Universiteit Leiden
The Netherlands

Opleiding Informatica

Benchmarking Lightweight Cryptography for TLS

Lars Ruigrok

Supervisors:
Nele Mentens & Kristian Rietveld

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

04/08/2022

Abstract

Lightweight cryptography is cryptography for constrained environments, where traditional cryptography imposes unacceptable resource costs for many applications or is otherwise unsuitable. NIST has started a process to solicit, evaluate and standardise such algorithms. We evaluate performance of lightweight cryptography algorithms part of the NIST lightweight cryptography competition in the context of a TLS session over WiFi. By focusing on a number of TLS usage scenarios, we hope to more accurately model real-world performance. Though further optimisations may be possible, we conclude that existing implementations of the Ascon and Xoodyak ciphers already outperform the established ChaCha20-Poly1305 cipher.

Contents

Chapter 1

Introduction

With the increasing prevalence of the Internet of Things (IoT), embedded systems and other resource-constrained devices are increasingly connected to the Internet. As these technologies become more influential, it is increasingly important to secure their communications. TLS[1] is the go-to protocol for network communication privacy and authentication. However, established encryption standards such as AES (and more recently ChaCha20-Poly1305) used in TLS impose resource costs which cannot be met by (few [AV19]) constrained devices. Lightweight cryptography aims to find new tradeoffs between resource costs (e.g. time/memory/storage/power/area/"realtimenessguarantees") and security for such devices. The absence of established standards in this field of cryptography is a hurdle to interoperability.

To fill this gap (and for other reasons we will ignore), in 2017 NIST[2] started a lightweight cryptography project to create a portfolio of recommended lightweight AEAD[3] and hashing algorithms through an open competition. 65 candidate algorithms were selected for the first round of evaluation; in 2021, the ten finalists were announced. NIST encourages public evaluation of the algorithms and their implementations and publication of the results throughout the process. [LWC]

We contribute to the evaluation of candidate algorithm performance by benchmarking software implementations in the context of a TLS session. By focusing on a number of TLS usage scenarios and evaluating full TLS sessions over WiFi, we hope to more closely model the real-world situations these ciphers may be deployed in.

Thesis overview In Chapter 2, we give a brief overview of the TLS protocol and the NIST LWC project. Chapter 2.3 lists other benchmarking initiatives and related work. Chapter 3 is centred around a number of TLS usage scenarios. In Section 3.1, we describe our implementation of these scenarios and our benchmarking setup. Section 3.2 presents our results, which we explore further in Section 3.4. In Section3.5 we compare our results with non-TLS performance measurements. Chapter 4 concludes the thesis and discusses potential future research.

This bachelor thesis is supervised by Prof.dr.ir. N. Mentens and Dr. K.F.D. Rietveld at the Leiden Institute of Advanced Compute Science (LIACS).

[1] Transport Layer Security
[2] U.S. Department of Commerce's National Institute of Standards and Technology
[3] Authenticated Encryption with Associated Data

Chapter 2

Preliminaries

2.1 TLS

TLS is a client-server network protocol providing authentication, confidentiality and integrity, best known for its use in HTTPS. To establish a TLS 1.3[1] session, the client and server perform a handshake procedure:

1. The client connects to the server requesting a secure connection (application-specific, e.g. connecting to port 443 for HTTP over TLS). It sends a ClientHello message containing a list of supported *ciphersuites* and key exchange information a.o.

2. The server responds with a ServerHello, indicating the chosen cipher suite, providing key exchange information and (optionally) including a certificate.

3. Using the exchanged information, client and server determine the shared key and both send an encrypted Finished message. When both peers successfully decrypt this message, the handshake is complete and application data can be sent, split into encrypted TLS records.

To save repeated cryptographic operations, after the first handshake an abbreviated handshake can be performed, skipping the full key exchange and certificate verification steps.

Before closing a TLS connection, an (encrypted) CloseNotify alert message is sent by the closing endpoint.[2]

The algorithms we evaluate are responsible for encrypting/decrypting the certificate, Finished and CloseNotify messages, and application data records. These tasks are normally performed by AES or ChaCha20-Poly1305 (the latter is preferred if there is no special hardware support for AES).

This is a simplified summary of the parts of TLS relevant to this thesis; for a complete specification of the protocol see RFC 8446.

2.2 NIST Lightweight cryptography standardisation process

NIST has initiated a competition to select a number of lightweight cryptographic algorithms (algorithms suitable for constrained environments such as sensor networks or embedded devices) for standardisation.

[1] When not specified, TLS will refer to TLS 1.3.

[2] while the standard mandates sending a CloseNotify, in practise it is often omitted if peers can be aware of the closing of the connection through some application-specific means, e.g. HTTP/1.1 response with connection: close, or even the TCP FIN flag

Candidates must have an AEAD[3] interface: encryption takes as input not only the key, plaintext and a nonce, but also some (possibly none) associated data which will be authenticated, but not encrypted. Decryption of the ciphertext then simultaneously verifies the authenticity of the associated data. (The bytes of the ciphertext expending past the length of the plaintext will sometimes be referred to as a 'tag'.) This mode of operation is particularly suitable for TLS, where records consist of an unencrypted header and encrypted message content, which both need to be authenticated.

At the time of writitn this thesis, ten finalists have been selected in the competition. A conclusion is expected at the end of 2022. For an overview of the NIST project, see csrc.nist.gov/projects/lightweight-cryptography; in particular, for algorithm and implementation requirements see csrc.nist.gov/CSRC/media/Projects/-Lightweight-Cryptography/documents/final-lwc-submission-requirements-august2018.pdf. For a proper specification of the aead interface, see RFC 5116.

2.3 Related Work

NIST has evaluated encrypt/decrypt speed of submitted implementations on a number of microcontrollers [LWCb]. Rhys Weatherley created optimised C implementations of most round 2 candidates and similarly evaluated their performance [Wea]. Renner et al. additionally measured RAM and ROM requirements of submitted implementations and those by Weatherley, and continue$\langle word:\ d \rangle$ testing new implementations[4][RPM20]. eBACS evaluated performance of various implementations on longer (associated) data using the SUPERCOP framework as part of its eBAEAD project, though mostly on higher-end processors [eBACS]. Campos et al. focused on RISC-V platforms, evaluating various optimisations for assembly and C implementations of a small selection of candidate algorithms [Cam+]. See also NIST's status report for a summary of results [LWC2, §4], including those for hardware implementations.

All of these focus on isolated encrypt/decrypt operations, whereas our research is concerned with performance in the more "real-world" setting of a full TLS session over WiFi.

[3] Authenticated Encryption with Associated Data

[4] They evaluate performance with the test vectors included in each submission; these are not the test vectors used by NIST in their performance evaluation, but are intended only to assess correctness of the implementation. Since they are limited to the range of 0 - 32 bytes, these tests are not necessarily representative of real usage.

Chapter 3

Experiments

In this chapter, we describe our test cases, their implementation, data collection and the collected results. We briefly consider the variability of our results and compare them to those of NIST's benchmarks.

To model real-world performance, we evaluate a number of client-server scenarios based on real use cases or common usage patterns. By looking at multiple scenarios, we may gain a better understanding of the relative effects of application data vs other TLS data (negotiation, alerts, etc.). The following scenarios are considered:

1. Wearable healthcare device: client transmits 33KiB every 10 seconds, receives small confirmation.
2. Weather station: client transmits 224B every 30 minutes.
3. File transfer: one-time HTTP-style GET or POST with 16 or 256KiB data.

The first scenario is based on a real use case as described by Winderickx et al. [Win+19]. The second is claimed by Winderickx [Win20] to be derived from the work of Roussey et al. [Rou+14][1]. The last is more generally oriented towards higher-end constrained devices (smart*stuff*) which often communicate using standard internet protocols[2].

3.1 Implementation

To evaluate these scenarios, we created simple TLS client and server programs based on a version of the Mbed TLS library [Mbed]. The library has been modified to use any cipher conforming to NIST's submission guidelines ([LWCc18]) for the cryptographic operations descrived in Section 2.1. We added checks to MbedTLS' cipher lookup to call our wrappers whenever our ciphersuite is used. These wrappers call the submission API, converting error codes and handling overlapping buffers and variable tag lengths. To speed up compilation, the whole library is compiled once and ciphers are linked against that. Because the design of MbedTLS depends on static constants for some cipher parameters, we override these at runtime. For reasons we did not properly document, the encryption/decryption functions called by the wrappers are also set at runtime rather than when linking. This may add a little overhead when calling these subroutines, but this overhead is constant and thus not a problem when comparing results. What could be a problem is that this may prevent

[1] This could not be confirmed. The every half (or quarter) hour constraint is supported by Roussey et al., but we have not implemented this for practical reasons. Perhaps Winderickx based their 224B estimate on the example CSV file, but the data is only transformed into CSV format after transmission to the "console". Looking at the example CSV file, assuming numbers are transmitted as double precision floating point and date and time together as a single 64-bit timestamp, we get a total size of 49 bytes per half hour. Perhaps the 224B figure is mentioned in a talk or some auxiliary publication, but this has been difficult to track down because IRSTEA merged into INRAE in 2020.

[2] Protocols of the internet protocol suite, not specifically *the* internet protocol, which is of course a protocol of the internet protocol suite.

some compiler optimisations such as inlining. Besides this, there are three more notable differences between this modification and the potential real implementation of a selected lightweight cipher. Firstly, because the submission guidelines do not require implementations to work for overlapping input/output buffers, we always copy the input buffer, leading to some additional overhead if the implementation *does* work on overlapping buffers. Secondly, to account for the allowed variable tag length, we append an additional byte to the ciphertext indicating the actual tag length, despite most ciphers using a fixed-length tag. Lastly, MbedTLS ciphers can maintain some state between operations, while evaluated submissions must recompute any internal key representation each time.

We use a set of Python scripts to build the candidates and TLS library and to run the benchmarks. The client and server applications communicate with these scripts through a simple plain-text protocol. The client is compiled by arduino-cli 0.18.3 using ARM GCC 7.2.1, the server is compiled with GCC 9.4.0.

Source code for the scripts, client/server applications and modified library can be found at git.liacs.nl/lwc_tls/lwc_mbedtls and git.liacs.nl/lwc_tls/lwc_tls_test.

Source code for evaluated implementations is hosted at lab.las3.de/gitlab/lwc/candidates. This includes implementations part of the submissions received by NIST and optimised C implementations by Weatherley [Wea].

In general, each session will require cipher operations as outlined in Table 3.1. The standard Fragment Length is 16384B. Response may be omitted entirely[3]. An endpoint may close the connection before receiving its peer's CloseNotify. The authenticated data length is an MbedTLS implementation detail: the standard TLS header only requires 5 bytes, but MbedTLS adds an additional 8 bytes.

	repeat	**en/de -crypt**	**en/decrypted data length (B)**	**authenticated data length (B)**
Client Finished	1	en	16	13
Server Finished	1	de	16	13
Application Data: request	$\lfloor P_1 \div F \rfloor$	en	F	13
Application Data: request	1	en	$P_1 \mod F$	13
Application Data: response	$\lfloor P_2 \div F \rfloor$	de	F	13
Application Data: response	1	de	$P_2 \mod F$	13
peer Close notify	optional	de	2	13
Close notify	1	en	2	13

Table 3.1: Typical cipher operations required on the client to complete a session. For operations on the server, invert en/de crypt. P_1 and P_2 are request resp. response Payload size, F is TLS Fragment length.

Hardware setup

We run the TLS client on an Arduino Nano RP2040 connect; this microcontroller features dual 32-bit Arm Cortex-M0+ cores and the U-blox Nina W102 2.4GHz IEEE802.11b/g/n WiFi module. The RP2040 is equipped with a 1-microsecond resolution timer.

The server runs on a laptop with Intel i5-7200U 2-core 2.50GHz CPU and 8GB RAM running Ubuntu 20. The

[3]Request could also be omitted, e.g. Time Protocol, but we will not consider such scenarios.

server device also acts as the IEEE 802.11n wireless access point.

The endpoints connect over IEEE 802.11n WiFi, with about 1 metre between the client device and the access point.

Data collection

Performance is measured on the client device. Measurement starts and ends when the client initiates resp. closes the connection.

For memory measurements, we measure maximum memory use during individual encrypt/decrypt operations, again on the client device. These are not the most memory-intensive parts of the TLS session, so the maximum memory use over the entire duration of the session is actually always the same[4]. The reported memory results are thus mainly relevant in a multi-threaded environment. (For non-TLS maximum memory use see Renner et al. [RPM]) Because our method of collecting memory usage information introduces non-constant delays, memory measurements are performed separately from performance measurements.

Storage/flash memory requirements are determined by analysing the hex file produced by arduino-cli.

3.2 Results

Results will typically be given relative to the best (minimum) result with no encryption, where 'no encryption' is represented by nocrypt.memcpy, a cipher which simply copies input to output. Thus results may be interpreted as 'additional resources used by encryption'. The original unprocessed results can be found at git.liacs.nl/lwc_tls/results.

In general, cipher implementations will be denoted *algorithm*[v*version*].*implementation*: Rhys Weatherley's ChaCha20-Poly1305 implementation, for example, is denoted chachapoly.rhys. Where possible the names used match the submission directory structure. That means all unoptimised reference implementations end in '.ref'.

Throughout the results we use Weatherley's implementation of ChaCha20-Poly1305 as reference for "existing standards" rather than the more commonly used AES-GCM, as AES performance is more dependent on hardware support.

When not otherwise specified, all ciphersuites use ECDHE-RSA key exchange, other parameters are left at the defaults set by MbedTLS, and transferred data consists of random bytes in the ASCII printable range [0x20–0x7F].

When scenarios are run multiple times with the same algorithm implementation, this occurs sequentially; No special provisions have been made regarding the time of testing of different implementations.

It should be emphasised this is not intended as a comparison between algorithms; the evaluated implementations have had varying levels of optimisation and none were specifically optimised for this application on this platform.

[4] bar some outliers and memory leaks

Evaluated ciphers are currently limited to reference implementations of finalists and Weatherley's optimised versions which run on our hardware with minimal modification. In particular, this excludes the reference implementation of finalist TinyJambu, which reinterprets octet strings as sequences of 32-bit words, causing misalignment faults on our specific hardware.

3.2.1 Scenario 1: Wearable Healthcare Device

In this scenario, the client initiates a new session and transmits 33KiB data every 10 seconds, receiving 5 bytes as confirmation. Session resumption is used, and the first (non-resumed) session is excluded from the results.

Table 3.2 shows median timing results over 19 sessions (excluding the first – non-resumed – session). Total time is divided into transfer – time to transfer application data – and setup – everything else, primarily handshake and CloseNotify. The baseline (minimum) total time taken by nocrypt.memcpy is 1.985 seconds per session, with a standard deviation of 0.072s around the mean of 2.013s.

Table 3.3 shows maximum stack and heap memory use for individual operations. Most additional heap use (+8/+16) can be attributed to allocation of extra space for non-zero-length tags; only the reference implementations of aceae128 and photonbeetleaead128rate128 actually use dynamic memory allocation. There is no difference in total maximum additional memory use between the initial session and further resumed sessions.

cipher.implementation	**setup (s)**	**transfer (s)**	**total (s)**
nocrypt.memcpy (minimum)	0.420	1.564	1.985
nocrypt.memcpy (median)	0.432	1.566	1.999
nocrypt.memcpy (mean)	0.436	1.577	2.013
nocrypt.memcpy (stdev)	0.035	0.061	0.072
aesgcm128.mbedtls	+0.130	+0.080	+0.208
chachapoly.rhys	+0.001	+0.060	+0.060
ascon128v12.ref	+0.128	+0.089	+0.214
ascon128v12.rhys	+0.127	+0.081	+0.206
elephant160v1.ref	+0.290	+35.234	+35.522
elephant160v1.rhys	+0.160	+2.853	+3.013
giftcofb128v1.ref	+0.141	+1.153	+1.292
giftcofb128v1.rhys	+0.003	+0.040	+0.042
grain128aead.rhys	+0.127	+0.198	+0.324
isapk128av20.ref	+0.056	+2.229	+2.283
isapk128av20.rhys	+0.013	+0.529	+0.542
photonbeetleaead128rate128v1.ref	+0.202	+11.304	+11.504
photonbeetleaead128rate128v1.rhys	+0.130	+0.605	+0.736
romulusn1.ref	+0.145	+3.373	+3.517
romulusn1.rhys	+0.134	+0.379	+0.514
tinyjambu128.rhys	+0.128	+0.094	+0.223
xoodyakv1.ref	+0.011	+0.277	+0.287
xoodyakv1.rhys	+0.001	+0.036	+0.037

Table 3.2: (relative) time per session for scenario 1: Wearable, median of 19

cipher.implementation	stack (B)	heap (B)
nocrypt.memcpy	2096	61243
chachapoly.rhys	+288	+16
ascon128v12.ref	+0	+16
ascon128v12.rhys	+56	+16
elephant160v1.ref	+184	+8
elephant160v1.rhys	+128	+8
giftcofb128v1.ref	+16	+16
giftcofb128v1.rhys	+400	+16
grain128aead.rhys	+88	+8
isapk128av20.ref	+248	+16
isapk128av20.rhys	+128	+16
photonbeetleaead128rate128v1.ref	+128	+16
photonbeetleaead128rate128v1.rhys	+136	+16
romulusn1.ref	+304	+16
romulusn1.rhys	+508	+16
tinyjambu128.rhys	+8	+8
xoodyakv1.ref	+184	+16
xoodyakv1.rhys	+72	+16

Table 3.3: (relative) maximum heap and stack memory use for individual en/de-crypt operations of scenario 1: Wearable

3.2.2 Scenario 2: Weather station

In this scenario, the client sends 224B of data every 30 minutes. All ciphersuites use pre-shared keys. For practical reasons, requests are not actually 30 minutes apart.

Table 3.4 shows median timing results over 20 sessions. The baseline (minimum) total time taken by nocrypt.memcpy is 0.738s per session, with a standard deviation of 0.035s around the mean of 0.765s.

Table 3.5 shows maximum memory use.

3.2.3 Scenarios 3: File transfer

In this set of scenarios, the client performs HTTP-style GET or POST requests. The client either sends a small ($<$ 128B) request and receives a small header and m bytes of body, or it sends a header and m bytes of body, receiving a small response. We tested each for $m \in \{16\text{KiB}, 256\text{KiB}\}$

Timing results are summarised in Table 3.6, and memory use in Table 3.7.

3.2.4 Storage/flash memory

Storage use does not depend on the scenario, and is listed in Table 3.8.

cipher.implementation	setup (s)	transfer (s)	total (s)
nocrypt.memcpy (minimum)	0.638	0.099	0.738
nocrypt.memcpy (median)	0.659	0.099	0.758
nocrypt.memcpy (mean)	0.666	0.099	0.765
nocrypt.memcpy (stdev)	0.035	0.000	0.035
aesgcm128.mbedtls	+0.010	+0.001	+0.012
chachapoly.rhys	+0.005	+0.001	+0.006
ascon128v12.ref	+0.011	+0.001	+0.012
ascon128v12.rhys	+0.004	+0.001	+0.005
elephant160v1.ref	+0.194	+0.301	+0.495
elephant160v1.rhys	+0.016	+0.025	+0.041
giftcofb128v1.ref	+0.016	+0.010	+0.025
giftcofb128v1.rhys	+0.005	+0.000	+0.005
grain128aead.rhys	+0.005	+0.001	+0.006
isapk128av20.ref	+0.053	+0.033	+0.086
isapk128av20.rhys	+0.014	+0.007	+0.021
photonbeetleaead128rate128v1.ref	+0.078	+0.096	+0.174
photonbeetleaead128rate128v1.rhys	+0.005	+0.005	+0.010
romulusn1.ref	+0.021	+0.027	+0.048
romulusn1.rhys	- 0.007	+0.003	- 0.003
tinyjambu128.rhys	+0.005	+0.001	+0.006
xoodyakv1.ref	+0.012	+0.003	+0.015
xoodyakv1.rhys	+0.007	+0.000	+0.007

Table 3.4: (relative) time per session for scenario 2: Weather station, median of 20.

cipher.implementation	stack (B)	heap (B)
nocrypt.memcpy	1856	43685
chachapoly.rhys	+252	+16
ascon128v12.ref	+0	+16
ascon128v12.rhys	+32	+16
elephant160v1.ref	+184	+8
elephant160v1.rhys	+128	+8
giftcofb128v1.ref	+16	+16
giftcofb128v1.rhys	+400	+16
grain128aead.rhys	+72	+8
isapk128av20.ref	+240	+16
isapk128av20.rhys	+128	+16
photonbeetleaead128rate128v1.ref	+128	+16
photonbeetleaead128rate128v1.rhys	+136	+16
romulusn1.ref	+304	+16
romulusn1.rhys	+508	+16
tinyjambu128.rhys	+0	+8
xoodyakv1.ref	+160	+16
xoodyakv1.rhys	+72	+16

Table 3.5: (relative) maximum heap and stack memory use for individual en/de-crypt operations of scenario 2: Weather station.

cipher.implementation	16KiB↑	16KiB↓	256KiB↑	256KiB↓
nocrypt.memcpy (minimum)	4.005	3.597	14.518	6.076
nocrypt.memcpy (median)	4.163	3.979	14.671	6.087
nocrypt.memcpy (mean)	4.151	4.018	15.006	6.148
nocrypt.memcpy (stdev)	0.123	0.158	3.245	0.179
aesgcm128.mbedtls	+0.236	+0.425	+0.717	+1.086
chachapoly.rhys	+0.211	+0.095	+0.696	+1.445
ascon128v12.ref	+0.236	+0.487	+0.907	+0.823
ascon128v12.rhys	+0.211	+0.091	+0.775	+0.713
elephant160v1.ref	+18.038	+18.344	+273.378	+273.256
elephant160v1.rhys	+1.466	+1.405	+22.153	+22.073
giftcofb128v1.ref	+1.110	+1.453	+9.495	+9.460
giftcofb128v1.rhys	+0.463	+0.521	+0.738	+0.809
grain128aead.rhys	+0.255	+0.352	+1.700	+1.796
isapk128av20.ref	+1.485	+1.764	+17.311	+17.723
isapk128av20.rhys	+0.712	+0.598	+4.447	+4.464
photonbeetleaead128rate128v1.ref	+6.235	+6.552	+88.036	+88.802
photonbeetleaead128rate128v1.rhys	+0.303	+0.305	+4.755	+4.705
romulusn1.ref	+2.248	+2.511	+26.352	+26.642
romulusn1.rhys	+0.415	+0.359	+3.253	+3.179
tinyjambu128.rhys	+0.481	+0.311	+1.124	+1.020
xoodyakv1.ref	+0.483	+0.584	+2.289	+2.217
xoodyakv1.rhys	+0.470	+0.277	+0.721	+0.598

Table 3.6: (relative) total time per session for scenarios 3. 16KiB↑: file upload (16KiB), median of 20; 16KiB↓: file download (16KiB), median of 20; 256KiB↑: file upload (256KiB), median of 10; 256KiB↓: file download (256KiB), median of 10. All in seconds.

	16KiB↑		16KiB↓		256KiB↑		256KiB↓	
cipher.implementation	stack	heap	stack	heap	stack	heap	stack	heap
nocrypt.memcpy	1872	64745	1872	64774	1872	64778	1872	64778
chachapoly.rhys	+288	+16	+288	+16	+288	+16	+288	+16
ascon128v12.ref	+0	+16	+0	+16	+0	+16	+0	+16
ascon128v12.rhys	+56	+16	+64	+16	+56	+16	+64	+16
elephant160v1.ref	+184	+8	+192	+8	+192	+8	+192	+8
elephant160v1.rhys	+104	+8	+136	+8	+128	+8	+136	+8
giftcofb128v1.ref	+16	+16	+16	+16	+16	+16	+16	+16
giftcofb128v1.rhys	+400	+16	+400	+16	+400	+16	+400	+16
grain128aead.rhys	+88	+8	+104	+8	+88	+8	+104	+8
isapk128av20.ref	+248	+16	+248	+16	+248	+16	+248	+16
isapk128av20.rhys	+128	+16	+128	+16	+128	+16	+128	+16
photonbeetleaead128rate128v1.ref	+128	+16	+128	+16380	+128	+16	+128	+16384
photonbeetleaead128rate128v1.rhys	+136	+16	+144	+16	+136	+16	+144	+16
romulusn1.ref	+304	+16	+304	+16	+304	+16	+304	+16
romulusn1.rhys	+508	+16	+508	+16	+508	+16	+508	+16
tinyjambu128.rhys	+8	+8	+8	+8	+8	+8	+8	+8
xoodyakv1.ref	+184	+16	+184	+16	+184	+16	+184	+16
xoodyakv1.rhys	+72	+16	+72	+16	+72	+16	+72	+16

Table 3.7: (relative) maximum stack memory use for individual en/de-crypt operations of scenarios 3. 16KiB↑: file upload (16KiB); 16KiB↓: file download (16KiB); 256KiB↑: file upload (256KiB); 256KiB↓: file download (256KiB). All in Bytes

cipher.implementation	size (B)
nocrypt.memcpy	330750
chachapoly.rhys	+2155
ascon128v12.ref	nan
ascon128v12.rhys	+1696
elephant160v1.ref	+2542
elephant160v1.rhys	+3944
giftcofb128v1.ref	+1426
giftcofb128v1.rhys	+7820
grain128aead.ref	+2660
grain128aead.rhys	+2592
isapk128av20.ref	+1861
isapk128av20.rhys	+1495
photonbeetleaead128rate128v1.ref	+1861
photonbeetleaead128rate128v1.rhys	+4408
romulusn1.ref	+3326
romulusn1.rhys	+8572
tinyjambu128.ref	+870
tinyjambu128.rhys	+2006
xoodyakv1.ref	+2061
xoodyakv1.rhys	+1338

Table 3.8: (relative) size of the compiled application binary.

3.3 Noise

Variable network latency threatens to make our results irreproducible and incomparable. Variation between test setups is accounted for by normalising results relative to some baseline. We aggregate results of multiple (sequential) tests to reduce the effect of short-term (in the order of minutes) variability, but we have made no such provisions for long-term (in the order of hours or days) variability. In this section we will assess the level of short- and long-term variability in our results. We assume variability measured in a test without encryption is at least as large as the equivalent with any encryption algorithm.

Table 3.9 shows variability of performance measurements for nocrypt.memcpy, characterising variability in the overall setup. While short-term variability (represented by mean of standard deviations) tends to be "greater" than long-term variability (represented by standard deviation of means), long-term variability is still a concern to comparability of results. The high standard deviation(s) for scenario 3a are in large part explained by a single outlier $\sim$ 12s above the median (excluding outliers) of $\sim$ 4s, as shown in Figure 3.1. This is why we chose to use the best (minimum) performance as our baseline rather than the mean; as Table 3.9 shows, the minimum is more stable than the mean and (by some metrics) even the median. This is justified by observing latency can always increase, but cannot arbitrarily decrease.

scenario	n	total stdev	mean stdev	stdev mean	stdev median	stdev minimum
1: wearable	19	0.00596	0.00376	0.00462	0.00480	0.00366
2: w.station	20	0.01369	0.00781	0.00299	0.00082	0.00299
3a: 16KiB↑	20	0.57626	0.19696	0.12881	0.05627	0.02367
3b: 16KiB↓	20	0.00367	0.00344	0.00089	0.00069	0.00128
3c: 256KiB↑	10	0.07694	0.06821	0.03302	0.06171	0.03003
3d: 256KiB↓	10	0.08242	0.05309	0.02190	0.00172	0.00157

Table 3.9: variability for **n** batches at 1-hour intervals of **n** runs of scenarios with nocrypt.memcpy. 'total' means aggregated over all runs of all batches; otherwise the second term refers to in-batch aggregation, the first to between-batch aggregation. stdev is $n-1$ corrected sample standard deviation. All in seconds.

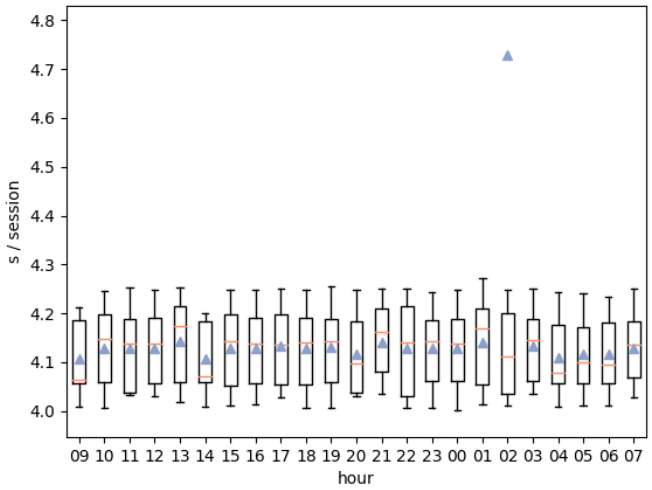

Figure 3.1: Boxplot of 23 batches at 1-hour intervals of 20 runs of scenario 3a: 16KiB↑ with nocrypt.memcpy. Triangles are means. A single 16.37s result at 02:00 has been cropped out to improve readability.

3.4 Discussion

While our intention is not to compare algorithms, in this section we will look at performance of several of Rhys Weatherley's general optimised C implementations [Wea] side-by-side. Figure 3.2 shows performance in seconds per session for the evaluated scenarios (chachapoly (ChaCha20-Poly1305) is not an LWC submission, but a standard algorithm for TLS 1.2+ included for reference.). elephant160v1.rhys (Elephant-Spongent-π[160] aka Dumbo) performs particularly poorly, but it should be noted this variant of elephant was designed for hardware implementations. In Figure 3.3 we show the same data, but relative to the baseline and excluding elephant to improve legibility. We can see GIFT-COFB and TinyJambu outperforming chacha on the smaller wearable and weather station tasks, while Ascon and Xoodyak do better in the larger file-transfer scenarios. Figure 3.4 shows relative maximum memory usage for the same ciphers. Again Ascon and Xoodyak do quite well, only beaten by the truly tiny TinyJambu, but besides GIFT-COFB and Romulus, all ciphers show better results than chachapoly on this front. Storage (Flash memory) requirements are shown in Figure 3.5. ISAP manages to do slightly better than Ascon in this aspect, with Ascon, ISAP, TinyJambu and Xoodyak requiring less storage than ChaChaPoly, followed closely by Grain.

Of course, these figures do not consider the cryptographic security of the algorithms, which might well make the slower ciphers preferable over the fastest ones. Perhaps more useful then would be comparing different implementations of the same algorithm. At present we have only benchmarked reference implementations and Weatherley's optimised versions, making such comparison a rather insipid exercise. Figure 3.6, for example, compares performance of various implementations of ascon128, with chachapoly for reference. As expected, the optimised implementation slightly outperforms the reference implementation in most scenarios. the reference implementation seems to have a .07s advantage in the decryption-heavy 16KiB↓ scenario, but this difference is insignificant given the levels of variability described in section 3.3. In fact, looking back at Tables 3.2, 3.4 and 3.6, you will see that no unoptimised implementation significantly outperforms their optimised counterpart.

Besides performance and security, existing infrastructure is an important criterion for cipher adoption. In the experience of Aumasson & Vennard [AV19], AES is usually lightweight enough, in part thanks to widespread hardware acceleration [TP15]. According to Dan Shumow, speaking for Microsoft Research [Shu15], current cryptographic standards (read: AES) are not a limit on IoT performance. IoT protocols have already been deployed with existing standards, making potential adoption of additional lightweight algorithms a slow and arduous process. The real use case for lightweight cryptography, Shumow says, is in hardware implementations, like for RFID tags [RW15]. In this light, we should not overestimate the real-world relevance of our results. While the evaluated scenarios may reflect real use cases, this does not imply lightweight algorithms would actually be deployed in these scenarios. Nevertheless, in a heterogeneous network, where different types of devices communicate with each other, the constrained processors we consider in this thesis might need to communicate with devices that have dedicated hardware support. In this kind of setup, our results do have real-world relevance, since the lightweight algorithms will have to be implemented on constrained processors.

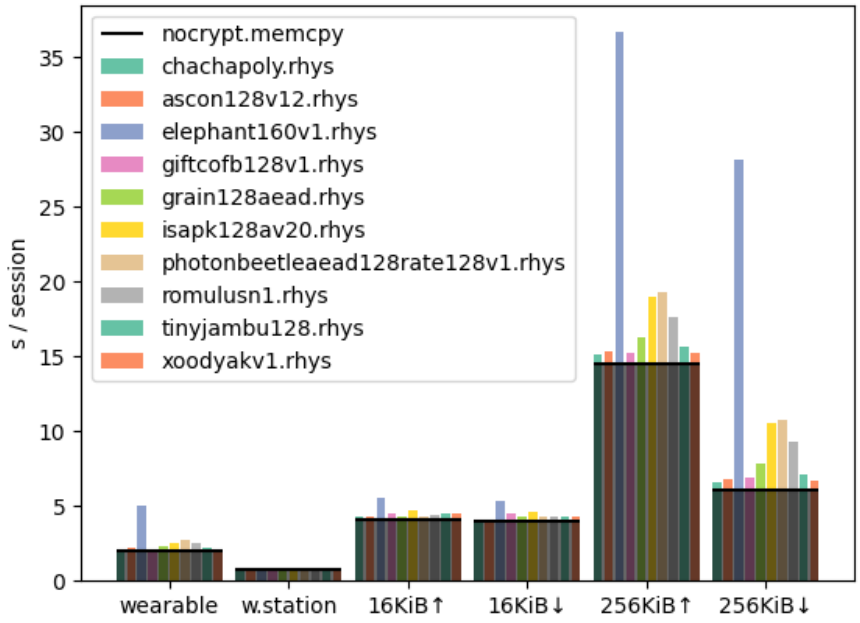

Figure 3.2: Performance of optimised implementations in various scenarios

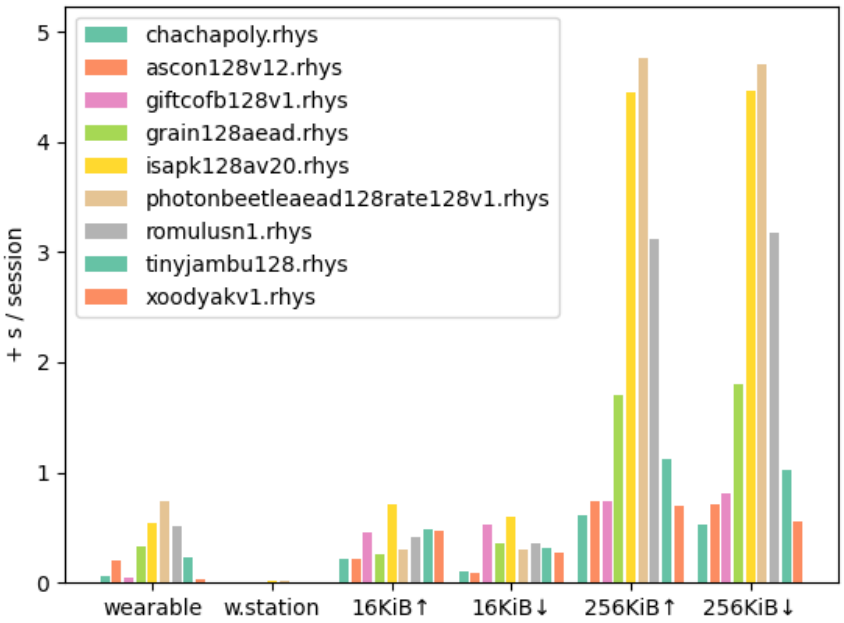

Figure 3.3: Performance of optimised implementations in various scenarios, relative to baseline.

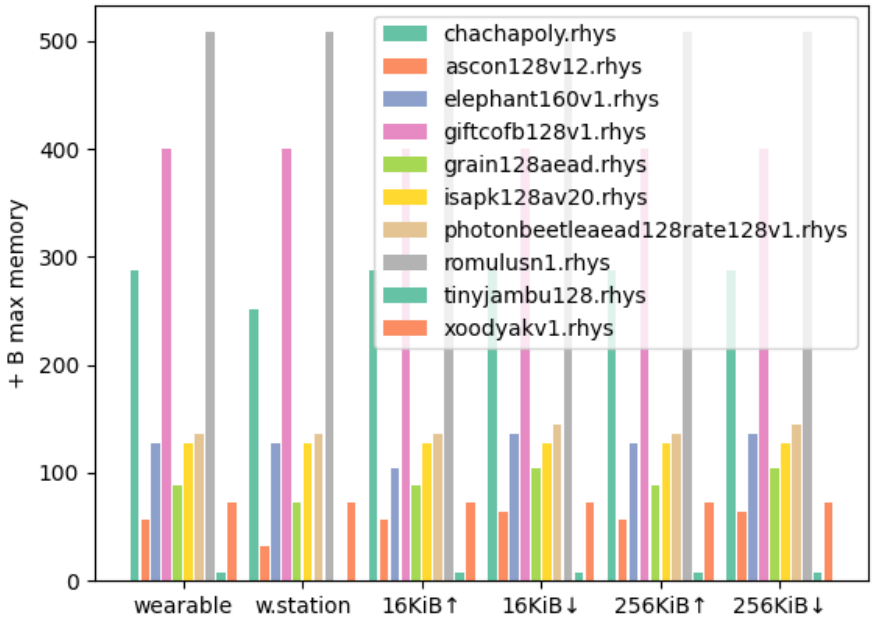

Figure 3.4: Memory usage of optimised implementations in various scenarios, relative to baseline.

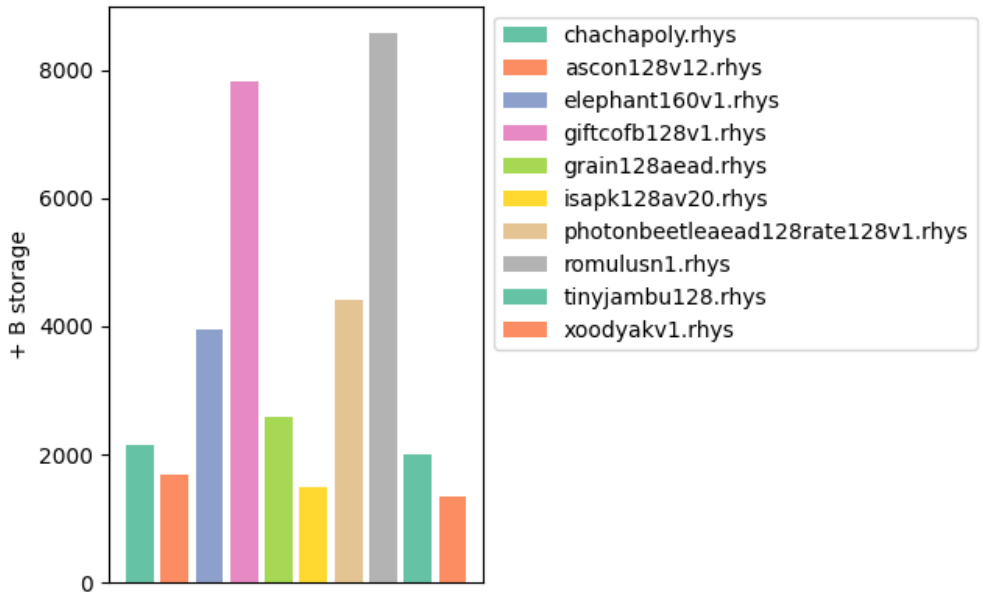

Figure 3.5: Storage requirements of optimised implementations, relative to baseline.

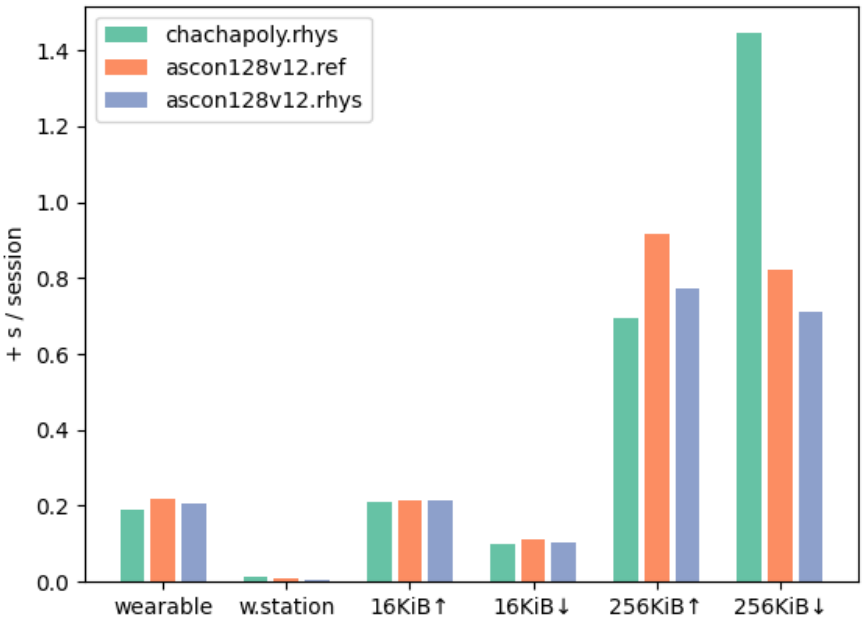

Figure 3.6: Performance of ascon implementations and chachapoly, relative to baseline.

3.5 Relation to other benchmarks

In Figure 3.7 we plot our median total timings per cipher[5] for each scenario against (fairly arbitrarily chosen) NISTs measurements of 128B plaintext + 128B additional data encryption on the Arduino Nano 33 BLE with `-O2` optimization[6]. The figure is a bit hard to read, but the main takeaway is that there could reasonably be a linear relation and thus the corresponding r^2 values listed in the first row of Table 3.10 are valid. Looking at those r^2 values, it would seem generic performance generalises reasonably well to the TLS setting, at least for implementations not specifically optimised for that setting. The remaining $\sim 20\%$ of variability would be partly explained by hardware differences and partly by the TLS setting. If we once again exclude elephant as an outlier, however, we get a different picture: isolated performance almost perfectly predicts performance in our TLS setup, suggesting NIST's measurements are just as representative of TLS performance as ours (at least for ciphers not specifically optimised for this setting).

$r^2_{ours,nists}$	wearable	w.station	16KiB↑	16KiB↓	256KiB↑	256KiB↓
incl. elephant	0.806801	0.816520	0.816922	0.818307	0.806321	0.808782
excl. elephant	0.999752	0.921680	0.996537	0.994744	0.999821	0.999851

Table 3.10: Coefficients of determination between our results and those of NIST, including and excluding elephant.

[5] only including ciphers which have matching versions in NISTs and our results, namely aceae128v1.rhys, ascon128v12.ref, ascon128v12.rhys, giftcofb128v1.ref, isapk128av20.ref, elephant160v1.ref, elephant160v1.rhys and photonbeetleaead128rate128v1.ref.

[6] as listed at github.com/usnistgov/Lightweight-Cryptography-Benchmarking

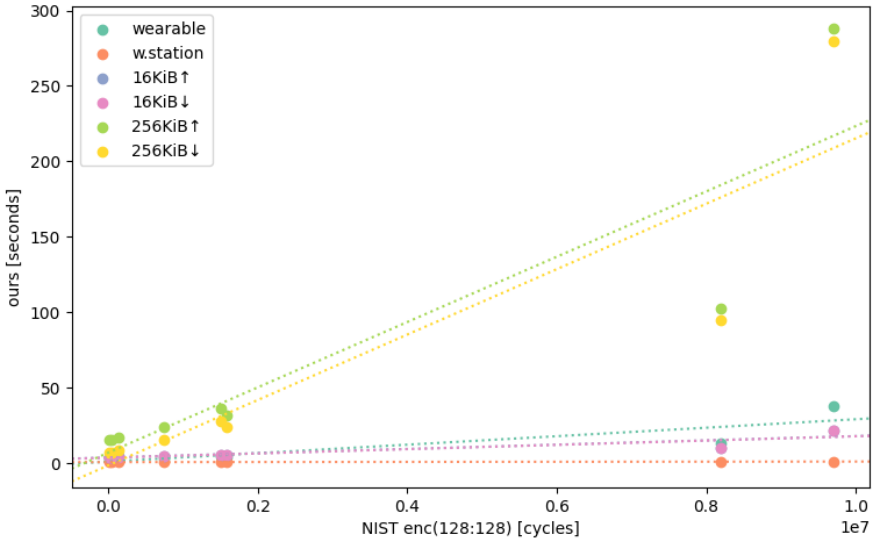

Figure 3.7: Our results / NIST's results. Lines are ordinary least squares lines. The Rightmost points are elephant160v1.rhys.

Chapter 4

Conclusions and Further Research

In this thesis, we explored performance characteristics of software implementations of NIST Lightweight Cryptography standardisation candidates. By focusing on a number of TLS usage scenarios and evaluating full TLS sessions over WiFi, we aimed to more closely model real use. We built a benchmarking setup around the open-source Mbed TLS library, measuring speed, memory use and storage requirements. Though drawing definitive conclusions about candidate algorithm performance at this stage would be unwise as more optimised implementations may yet emerge, we found existing implementations of the Ascon and Xoodyak ciphers already outperform the standard ChaCha20-Poly1305 algorithm on speed, storage and memory use.

Our results are limited to finalist reference implementations and Weatherley's optimised versions. In the future, this could quite easily be expanded to include other algorithms and implementations. Expanding the benchmarking setup to use different hardware would be more difficult; measurement methods rely on details of the Arduino-MbedOS platform. It might be more useful to take the modified TLS library and scenarios and integrate them into an existing multi-platform benchmarking framework, such as that of Renner et al. [RPM20].

Bibliography

[AV19] Jean-Philippe Aumasson and Antony Vennard. "Cryptography in industrial embedded systems: our experience of needs and constraints". In: NIST Lightweight Cryptography Workshop. 2019. URL: https://csrc.nist.gov/Presentations/2019/cryptography-in-industrial-embedded-systems.

[Cam+] Fabio Campos et al. *RISC-V Benchmarking*. URL: https://github.com/AsmOptC-RiscV/Assembly-Optimized-C-RiscV.

[eBACS] Daniel J. Bernstein and Tanja Lange (editors). *eBACS (ECRYPT Benchmarking of Cryptographic Systems): General-purpose Processor (Intel, AMD, ARM Cortex-A, Qualcomm) Benchmarking*. URL: https://bench.cr.yp.to/results-nistlwc-aead.html.

[LWC] NIST LWC team. *Lightweight Cryptography*. URL: https://csrc.nist.gov/projects/lightweight-cryptography.

[LWC2] Meltem Sönmez Turan et al. *Status Report on the Second Round of the NIST Lightweight Cryptography Standardization Process*. Tech. rep. NIST, 2021. URL: https://csrc.nist.gov/publications/detail/nistir/8369/final.

[LWCb] NIST LWC team. *Microcontroller Benchmarking*. URL: https://github.com/usnistgov/Lightweight-Cryptography-Benchmarking.

[LWCc18] NIST LWC team. *Submission Requirements and Evaluation Criteria for the Lightweight Cryptography Standardization Process*. 2018. URL: https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/final-lwc-submission-requirements-august2018.pdf.

[Mbed] The Mbed TLS Contributors. *Mbed TLS*. URL: https://tls.mbed.org/.

[NIS15] NIST, ed. *NIST Lightweight Cryptography Workshop*. 2015. URL: https://www.nist.gov/news-events/events/2015/07/lightweight-cryptography-workshop-2015.

[Rou+14] Catherine Roussey et al. "Short Paper: Weather Station Data Publication at IRSTEA: An Implementation Report". In: *Proceedings of the Joint Proceedings of the 6th International Workshop on the Foundations, Technologies and Applications of the Geospatial Web and 7th International Workshop on Semantic Sensor Networks (TC-SSN)* (Riva del Garda, Trentino, Italy, Oct. 20, 2014). Ed. by Kostis Kyzirakos et al. CEUR Workshop Proceedings 1401. Aachen, 2014, pp. 89–104. URL: http://ceur-ws.org/Vol-1401/#paper-07.

[RPM] Sebastian Renner, Enrico Pozzobon, and Jürgen Mottok. *AVR/ARM/RISC-V Microcontroller Benchmarking*. URL: https://lwc.las3.de/.

[RPM20] Sebastian Renner, Enrico Pozzobon, and Jürgen Mottok. "A Hardware in the Loop Benchmark Suite to Evaluate NIST LWC Ciphers on Microcontrollers". In: *Information and Communications*

Security. Ed. by Weizhi Meng et al. Springer International Publishing, 2020, pp. 495–509. ISBN: 978-3-030-61078-4. URL: https://lwc.las3.de/paper.pdf.

[RW15] M.J.B. Robshaw and T. Williamson. "RAIN RFID and the Internet of Things: Industry Snapshot and Security Needs". In: *NIST Lightweight Cryptography Workshop*. Ed. by NIST. 2015. URL: https://csrc.nist.gov/csrc/media/events/lightweight-cryptography-workshop-2015/documents/papers/session4-robshaw-paper.pdf.

[Shu15] Dan Shumow. "Thanks, But No Thanks. Current Cryptographic Standards Are Sufficient for Software". In: *NIST Lightweight Cryptography Workshop*. Ed. by NIST. 2015. URL: https://csrc.nist.gov/csrc/media/events/lightweight-cryptography-workshop-2015/documents/presentations/session4-shumow.pdf.

[TP15] Hannes Tschofenig and Manuel Pegourie-Gonnard. "Performance of State-of-the-Art Cryptography on ARM-based Microprocessors". In: *NIST Lightweight Cryptography Workshop*. Ed. by NIST. 2015. URL: https://csrc.nist.gov/csrc/media/events/lightweight-cryptography-workshop-2015/documents/presentations/session7-vincent.pdf.

[Wea] Rhys Weatherley. *AVR/ARM Microcontroller Benchmarking*. URL: https://rweather.github.io/lightweight-crypto.

[Win+19] Jori Winderickx et al. "Communication and Security Trade-Offs for Wearable Medical Sensor Systems in Hospitals: Work-in-Progress". In: *Proceedings of the International Conference on Embedded Software Companion*. EMSOFT '19. New York, New York: Association for Computing Machinery, 2019. ISBN: 9781450369244. DOI: 10.1145/3349568.3351548.

[Win20] Jori Winderickx. "Energy-efficient and secure implementations for the IoT". PhD thesis. KU Leuven, 2020. URL: https://lirias.kuleuven.be/retrieve/567362.