This is a preprint. Final version appears in IEEE CAMAD '17 (19-12 June 2017, Lund)
DOI: http://ieeexplore.ieee.org/servlet/opac?punumber=1002097

Towards compactly encoded signed IoT messages

Henrich C. Pöhls,
Institute of IT-Security and Security Law (ISL)
University of Passau
Innstr. 43
94032 Passau, Germany
Email: hp@sec.uni-passau.de

Benedikt Petschkuhn
University of Passau
Innstr. 43
94032 Passau, Germany

***Abstract*—In this paper, we measure the overhead in message size and in firmware code-size for exchanging signed messages with constrained devices typically found in the Internet-of-Things (IoT) domain. The application of digital signatures on every important message in the IoT helps to increase its resilience against attacks and increases the data quality level. Other papers have shown that Elliptic Curve Cryptography (ECC) signatures can be applied on modern constrained devices. Our focus is on message size as we identified this to be the most prevailing problem as it relates to the storage required at each handling node and directly relates to the communication overhead, which in turn results in an increased energy consumption. We give analytical information for typical message sizes in two different protocols (MQTT and COAP), as well as real measurements on a Zolertia ReMOTE when it comes to firmware code-size overheads. Our measurements indicate that a suitable encoding of message level security can decrease the message overhead of signatures by at least 30 percent and still balance its usability by keeping it decodable into JSON, the prevailing exchange format of many IoT upper layer protocols. Our new HYBRID format is suitable to achieve end-to-end, i.e., device-to-application, integrity.**

I. Introduction

With the term Internet-of-Things (IoT) we describe a complex system that reacts via a set of constrained subsystems to physical changes (sensors) and is able to influence the physical world through another set of subsystem (actuators). Complex system design lends itself to the use of additional layers of abstraction. The IoT is no exception: An IoT middleware abstracts from the underlying deployment of the actual sensors and actuators (the devices). Further, the middleware allows applications to easily request sensor data and communicate with the virtual representations of the physical world. The implementation of middleware components means that an IoT communication will travels through a multitude of systems before it reaches the application, e.g., through different constrained devices, routers, the Internet, and middleware components, such as message queuing systems, databases that might be partly cloud deployed systems. With this many systems the possibilities for a malicious attack, i.e., loss of integrity, are endless. Following good IT security practice, we require cryptographic integrity protection and strong origin authentication. Only this ensures that an application, which does make decisions based on IoT sensor data, is working on valid and un-tampered information. Furthermore, we want that commands issued to actuators are being only carried out after the actuator validated their authenticity. As we can not trust all intermediate systems in all deployments, we want this protection to be end-to-end, i.e., sensed data must be protected from the constrained device to the application server and vice-versa for commands.

Digital signatures are the cryptographic tool to gain such an integrity protection for a message. The signature is generated using the secret signing key and the message. The verifications algorithm takes as an inout the signature, the message and the public verification key. If the verification result is positive the message has not been altered and it has been created involving the secret key corresponding to public key used for verification. The latter gives us authentication of origin.

With elliptic curve cryptography (ECC) there are good cryptographic algorithms that are able to run on constrained devices. ECC was independently described by *Miller* [1] and *Koblitz* [2] in 1986. It got already standardised, i.e., is well known. Recently, there was criticism of the NIST standard on Elliptic Curve Digital Signature Algorithm (Elliptic Curve Digital Signature Algorithm (ECDSA)) [3] which deemed it potentially insecure [4]. However, new curves have been standardized [5], like Curve25519 proposed by *Bernstein*.

Also many implementations of ECC which are well-suited for constrained devices exist, e.g., NanoECC [6], or NIST's ECClight [7]. That ECC can be very efficient, such as the curve with 160-bit key length implementation, is shown by *Kern* and *Feldhofer* [8]. A very lightweight ECC-based construction for authentication to run on RFID-type devices was presented by *Braun*, *Hess* and *Mayer* [9].

Hence, ECC signatures can be used to protect the integrity and allow via asymmetric keys to authenticate the origin of messages. While encryption does not impact on message length, rather on CPU usage and speed, integrity always additionally increases the message length. The signature for the message has to be generated and then communicated alongside the actual payload. The length of the signature varies not with the payload length, but depends on the security level and the chosen mathematical algorithms. We found that sticking with ECC allows to harvest the existing code base and the existing standardised cryptographic algorithm, so we choose this as our underpinning.

A. Contribution and outline

This paper's contribution is two-fold: First, it gives analytical results showing the overhead in size in reality by analysing existing proposals in Sect. II. Secondly, we propose a new hybrid encoding scheme that limits the increase to a bare minimum in Sect. III. We measured the overhead in message size and also the code-size on a constrained device. The numbers presented in Sect. IV can be used as input to simulations and helps to calculate the impact of enabling end-to-end integrity and strong origin authentication.

II. Existing Approaches and Encodings

Transmitting signed messages and keys in a suitable manner requires data formats to offer an underlying structure and syntax scheme. Signatures and keys consist of binary data, hence efficient encodings are required to minimize overhead. If one take a look at even privacy aware system frameworks for the IoT, like [10], [11], then one sees that one would require end-to-end protection as too many components are involved to establish connections to communicate reliably and in an abstract manner with the IoT's devices (sensors and actuators). Of course the encryption, like Datagram Transport Layer Security (DTLS) [12], [13] "provides authenticated, confidentiality- and integrity-protected communication between two endpoints" [12]. Thus they would protect integrity alongside. However, encryption renders data unreadable by intermediaries and thus would remove the ability to read the data unless they are an end-point. Also DTLS is connection oriented. For example assume that we read temperatures and need to take severe actions in case we detect overheating, than you do not want to react on falsified injected readings. Thus we need integrity and authenticity, i.e., authentication of origin. Assume further that the middleware would have an intelligent filtering message queuing system which would prioritize high temperature readings to facilitate responsiveness of alerts for overheating. If the data were encrypted and the message queue would not be an end-point then it would not be able to prioritize. It it were an end-point of DTLS, then it would be able to manipulate the data before sending it over another DTLS secured connection, due to the connection orientation. Thus, this paper chose to use only integrity

A. Data Formats and Encodings: JSON and CBOR

In our context, the term data format refers to language-independent data-interchange formats that offer a common syntax and scheme for different data types and structures, and allow a mutually agreed representation of structured data. The JavaScript Object Notation (JSON) data format "is a lightweight, text-based, language-independent data interchange format" [14]. JSON offers a widespread usage, being one of the "two [..] most common serialisation formats" [15]. It is standardized [16]. However, JSON is a textual-based format, binary data must be encoded - most likely with one of the Base-N encodings specified in RFC4648 [17] .

Another approach is offered by the Concise Binary Object Representation (CBOR) [18]: this format is a completely binary data format, that enables different structures and data types like JSON, by offering "extremely small code-size, fairly small message size, and extensibility without the need for version negotiation" [18]. In principle, CBOR features almost the same structure and types as JSON. In contrast to the text-based JSON format, CBOR does not require the encoding of binary data: it can be directly embedded as byte string.

```
{ "protected":{<plain headers>},
  <plain payload>,
  "signature":"<base64url-encoded signature>" }
```

Fig. 1. JSS structure and syntax.

```
COSE_SIGN_MESSAGE = #6.nnn [
  protected: bstr
  unprotected: {
    * label => value  },
  payload: bstr
  signature: COSE_Signature ]
```

Fig. 2. Structure and syntax of a COSE signature message.

B. Existing Message Format: JSS

The JSON Sensor Signatures (JSS) [19] message format was developed to allow "End-to-End Integrity Protection from Constrained Device to Internet of Things (IoT) Application" [19]. It was also used and deployed within a trial of the EU research project RERUM [20]. We follow the design goals for JSS, which have been defined in [19] as follows:

1) keep data in signed messages still accessible to the IoT value chain in the same manner as if it would not have been signed
2) keep JSON's simplicity, as it might be the reason why JSON has seen widespread adoption especially higher up in the IoT data chain
3) limit the increase in message size due to meta-data
4) signature is generated/verified on constrained devices

The underlying data format for JSS is JSON. The overall structure and syntax of a JSS message is depicted in Lst. 1. To keep "the level at which the payload's values reside" [19] and to meet the first design goal's requirement, a JSS payload is embedded in an enveloped manner. Thus, the payload is enclosed between a protected header map and the signature. The parameters of the protected header are defined to follow the JSON Web Algorithms specification [21]. The signature itself is embedded as a Base64URL (RFC4648 [17] , section 5) encoded string, as JSON requires binary data to be encoded.

C. Existing Message Format: COSE

The CBOR Object Signing and Encryption (COSE)[22] specification "describes how to create and process signature, message authentication codes and encryption using CBOR for serialization" and "additionally specifies how to represent cryptographic keys using CBOR." [22]. The overall goal for COSE is described to define a specification, that is "designed for small code-size and small message size" [22].

The CDDL structure definition of a COSE signature message is depicted in Fig. 2. The surrounding structure is a CBOR array, thus the different elements of a COSE signature message are identified by their position in this array. The first two elements are two header buckets. One protected, and one unprotected header, both containing "information about content, algorithms, keys, or evaluation hints" [22] related to the message. The former will be included in the signing procedure, the latter not. Both headers are defined to be a CBOR map, representing their entries as key-value pairs. These entries feature a completely numerical representation and are defined as *COSE Common Header Parameters* in table 2 of the COSE specification [22]. The protected header is serialised and included as a CBOR byte string.

Similar to JSS, the content or payload(s) of a COSE signature message is included in between the headers and the signature as a byte string. The signature is represented within a `COSE_Signature` structure and is structured as an array. The `COSE_Signature` structure contains the two formerly described header buckets in the same manner as a COSE signature message. The signature itself is embedded as a CBOR byte string, as no further encoding is required due to the binary CBOR format.

III. New Message Format and Encodings

We see two main requirements for message formats and encodings in IoT environments: compactness and compatibility. For obvious reasons, the demand for a compact message size is paramount. This particularly applies to an efficient embedding of the binary signature, but also to a minimal representation of a message's headers and attributes, as well as its structure. However the compatibility also wields influence on an efficient application of signed messages in the IoT. The IoT consists of a variety of different systems and actors. Thus, the introduction of a message format should not impose the need to rewrite an application and should "keep the original data accessible by legacy parsers" [19]. In regards to these two requirements, both, JSS and COSE offer certain assets, but also feature some drawbacks. On the one hand, JSS features more compatibility, but lacks in the compactness of a message's size. On the other hand, the COSE signature message offers a minimal message size, but a CBOR and COSE ecosystem is required to understand the plain data, thus lacking in compatibility.

We propose a new a JSON-CBOR *HYBRID* format, which combines the benefits of both formats: a JSON structure for the payload in which a CBOR structure for the signature meta data is embedded. The JSON part of this format serves as a surrounding container for the payload. This preserves the compatibility of JSON by simultaneously facilitating JSS' design goal to "keep data in signed messages still accessible to the IoT value chain in the same manner as if it would not have been signed" [19]. The inner CBOR structure acts as container for everything related to the signature, as well as the signature itself. The structure and syntax of this CBOR object is based on the COSE specification to feature its compact structure and syntax. Due to its binary encoding, the CBOR object is Base64URL encoded to allow the integration into the surrounding JSON message.

These principles result in a format that offers the compatibility of JSS on the one hand, and the compactness of CBOR structures and the COSE syntax on the other. Consequently, the payload of a signed message remains processable by entities that only support JSON, whereat support for CBOR is only required for entities that need to verify or sign a message.

These foundations combined with the design goals of JSS from [19] (see Sect. II-B) and COSE's principle of compactness were the basis for the design of a new HYBRID format. Figure 3 depicts the JSON structure of the HYBRID format. The surrounding JSON structure is realised as a JSON map. By embedding the payload without any encapsulation or

```
{ <plain payload>,
  "sig":"<base64url CBOR-signature-object>" }
```

Fig. 3. Basic JSON structure of the HYBRID format.

```
SignatureObject = [
  protected : bstr,
  ? unprotected : HeaderMap,
  signature : bstr  ]
HeaderMap = {
  * HeaderLabel => HeaderValue  }
```

Fig. 4. Structure of the CBOR signature object and the HeaderMap of the HYBRID format.

rearrangement and by directly including the encoded CBOR signature object as a key-value pair, a flat hierarchy for the message is enabled, accomplishing the first design goal.

The embedded CBOR signature object is the central message part which contains all signature related information, as well as the signature itself. The structure of this CBOR structure is depicted in Lst. 4. The surrounding structure is defined to be an array. Alike COSE, the array's elements are in a defined order with each element being identified by its position in this array: at first the protected header, followed by the unprotected header and the signature at the end. Besides the CBOR signature object, Fig. 4 additionally includes the `HeaderMap` structure of the two headers, which is a map containing a variable amount of `HeaderLabel => HeaderValue` pairs. The unprotected header's map is directly included in the surrounding CBOR signature object. The `HeaderMap` of the protected header is encoded as a CBOR byte string and included in the CBOR signature object's array. This indirect embedding allows "the protected map to be transported with a greater chance that it will not be altered in transit" [22] and further "avoids the problem of all parties needing to be able to do a common canonical encoding" [22] for the protected header in the signing or verification process. The corresponding header parameters for the HYBRID format are adopted from the *Common Header Parameters* in table 2 of the COSE specification [22].

The HYBRID format requires the signature to be encoded due to its textual JSON data format. The associated increase of a message's size imposes a large drawback in constrained environments like the IoT. When only a minimal machine-to-machine messaging is possible usually a gateway enables the communication to the outer "not-constrained" world. In such environments we propose a binary transport compression for the HYBRID format. In principle this compression enables the direct embedding of binary data (the signature) within a compressed HYBRID message, further allowing to generically uncompress the message to the regular JSON HYBRID format. This is performed by a direct transcoding between CBOR and JSON as the underlying data formats for the compressed and uncompressed HYBRID format and vice versa. In this way, the CBOR compressed format can be used for transmissions between constrained devices or where every byte matters. The compressed message can then further be transcoded to the JSON HYBRID format, when the widespread compatibility of JSON is more important than the compactness of the message.

```
CompressedHybrid = {
  * tstr => value,
  sig => bstr }
value = { * tstr => value } //
    [ * value ] //
    simple
simple = tstr / int / bool / nil
```

Fig. 5. CBOR structure of the compressed HYBRID format.

Basically, a compressed and an uncompressed HYBRID message are composed of a map and feature the same structure - besides two key differences: the uncompressed message is a JSON map and includes the CBOR signature object as a Base64URL encoded character sting - the compressed message is a CBOR map and includes the signature object as a CBOR byte string. With the CBOR signature object being included as a byte string, no parsing or reformatting is involved when the compressed message is transcoded to its JSON counterpart. To the contrary, this CBOR byte string can directly used as input for the Base64URL encoding. The payload of the compressed and uncompressed message is substantially exactly the same - one represented with CBOR and one with JSON.

The transcoding from JSON to CBOR is straightforward. CBOR offers a counterpart for every JSON type and structure, thus enabling a linear element-by-element transcoding for the payload. The other way round - from CBOR to JSON - slightly more attention must be paid. CBOR offers some types and structures that cannot be represented with JSON. To meet this challenge, a strict definition of the compressed HYBRID format was established and is depicted in Fig. 5.

IV. Results

To evaluate the described and proposed formats towards their applicability in the IoT we set up an small environment with communication over different IoT protocols. A Zolertia RE-mote development board [23] was set up with a CoAP client from the EU-project RERUM[1] [20]. The ARM-based RE-mote hardware platform features the CC2538 ARM Cortex-M3 with a 2.4Ghz IEEE 802.15.4 radio, as well as a CC1200 RF transceiver. The RE-mote offers 512KB of programmable flash and 32KB of RAM, as well as a System on Chip (SoC) clock rate of up to 32MHz. The RERUM CoAP client already offered a REST-like CoAP interface and support for elliptic curve based signatures (curve `secp192r1`) delivered in the JSS format. We enhanced the modular Contiki firmware: we implemented a CBOR library and changed the message format to gain support for the HYBRID format and for COSE. To further analyse the message formats in regards to the Message Queue Telemetry Transport (MQTT) protocol, the Contiki MQTT-Demo Client[2] was enhanced to to enable support for JSS, COSE and the HYBRID format. The source code based on Contiki has been released[24].

[1]RERUM: REliable, Resilient and secUre IoT for sMart city applications EU FP7 project: Sept. 2013 –Oct. 2015; https://ict-rerum.eu/

[2]https://github.com/contiki-os/contiki/tree/master/examples/cc2538-common/mqtt-demo

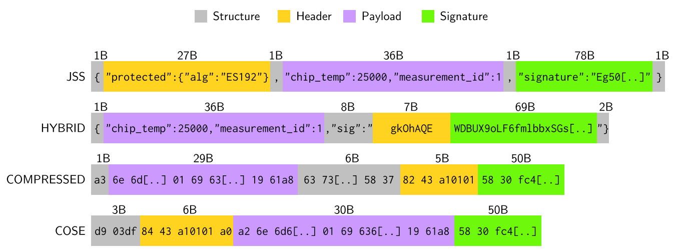
Structure
Header
Payload
Signature
JSS
1B
27B
1B
36B
1B
78B
1B
{ "protected":{"alg":"ES192"} , "chip_temp":25000,"measurement_id":1 , "signature":"Eg50[..]" }
HYBRID
1B
36B
8B
7B
69B
2B
{ "chip_temp":25000,"measurement_id":1 ,"sig":" gkOhAQE WDBUX9oLF6fmlbbxSGs[..] "}
COMPRESSED
1B
29B
6B
5B
50B
a3 6e 6d[..] 01 69 63[..] 19 61a8 63 73[..] 58 37 82 43 a10101 58 30 fc4[..]
COSE
3B
6B
30B
50B
d9 03df 84 43 a10101 a0 a2 6e 6d6[..] 01 69 636[..] 19 61a8 58 30 fc4[..]

Fig. 6. Short temperature measurement in the different formats.[3]

Format	JSS	HYBRID	COMPRESSED	COSE
bytes	145	123	91	89
Savings	-	15.17%	37.24%	38,62%

TABLE I. Message sizes.

A. Overhead in message size

To obtain a meaningful evaluation of message sizes for the different signed message formats, a real-life message was chosen as foundation: the RERUM CoAP client's chip-temp message. The payload of this message consists of the current temperature of the RE-mote's CC2538 SoC in millidegree Celsius and a numerical measurement id. The message further contains one header, indicating which signature and hashing algorithm is used, as well as the signature itself. This signature is processed from the payload and the header utilising SHA256 as hash function and the ECC curve `secp192r1` for signature generation.

The respective messages for each evaluated format are depicted in Fig. 6. The different parts (structural indicators, header, payload and signature) are colour coded and their byte sizes are indicated over each section. Comparing the default RERUM message in the JSS format with its HYBRID equivalent, we can see that the payload remains completely untouched, whereas the header and signature of the HYBRID message are encapsulated within the Base64 encoded signature object at the end of the message. Despite both formats require the signature to be encoded, the JSS header and signature require more bytes than the encoded HYBRID signature object - 27 bytes header and 78 bytes signature result in 105 bytes for JSS, compared to 76 bytes for the HYBRID format (7 bytes header and 69 bytes signature). The remaining two messages in the COSE and the HYBRID CBOR compressed format are both depicted in hexadecimal encoding for a better visualisation of the binary data. As it is not required to encode the signature for these two binary formats, the signature's byte size is significantly decreased in comparison to JSS and the uncompressed HYBRID format. Besides this, also the header, the payload and the structural indicators feature a reduction of bytes needed for each part of the message. The total size in bytes for each format is listed in table I.

Besides the evaluation of the different format's message sizes, it is obligatory to have a look at each format's impact when the messages are transmitted in practice. Each message will be transmitted in one ore several IPv6 packets, each packet

[3] `[..]` indicates truncated bytes or base64 encoded characters to provide better readability.

Format	CoAP Packets	CoAP Bytes	CoAP Savings	MQTT Packets	MQTT ACKs	MQTT Bytes	MQTT Savings
JSS	3	316	-	5	5	755	-
HYBRID	2	237	25.00%	4	4	611	19.07%
COMPRESSED	2	205	35.12%	3	3	457	39.47%
COSE	2	203	35.75%	3	3	455	39.73%

TABLE II. TRANSMITTED BYTES FOR DIFFERENT MESSAGE FORMATS.

	Encoder		Decoder	
Format	JSON	CBOR	JSON	CBOR
`text`(bytes)	687	732	999	583

TABLE III. SIZE OF JSON AND CBOR CONTIKI OBJECT FILES.

containing additional data for protocol specific headers. Thus, the number of packets in which a message is fragmented and transmitted can cause a significant overhead. This overhead is heavily influenced by the protocols to be used, as they differ in maximum payload per packet. Each packet features a fixed 40 bytes IPv6 header. Furthermore, a MQTT message requires 20 bytes for its TCP and 2 bytes for its MQTT header, while a Constrained Application Protocol (CoAP) message demands for an 8 bytes UDP header and 9 bytes for CoAP. The total packet size for each protocol is defined by Contiki's network core configuration. As the protocol specific headers require a fixed amount of bytes, the amount of bytes left for the payload heavily depends on the total size of each packet. As a CoAP packet is defined in Contiki to be up to 121 bytes of size, and the headers are 57 bytes long, each CoAP packet can transmit up to 64 bytes of data. The MQTT packets in Contiki only allow for a maximum of 32 bytes of payload in each packet, which is defined be the `MAX_TCP_SEGMENT_SIZE 32` preprocessor `#define` in the configuration of the Contiki MQTT demo client. As the headers demand for 62 bytes, the whole packet has a total size of up to 94 bytes. Since MQTT relies on the TCP protocol, an additional Acknowledgement (ACK) is required for each transmitted packet. These ACKs only consist of an IPv6 and a TCP header, resulting in a total of 60 bytes that must be transmitted additionally in response to each packet sent.

B. Overhead in code-size

The code-size evaluation is performed in two parts: on the one hand, the byte size of the involved Contiki libraries is measured, on the other hand the absolute byte size of the RERUM CoAP Client is analysed when compiled in combination with each of the message formats. All symbols and debugging information were discarded from the firmware by the strip program from the GNU ARM embedded toolchain (`arm-none-eabi-strip`) to obtain meaningful results. The firmware's or object files' byte size is further metered by the `arm-none-eabi-size` tool. Besides the total size, this program outputs the sizes for the `text`, `data` and `bss` sections separately. The `text` sections contains everything that will be located in the devices FLASH memory. That are all functions and constant data. The `data` section contains all initialised, the `bss` section all uninitialised data or variables.

	Original	JSS	HYBRID	COMPRESSED	COSE
`text` (bytes)	63.988 -	64.565 +577	65.437 +1449	64.889 +901	64.813 +825
`data` (bytes)	2340 -	2464 +124	2496 +156	2508 +168	2536 +196
`bss` (bytes)	12.664 -	13.004 +340	13.108 +444	13.112 +448	13.032 +368
Total (bytes)	78.992 -	80.033 +1041	81.041 +2049	80.509 +1517	80.381 +1389

TABLE IV. SIZE OF CONTIKI FIRMWARE IMAGE FOR RE-MOTE.

The sizes of the JSON and CBOR libraries' object files are depicted in table III. As both apps are compiled detached and will be statically linked to the final firmware, the corresponding object files only contain bytes in the text section. All evaluated object files are under one kilobyte in size, whereas the encoders are nearly the same size for both apps, and the CBOR decoder being nearly half the size of its JSON counterpart.

To obtain an in practice evaluation on code-size, the signed message format's and libraries' impact on the RERUM CoAP client firmware size was measured. The firmware was configured to support one format at a time and further compiled, linked and striped to get a fully functional firmware image for the Zolertia RE-mote. The byte sizes depicted in table IV show the sizes of each memory section and the whole firmware image, as well as the overhead for each configuration compared to the unaltered RERUM firmware. The overhead of the different configuration's `text` section sizes roughly correlates with the libraries' encoder byte sizes in table III. The overhead of the JSS configuration is only 577 bytes, being smaller than the raw libraries' size, which is caused by some functions being removed from the unaltered firmware. The HYBRID configuration requires both the JSON and CBOR encoder, which is why it requires the most bytes for its `text` section. The `data` and `bss` section's relative overhead is larger compared to the `text` or overall overhead. This is mainly because the message format encoding apps require a certain amount of initialised variables like headers and structural elements, as well as uninitialised variables like the payload and callbacks. In total the JSS configuration offers the smallest overhead of only 1.3% or 1041 bytes. The COMPRESSED and COSE firmwares result in a slightly larger overhead, that is 1.9% or 1517 bytes for the COMPRESSED, and 1.7% or 1389 bytes for the COSE configuration. The HYBRID one ensues the largest overhead of 2.5% or 2049 bytes, which is caused by the need for two separate encoding libraries. Compared to the total firmware size of at least 78, 992 bytes, this overhead is in acceptable boundaries.

V. CONCLUSION

We have shown that a non-binary message format, while offering a nice compatibility with the existing parsers and infrastructure, has a significant impact on the message size due to suboptimal encoding. Besides the total size, the relative savings of the HYBRID format (compressed and uncompressed) and COSE in comparison to JSS are denoted. With 123 bytes and savings of 15% the HYBRID format offers a notable reduction in the message's size. The two binary formats reduce the message's size by nearly 40% or to be exact, 91 bytes for COSE and 89 bytes for the compressed HYBRID format. However, by obtaining unchanged JSON-compatibility for the message and the payload the HYBRID format allows to retain compatibility.

We have released the prototypical implementation's source code.[24] These numbers, together with other results showing that ECC based cryptography can be accelerated and fast on constrained device, are so encouraging that strong end-to-end integrity protection should become the by-design approach to secure the IoT.

Acknowledgment

H. C. Pöhls was supported by the Bavarian State Ministry of Education, Science and the Arts as part of the FORSEC research association. H. C. Pöhls has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement no 644962 (PRISMACLOUD). Some of the early work of B. Petschkuhn and H. C. Pöhls were also supported by the European Unions 7th Framework Programme (FP7) under grant agreement no 609094 (RERUM). 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] V. Miller, "Use of elliptic curves in cryptography," in *Proc. of Advances in Cryptology (CRYPTO85).* Springer, 1986, pp. 417–426.

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

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

[4] 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.

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

[6] 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.

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

[8] 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.

[9] 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.

[10] S. Meissner, D. Dobre, M. Thoma, and G. Martin, "Internet of Things Architecture IoT-A Project Deliverable D2. 1–Resource Description Specification," www.meet-iot.eu/deliverables-IOTA/D2_1.pdf [last accessed: Jan 2017], 2012.

[11] H. C. Pöhls, V. Angelakis, S. Suppan, K. Fischer, G. Oikonomou, E. Z. Tragos, R. D. Rodriguez, and T. Mouroutis, "RERUM: Building a Reliable IoT upon Privacy- and Security- enabled Smart Objects," in *Proc. of the IEEE WCNC 2014 Workshop on Internet of Things Communications and Technologies.* IEEE, 2014. [Online]. Available: http://dx.doi.org/10.1109/WCNCW.2014.6934872

[12] H. Tschofenig and T. Fossati, "Transport Layer Security (TLS) / Datagram Transport Layer Security (DTLS) Profiles for the Internet of Things," RFC 7228 (Proposed Standard), Internet Engineering Task Force, Tech. Rep. 7925, July 2016.

[13] N. Modadugu and E. Rescorla, "The design and implementation of datagram TLS," in *Proc. of NDSS'04*, 2004.

[14] T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format," RFC 7159, Internet Engineering Task Force, Tech. Rep. 7159, Mar. 2014.

[15] N. Bessis, F. Xhafa, D. Varvarigou, R. Hill, and M. Li, Eds., *Internet of Things and Inter-cooperative Computational Technologies for Collective Intelligence*, ser. Studies in Computational Intelligence. Springer, 2013, vol. 460.

[16] ECMA-International, "Standard ECMA-404: The JSON Data Interchange Format," Standard ECMA-404, ECMA International, Tech. Rep. 404, October 2013.

[17] S. Josefsson, "The Base16, Base32, and Base64 Data Encodings," RFC 4648, Internet Engineering Task Force, Tech. Rep. 4648, October 2006.

[18] C. Bormann and P. E. Hoffman, "Concise Binary Object Representation (CBOR)," RFC 7049, Internet Engineering Task Force, Tech. Rep. 7049, October 2013.

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

[20] G. Moldovan, E. Z. Tragos, A. Fragkiadakis, H. C. Pöhls, and D. Calvo, "An IoT middleware for enhanced security and privacy: the RERUM approach," in *Proc. of 8th IFIP Int. Conf. on New Technologies, Mobility and Security (NTMS 2016).* IEEE, 2016, pp. 1–5.

[21] M. Jones, "JSON Web Algorithms (JWA)," RFC 7518, Internet Engineering Task Force, Tech. Rep. 7518, May 2015.

[22] J. Schaad, "COSE: A Message Based Security Solution for CBOR," Internet Engineering Task Force, Internet-Draft draft-ietf-cose-msg-13, work in Progress. [Online]. Available: https://tools.ietf.org/html/draft-ietf-cose-msg-13

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

[24] B. Petschkuhn, "Source Code," https://web.sec.uni-passau.de/papers/2017_JSS_Code.zip or http://henrich.poehls.com/papers/2017_JSS_Code.zip, 2017.