
10.1016/j.ifacol.2022.06.044

symmetric cryptographic algorithms. Section 3 describes our experimental setup that we use to measure the performance of cryptographic algorithms. In Section 4 the results obtained from our measurement campaign are presented. Section 5 finally summarizes and concludes our paper.

2. BACKGROUND

Cryptography is the major means to achieve, for example, secure communications in open networks. Symmetric cryptographic algorithms facilitate the protection of bulk data against disclosure (by encryption), manipulation (by securing its integrity) and spoofing (by authentication) using a cryptographic key shared between the legitimate communication entities. Asymmetric cryptographic algorithms are the complementary and may help to share or distribute the required symmetric keys securely. For the work presented within this paper, we purely focus on symmetric cryptographic algorithms.

Classically, symmetric encryption (privacy protection) and authentication algorithms were two distinct building blocks (Bellare and Namprempre (2008)). Today, state-of-the-art is to use *authenticated encryption with associated data* (AEAD), a category of cipher operation modes which combine both encryption and authentication into a single primitive (Rogaway (2002)).

An AEAD primitive supports two distinct operations, which are called *authenticated encryption* and *authenticated decryption*. Input to an authenticated encryption operation is a fixed-length secret cryptographic key, a fixed-length unique initialization vector (IV), a variable-length bit string of *plaintext* (PT) and a variable-length bit string of *additional associated data* (AAD). It outputs a variable-length bit string of ciphertext (CT), which is the encrypted plaintext (same length as plaintext but encrypted), and a fixed-length *integrity check value* (ICV). The integrity protection covers both the plaintext and the additional associated data.

The input to an authenticated decryption operation, which is the inverse operation to authenticated encryption, is the secret cryptographic key, the initialization vector, the ciphertext, and the additional associated data. The operation outputs the decrypted plaintext after successful verification of the ICV or an error indication if the verification failed.

An interesting feature of AEAD modes as described above is the option to achieve both authentication-only protection as well as authenticated encryption with the same algorithm. The choice between these two options is just governed by the input to which to-be-protected data is supplied: if given to the plaintext (PT) input, data is authenticated *and* encrypted; if given to the additional associated data (AAD) input, data is only authenticated. However, it is perfectly fine (and a quite typical case) to supply one portion of data (e.g., a message's payload) to the PT input and another portion of data (e.g., a message's header) to the AAD input.

There are two interesting things to note here. First, pure encryption algorithms (i.e., those that do not offer authentication) are not enough to form valid authenticated encryption and decryption operations. This is because encryption without authentication has inherent security issues (Bellovin (1996)). Second, pure authentication algorithms (i.e., those that do not offer encryption), on the other hand, can be used to form limited authenticated encryption and decryption operations that do not accept non-empty input to their plaintext (PT) and ciphertext (CT) inputs, respectively.

Out of the algorithms we are studying (i.e., AES-GCM, ChaCha20-Poly1305, HMAC-SHA256, KMAC, and SipHash), AES-GCM and ChaCha20-Poly1305 are native AEAD algorithms supporting both encryption and authentication. HMAC-SHA256, KMAC, and SipHash, only offer authentication. In accordance with the note above, we nevertheless treat these authentication-only algorithms as limited AEAD algorithms, using them only with zero-length plaintext and ciphertext inputs.

Several parameters affect the performance of cryptographic operations. The strongest impact among them has the choice of the cryptographic algorithm (e.g., ChaCha20-Poly1305), the choice of a corresponding implementation, and the length of the input data to be processed by the algorithm respectively the implementation. Note that parameters like the length of the cryptographic key (input) or the length of the ICV (output) are most often determined by the cryptographic algorithm, making them parameters that allow to *directly* affect performance only for cryptographic algorithms which support multiple parameter values (like, e.g., the three key lengths for AES or the two ICV lengths for SipHash).

Not only the input data length has an impact on the execution times of an cryptographic algorithm, but also the key length might affect it. AES, which is the underlying cipher of AES-GCM, supports different key lengths: 128, 192 or 256 bits. It has an effect on the execution times of AES, because the number of rounds performed during the execution of the algorithm depends on the key size. A key size of 128 bits results in 10 rounds, a 192 bit key in 12 rounds, and a 256 bit key in 14 rounds.

The ChaCha20-Poly1305 algorithm always uses a 256 bit key and does not offer to vary this key length.

When using HMAC-SHA256 the block size is determined by the underlying hash function, which is 256 bit for SHA256. The size of the authentication tag is same as the block size (i.e., 256 bit). It is common to truncate this value (e.g., to 128 bit), however this does not affect the computation time. The key size is variable. It is not recommended to use a shorter than the block size, as it would degrade the security strength. When using a key longer than the block size, it would be hashed first, and then the resulting hash would be used as the key.

SipHash uses a 128 bit key and allows no variation in key length.

3. EXPERIMENTAL SETUP

3.1 General

In our study we have used a *STM32F429I-DISC1* development board from STMicroelectronics. The board is

Table 1. Cryptographic algorithms and the implementations we use in this benchmark. Mode A refers to authentication, and A+E to authenticated encryption

#	Algorithm	Library	Modes
1	AES-GCM/GMAC	mbedTLS 2.16.2	A, A+E
2	HMAC-SHA256	mbedTLS 2.16.2	A
3	ChaCha20-Poly1305	mbedTLS 2.16.2	A, A+E
4	KMAC	CycloneCRYPTO	A
5	SipHash	Author's ref. implementation	A

equipped with an STM32F4ZIT6U [4], which is a single-core ARM Cortex-M4 microcontroller with 2 MB of flash memory and 256 kB of SRAM. We clocked the microcontroller at 100 MHz, which is below the maximum allowed clock frequency of 180 MHz.

On the microcontroller is running a bare-metal application that means that no (real-time) operating system or any scheduling effects have to be concerned. The software was built using the official STM32CubeIDE [5] for all the platform-specific code plus own software components dealing with the actual benchmark.

Although STM32-F469 microcontrollers have an on-chip cryptographic accelerator, we do not use it, because we are solely interested in the software-only performance. Another reason why we decided against the cryptographic accelerator was its limited support of our desired algorithms. For example it does not provide support for ChaCha20-Poly1305.

To sample the actual execution time the *microtags* (Walz (2017)) library was used. It provides lightweight methods for time and event logging. Each tag consists of a current 32 bit timestamp plus an event code and/or additional data. The timestamp originates from a platform-dependent source. For this purpose we use TIMER2 with no pre-scaler in free-run mode, which results in a 32 bit time value counting upwards with half the system clock (i.e., 50 MHz or 20 ns per tick). Every time we capture a tag, the current value of this 32 bit timer is used for the current timestamp. Those tags are then written into a statically (pre-)allocated section in memory to interfere with the running code as less as possible. After having executed the critical code section (e.g., the encryption/decryption operation) the tags in the buffer are written to a serial interface (USART1 in our case) for further analysis.

The output is then received on a computer to save it to a file. Afterwards, the file is analyzed and plots like those shown in the next chapter are generated.

A typical application programming interface (API) for such cryptographic algorithm is in the form of `start`, `update`, `finalize` operations. We always take the time before and after each of those operations. However, in the end we are mainly interested in the total operation time of the algorithms shown in table 1. The memory footprint of those algorithms and implementations is not part of this analysis.

We studied two different scenarios, which will be explained in the next sections.

[4] https://www.st.com/resource/en/datasheet/stm32f427vg.pdf
[5] https://www.st.com/en/development-tools/stm32cubeide.html

3.2 Authentication Only

This scenario represents the use-case where the data are not confidential, that is, there is no interest in keeping them secret. As shown in the introduction, AEAD algorithms allow to let the plaintext input empty and just cryptographically protect the additional associated data with an integrity check value. In this scenario we vary the length of the AAD between 40 and 1440 bytes.

From our selected algorithms in table 1 AES-GMAC (1), HMAC-SHA256 (2), ChaCha20-Poly1305 (3), KMAC (4) and SipHash (5) support this operating mode, which transform those AEAD algorithms effectively into a MAC.

3.3 Authenticated Encryption

In contrast to the previous scenario, we consider part of the data as confidential, resulting in a non-empty plaintext input. However, we still use the AAD input because in a real-world setting some part of the data must not be encrypted in order to provide a way to transport the data to the recipient (e.g., IP address, MAC address, header fields, etc). We still vary the total length of the data between 40 and 1440 bytes, however a fixed part of 4 bytes is treated as AAD. This results in a PT length between 36 and 1436 bytes.

From our selected algorithms in table 1 only AES-GCM (1) and ChaCha20-Poly1305 (3) support this operating mode.

4. EXPERIMENTAL RESULTS

We define a benchmark with a certain configuration (e.g., encryption of a payload with length 40 bytes with AES-GCM and a key size of 128 bits) as a single run. Each run was repeated three times. The variation between those runs is negligible, which can be explained by the absence of an operating system with concurrent threads and the absence of interrupts.

The timing values below include only the time required to perform the actual encryption/protection operation and writing the resulting cipher text and ICV to the output buffers. The time to set up a cryptographic context, initialize the algorithm dependent registers (e.g., round keys) is excluded from the measurements.

For the remainder of this paper we designate cryptographic algorithms according to this scheme:

`<name>-<icvLen>#<keyLen>`.

The cryptographic primitive is referenced by the `<name>` (e.g., AES-GCM or KMAC), the length of the generated ICV is stated by `<icvLen>` in bits and, finally, the length of the cryptographic key being used is given by `<keyLen>` in bits.

4.1 Authentication Only

The benchmark results for the authentication only scenario is visualized in figure 1. It can be clearly seen that KMAC with an output MAC length of 128 bits and a 256 bit key (thus the algorithm is designated as KMAC-128#256) is the slowest implementation for all payload sizes. The ranking of the other implementations require a more detailed look.

For payload sizes below 600 bytes SipHash with two compression, four finalization rounds, an output MAC length of 64 bits and using a 128 bit key (SipHash-2-4-64#128) is always the fastest algorithm. Above this payload size ChaCha20-Poly1305 with an output MAC length of 128 bits and a 256 bit key is the fastest algorithm. In general this algorithm performs very well throughout all payload sizes. The slope of its curve is shallow, resulting in good performance even for bigger payloads.

For payload sizes below 100 bytes AES-GMAC, both with a 128 bit as well as a 256 bit key, is faster than the HMAC-SHA2-256 implementation with a 128 bit key. The small performance difference between AES-GMAC-128#128 and AES-GMAC-128#256 although more AES rounds have to be computed, can be explained by the fact, that the plaintext input is left empty and now the GHASH unit acts as the bottleneck. This unit is equal for both key lengths.

For payload sizes above 100 bytes the HMAC-SHA2-256#128 outperforms both AES-GMAC algorithms in our used implementation.

4.2 Authenticated Encryption

For our second scenario (authentication and encryption, A+E) the results are visualized in figure 2. For comparison also the performance figures for AES-GMAC and ChaCha20-Poly1305 from the previous authentication only scenario are shown here. Again, ChaCha20-Poly1305-128#128 outperforms AES-GCM for all payload sizes.

The difference in the key lengths for AES-GCM becomes more clear in this scenario, where the (to-be-encrypted) plaintext is not empty anymore. Now the increased amount of AES rounds (14 vs. 10) has an effect on the execution time, especially for longer payloads.

5. SUMMARY AND CONCLUSION

In this paper, we presented our measurements of the runtime of different symmetric cryptographic algorithms on an embedded STM32 platform with an ARM Cortex-M4 microcontroller in a bare metal setting. Depending on each algorithm's capabilties, we measure authentication-only and authenticated encryption scenarios.

Our study experimentally shows that, as one would expect, the runtime of each algorithm is approximately linear in the length of the input (to-be-encrypted-and-authenticated plaintext or to-be-authenticated additional associated data). However, for different algorithms, both the slope and the offset differ significantly. The implication is that there is no generally best-performing algorithm. However, it can be stated that, under the conditions defined by our setup (execution platform, algorithm implementations, compiler optimizations), ChaCha20-Poly1305 is by several factors faster than AES-GCM/GMAC. This result should not surprise, as on of the design goals of ChaCha20-Poly1305 is to be very efficient when implemented solely in software. Keep in mind that we did not use any AES hardware acceleration, which would potentially execute AES much faster than in software (Skuballa et al. (2021)). For authentication-only scenarios, SipHash-2-4-64 is the fastest algorithm for smaller input lengths, being overtaken by ChaCha20-Poly1305 for larger input lengths. One downside of SipHash is the smaller security level it provides due to its output length of only 64 bits, in comparison to the other algorithms providing 128 bit or even 256 bit authentication tags [6]. HMAC-SHA256 is somewhere in the middle between AES-GCM/GMAC and ChaCha20-Poly1305 for most input lengths, and is slower than AES-GCM/GMAC for small payloads. Finally, we want to clearly state, that these findings are highly dependent on the setup (see above) and results may differ when varying parameters of the setup.

REFERENCES

(2001). Federal Information Processing Standards Publication 197: Announcing the Advanced Encryption Standard (AES). Technical report, NIST.

(2007). National Institute of Standards and Technology Special Publication 800-38D: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. Technical report, NIST.

(2016). National Institute of Standards and Technology Special Publication 800-185: SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash and ParallelHash. Technical report, NIST.

Aumasson, J.P. and Bernstein, D.J. (2012). Siphash: A fast short-input prf. In S. Galbraith and M. Nandi (eds.), *Progress in Cryptology - INDOCRYPT 2012*, 489–508. Springer Berlin Heidelberg, Berlin, Heidelberg.

Bellare, M. and Namprempre, C. (2008). Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *Journal of Cryptology*, 21, 469–491. doi:10.1007/s00145-008-9026-x.

Bellovin, S. (1996). Problem areas for the IP security protocols. In *6th USENIX Security Symposium (USENIX Security 96)*. USENIX Association, San Jose, CA. URL `https://www.usenix.org/conference/6th-usenix-security-symposium/problem-areas-ip-security-protocols`.

Ferguson, N., Schneier, B., and Kohno, T. (2011). *Cryptography engineering: design principles and practical applications*. John Wiley & Sons.

Hansen, T. and Eastlake, D.E. (2011). US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF). RFC 6234. doi:10.17487/RFC6234. URL `https://rfc-editor.org/rfc/rfc6234.txt`.

Krawczyk, D.H., Bellare, M., and Canetti, R. (1997). HMAC: Keyed-Hashing for Message Authentication. RFC 2104. doi:10.17487/RFC2104. URL `https://rfc-editor.org/rfc/rfc2104.txt`.

[6] Note that SipHash also exists in a flavor that features a 128 bit authentication tag (see `https://github.com/veorq/SipHash`).

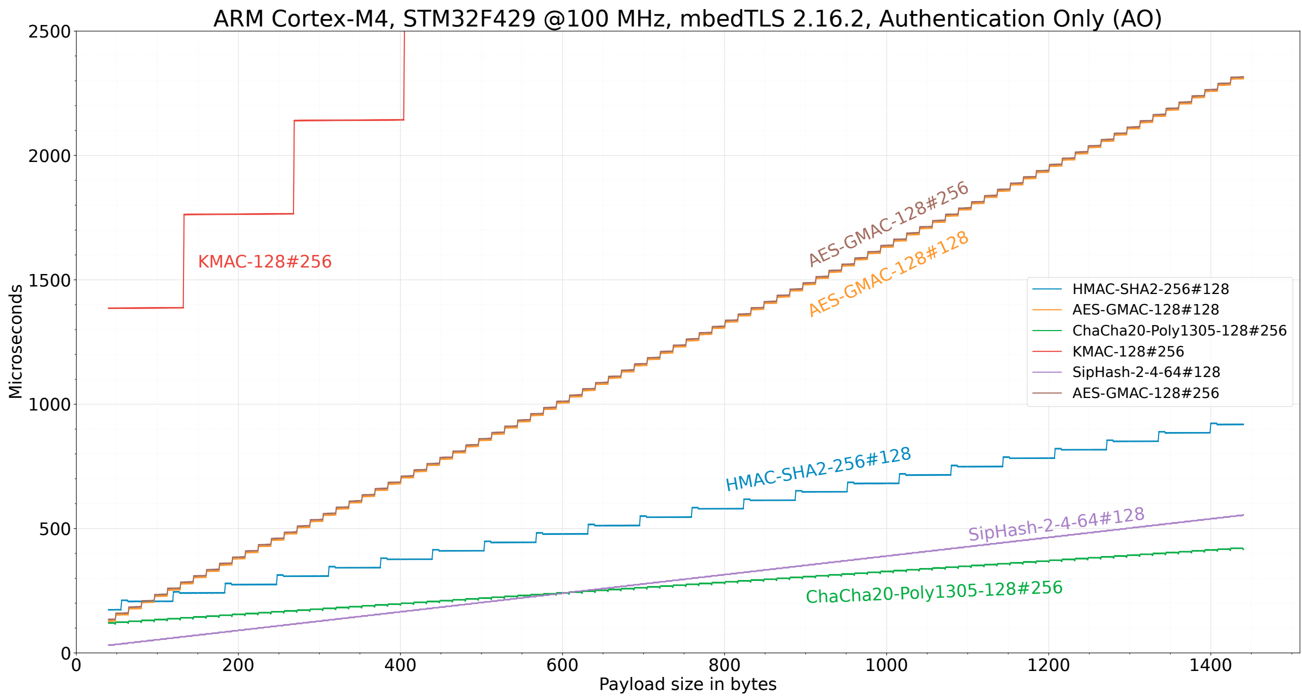

Fig. 1. Authentication-only scenario. On the y-axis the time in μs required to generate the ICV for the data. On the x-axis the length of the to-be-authenticated data in bytes. SipHash is the fastest algorithm for payloads below 600 bytes, otherwise ChaCha20-Poly1305 performs faster.

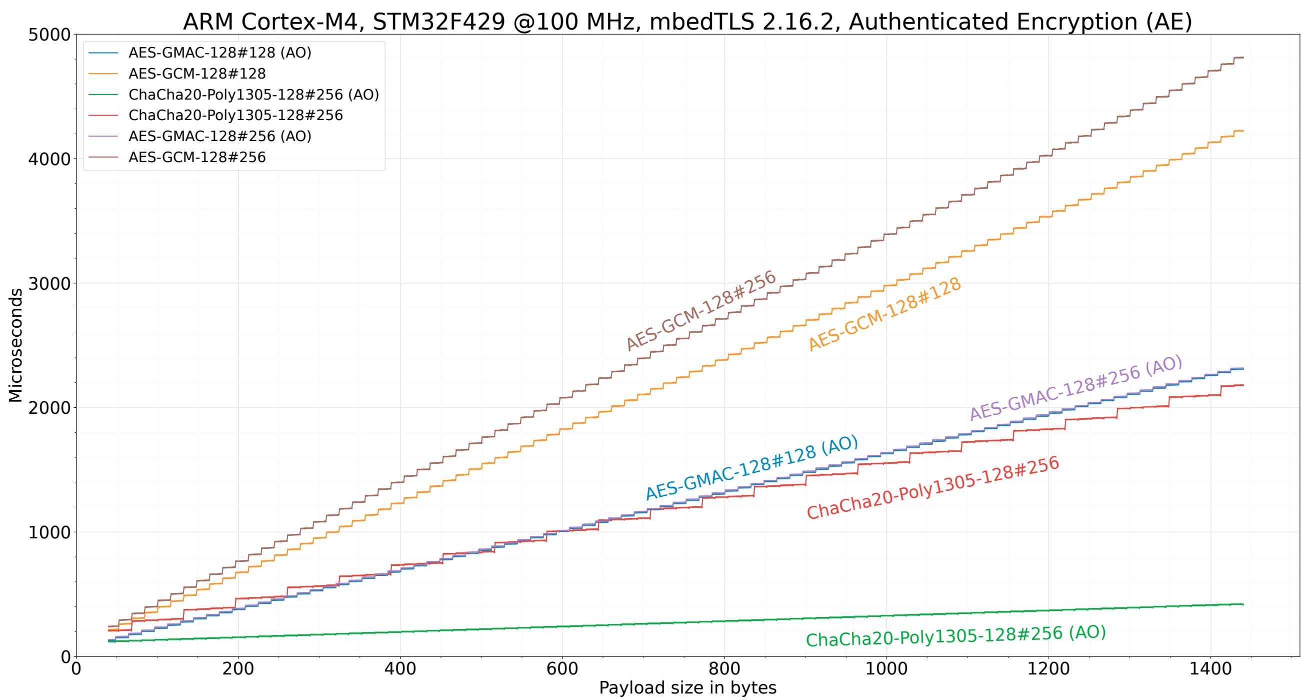

Fig. 2. Authenticated encryption scenario. On the y-axis the time in μs required to encrypt the data and calculate the ICV. On the x-axis the length of the to-be-processed data in bytes. For reference the performance figures from the previous scenario are indicated with *(AO)*. Again, ChaCha20-Poly1305 performs best in our test scenario.

Nir, Y. and Langley, A. (2015). ChaCha20 and Poly1305 for IETF Protocols. RFC 7539. doi:10.17487/RFC7539. URL https://rfc-editor.org/rfc/rfc7539.txt.

Rogaway, P. (2002). Authenticated-encryption with associated-data. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, CCS '02, 98–107. Association for Computing Machinery, New York, NY, USA. doi:10.1145/586110.586125. URL https://doi.org/10.1145/586110.586125.

Skuballa, M., Walz, A., Bühler, H., and Sikora, A. (2021). Cryptographic protection of cyclic real-time communication in ethernet-based fieldbuses: How much hardware is required? In *IEEE 26th International Conference on Emerging Technologies and Factory Automation (ETFA)*.

Walz, A. (2017). microtags. URL https://github.com/phantax/microtags. Original-date: 2017-08-22T14:29:43Z.

Walz, A., Kehret, O., and Sikora, A. (2016). Comparison of cryptographic implementations for embedded TLS. In *Proceedings of the Embedded World Conference*.