ECDSA on Things: IoT Integrity Protection in Practise

Johannes Bauer[1], Ralf C. Staudemeyer[1(✉)], Henrich C. Pöhls[1], and Alexandros Fragkiadakis[2]

[1] Institute of IT-Security and Security Law (ISL), University of Passau, 94032 Passau, Germany
{rcs,hp}@sec.uni-passau.de
[2] Institute of Computer Science, Foundation for Research and Technology-Hellas, Heraklion, Crete, Greece
alfrag@ics.forth.gr

Abstract. This paper documents some experiences and lessons learned during the development of an IoT security application for the EU-funded project RERUM. The application provides sensor data with end-to-end integrity protection through elliptic curve digital signatures (ECDSA). Here, our focus is on the cost in terms of hardware, runtime and power-consumption in a real-world trials scenario. We show that providing signed sensor data has little impact on the overall power consumption. We present the experiences that we made with different ECDSA implementations. Hardware accelerated signing can further reduce the costs in terms of runtime, however, the differences were not significant. The relevant aspect in terms of hardware is memory: experiences made with MSP430 and ARM Cortex M3 based hardware platforms revealed that the limiting factor is RAM capacity. Our experiences made during the trials show that problems typical for low-power and lossy networks can be addressed by the chosen network stack of CoAP, UDP, 6LoWPAN and 802.15.4; while still being lightweight enough to drive the application on the constrained devices investigated.

1 Introduction

The Internet-of-Things will generate a plethora of communication messages, that are stored, forwarded and used by higher applications to make decisions. To provide adequate protection in case of a cyber attack, strong security mechanisms must be in place and enabled at an early stage. Here we focus on integrity protection and strong authentication through digital signatures. We aim to push security mechanisms towards the edge of the IoT, ideally running on 'things' itself, i.e. on constrained devices. In this paper, we share the lessons learned within the European and national research projects we participated in[1]. The RERUM framework [1] allows to build IoT applications while considering security and privacy mechanisms early in their design phase.

[1] EU-funded project RERUM ict-rerum.eu (last accessed 03 Oct 2016).
BMBF-funded project FORSEC bayforsec.de (last accessed 03 Oct 2016).

© Springer International Publishing AG 2016
K.-Y. Lam et al. (Eds.): ICICS 2016, LNCS 9977, pp. 3–17, 2016.
DOI: 10.1007/978-3-319-50011-9_1

Attacks we address and mitigate concern the manipulation of IoT data in transit and the spoofing of an authorised message origin. Digital signature's offering of strong origin authentication primarily protects against attackers inserting messages. Inserted messages can have unwanted consequences, like sending an 'open window' command or falsifying a 'lock door' command. Integrity protection paired with the knowledge of the sensor-data source, allows us to make better decisions on the reputation of data based on its authenticated origin.

All in all integrity alone would help against malicious tampering and could be achieved by symmetric MACs, but only with digital signatures we also reach strong authentication of the data's or command's source. By this the IoT's last mile is not any more the weakest link, but the attacker must now be able to compromise back-ends and control systems, which we already have good defence mechanisms for. Thus, showing that we can put ECDSA on things in practice proves that another important information security goal [2] for protection in depth against cyber attacks in the IoT is possible.

2 Background Information

In the EU-funded project RERUM we developed a framework which enables IoT applications to consider security and privacy mechanisms early in their design phase. One of the goals of RERUM is to provide end-to-end integrity protection down to the wireless sensor network. Part of this framework is an application that runs on constrained devices and provides ECDSA signed sensor data using standard IoT protocols. This paper presents the experiences and lessons learned during development and testing of the signing application. We described early results that present the overhead of signatures in terms of runtime, memory, energy consumption and communication in [3], with energy overhead being the focus. Here we discuss the practical impact of signatures by showing how long the sensor device used in the trials can actually work running on battery. This paper extends previous findings by adding new measurements we did using a hardware accelerated ECDSA implementation. We also examine the impact of problems like an unreliable network connection, a typical aspect of low-power and lossy networks, and complement a number of considerations done previously [1,3,4].

2.1 Hardware Platform

For the proof-of-concept implementation of ECDSA integrity protection, we initially used the Z1 platform[2] [5]. It is based on a MSP430 16bit RISC CPU with 8 KB RAM, approximately 60 KB (usable) flash memory, and the CC2420, a 2.4 GHz IEEE 802.15.4 [6–8] compliant RF transceiver.

Later we switched hardware platform to a Zolertia RE-Mote[3] [9]. The RE-Mote incorporates the CC2538 Cortex-M3 SOC from Texas Instruments with

[2] zolertia.io/product/hardware/z1-platform (last accessed 02 Oct 2016).

[3] zolertia.io/product/hardware/re-mote (last accessed 02 Oct 2016).

32 MHz, 512 KB flash and 32 KB RAM[4] [10]. The CC2538 contains a hardware crypto engine that accelerates cryptographic operations like SHA2, AES and ECC. The RE-Mote did run Contiki[5] [11], a lightweight operating system designed with Internet-of-Things, and the restrictions and needs of constrained devices in mind. We used the RE-Mote hardware platform during all trials and all measurements, and explanations in this paper are based on it, unless explicitly stated otherwise.

2.2 JSS and Signed CoAP Message

JSON Sensor Signatures (JSS) is a valid JSON format that contains additional meta-data about the embedded signature and the algorithm applied. It is an encoding scheme that transports the digital signature and its meta data alongside the JSON data [4]. This enables to keep the plain information accessible to all involved processes that can handle JSON. By this the signature can remain attached for end-to-end protection as long as JSON data can be stored. This is the case for a number of IoT storage backends, like Couchbase or MongoDB.

In IoT terminology the device will expose its sensed data as resources. Here, following IoT-A [6] and RERUM terminology [12], the device's resources can be accessed via a RESTful interface [13] and return data in the JSON format. The resources can be requested in two representations: unsigned with the Constrained Application Protocol (CoAP) content-format `application/json` and signed with a newly introduced content-format `application/jss`.

```
{
  "jss.protected":
  {
     "alg": "ES192"
  },
  "amb_temp": 20965,
  "measurement_id": 21,
  "jss.signature": "le4uz7vWD_z•••WL"
}
```

Fig. 1. Sample JSS message. Omitted characters are indicated with dots (•••). Whitespace is added for better readability. The framed message parts are the JSS related additions. Note that unsigned messages contain only the sensor value itself.

While `application/jss` is no official standard mime-type, this design decision was made to maintain backwards compatibility to clients that aren't capable of handling signatures. The sample JSS message depicted in Fig. 1 shows its internal structure: a header section (`jss.protected`) describing the ECC-based signature scheme [14], a field `measurement_id`, and the signature itself,

[4] ti.com/product/cc2538 (last accessed 02 Oct 2016).
[5] contiki-os.org (last accessed 02 Oct 2016).
[6] iot-a.eu/public/terminology (last accessed 02 Oct 2016).

encoded in BASE64URL [15] (`jss.signature`). The payload itself was hashed with SHA-256 [16] and then signed with `secp192r1` (indicated by `ES192`[7]).

We build the signature itself following the steps we suggested in [4]:

1. clean JSON keys that are part of payload (i.e. not part of signature metadata; clean = remove non-alphanumerical characters)
2. sort key-value pairs of previous step alphanumerically
3. encode measurement_id in BASE64URL
4. encode payload in BASE64URL
5. concatenate encoded measurement_id and payload with dot character ('.')
6. build SHA-256 hash of concatenated string
7. sign hash generated
8. encode signature in BASE64URL

For verifying a received JSS message, the first six steps are performed to generate the hash. Then the received BASE64URL encoded signature is decoded and fed together with the hash and public key to the verification function.

The JSS field `measurement_id` was introduced for signed JSS messages to protect against replay attacks. In replay attacks a potential attacker could sniff packets and replay them later. The client has no means to distinguish between the valid messages and the replayed ones, as the replayed packets do contain valid signatures.

The measurement_id gets incremented after each measurement (here every 30 s). These are used for generating the signature and a client can use them to detect previous measurements. Of course, this protects against replay attacks only, if the measurement_ids are not repeating.

For reducing the overhead, we use a 32bit unsigned integer for storing the measurement_id, allowing for $2^{32} = 4,294,967,295$ messages with different measurement_ids. We consider this as safe, as with a typical sensing interval of 30 s, it would take more than 4.000 thousand years for one measurement_id to repeat. Furthermore, possible restarts or crashes of the sensing device due to updates, power issues, battery changes that lead to resetting of the measurement_id counter to zero have to be considered. To circumvent this, the measurement_id should additionally be stored on a non-volatile memory, like the internal flash.

3 ECDSA Signatures Implementation

For the proof-of-concept implementation, we used a reference implementation from NIST[8] together with the p160 curve on the Zolertia Z1. It revealed, that the Z1s capabilities are too restricted to provide security on an adequate level ($\geq$192bit curve size [17]): while runtime performance proved fine, the RAM and flash memory sizes were too limited [4]. Our Performance and overhead measurements [3] revealed MicroECC [18] to be a well suited ECDSA implementation.

[7] RFC7518 [14] actually does not provide an identifier for SHA-256 hashing in combination with `secp192r1` signing. However, it does for SHA-256 hashing in combination with P-256 signing (`ES256`). Therefore we use `ES192` analogue to this convention.

[8] github.com/nist-emntg/ecc-light-certificate (last accessed 02 Oct 2016).

3.1 Implementation Details

The RERUM sensing application consists of multiple parts:

1. Implementation of sensing routines for several internal and external sensors (e.g. ambient temperature, humidity, noise, and various analogue sensors).
2. ECDSA signature generation with MicroECC.
3. Encoding and formatting of sensed value with JSS.
4. Exposing resources, i.e. signed sensor data through a standards compliant RESTful CoAP interface.

MicroECC was adapted to the needs of our use case as follows: To generate SHA-256 hashes, we used the CC2538's hardware acceleration engine. The CC2538 random number generator delivers the nonce[9] required for the signing process. We configured MicroECC to use the optional in-line assembly optimisations (here referred as `ASM fast`). In order to encode the signatures, we extended a BASE64 implementation to support BASE64URL compliant encoding.

Contiki ships with various integrated applications like the IPv6 network stack and application layer protocols. This simplified the implementation of signatures significantly, since it allowed increasing the focus on signature generation and encoding rather than on the network stack. That's why all network related libraries were taken directly from Contiki like CoAP, the REST implementation, and the lower layer implementations as well (like UDP, IPv6, and 6LoWPAN). Contiki's CoAP implementation was extended by a monitor concept that encapsulated the resource representation, exchange and signature generation and allows to easily add new sensors without rewriting network specific operations.

This approach permitted us to reduce code complexity and maintainability, and as well simplified the integration of different external sensors.

3.2 Hardware Acceleration

We compared different ECC signature libraries in terms of runtime overhead, firmware size and energy consumption on the RE-Mote hardware [3]. Among those examined libraries is MicroECC [18], a library implementing the standard NIST curves `secp160r1`, `secp192r1`, `secp224r1`, `secp256k1` and `secp256r1`[10]. MicroECC is a platform-independent ECC implementation written in C that was especially designed with constrained devices in mind. It has a small code footprint, does not require dynamic memory allocation, and runs on 8-, 32-, and 64bit CPUs. To increase performance, MicroECC can be optionally configured to use in-line assembler for the AVR, ARM, and Thumb platforms in two modes with speed vs. code size optimisations: `ASM small` (little code size overhead) and `ASM fast` (optimised on speed, but with more code size overhead). Since measurements showed acceptable performance of MicroECC, we selected its ECDSA library for further development.

[9] nonce: arbitrary number only used once.

[10] github.com/kmackay/micro-ecc (last accessed 02 Oct 2016).

We presented the MicroECC runtime overhead in relation to the different ECC curves and assembly optimisations [3]. This paper extends those results by new measurements that contemplate the impact of hardware acceleration.

In this section, we present the runtime of a single hashing and signing step with the different implementations. To avoid any influences that could distort the results, i.e. hardware interrupts, the firmware images flashed on the RE-Motes contain only the necessary parts to perform the measured operations; the Contiki network stack was disabled.

For encoding the signatures, we improved a plain C implementation of the BASE64 encoding scheme in order to support BASE64URL compliant encoding. The message buffer that gets signed depends only on the signature size, i.e. the size of the ECC curve. The runtime overhead of BASE64URL encoding either for `secp192r1` (signature with 48bytes) or `secp256r1` (signatures with 64bytes) is negligible with both less than 1 ms. Thus we did not further examine this step, since it has little impact on the signing process.

The RE-Mote's CC2538 features a hardware encryption engine that is capable of accelerating the generation of SHA2 hashes. The RERUM application already uses this feature for hashing the message payload prior getting signed; where the message that gets hashed is between 30 and 50 characters long (depends on the measurement_id and resource identifier).

Table 1. Runtime overhead of SHA256: 39bytes refers to a typical JSS payload (here: amb_temp for ambient temperature resource); the 1024bytes where randomly chosen.

Size [byte]	Un-accelerated [ms]	Accelerated [ms]
39	0.669	0.059
1024	5.834	0.471

The results are shown in Table 1: Hashing 39bytes, a typical payload length within the RERUM application, has an negligible overhead of less than 1 ms. However, the values of hardware accelerated hashing compared to the un-accelerated ones indicate a speed-up by a factor of approximately eleven. We verified the factor in another measurement by randomly chosen 1024bytes buffers to exclude any inaccuracies that could come from the timer at such small intervals. These measurements show a similar difference: while the plain C implementation needs ~5 ms for hashing, the accelerated one does not even need half a millisecond leading to a performance boost by a factor of approximately twelve.

Besides the hardware acceleration for SHA2 hashes, the RE-Mote's CC2538 features also a ECC hardware acceleration engine. Fortunately Contiki contains a ECC implementation (`secp192r1` and `secp256r1`) that utilizes hardware acceleration since October 2015. Table 2 shows the results: the measurements for un-accelerated, `ASM small` and `ASM fast` signing with MicroECC were taken from [3] and extended by the results from the hardware accelerated implementation. While the in-line assembly optimisations `ASM small` reduces the runtime

for signing about a quarter and `ASM fast` even more than a half, hardware acceleration reduces the runtime about almost three-quarters. Surprisingly, for the verification MicroECC with `ASM fast` beats the hardware accelerated implementation. Currently, we can't explain this behaviour.

Table 2. Runtime for signing and verifying with `secp256r1` curves; un-accelerated, `ASM small` and `ASM fast` refer to MicroECC measurements we presented in [3].

Configuration	Un-accelerated	ASM small	ASM fast	Accelerated
secp256r1 sign [ms]	1177	855	537	341
secp256r1 sign [%]	100	72.6	45.6	29
secp256r1 verify [ms]	1320	957	595	697
secp256r1 verify [%]	100	72.5	45	52.8

We note that the hardware accelerated implementation leaves it up to the user of this library to choose the random nonce used for signing. It is important to be aware of the fact that using the same nonce more than once [19] or relying on a bad random number generator [20] offers attackers the possibility to extract the private key used for signing. MicroECC instead keeps care of choosing and processing this random number correctly and furthermore considers possible side channel attacks [21].

Table 3. Signing and verifying runtimes for TweetNaCl Ed25519 implementation.

Configuration	Un-accelerated [ms]
Ed25519 sign	3332.9
Ed25519 verify	6646.9

To compare the results of Table 2, we used the Ed25519 curve implementation of the TweetNaCl [22] library[11]. This implementation is quite compact[12], but not optimised for speed. Signing and verifying takes with roughly 3.3 and 6.6 s (see Table 3) significantly longer than with the `secp256r1` curve implementations described previously.

Unfortunately, like MicroECC's `secp256r1`, Ed25519's RAM requirements are too high in order to fit in the RERUM application. Even with further optimisations, we did not succeed to use TweetNaCl's Ed25519 curve alongside with the CoAP interface, the sensor library, and the other routines required.

3.3 Challenges Overcome

The runtime overhead of generating signatures is smaller than expected, however, a lot of signing operations within a short period can reduce the availability and

[11] tweetnacl.cr.yp.to.

[12] TweetNaCl fits into 100 Twitter Tweets.

responsiveness for clients issuing requests. This increases also the danger for easy Denial-of-Service attacks for attackers flooding one sensing RE-Mote with too many requests. For this reason we decided not to sense and sign new sensor values for every incoming request. Instead, sensing and signing takes place once within a configurable interval down to five seconds.

The RE-Mote made it possible to implement the use-case, unlike the Z1 which was too constrained. However also the RE-Mote's RAM size was limiting: Signing with `secp256r1` together with the described network stack and the JSS format caused stack overflows during runtime. Attempts to circumvent this by increasing the stack was not helping.

Therefore we decided to use `secp192r1`. It has 80 minimum bits of security [23] and is still considered as an adequate security level with an equivalent symmetric key size of 96bits [17]. Also the RAM limits the number of resources that can get provided with signatures to approximately five. The reason is that for each signed resource we pre-allocate memory for the buffers to store the last signed value in order to be able to respond to queries quickly. In practice this restriction is mitigated as we implemented the application in such a way that is easy to configure signing on a per-resource basis during build time. This fulfils the different needs for different use-cases and sensors while the code can be still developed centrally from one consistent code base.

4 Trial Scenario

Our trial scenario was placed in Heraklion, Greece: RE-Motes together with attached sensors were installed. They measured and provided resources like ambient temperature and ambient humidity.

4.1 Network Architecture and Components

For the trials, we used a 3-tier network architecture, as shown in Fig. 2, consisting of a RERUM device, RERUM gateway, and RERUM middleware. RERUM devices are our wireless RE-Motes that we use for sensing the environment. Each RE-Mote carries several sensors that collect data like ambient temperature and humidity, noise, light, etc. Each distinct type of sensory data is made available through a CoAP resource.

The RERUM gateway has several roles. First, it interconnects the devices through an 802.15.4 wireless sensor network by initiating the RPL [24] routing protocol. Second, it also provides the interconnection between the sensor network and the outside world. Third, it keeps a list of the registered sensors and performs several housekeeping operations (device resetting, etc.). Furthermore, we note that it is possible to verify signatures on the gateway. However, additional functionality triggered by the verification result of the gateway is of separate interest, and further discussed by us in [25,26].

The RERUM middleware comprises the backend, where the sensory data is stored. Authorised end-used applications can request sensory data through

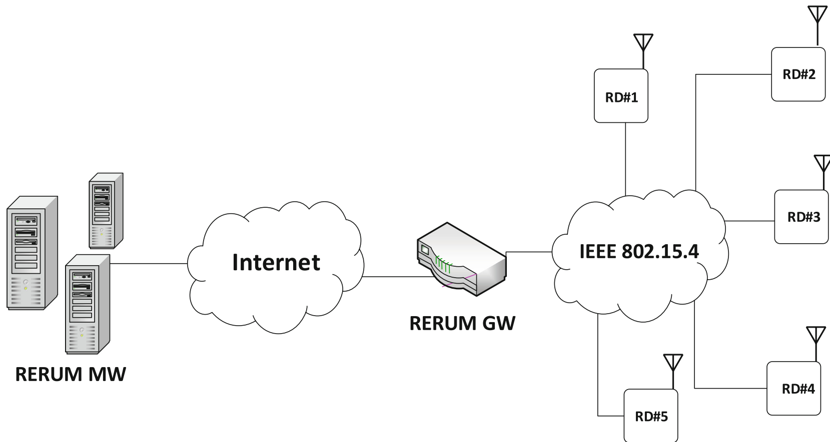

Fig. 2. Network topology used for the trials

the middleware. For every distinct client request, the middleware communicates with the gateway using an appropriate protocol over HTTP. Next, the gateway translates this request to a CoAP one, and transmits the corresponding sensory data back to the middleware. We note, that the communication between the middleware and the gateway is performed through a secure VPN connection.

The trials were performed in a wireless sensor network with one RERUM gateway and eleven RERUM devices. All components were installed in a building owned by the Heraklion municipality. The RERUM devices were sensing, signing and transmitting new values every 30 s.

4.2 Battery Runtime and Energy Consumption

To get a practical comparison of signed sensing in a real world environment, we installed a test setup of two RE-Motes with coin cells and AAA batteries. Those RE-Motes sensed the ambient temperature and provided it throughout the CoAP interface. The firmware images deployed were exact the same. The only difference was, that one device had signing enabled. The signed messages were provided in JSS format. For signing, we used MicroECC with enabled in-line assembly optimisations and `secp192r1` curve. The CoAP client requested the resources once in a 30 s interval. It was running on a standard PC connected via a border router to the sensing devices.

First tests with standard 3V CR2430 coin cells revealed that coin cells are not well suited to drive the RE-Motes: both ran for a maximum of roughly three and a half hours before shutting down. There was no notable difference in runtime of the device signing messages and the one emitting just plain messages. Slight differences were most probably caused by variations of the coin cells capacities.

To increase runtime and reveal real runtime differences, we decided to switch to standard AAA/LR03 batteries with a voltage of 1.5 V. For each of the RE-

Motes we used two of those batteries in series, picked from the same batch (to reduce the influence of the variations of different battery manufacturers).

Compared to the coin cell, the runtime with two AAA batteries was significantly higher. The signing device achieved an average runtime of 6228 min, whereas the other achieved approximately an additional half an hour, with an average runtime of 6264 min. The difference in runtime between two signing-enabled runtime measurements and two signing-disabled runtime measurements was 2% at most; potentially still caused by battery capacity variations.

To increase total performance and to avoid easy Denial-of-Service attacks on sensing, the RE-Mote is not generating a new signature for every request. Instead, the signature is generated only once within the (configurable) sensing interval. This decreases the impact of signatures for the overall runtime effectively. Our lab measurements [3], revealed that the energy overhead of signatures basically originates from the increased time period in which the CC2538 is not in sleep mode. Therefore we decided for the trials to measure and sign only once in every 30 s for each resource. The experiments show that signing with such a low frequency does have little to no measurable impact on the overall battery runtime in a real-world scenario.

5 Experimental Validation

The investigated logs cover a period of two days of measurements resulting in roughly 2.800 messages each for the resources ambient temperature and ambient humidity; both in signed JSS format and as well plain, unsigned, JSON. The signatures were all verified as valid, which indicates that the network error correction mechanisms described were working successfully and no message was manipulated. To rule out implementation errors, we tampered signatures manually by introducing single bit errors, which immediately caused the verifier to correctly recognize them as manipulated.

The RE-Mote is using the 6LoWPAN adaption layer over 802.15.4 for sending the JSS messages. Since the MTU is only 127bytes before packets get fragmented, JSS messages with typical length of about 160bytes and more do not fit into a single 6LoWPAN packet. While RFC7252 [27] states that a "CoAP message [...] SHOULD fit within a single IP packet" in order to avoid fragmentation on the IP layer, no strict requirements considering message size for constrained networks are given. This problem gets addressed by the CoRE Working Group: "Block-wise transfers in CoAP" [28]. This draft extends CoAP "with a pair of 'Block' options, for transferring multiple blocks of information from a resource representation in multiple request-response pairs".

The block options exist for both, requests (option `Block1`, e.g. `POST` and `PUT`) and responses (option `Block2`, e.g. responses for `GET` requests), but for the use-case of the RERUM application, only the `Block2` option for (JSS) responses is relevant. The block sizes can be chosen by the client and must be of the power-of-two, ranging from $2^4 = 16$ to $2^{10} = 1024$bytes. The RERUM application supports only sizes up-to 64bytes, since greater block sizes would again lead to

fragmentation on the 6LoWPAN/802.14.5 layers. The block sizes used during development and the trials is indeed 64bytes, since lower values would increase the number of messages exchanged and thus the overall transmission overhead.

A sample interaction between a client requesting a resource (e.g. ambient-temperature) on a RE-Mote is depicted in Fig. 3. This sample resource is requested as a signed JSS representation and has total length of 164bytes. Block-wise transfer means that these JSS messages get divided into three blocks of $64 + 64 + 36$bytes (the last block contains the remainder and is typically smaller than the block size). The client specifies the desired block sizes and sends the request to the RE-Mote, which answers with the first block of the requested resource. Furthermore, the RE-Mote specifies the number of this block in the sequence and if there are more blocks available for this resource. The client subsequently requests the following blocks until it receives the last one. The RE-Mote itself acts completely stateless and does neither need to do flow control nor maintain any session information.

The RERUM application is deployed on constrained devices in low-power and lossy networks [29]. Unreliable communication channels and high packet loss are some of the challenges of those constrained networks. Since the RE-Mote itself is a constrained device with limited memory and CPU performance, the overhead of lost packets and transmission errors, and their impact are important to consider. Here we examined the transmission overhead in case of lost packets and transmission errors.

The sensing RE-Mote is exposing the resources through the RESTful CoAP interface. A normal personal computer with installed Firefox and the Firefox CoAP add-on Copper [30][13] is acting as a client. If the client issues a CoAP `GET` request, this request gets passed to the network interface 'tun0' that is created

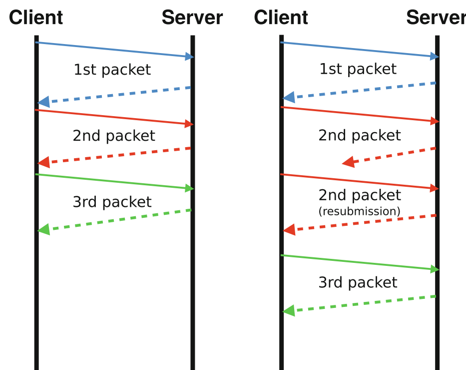

Fig. 3. A client is requesting a CoAP resource represented as a JSS message in three CoAP blocks. The left side shows a communication without errors. To the right the 2nd CoAP block gets lost and the client requests re-submission prior requesting the 3rd block. Note that requests are represented as straight lines and responses as dashed.

[13] people.inf.ethz.ch/mkovatsc/copper.php (accessed 23 Aug 2016).

by Contiki's `tunslip` script[14]. This network tunnel is forwarding the IP packets to the border router via the serial interface [31]. A second RE-Mote is acting as the border router, flashed with the border router firmware image provided by the Contiki project. The border router is responsible for translating between 6LoWPAN packets from the wireless sensor network and the other, 'normal' IPv6 packets. Once it receives packets through the serial line, it converts and sends them to the wireless sensor network (which consists in our setup of one sensor host only).

In the focus of this setup are packets that were coming from the sensing RE-Mote; we did not manipulate packets from the client to the RE-Mote. We altered the border router source code to simulate different network behaviours. We implemented a feature to configure a drop frequency n, that triggers the border router to drop every n-th packet coming from the sensing RE-Mote. This allows to effectively simulate an unreliable communication link, where packets get lost during transmission with a configurable 'unreliability'.

The second feature triggers bit errors such as flipping bits: a 'flip_signature_bit' parameter m can be configured to flip single bits of every m-th JSS signature transmitted. We examined the network traffic that goes through the `tunslip` tunnel on the client machine with Wireshark[15].

The sample GET interaction depicted in Fig. 3 shows that the client is requesting the individual CoAP blocks one by one. Each request is marked as 'confirmable' and the client only succeeds if it has received an acknowledgement ('ACK') for its request. To minimise the messages exchanged, the sensing RE-Mote piggy-packs the acknowledgements onto the responses. If a packet loss occurs and the client doesn't receive the second CoAP block, the client waits for the two seconds timeout and requests this block again. After successfully receiving the missing block, the client proceeds with requesting the last block. This interaction shows, that a single packet loss doesn't cause the whole JSS message to be retransmitted, but only that part, i.e. the CoAP block, that got lost. In this scenario, the sensing RE-Mote acts completely stateless and is not involved in the recovering of the lost packet, it just responds to the additional request.

Now to simulate transmission errors, we configured the border router to flip the last bit of the first character of the signature. Again, like in the scenario of the packet loss, the UDP checksum [32] signals to the client a transmission error and only the UDP datagram, i.e. the CoAP block that contained this bit error is retransmitted, while the other blocks of the JSS message are unaffected.

6 Conclusion and Future Work

In this paper, we presented the practical impact of using integrity protected communication through elliptic curve based signatures performed in a real-world

[14] github.com/contiki-os/contiki/blob/master/tools/tunslip6.c.
[15] wireshark.org.

IoT trial scenario. We proved that integrity protection on constrained devices is possible now, however with some restrictions.

We showed that the Zolertia Z1 with the MSP430 chip was too constrained to build a secure IoT application [4]. We did not succeed to build a sensing application that incorporates a standard IoT operating system like Contiki, a standards compliant CoAP and REST interface in combination with ECC signatures on an adequate security level (more or equal than 192bit curve size) on a chip with only 8 KB RAM and around 60 KB flash memory.

However we showed that the resources provided by the Zolertia RE-Mote using a CC2538 chip with a ARM Cortex-M3 32bit chip, 32 KB RAM and 512 KB flash are sufficient [3]. The lab results proved to be sufficiently stable using a 192bit curve size to go into trials.

In Lab testing we analysed several existing elliptic signature implementations and libraries [3,4]. In terms of runtime and power consumptions MicroECC provided the best results and was therefore selected. Our new results show that this turned out to be a good choice. MicroECC is in active development, easy to use and performs well in terms of runtime and code footprint overhead.

Switching to the RE-Mote and the measurements, however, proofed that it is possible to build a IoT application that provides integrity protection through digital signatures and communicates with modern, standard-compliant protocols. However, we did not tackle the key-distribution problem which needs to be considered when building IoT applications with end-to-end security.

Besides the results from the trial scenario, additional runtime overhead measurements were done. These extend the comparison of different ECC libraries presented in [3] by an implementation that utilises the hardware crypto engine capabilities of the CC2538 chip significant. Our results reveal that the hardware accelerated implementation is performing better in terms of runtime, than the already well performing MicroECC library. However, besides the runtime other aspects like the problem of side channel attacks that MicroECC addresses have to be considered in depth in future work.

This paper also extends the theoretical communication overhead considerations done in [3], by practical aspects that influence the number of messages exchanged in low-power and lossy networks, such as packet loss and transmission errors. Our results show that CoAP and UDP offer robust and reliable communication, while still being lightweight. Transmission errors are treated in such a way that also constrained devices, like the RE-Mote platform can handle them with little overhead.

It is worth noting that our JSS message format used is not optimal for exchanging messages in terms of size. Therefore we expect that one can reduce the communication overhead further by improving the formatting of the message and the embedded signature. Another option is to limit signatures to average values as we suggest in [25,26].

Acknowledgements. J. Bauer, R.C. Staudemeyer, H.C. Pöhls, and A. Fragkiadakis were supported by the European Unions 7th Framework Programme (FP7) under grant agreement n°609094 (RERUM). H.C. Pöhls and R.C. Staudemeyer were also

supported by the European Unions Horizon 2020 Programme under grant agreement n°644962 (PRISMACLOUD). Additionally J. Bauer and H.C. Pöhls were supported by the Bavarian State Ministry of Education, Science and the Arts as part of the FORSEC research association.

References

1. Pöhls, H.C., Angelakis, V., Suppan, S., Fischer, K., Oikonomou, G., Tragos, E.Z., Diaz Rodriguez, R., Mouroutis, T.: RERUM: building a reliable IoT upon privacy- and security-enabled smart objects. In: Wireless Communications and Networking Conference Workshop on IoT Communications and Technologies (WCNC 2014), pp. 122–127 (2014)
2. Staudemeyer, R.C., Pöhls, H.C., Watson, B.W.: Security & privacy for the internet-of-things communication in the SmartCity. In: Angelakis, V., Tragos, E., Pöhls, H.C., Kapovits, A., Bassi, A. (eds.) Designing, Developing, Facilitating Smart Cities: Urban Design to IoT Solutions, 30 p. Springer, Heidelberg (2016)
3. Mössinger, M., Petschkuhn, B., Bauer, J., Staudemeyer, R.C., Wojcik, M., Pöhls, H.C.: Towards quantifying the cost of a secure IoT: overhead and energy consumption of ECC signatures on an ARM-based device. In: Proceedings of the 5th Workshop on the Internet of Things Smart Objects and Services (IoTSoS 2016), 6 p. (2016)
4. Pöhls, H.C.: JSON sensor signatures (JSS): end-to-end integrity protection from constrained device to IoT application. In: Proceedings of the Workshop on Extending Seamlessly to the Internet of Things (esIoT 2015), pp. 306–312 (2015)
5. Zolertia, "Z1 datasheet," 20 p. (2010)
6. Montenegro, G., Kushalnagar, N., Hui, J., Culler, D.: RFC4944 – transmission of IPv6 packets over IEEE 802.15.4 networks. Requests for Comments (2007)
7. IEEE Standards Association, Part 15.4g: Low-Rate Wireless Personal Area Networks (LR-WPANs) Amendment 3: Physical Layer (PHY) specifications for low-data-rate, wireless, smart metering utility networks. IEEE (2012)
8. Olsson, J.: "6LoWPAN demystified," Texas Instruments, 13 p. (2014)
9. Zolertia, "RE-Mote datasheet," 2 p. (2015)
10. Texas Instruments, "CC2538 datasheet," 32 p. (2015)
11. Dunkels, A., Grönvall, B., Voigt, T.: Contiki – a lightweight and flexible operating system for tiny networked sensors. In: 29th Annual IEEE International Conference on Local Computer Networks (LCN 2004), pp. 455–462 (2004)
12. Angelakis, V., Cuellar, J., Fischer, K., Fowler, S., Gessner, J., Gundlegård, D., Helgesson, D., Konios, G., Lioumpas, A., Lunggren, M., Mardiak, M., Moldovan, G., Mouroutis, T., Nechifor, S., Oikonomou, G., Pöhls, H.C., Ruiz, D., Siris, V., Suppan, S., Stamatakis, G., Stylianou, Y., Traganitis, A., Tragos, E.Z.: The RERUM system architecture (RERUM Deliverable D2.3). Technical report (2014)
13. Fielding, R.T.: Architectural styles and the design of network-based software architectures. Ph.D. dissertation, Department of Information and Computer Science (2000)
14. Jones, M.: RFC7518 – JSON Web Algorithms (JWA). Technical report, Requests for Comments, Internet Engineering Task Force (2015)
15. Josefsson, S.: RFC4648 – The Base16, Base32, and Base64 data encodings. Technical report, Requests for Comments, Network Working Group (2006)
16. Dang, Q.H.: Secure hash standard. National Institute of Standards and Technology, Gaithersburg, MD, Technical report, August 2015

17. European Network of Excellence in Cryptology II: ECRYPT II Yearly Report on Algorithms and Keysizes (2011–2012). Katholieke Universiteit Leuven, Technical report (2012)
18. MacKay, K.: micro-ecc. http://kmackay.ca/micro-ecc/
19. fail0verflow (bushing, marcan, segher, sven), "Console Hacking 2010 - PS3 Epic Fail (slides)," 27th Chaos Communications Congress (27C3) (2010). http://events.ccc.de/congress/2010/Fahrplan/attachments/1780_27c3_console_hacking_2010.pdf
20. Klyubin, A.: Some SecureRandom Thoughts (2013). http://android-developers.blogspot.de/2013/08/some-securerandom-thoughts.html
21. Brumley, B.B., Tuveri, N.: Remote timing attacks are still practical. In: Atluri, V., Diaz, C. (eds.) ESORICS 2011. LNCS, vol. 6879, pp. 355–371. Springer, Heidelberg (2011). doi:10.1007/978-3-642-23822-2_20
22. Bernstein, D.J., Gastel, B., Janssen, W., Lange, T., Schwabe, P., Smetsers, S.: TweetNaCl: a crypto library in 100 tweets. In: Aranha, D.F., Menezes, A. (eds.) LATINCRYPT 2014. LNCS, vol. 8895, pp. 64–83. Springer, Heidelberg (2015). doi:10.1007/978-3-319-16295-9_4
23. Turner, S., Brown, D., Yiu, K., Housley, R., Polk, T.: RFC5480 - elliptic curve cryptography subject public key information. Technical report, Requests for Comments, Network Working Group (2009)
24. Brandt, A., Hui, J., Kelsey, R., Levis, P., Pister, K., Struik, R., Alexander, R.: RFC6550 – RPL: IPv6 routing protocol for low-power and lossy networks. Technical report, Requests for Comments, Internet Engineering Task Force (2012)
25. López, D.R., Cuellar, J., Staudemeyer, R.C., Charalampidis, P., Fragkiadakis, A., Kasinathan, P., Pöhls, H.C., Suppan, S., Tragos, E., Weber, R.: Modelling the trustworthiness of the IoT (RERUM Deliverable D3.3), Technical report (2016)
26. Tragos, E.Z., Bernabe, J.B., Staudemeyer, R.C., Luis, J., Ramos, H., Fragkiadakis, A., Skarmeta, A., Nati, M., Gluhak, A.: Trusted IoT in the complex landscape of governance, security, privacy, availability and safety. In: Digitising the Industry – Internet of Things Connecting the Physical, Digital and Virtual Worlds. River Publishers Series in Communications, pp. 210–239 (2016)
27. Shelby, Z., Hartke, K., Bormann, C.: RFC7252 – The Constrained Application Protocol (CoAP). Technical report, Requests for Comments, Internet Engineering Task Force (2014)
28. Bormann, C., Shelby, Z.: Block-wise transfers in CoAP. Working Draft, IETF, Internet-Draft (2016)
29. Shelby, Z., Hartke, K., Bormann, C.: RFC7228 – terminology for constrained-node networks. Technical report, Requests for Comments, Internet Engineering Task Force (2014)
30. Kovatsch, M.: Demo abstract: human-CoAP interaction with Copper. In: International Conference on Distributed Computing in Sensor Systems and Workshops (DCOSS 2011), pp. 1–2, Barcelona, Spain (2011)
31. Romkey, J.: RFC1055 – Nonstandard for transmission of IP datagrams over serial lines: SLIP. Technical report, Requests for Comments, Network Working Group (1988)
32. Postel, J.: RFC768 – User Datagram Protocol. Technical report, Requests for Comments, Internet Engineering Task Force (1980)