Towards quantifying the cost of a secure IoT: Overhead and energy consumption of ECC signatures on an ARM-based device[*]

Max Mössinger[1], Benedikt Petschkuhn[1], Johannes Bauer[1], Ralf C. Staudemeyer[1], Marcin Wójcik[2] and Henrich C. Pöhls[1]

[1]Institute of IT-Security and Security Law (ISL), University of Passau, Innstr. 43, 94032 Passau, Germany
[2]Computer Laboratory, University of Cambridge, 15 JJ Thomson Avenue, Cambridge CB3 0FD, UK
[1]hp|rcs@sec.uni-passau.de
[2]marcin.wojcik@cl.cam.ac.uk

Abstract

In this paper, we document the overhead in terms of runtime, firmware size, communication and energy consumption for Elliptic Curve Cryptography (ECC) signatures of modern ARM-based constrained devices. The experiments we have undertaken show that the cryptographic capabilities of the investigated Zolertia Re-Mote based on a TI's CC2538 chipset running Contiki OS is indeed suitable for the Internet-of-Things (IoT): Computing a signature using a curve with a 192-bit key length adds an additional runtime of roughly 200 ms. However, we found that in comparison to sending an unsigned message approximately two-thirds of the runtime overhead is spent on cryptographic operations, while sending the signed message accounts for the remainder.

We give real measurements which can be used as a basis for analytical models. Our measurements show that the saving gained by using curves with lower security levels (i. e., 160-bit key length) is not worth the sacrifice in protection. While signatures add non-negligible overhead, we still think that the additional 200 ms (signing with `secp192r`) is worth consideration. This paper gives an indication of the true costs of cryptographically protected message integrity which is greater or equal to the cost of encryption. We show what needs to be spent in order to verify the origin of the data in the application, since in the IoT it will have travelled through many 'things'.

[*]This is a pre-print.

1 Introduction

Deploying cryptographically strong security on constrained devices used as the underlying platforms in the Internet of Things (IoT) was a long outstanding quest. Still—according to a report from HP in 2015—this critical protection of last mile communication remains an open issue [1]. The tools that can make the significant impact on this issue are evolving constantly: Elliptic Curve Cryptography (ECC) is known to be more resource efficient and is currently seeing an update of standards that include new designs. These will aid in security, as well as in interoperability.

Initially, ECC was presented independently by *Miller* [2] and *Koblitz* [3] in 1986. More recently, ECC has become the standard tool for strong cryptographic algorithms in the IoT device context. Widely known is the NIST standard containing the Elliptic Curve Digital Signature Algorithm (Elliptic Curve Digital Signature Algorithm (ECDSA)) [4]. After recent criticism of ECC parameters being selected in a non-deterministic fashion, thus less trusted and potentially insecure [5], new curves have been standardized [6], like Curve25519 proposed by *Bernstein*. Based on Curve25519 a signature algorithm named Ed25519 is currently en route to standardization [7].

The question we strive to answer here is: *What is the real overhead for ECC signatures on constrained devices?*[1]

Many implementations of ECC exist, which are well-suited for Wireless Sensor Networks (WSN), like TinyECC [8], NanoECC [9], or NIST's ECClight [10]. ECC on constrained devices has been further evaluated and optimised in [11]. That ECC can be very efficient, such as the curve with 160-bit key length implementation, is shown by *Kern* and *Feldhofer* [12]. A very lightweight ECC-based construction for authentication to run on RFID-type devices was presented by *Braun*, *Hess* and *Mayer* [13]. However, hardware has evolved and many of the IoT devices already use ARM-based System on Chip (SoC) [14][2].

In our experiments, we evaluate the overhead in terms of runtime, code footprint, energy consumption, and communication of signing and verifying a message's payload with ECC signatures. We start by giving some background information about the cryptography, the hardware and the tested libraries in Section 2. In Section 3 we explain the experimental setup. The evaluation of runtime overhead and code footprint is given in Section 4, and for energy consumption in Section 5. Finally, we look at the overhead of a Constrained Application Protocol (CoAP) communication of a signed message in Section 6, before we conclude in Section 7.

[1]based on research in the EU FP7 project RERUM (`ict-rerum.eu`)

[2]We imply ECC remains hardware supported, as in TI's CC2538 chipset. Recently ECC support has been dropped, but we presume the co-processor can be micro-coded to speed up new standardized curves (see: `blog.spd.gr/2015/04/to-cc-or-not-to-cc.html`).

2 Background

We investigate the overhead of a cryptographic operation called digital signature. Signatures protect the integrity of messages and allow to verify their origin. In the following, we evaluate several related cryptographic libraries.

2.1 ECC signatures

Integrity is the "property that data has not been altered or destroyed in an unauthorised manner" [15]. While integrity can be achieved on transport- and message-level, we focus on message-level integrity, since it creates an integrity check value over the actual message. This value can be verified even after the message was sent over an unsecured communication channel or stored at a non-trusted system to guarantee integrity. This is suitable to the IoT as information from sensors is gathered by constrained devices and either forwarded to other devices or stored in message queues to be picked up by applications [16].

To generate ECC-based signatures two distinct keys are required, likewise to all asymmetric based digital signature schemes. One is called private key used to generate signatures, the other one is called public key used to verify signatures. The secret key must be generated, stored, and used, in such a way that confidentiality is not violated at any time. In addition, the ECC-based signature algorithm usually involves a hash function (e. g., SHA-256 [17], or better).

In order to use signatures in practice, a key pair per device is required. In contrast to symmetric-key cryptography this increases the security, as an extracted key empowers an attacker to only impersonate that single device. For the tests conducted in our experiments, we did not address the key distribution problem.

2.2 Cryptographic libraries

From the existing sets of cryptographic libraries, we selected those, which were suitable without significant underlying changes for both: (i) running under Contiki and (ii) running on the ARM Cortex-M3 core. The following libraries were short-listed as candidates for further investigation on our Re-Mote platform: TweetNaCl [18], Piñol [19] and MicroECC [20]. In order to run the libraries on the Re-Mote we ported them to Contiki and adjusted the code whenever necessary [21].

A better diversification of results was a main intent of our selection process. Thus, we focused on selecting libraries that support a greater variety of features and standards, i. e., MicroECC implements curves across different security levels, whereas Piñol focuses on one security level but different coordinates. While this approach makes a direct comparison more challenging, the results give a better understanding of the actual ECC signature overhead on ARM-based constrained devices, being the objective of this paper.

2.2.1 TweetNaCl library

TweetNaCl [18] is a compact implementation of the NaCl [22] library. It only contains a C source file with its corresponding header file. The library is generic, but some of the design choices are suited to 32-bit architectures. In addition, the library does not perform any dynamic memory allocation and provides protection against cache-timing attacks. Although TweetNaCl includes all cryptographic primitives of the original NaCl library, we selected only the Ed25519 [23] primitive for signature evaluation.

2.2.2 Piñol library

the Piñol [19] library implements the `secp256r1` standardized NIST curve [4] using three different coordinate representations: affine, homogeneous and jacobian. The library is generic, although it features several optimisation techniques and design choices, i. e., it allows a different word size for a point representation and thus one can easily configure the library for both 16-bit and 32-bit platforms. In addition, to perform a more efficient point doubling computation the library supports a sliding window size technique.

2.2.3 MicroECC library

Similarly to Piñol, the MicroECC library implements five standard NIST curves [4]: `secp160r1`, `secp192r1`, `secp224r1`, `secp256k1`, and `secp256r1` which can be used for ECDSA. The library is implemented in C, but to increase performance, it can be adjusted to use the inline assembly feature for our target ARM architecture. The MicroECC library can optionally be optimised for either speed or code size and, similar to TweetNaCl, features a protection against cache-timing attacks.

3 Experiments

We use the Zolertia Re-Mote [14] development board as the underlying platform, which houses the CC2538 chipset—a recent SoC design by Texas Instruments. The device contains an ARM Cortex-M3 as core processor, on-chip memory modules, an IEEE 802.15.4-compliant radio transceiver module, and a co-processor capable of performing cryptographic operations. The Re-Mote is well-suited to run Contiki [24], an modern operating system specifically designed to support a wide-range of constrained devices. All our experiments run as applications on top of Contiki.

For the cryptography-only measurements, we disabled the network stack in order to eliminate unnecessary interrupts, additional energy consumption, and all other side effects, that might be imposed by the integrated radio module and related software code.

3.1 Code footprint tests setup

To investigate the code footprint of the target primitives and schemes, we used a standard *gcc-arm* toolchain available for CC2538 devices. Static code size values can be obtained from the Executable and Linkable Format (ELF) file, by using aforementioned toolchain. The ELF file represents a compiled and already linked executable and can be uploaded into the flash memory of the Re-Mote device.

The static code size of the target function was obtained as follows: *(i)* we measured a static code size of the whole project, *(ii)* all code dependencies to our target library were removed, and *(iii)* the resulting elf file was stripped and the code size measured. Finally, *(iv)* we compared the measured size to the original project size, therefore obtaining the requested code footprint.

3.2 Runtime tests setup

For measuring timings under Contiki, we developed our own tool called *timelib*, due to the lack of any suitable library. *timelib* consists of multiple useful functions to automate the measurement process and to provide standardised output, well suitable as an input to other tools. Our tool supports two modes: timing and power-trace. In timing mode, *timelib* prints the values of the internal Re-Mote timer for each measured task to standard output. Alternatively, in power-trace mode *timelib* triggers power measurement periods by activating/inactivating an external pin.

For runtime measurements, we used the internal real-time clock of the CC2538 chip, which runs at 32.768 kHz [25]. We observed each cryptographic operation over a period of 10 minutes, performing the measurements multiple times. The *timelib* tool analyses the output, aggregates the measured time of each task and computes the average over several iterations.

3.3 Energy consumption tests setup

We did tap the power supply lines going via USB to the Re-Mote device. To perform the power measurements an analog-digital-converter (AD-converter) of type MCP3008 [26] was used. The said converter is triggered by a Raspberry Pi over the Serial Peripheral Interface (SPI) and results are read back. Each measurement contains time-stamp and measured value. The python scripts and wiring diagrams, are based on the work of Erik Bartmann [27]. The MCP3008 is supplied by 3.25V and has a resolution of 1024 bits, these parameters limit the maximal resolution to 3.17mV/bit [26]. The AD converter works within the range of 2.5 ms to 10 ms per measurement. To be safe, we assume the worst resolution and set the shortest measurable task cycle to 10 ms.

To automate the process of analysing these traces, we developed another tool called *powertracer-tool.* The tool generates two kinds of traces: a time trace and a power trace. The time trace consists of the task names and their duration, whereas the power trace consists of the voltage on the measurement points and the marker positions of the glowing LED marker points. The measurement process starts by the generation of a timing trace, which is next saved as a log file. Furthermore the Raspberry Pi generates the power data, which is then analysed together with the timing trace. In the end the *powertracer-tool* syncs the information from the timing of each selected task with the power trace and thus gives energy statistics for each given task. The output of a power trace is partially shown in Figure 1.

To achieve the aforementioned synchronisation, task start/stop checkpoints are marked by activating the LEDs and its higher—clearly observable—power drain is used as a marker. Consequently, the energy consumption for each power measuring point within this interval is calculated. The interval does not contain the power measuring point for the marker. The energy consumption is calculated in mJ. Please note that the limitation of the used hardware gives a minimum timing interval of 10 ms. Afterwards the sum of all measuring points between the two markers will be used to calculate the total consumption of a given task. In a test run each task is executed several times. The measurements stated in this paper's tables are the average value of several runs.

3.4 Cryptographic libraries adjustments

No optimisation for the processor, Contiki, or otherwise was done to the libraries. In order to achieve consistent and reproducible measurements, we crippled security and all calls to Pseudo-Random Number Generator (PRNG) are replaced with a function generating a fixed output. For all curves, the used hash function is included into the measurement. We have timed the hash function explicitly for each task and found the overhead to be minimal, i.e. in the region of a few ms. Note that we fixed the length of the message to sign in our tests to 15 bytes. Since our selected cryptographic primitives heavily use the memory stack, we increased the stack size from Contiki's default value for the CC2538 chipset of 2048 bytes to 4096 bytes.

TweetNaCl: Although the source code does not target Contiki, the code integration was straightforward; with little need for adjustments. Our investigated implementation of Ed25519 uses an internal SHA-512 hash function.

Piñol: The library was originally designed for Contiki, thus no source code tweaks were required. Similarly to TweetNaCl the implementation includes a hash function, but this time the SHA-256 version. We investigated all three different coordinate systems provided by the author and two 16-bit and 32-bit word size options. For brevity we only present the 32-bit results, since this is the target architecture on the Re-Mote.

MicroECC: Here the original library is not perfectly suited to run under Contiki, thus we performed a few minor tweaks. Our investigated library version excludes a hashing step, therefore, we applied SHA-256 to be consistent across libraries. For each of the curves there are three possible configurations, that use different inline assembly code optimisations: no inline assembler code, inline assembler code optimised for compact size, and inline assembler code optimised for fast execution.

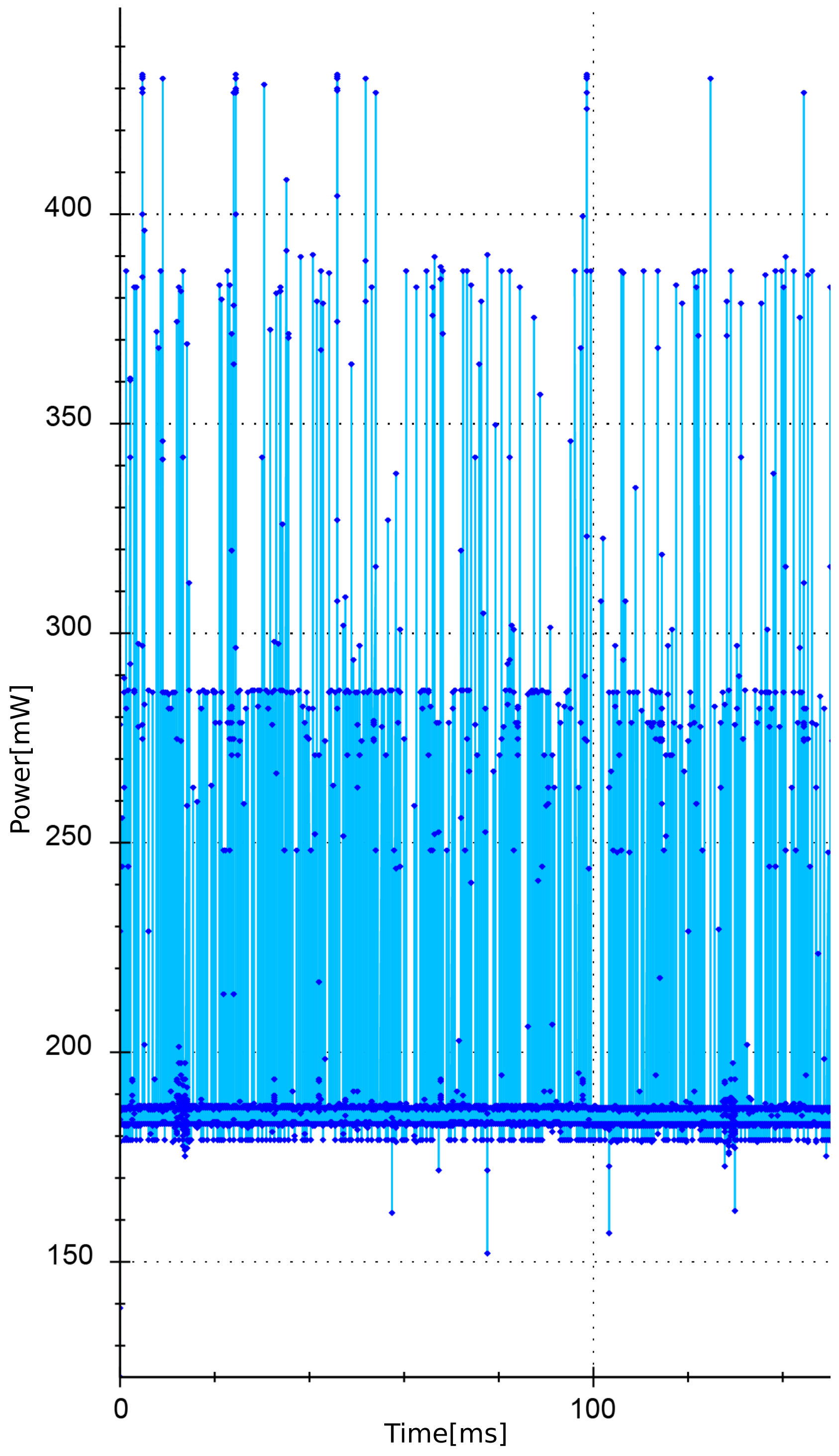

Figure 1: Power trace output for first 150 ms of the test's duration

4 Performance and Code Footprint Evaluation

In this section, we state and compare the results of time and code footprint measurements for the implementations of our TweetNaCl, Piñol and MicroECC target libraries. This includes all possible configuration options supported.

TweetNaCl: Table 1 shows the timing and the code footprint results of the TweetNaCl library. TweetNaCl accomplished the key generation in 3.518 s, signature generation in 3.532 s, and verification in 7.047 s.

Piñol: Table 2 shows the timing and the code footprint results of the Piñol library. We measured all 12 possible configurations supported by the library, i. e., 16-bit and 32-bit options with affine (af), homogeneous (hm) and Jacobian (jb) coordinates. For the purpose of the paper, we included only results for 32-bit option. We successfully replicated timing and code footprint results documented in [19].

As shown in Table 2, the best performing configuration is with the Jacobian coordinate system without sliding window and optimised for 32-bit operations. However, in all variants the timing differences in the configuration with and without sliding window are either negligible or slightly (within the range of 100 ms on average) in disfavour of the latter.

As shown in Table 2, the smallest code size is accomplished by using the affine coordinates optimised for 32-bit architecture. In this case the compactness is achieved at the expense of speed. In addition, there is no difference in the static code size of the implementations between versions with and without a sliding window. The largest static code footprint results from the implementation that achieved the best performance results. The difference between the smallest and the largest code size is in the range of 1418 bytes.

MircoECC: Table 3 shows the timing and the code footprint results of the MicroECC library. We measured all possible configurations, i. e., all five curves with all possible speed versus code size optimisation (fast, standard, small). The optimisations are possible through inline assembly insertion in critical parts of the code.

Since MicroECC includes elliptic curves on different security levels, a direct comparison is misleading. Instead, we noticed differences between a generic implementation and a platform-dependent assembler-optimised version. For example, `secp256r1` without any inline assembler optimisations runs in 1.129 s, while the platform-dependent fast version took only 0.489 s on average. This is a notable advantage of 0.640 s, which is around 2.3 times faster.

As shown in Table 3, the general difference of the code size between a curve with a short key size and one with a larger key size is considerably small. There is only a difference of 168 bytes between the curve with the lowest security level (`secp160r1`) and the highest security level `secp256r1`. In addition, savings in terms of code size between the versions optimised in this regard compared to those without are of around 200 bytes.

Table 1: TweetNaCl timing and code footprint results

Configuration	KeyGen [s]	Sign [s]	Verify [s]	Code Size [bytes]
Ed25519_small	3.518	3.532	7.047	6776

Table 2: Piñol timing and code footprint results

Configuration	KeyGen [s]	Sign [s]	Verify [s]	Code Size [bytes]
secp256r1_af_32	26.644	26.420	52.977	6915
secp256r1_af_32sw	26.712	26.503	52.832	6915
secp256r1_hm_32	10.829	10.820	21.449	8153
secp256r1_hm_32sw	10.842	10.858	21.798	8153
secp256r1_jb_32	6.425	6.439	13.075	8317
secp256r1_jb_32sw	6.521	6.588	13.175	8317

5 Power consumption evaluation

Now we shift our focus to the evaluation of the energy efficiency. In this section, we present the energy consumption results of all investigated signature schemes. Similarly to timing and code size results, we perform our tests using all possible configurations available in the libraries. All values are expressed in joule.

5.0.1 TweetNaCl

Table 4 shows the measured energy consumption results of the TweetNaCl library, i. e., the signature scheme Ed25519. As it can be seen, energy consumption of `Verify` is much higher than `KeyGen` and `Sign`. This is expected since the execution time of `Verify` is longer.

5.0.2 Piñol

The measured power consumption results of the Piñol library are given in Table 5. It is not surprising, that the energy consumption of a particular primitive is strongly related to its execution time, similarly to TweetNaCl.

5.0.3 MicroECC

In Table 6 we present the results of the measured energy consumption for the MicroECC library. Similarly to TweetNaCl and Piñol the results are strongly related to execution times.

Table 3: MicroECC timing and code footprint results

Configuration	KeyGen [s]	Sign [s]	Verify [s]	Code Size [bytes]
secp160r1_fast	0.177	0.211	0.225	5301
secp160r1_std	0.318	0.356	0.386	4679
secp160r1_small	0.260	0.296	0.320	4522
secp192r1_fast	0.181	0.206	0.224	5225
secp192r1_std	0.409	0.437	0.485	4223
secp192r1_small	0.305	0.333	0.368	4042
secp224r1_fast	0.262	0.297	0.324	5825
secp224r1_std	0.593	0.630	0.700	4311
secp224r1_small	0.438	0.475	0.525	4130
secp256k1_fast	0.420	0.467	0.476	6301
secp256k1_std	0.925	0.972	1.010	4227
secp256k1_small	0.665	0.714	0.737	4050
secp256r1_fast	0.489	0.537	0.595	6605
secp256r1_std	1.129	1.177	1.320	4531
secp256r1_small	0.805	0.855	0.957	4354

Table 4: TweetNaCl energy consumption results

Configuration	KeyGen [J]	Sign [J]	Verify [J]
Ed25519_small	0.332	0.333	0.665

6 Communication of signed JSON messages

Finally, we measure a full interaction by involving a Re-Mote platform, which was able to answer to CoAP requests with signed messages. To generate ECC-based signatures we selected the `secp192r1` curve and the MicroECC library. We ran the CoAP stack that can be configured within Contiki.

For the messages exchanged using CoAP we selected the JSON Signature Scheme (JSS) [16]. The JSS header contains meta-information about the signature algorithm embedded in a JavaScript Object Notation (JSON) message as a dedicated element `jss.protected`. The signature value itself gets encoded in BASE64URL [3] and embedded as the element `jss.signature`. In contrary to JSON Web Signatures (JWS) [28], only the signature gets encoded in BASE64URL and the payload is untouched.

One sample JSON Sensor Signatures (JSS) message—containing a temper-

[3]Base64 Encoding with URL and Filename Safe Alphabet (RFC4648)

Table 5: Piñol energy consumption results

Configuration	KeyGen [J]	Sign [J]	Verify [J]
secp256r1_af_32	2.712	2.709	5.494
secp256r1_af_32sw	2.569	2.542	5.073
secp256r1_hm_32	1.050	1.048	2.112
secp256r1_hm_32sw	1.045	1.032	2.094
secp256r1_jb_32	0.632	0.638	1.276
secp256r1_jb_32sw	0.625	0.623	1.248

ature value of a sensor and a measurement ID as payload—sent by our application is depicted in Figure 2. The length of this message is 154 characters, but can vary, depending on the size of the measurement ID. The signature algorithm identifier `ES192` shows that we were using `secp192r1` in combination with `SHA256`. The same message without integrity protection, i. e., no signature (it does only contain the red marked area in Figure 2) results in a length of 39 characters. Hence, the overhead of sending a JSS message is 115 characters (bytes) or 195%.

```
{
  "jss.protected": {
    "alg":"ES192"
  },
  "chip_temp":20953,
  "measurement_id":34,
  "jss.signature":"fgj•••lR0JMj"
}
```

Figure 2: Sample JSS message (above ••• indicates removed characters; spaces were added for better readability)

Before the message was signed, it was hashed with the built-in, hardware-accelerated SHA-256 function of the Re-Mote's CC2538 chip. Afterwards the generated signature was encoded in BASE64URL and put into the JSS message. The runtime overhead of the signing process was determined by using the integrated timer of the CC2538 chip. These measurements were performed multiple times for each step of the signing process. It turned out that the SHA-256 hashing and the BASE64URL encoding had a negligible affect on the runtime: The total duration of hashing and encoding was less than 1 ms. The average runtime was 218 ms for generating a signature using the `secp192r1` curve implementation from MicroECC (as shown in Table 7).

The code size overhead results for ECC-based signatures on the Re-Mote using the MicroECC library, the BASE64URL encoding algorithm, and additional

Table 6: MicroECC energy consumption results

Configuration	KeyGen [J]	Sign [J]	Verify [J]
secp160r1_fast	0.016	0.020	0.021
secp160r1_none	0.030	0.033	0.036
secp160r1_small	0.024	0.027	0.030
secp192r1_fast	0.016	0.019	0.020
secp192r1_none	0.037	0.040	0.044
secp192r1_none	0.028	0.031	0.034
secp224r1_fast	0.024	0.027	0.029
secp224r1_none	0.055	0.058	0.065
secp224r1_small	0.041	0.044	0.049
secp256k1_fast	0.038	0.042	0.043
secp256k1_none	0.085	0.089	0.093
secp256k1_small	0.064	0.069	0.071
secp256r1_fast	0.044	0.048	0.054
secp256r1_none	0.105	0.109	0.123
secp256r1_small	0.075	0.080	0.089

variables like the signing buffer are shown in Table 7.

To measure the code footprint the previously described setup was used. A verification with the *gcc-arm-none-eabi-objdump* tool confirmed that the MicroECC and BASE64URL libraries were not linked into the firmware image when measuring the code footprint of the image with disabled signatures. Comparing the sizes of both images showed a total code size overhead of 5.89 kb.

Finally, we compared the timing overhead for requesting a signed JSS message and an unsigned JSON message. The request for an unsigned JSON message was answered by the application with one COAP message, whereas the same message as signed JSS was transmitted in three COAP blocks. The timing overhead for transmitting messages is also shown in Table 7. Note that

Table 7: TIMING AND CODE FOOTPRINT OVERHEAD FOR SENDING JSS MESSAGES WITH `SECP192r` AND MICROECC

JSS	Signing [ms]	Total Transm. [ms]	Overhead Transm. [%]	Code Size [kb]	Overhead Code [%]
disabled	-	92	-	79.58	-
enabled	218	361	292	85.47	7.4

the application doesn't answer each request with a new signature: reading the current sensor value and signing it is done once in an interval of 30 seconds and then stored to an internal buffer. Consequently, the overhead for transmitting a JSS message is almost 300% compared to an unsigned JSON message due to the increased size of COAP messages (blocks).

7 Conclusion and future work

In this paper, we document the overhead in terms of runtime, firmware size, communication, and energy consumption for ECC signatures. The experiments undertaken show that the cryptographic capabilities of modern ARM-based constrained devices (Zolertia Re-Mote based on TI's CC2538) running a modern OS (Contiki OS) are suitable for a *more secure* Internet-of-Things (IoT): Computing a signature using a curve with a 192-bit key length adds an additional runtime of roughly 200 ms. However, in comparison to sending an unsigned message, which would fit in one COAP message, we find that roughly two-third of the runtime overhead is spent on cryptographic operations, and one third is spent on sending the signed message. For the implementation done by Piñol, which also targets Contiki, we replicated the results on timing and extended them by providing real energy consumption measurements complementing the estimations provided in said work. Note, that the real measurements we present can be used as a basis for analytical models.

Our measurements further reveal that the saving gained while using curves with lower security levels (i. e., using a 160-bit key length) is not worth the sacrifice in protection. While signatures add a non-negligible overhead, we still think that the additional 200 ms (signing with `secp192r`) is worth consideration. This paper gives an indication of the true costs to gain cryptographically protected message integrity. This can be seen as an upper bound to the cost for message encryption to protect confidentiality. We show what resources need to be spent in order to verify the origin of the data, in the light that in the IoT this data will have travelled through many 'things'.

It is worth noting that some of the cryptographic primitives that we investigated (i. e., NIST ECCs) are suitable for a hardware acceleration using a cryptographic co-processor available in some ARM-based chipsets, like TI's CC2538 System-on-Chip. Therefore we expect we expect further improvements in terms of computational overhead and memory footprint. However, facilitating hardware acceleration is out of the scope of this paper. Finally, our results hint that it would be worthwhile to investigate if the increase in message size for ECC-based signatures and the induced communication overhead can be reduced by using signatures offering message recovery, e. g., [29, 30, 31].

Acknowledgment

We would like to thank Andrew Moore and George Oikonomou for their invaluable help during the course of this work. J. Bauer, B. Petschkuhn, H. C. Pöhls, R. C. Staudemeyer, M. Wójcik were supported by the European Unions 7th Framework Programme (FP7) under grant agreement no 609094 (RERUM). H. C. Pöhls was also partly supported by the European Unions Horizon 2020 Programme under grant agreement no 644962 (PRISMACLOUD). M. Wójcik has also partly received funding from the European Union's Horizon 2020 research and innovation programme 2014-2018 under grant agreement No. 644866. This paper reflects only the authors' views and the European Commission is not responsible for any use that may be made of the information it contains.

References

[1] Hewlett Packard Enterprise, "Internet of Things research study," Tech. Rep. July, nov 2015.

[2] V. Miller, "Use of elliptic curves in cryptography," in *Proc. of Advances in Cryptology (CRYPTO85).* Springer, 1986, pp. 417–426.

[3] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of Computation*, vol. 48, no. 177, pp. 203–203, jan 1987.

[4] Federal Information Processing Standards Publication, "Digital Signature Standard (DSS)," Gaithersburg, MD, Tech. Rep., jul 2013.

[5] D. J. Bernstein, T. Chou, C. Chuengsatiansup, A. Hülsing, E. Lambooij, T. Lange, R. Niederhagen, and C. van Vredendaal, "How to manipulate curve standards: a white paper for the Black Hat," in *IACR Cryptology ePrint Archive*, 2015, vol. 2014, pp. 109–139.

[6] A. Langley, M. Hamburg, and S. Turner, "Elliptic curves for security," IRTF RFC 7748, Tech. Rep. 7748, jan 2016.

[7] N. Moeller and S. Josefsson, "IETF Draft: EdDSA and Ed25519," 2015.

[8] A. Liu and P. Ning, "TinyECC: a configurable library for elliptic curve cryptography in wireless sensor networks," in *Int. Conf. on Information Processing in Sensor Networks (ipsn 2008)*, apr 2008, pp. 245–256.

[9] P. Szczechowiak, L. B. Oliveira, M. Scott, M. Collier, and R. Dahab, "NanoECC: testing the limits of elliptic curve cryptography in sensor networks," in *Wireless Sensor Networks*, Berlin, Heidelberg, 2008, pp. 305–320.

[10] National Institute of Standards and Technology (NIST), "ecc-light-certificate library," 2014. [Online]. Available: https://github.com/nist-emntg/ecc-light-certificate

[11] J. Ayuso, L. Marin, A. Jara, and A. F. G. Skarmeta, "Optimization of Public Key Cryptography (RSA and ECC) for 16-bits Devices based on 6LoWPAN," in *1st Int. Workshop on the Security of the Internet of Things*, Tokyo, Japan, 2010.

[12] T. Kern and M. Feldhofer, "Low-resource ECDSA implementation for passive RFID tags," in *17th IEEE Int. Conf. on Electronics, Circuits, and Systems (ICECS'10)*, 2010, pp. 1236–1239.

[13] M. Braun, E. Hess, and B. Meyer, "Using elliptic curves on RFID tags," *International Journal of Computer Science and Network Security*, vol. 2, pp. 1–9, 2008.

[14] Zolertia, "Re-MOTE," 2015. [Online]. Available: http://zolertia.io/product/hardware/re-mote

[15] ISO/IEC, "ISO-IEC 7498-2: Information processing systems Open Systems Interconnection Basic Reference Model. Part 2: Security Architecture," Tech. Rep., 1989.

[16] H. C. Pöhls, "JSON Sensor Signatures (JSS): end-to-end integrity protection from constrained device to IoT application," in *Proc. of the Workshop on Extending Seamlessly to the Internet of Things (esIoT)*, 2015, pp. 306–312.

[17] Q. H. Dang, "Secure Hash Standard," National Institute of Standards and Technology, Gaithersburg, MD, Tech. Rep. August, jul 2015.

[18] D. J. Bernstein, B. van Gastel, W. Janssen, T. Lange, P. Schwabe, and S. Smetsers, "TweetNaCl: A Crypto Library in 100 Tweets," in *Progress in Cryptology–LATINCRYPT 2014*, 2015, pp. 64–83.

[19] O. Piñol Piñol, "Implementation and evaluation of BSD Elliptic Curve Cryptography," Master thesis (pre-Bologna period), Universitat Politècnica de Catalunya, nov 2014.

[20] K. MacKay, "Micro-ecc," 2016. [Online]. Available: http://kmackay.ca/micro-ecc/

[21] M. Mössinger, "Measurement of Elliptic Curve Cryptography implementations for Contiki on a Re-MOTE," M.Sc. thesis at University of Passau, Germany, 2016.

[22] D. J. Bernstein, T. Lange, and P. Schwabe, "The security impact of a new cryptographic library," in *Progress in Cryptology–LATINCRYPT 2012*, 2012, pp. 159–176.

[23] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, "High-speed high-security signatures," *Journal of Cryptographic Engineering*, vol. 2, no. 2, pp. 77–89, 2012.

[24] A. Dunkels and E. al., “The Contiki operating system,” 2016. [Online]. Available: http://www.contiki-os.org/

[25] T. Instruments. CC2538 a powerful System-On-Chip for 2.4-GHz IEEE 802.15.4 and ZigBee applications. [Online]. Available: http://www.ti.com/product/cc2538

[26] Microchip Technology Inc., “MCP3004/3008,” p. 40, 2005. [Online]. Available: http://www.adafruit.com/datasheets/MCP3008.pdf

[27] E. Bartmann, “Raspberry Pi - AddOn - Der A/D-Wandler MCP3008 v1.3,” adafruit, Tech. Rep., 2012. [Online]. Available: www.erik-bartmann.de

[28] M. Jones, J. Bradley, and N. Sakimura, “IETF draft: JSON Web Signatures (JWS),” 2015.

[29] R. R. Ramasamy and M. A. Prabakar, “Digital Signature Scheme with message recovery using Knapsack-based ECC,” *IJ Network Security*, vol. 12, no. 1, pp. 7–12, 2011.

[30] S.-F. Tzeng and M.-S. Hwang, “Digital signature with message recovery and its variants based on elliptic curve discrete logarithm problem,” *Computer Standards & Interfaces*, vol. 26, no. 2, pp. 61–71, 2004.

[31] Z.-m. Zhao, Y.-g. Wu, and F.-y. Liu, “A signature scheme with message recovery based on elliptic curves,” *Computer Engineering & Science*, vol. 2, p. 2, 2005.