How Little is Enough? Implementation and Evaluation of a Lightweight Secure Firmware Update Process for the Internet of Things

Silvie Schmidt[1], Mathias Tausig[1], Manuel Koschuch[1], Matthias Hudler[1], Georg Simhandl[2], Patrick Puddu[3] and Zoran Stojkovic[3]

[1]*Competence Centre for IT-Security, FH Campus Wien, University of Applied Sciences, 1100 Vienna, Austria*

[2]*Adaptiva GmbH, 1010 Vienna, Austria*

[3]*Embed-IT GmbH, 1040 Vienna, Austria*

Keywords: Bootloader, Firmware Update, Security, Internet of Things, Performance Evaluation, RIOT-OS, Cryptography, AES, Elliptic Curve Cryptography, ECDSA, Cortex M0+.

Abstract: With an ever growing number of devices connecting to each other and to the Internet (usually subsumed under the ”Internet-of-Things” moniker), new challenges arise in terms of keeping these devices safe, secure and usable. Against better judgement, a large number of such devices never gets updated after being deployed, be it from negligence, inconvenience or sheer technical challenges. And all that while a plethora of valid approaches already exists for secure wireless remote update processes for such devices. In this work, we present another approach to solve this problem, with a special focus on the ease of integration into existing systems: we try to provide the absolute bare minimum to enable a secure over-the-air update process, analyze the security of this approach, and evaluate the performance impact of the implementation. We show that our solution can deal with nearly 80% of the identified threats, with a negligible impact on practical performance in terms of processing power and energy consumption.

1 INTRODUCTION

There are many definitions of what exactly the ”Internet-of-Things” (IoT) is (and how it differs from respectively overlaps with cyberphysical systems), yet the fundamental characteristic is the presence of a (usually large) number of (usually heterogeneous) devices, communicating wirelessly with each other and the Internet. Depending on the specific application area, the data exchanged, processed and acted upon by these devices can have impact on physical entities (e.g. plant monitoring sensors, controlling the production process), human beings (e.g. smart home systems, controlling light and heating), or personal, possible sensitive, information (e.g. medical monitoring devices).

A single IoT node is essentially a microcontroller, together with some sensors and interfaces for wireless communication, usually running an embedded operating system. Recent research results (e.g. (Tweneboah-Koduah et al., 2017; Lee et al., 2017; Desnitsky and Kotenko, 2018)) have confirmed that these implementations suffer from many of the same security problems known from interconnected desktop-PCs and servers. But where in the latter case most vendors have moved to deploying patches for the most obvious problems at least during the lifetime of the system, in the IoT domain this continuous update process is still severely lacking. Not for the lack of possible solutions, there is a huge amount of research dealing with secure update processes for such devices ((Choi et al., 2016; Jain et al., 2016; Kachman and Balaz, 2017; Jurkovic and Sruk, 2014; Lee and Lee, 2017; Idrees et al., 2011; Fuchs et al., 2016; Rico et al., 2015), only to name a few). And while in some – usually tightly controlled – areas, like the automotive domain, different update solutions are already in place, there are a lot of other areas that still suffer from unpatched, vulnerable IoT components, despite workable solutions being readily available. We argue that part of this reluctance in adoption comes from the difficult, complex and often expensive integration of the existing solutions into systems designed without a secure remote software update process in mind. All of the solutions cited above either require special hardware, complex update processes, or lack practical

Schmidt, S., Tausig, M., Koschuch, M., Hudler, M., Simhandl, G., Puddu, P. and Stojkovic, Z.
How Little is Enough? Implementation and Evaluation of a Lightweight Secure Firmware Update Process for the Internet of Things.
DOI: 10.5220/0006670300630072
In *Proceedings of the 3rd International Conference on Internet of Things, Big Data and Security (IoTBDS 2018)*, pages 63-72
ISBN: 978-989-758-296-7

evaluations that would enable possible implementers to reliably estimate the impact of an actual implementation.

In this work, together with two companies, we design a secure bootloader for an embedded device, enabling secure remote firmware updates, by trying the easiest and most bare-bones process possible, in the hope that such a small, non-invasive solution might provide for easier, faster adoption. We analyze the threats that can be thwarted by this approach, as well as its impact on device performance (in terms of computing power, memory requirements and energy consumption), by using combinations and extensions of well tested existing solutions.

For this work we used an Atmel SAMR21-xpro with an Cortex-M0+ CPU (48MHz, 32kB RAM and 256k flash memory) (Atm, 2015). The Cryptographic libraries used are tinyAES[1] and micro-ecc[2]. Regarding elliptic curve cryptography (ECC) the SECG[3] standard curves were implemented and tested, as well as Bernstein's curve 25519 (in combination with Ed25519).

In Section 2 we give an overview of the most important security requirements of an embedded system, as well as of the challenges that arise during the firmware update process. We follow with the definition of our threat model and possible countermeasures. Section 3 then details our implementation, followed by a discussion of the achieved performance in Section 4. Finally, we conclude and state the still open questions concerning our solution in Section 5.

2 SECURITY REQUIREMENTS AND THREAT MODEL

The most sensitive issue concerning embedded systems security is the process of updating the firmware, since attacks on this process could render the device completely useless, or inject malicious functionality. Nevertheless, it is important to look at all arising aspects of security in embedded systems.

2.1 Security Requirements for an Embedded Operating System

The general security requirements for operating systems (OS) for embedded devices are (Kleidermacher and Kleidermacher, 2012):

- *Memory Protection* is the primary requirement for an embedded system; it can be achieved by hardware, as well as software solutions.
- *Virtual Memory* provides further security through guard pages and location obfuscation.
- *Fault Recovery* has to be assured by implemented mechanisms.
- *Guaranteed Resources* are a major concern since resource-hogging malware is still possible even when *Memory Protection* and *Virtual Memory* are enabled.
- *Virtual Device Drivers* ensure that one of the most important elements of the system are protected, i.e. the device drivers.
- *Impact of Determinism & Secure Scheduling* is a main concern of real-time embedded devices - it assures secure time partitioning.

Access Control (Kleidermacher and Kleidermacher, 2012) is another issue concerning security in embedded systems. In this context this means that applications have to be assured to get the resources needed, but have to be restricted from all the resources not needed.

Since the communication between the server and the client requires a lot of energy, the power-consumption during a firmware update is a critical issue as well (as IoT devices are mostly battery-based).

2.2 Security Requirements for a Remote Firmware Update

Integrity of the firmware has become the most important security issue for embedded devices because its protection solves various security and safety issues regarding the update process. Signing the firmware update is easy to implement, however it should never be the only security feature provided (Cui et al., 2013). Securing the integrity means to discover tampered, wrong, and incomplete firmware, i.e. any errors (intentionally or unintentionally) or failures regarding the transmission of the new firmware have to be detected, as well as information loss. Not all of these issues can be addressed by digital signatures.

Table 1 defines several requirements regarding security issues derived from the process of updating firmware. Nevertheless, we are aware that these requirements may differ slightly depending on the device and/or the specific application.

Some non-security issues have to be taken into consideration as well: how and by whom (client, server, user,...) is the update-process initiated? How often is an update necessary - on the average? Does

[1]https://github.com/kokke/tiny-AES128-C
[2]https://github.com/kmackay/micro-ecc
[3]http://www.secg.org/

Table 1: Major requirements of a secure remote firmware update process.

Authentication	The device may only accept software from a specific source.
Version Control	Only the version intended for the device shall,be accepted; this also prevents the installation of outdated software.
Code Integrity	Tampered or incomplete firmware shall not be accepted by the device.
Complete & Error-Free Transmission	After its transmission the update package has to be checked for errors and it has to be transferred completely.
Operability Check	The new firmware has to be checked if it is working as intended.
Reduced User Interaction	The user should not have to be overly involved in the update process, thereby reducing error sources and increasing usability.

each device need the same update? Are there different updates for certain devices?

In the following Subsection we use these observation to construct a formal threat model for the process of updating an embedded device's firmware, together with the appropriate countermeasures addressed by this work.

2.3 Threat & Evaluation Model

We designed a threat & evaluation model by using Microsoft's STRIDE approach[4]. All threats caused by physical access to the device were excluded for the implementation and design of this secure update process. Furthermore, the threat regarding the authentication of the server to the device is not part of this work, these issues are being treated in the near future (currently we employ implicit authentication of the server using a shared secret and the server's signature). See Table 7 in the Appendix for the entire threat model.

Analyzing this model and our solution, we find that our simple approach addresses 15 out of 19 of these threats (i.e. nearly 80%). Three of the four unaddressed threats arise from malicious physical access to the device.

3 SYSTEM DESIGN AND IMPLEMENTATION

This Section first gives an overview of the entire system including our chosen parameters, followed by a deeper discussion of the individual building blocks.

3.1 System Overview

Our prototypical system consists of a management server, storing the firmware (possible many different images, depending on the devices controlled by the server), and the IoT devices themselves that can wirelessly communicate with the server, over a basically insecure channel.

The server sends encrypted and signed firmware packages to the client(s), that can verify their integrity using the server's pre-deployed public key and decrypt them using a pre-shared symmetric key. The implemented secure bootloader on the client is responsible for these checks and, after their successful completion, installs the new firmware on the next reboot. Our main design philosophy for all parts was, as already stated in the sections above, to follow a very lean and non-intrusive approach that relies on as little additional infrastructure as possible. The following subsections now detail our chosen asymmetric algorithms (Subsection 3.2) as well as the concrete bootloader implementation (Subsection 3.3). We purposely omit the description of the update server's implementation in this work, since it depends a lot on the specific environment it is used in, and we basically impose no special requirements on this server.

After evaluations of performance, memory requirements and power consumption (as detailed in Section 4), we decided to use *ECDSA* over the 256-bit prime-field curve *secp256r1* for the signatures, *SHA-256* for hashing, and *AES128-CBC* as symmetric blockcipher primitive.

3.2 Elliptic Curve Cryptography

An elliptic curve is formed by all the tuples (x,y) satisfying the simplified Weierstrass equation $y^2 = ax^3 + bx + c$, where $a,b \in$ any finite field (Hankerson et al., 2004). For the remainder of this work we focus on prime fields $GF(p)$, containing the integers up to $p-1$, where p is prime. Thus all arithmetic in $GF(p)$ has to be done modulo p. The points on an elliptic curve, together with a so-called "point-at-infinity" serving as the identity element, form an additive group, with the operations point addition and point doubling. A single operation in the elliptic curve group requires several operations in the underlying

[4]https://msdn.microsoft.com/en-us/library/ee823878(v=cs.20).aspx

field, the exact number depending on the calculation method used and the representation of the elements. The basic building block for secure asymmetric cryptographic systems utilizing elliptic curve groups is the assumed intractability of the so-called “Elliptic Curve Discrete Logarithm Problem (ECDLP)”. Given two points P and Q on a curve, where Q resulted from adding P k-times to itself (so $Q = k * P$, an operation called “scalar multiplication”), there are no efficient methods known to determine k. It is generally agreed upon that the hardness of solving this problem for a 160-bit underlying finite field is equivalent to solving the integer factorization problem for a 1,024-bit composite number (Lenstra and Verheul, 2001; Krasner, 2004). So compared to e.g. RSA only a sixth of the bit length is needed to achieve a comparable level of security. This property makes elliptic curves especially attractive in the context of resource constrained devices, since it means smaller intermediate values to store, and also smaller signatures and messages to be exchanged (Potlapallyy et al., 2002; Ravi et al., 2002).

Basically, two widely used cryptographic primitives using elliptic curves are usually employed today: key exchange, using elliptic curve Diffie-Hellman (ECDH), and signing, using the elliptic curve digital signature algorithm (ECDSA). In this work, we only employ the latter.

In addition to an elliptic curve key pair a secure hash function H is needed, whose output is not longer than n. Algorithm 1 describes the signature generation process for ECDSA. Note that the transformation of x to an integer in step 3 can be easily done by just looking at its binary representation, regardless whether the involved field is a prime field or a binary extension field. In addition, calculations in two different finite fields have to be performed: the scalar multiplication involves computation in $\mathbb{F}_q$, but $\bar{x}$ in step 4 is calculated modulo the order n of the base point P. The entire signature generation process requires one scalar multiplication, one modular inversion and two modular multiplications, for this work in the context of 256-bit fields.

Algorithm 18 shows the verification of an ECDSA signature. As in the generation of the signature, calculations with two different moduli are also involved in the verification process. For signature verification, one modular inversion, two modular multiplications and 2 scalar multiplications are required, although the latter can be interleaved to take in fact only negligible longer than a single scalar multiplication. In this work we use the micro-ecc[5] library to deal with the ECC part of the implementation. It is small, adequately fast and easy to use, and also employs basic protection

Algorithm 1: ECDSA Signature Generation.

```
Input: Domain Parameters $D = (q, FR, S, a, b, P, n, h)$, private key $d$, message $m$,
    hash function $H$
Output: Signature $(r, s)$
 1: Select $k \in [1, n-1]$ at random
 2: $P_1 \leftarrow k * P = (x_1, y_1)$
 3: Convert $x_1$ to an integer $\bar{x}_1$
 4: $r \leftarrow \bar{x}_1 \bmod n$
 5: $e \leftarrow H(m)$
 6: $s \leftarrow k^{-1}(e + dr) \bmod n$
 7: return $(r, s)$
```

Algorithm 2: ECDSA Signature Verification.

```
Input: Domain Parameters $D = (q, FR, S, a, b, P, n, h)$, public key $Q$, message $m$,
    signature $(r, s)$, hash function $H$
Output: ACCEPT or REFUSE message
 1: if $r, s \notin [1, n-1]$ then
 2:     return REFUSE
 3: end if
 4: $e \leftarrow H(m)$
 5: $w \leftarrow s^{-1} \bmod n$
 6: $u_1 \leftarrow ew \bmod n$
 7: $u_2 \leftarrow rw \bmod n$
 8: $X \leftarrow u_1 P + u_2 Q = (x_1, y_1)$
 9: if $X = O$ then
10:     return REFUSE
11: end if
12: Convert $x_1$ to an integer $\bar{x}_1$
13: $v \leftarrow \bar{x}_1 \bmod n$
14: if $v = r$ then
15:     return ACCEPT
16: else
17:     return REFUSE
18: end if
```

against most side-channel attacks (like timing- or power-analysis), fitting well with our approach of trying to implement a usable, lightweight and mostly non-intrusive solution. Note that in the course of this work, we detected and fixed a subtle bug in this library: the library uses the technique of implicitly updating the Z-coordinate of the projective curve point when performing the scalar multiplication using the Montgomery ladder (as discussed in (Meloni, 2007), (Rivain, 2011), and (Montgomery, 1987)). This approach works fine as long as the scalar used is smaller than the order of the point it gets multiplied with. However, since all scalars are always extended to a common length by the library (to thwart side-channel-attacks on scalars of varying length), in rare edge

[5]https://github.com/kmackay/micro-ecc

cases (of which we ran into one), this leads to scalars larger than the group order, and as such wrong results of the scalar multiplication. We fixed this by transforming the scalar from the range $0 - r$ to the range $\frac{-r}{2} - \frac{+r}{2}$.

3.3 Secure Bootloader

The development of a secure bootloader in the Internet of Things has to take into account that the IoT mainly consists of devices with very constrained resources. The main parts of this bootloaders' development are:

- An update package which provides the foundations to ensure integrity, authenticity and confidentiality.
- A memory table holding all relevant data regarding the currently running firmware.

The firmware itself is embedded into an update application; both parts are based on RIOT-OS[6]. In the context of this work the actual firmware combined with the updater is simply called firmware or application.

The **bootloader's update process** starts with the validation of the update header's signature (for a description of the header see Subsection 3.3.1); if it is valid, the version number of the firmware update is checked. If the version check is passed, the signature of the payload is verified. After all these tests are completed successfully, the remainder of the update package is downloaded and saved into the update area. Finally, the encrypted payload (firmware) is decrypted on-the-fly and saved into the application area. After rebooting the new firmware is started.

The bare metal developed bootloader contains five **callback functions** in order to be able to "communicate" with the application; i.e. these functions may be called by the firmware:

- *app_getUpdatePageSize:* returns the size of one flash page.
- *app_writeUpdatePage:* writes one full flash page of data to the update area.
- *app_retrieveAppVersion:* returns the version number of the currently running firmware (as stored in the memory table). The version is stored (and returned) as a 16 bit BCD (binary-coded decimal). The lowest 4 bits hold the patch version, the second nibble the minor version and the third one the major version; e.g. 0x0123 $\rightarrow$ 1.2.3.
- *app_willAcceptUpdateWithHeader:* verifies if the image header stored in the update area can be used for a valid update. It checks if the version is higher than the currently installed one. If the field *header_signature* is not NULL, then the signature is verified against the header in the update area using a matching public key from the memory table.
- *app_verifyUpdatePackage:* verifies an update package currently stored in the update area by doing a signature verification.

[6]https://github.com/RIOT-OS/

3.3.1 The Update Package

The update package consists of two parts: the update header and the payload. The payload represents the new application. The **header** contains:

- the version of the new firmware,
- the payload size, and
- the key IDs (for signature and/or encryption).

The update package is created by a developed auxiliary tool called **MKIMG**. It is written in C++ and signs and encrypts the raw firmware (provided in binary code). Furthermore, it creates and signs the header. The final outcome is an image of the update package containing the signed header and the encrypted and signed firmware (payload).

3.3.2 Memory Partitioning

The bootloader's memory (256 KiB) is split into four areas - as shown in Table 2.

Table 2: Memory Partitioning.

Area	Start address	Size (KiB)
Bootloader	0x00000000	24
Memory table	0x00006000	1
Firmware	0x00006400	103
Update area	0x00020000	128

The **Bootloader** area contains the running bootloader and the **Firmware** area holds the currently running firmware.

The **Update area** represents the upper 128 KiB of the flash memory and saves the transmitted update package. In this area the checks for version control and signatures are executed. The **memory table** contains all informations provided by the update header as described in Section 3.3.1 plus the keys for signature and encryption.

3.3.3 Data Encoding

All data in the update header and the memory table is stored as DER encoding of custom ASN.1 types specified for this purpose. This approach was chosen in order to guarantee a clean and precise definition of the data and to ensure extensibility for future developments.

Since available ASN.1 encoding libraries are usually quite memory intensive, especially concerning dynamic memory, they are not suitable to be easily used on a microcontroller platform. Because of that, a new library *tiny-asn1* was developed which can do arbitrary DER encoding and decoding relying solely on stack memory.

4 PRACTICAL EVALUATION

All measurements were done using a single purpose RIOT-OS application on an Atmel SAMR21-XPRO evaluation board with a Cortex M0+ CPU. We used either the then current version of the master branch (between February and April 2015) or the release version 2016.04. This Section presents all our test results as well as descriptions of the respective parameters and settings. Note that we give results for more than just the parameters we decided to use in our final implementation, in order to provide comparative reference values as a decision base for different implementations.

4.1 Cryptographic Primitives

Three common ECDSA operations were tested on various curves using the *micro-ecc* package running on the micro-controller:

I Key generation

II Signature generation (with the key created in I)

III Verification of the signature created in II

Note that in our actual implementation, only operation III has to be performed on the IoT device, after receiving the updated firmware. All the other operations currently have to be performed by the update server. All operations were tested with the standard SECG curves (Brown, 2010) *secp160r1, secp192r1, secp224r1, secp256r1*, and *secp256k1*.

Furthermore, we evaluated the implications of certain optimization settings within the *micro-ecc* package, providing an easy and fast way to practically influence the performance in real-world implementations, without requiring deep knowledge of the inner workings on the implementer's side. Table 4 shows the improvements in relation to Table 3 for the 160-bit curve. Improvements for the other curves are on a similar level. The `uECC_OPTIMIZATION_LEVEL (OL)` is set to 2 by default.

The evaluation of the memory consumption (ROM/flash and RAM) by using the *micro-ecc* package is shown in Table 5 and is given in relation to the baseline without any inclusion of ECC, indicated in the first table row. The test code solely contains calls to all standard functions (key generation, signature creation, signature verification).

In order to find a blockcipher algorithm suitable for the given environment, we measured the runtime of encryption and decryption of one block of data in different modes of operations, using the RIOT-OS implementation of these primitives. The results are given in Table 6.

4.2 Power Consumption

For power consumption measurements we developed a RIOT-OS application that executes various cryptographic operations. All measurements were performed on the Atmel SAMR21-XPRO evaluation board; at the beginning of the operation a GPIO-PIN is set and at the end of the operation it is deleted, in order to be able to map the power consumption of the board to the actual operations. At the beginning of this application all used pins are set to 0.

The following GPIO-PINs are used:

```
Extension Header EXT1:

[GPIO1] = PA13 = port 1 pin 3
[GPIO2] = PA28
[SPI_SS_B/GPIO] = PA23

Extension Header EXT3:

[GPIO1] = PA15
[SPI_SS_B/GPIO] = PA08
```

The test setup for measuring the power consumption is shown in Figure 1.

A digital-trigger-oscilloscope was used in order to be able to test all algorithms separately. Due to current differences in a sub-milliampere area, the measurements were done using a shunt and a measurement amplifier to achieve significant results. Our results show that adding security using our approach results in an additional 10% to 30% higher power consumption compared to the baseline without employing any cryptographic operations.

Table 3: Comparison of ECDSA signature key generation, signature generation and signature verification for different curves, mean time in *ms* from $n = 50$ runs.

	Key Generation	Signing	Verification
secp160r1	144.3 *ms*	161.1 *ms*	170.9 *ms*
secp192r1	179.4 *ms*	194.5 *ms*	213.7 *ms*
secp224r1	262.0 *ms*	281.7 *ms*	214.0 *ms*
secp256r1	426.1 *ms*	451.0 *ms*	504.3 *ms*
secp256k1	502.5 *ms*	527.3 *ms*	554.2 *ms*

Table 4: Runtime Improvements for different compiler settings, in relation to Table 3.

	Signing	Verification
secp160r1, `OL=0`	4,446.4 *ms*	5,008.940 *ms*
Change	+2,654%	+2,860%
secp160r1, `OL=1`	145.7 *ms*	145.7 *ms*
Change	-9.8%	-13.9%
secp160r1, `OL=3`	144.1 *ms*	144.1 *ms*
Change	-10.7%	-14.8%

Table 5: Memory Consumption & RAM Requirement (micro-ecc), given in Bytes relative to the baseline implementation without any ECC.

	Image size	RAM needed
no micro-ecc	8,424B	9,072B
Only secp160r1	+12,436B	+72B
Only secp192r1	+12,464B	+72B
Only secp224r1	+12,824B	+72B
Only secp256r1	+12,600B	+72B
Only secp256k1	+12,234B	+72B
All curves `OL=1`	+15,152B	+72B
All curves `OL=2`	+15,296B	+72B
All curves `OL=3`	+15,278B	+72B
All curves Only verification	+3,576B	+0B

Figure 2 shows the results of the power consumption measured for each operation listed.

Based on these results an estimation for the runtime and energy necessity for the security functions within the update process was provided. The parameters for this estimation are:

- hardware: Atmel SAMR21-XPRO evaluation board,
- operating system: RIOT-OS,
- software elliptic curves: micro-ecc,
- cryptographic implementation AES: RIOT-OS,
- cryptographic implementation SHA-2: RIOT-OS
- update package size: 128 KiB.

For this test-setting the verification and installing an update three cryptographic functions are called:

- HASH calculation using SHA-256,

Table 6: Runtime for En- and Decryption, respectively, for different symmetric blockciphers.

Algorithm	Encryption	Decryption
AES-ECB (16 byte data)	101 *µs*	171 *µs*
AES-CBC (64 byte data)	347 *µs*	349 *µs*
AES-CCM (24 byte data, 8 byte auth data)	626 *µs*	623 *µs*
TWOFISH-128	1,137 *µs*	1,137 *µs*

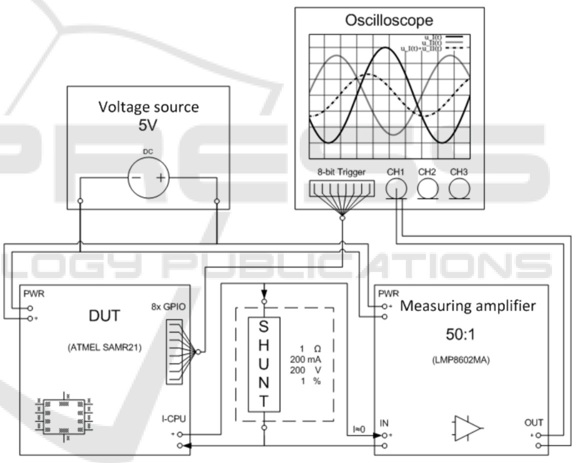

Figure 1: Test setup for power consumption measurements.

- signature verification using ECDSA over curve SECP256r1, and
- decryption using AES-CBC.

The results for this update test process are:

- runtime: 2,279 ms
- power consumption: 178.91 mJ

The part for verifying the signature is constant (498 ms, 36.6 mJ) and the remainder of the process (1781 ms, 142.26 mJ) is linearly dependent on the size of the update package.

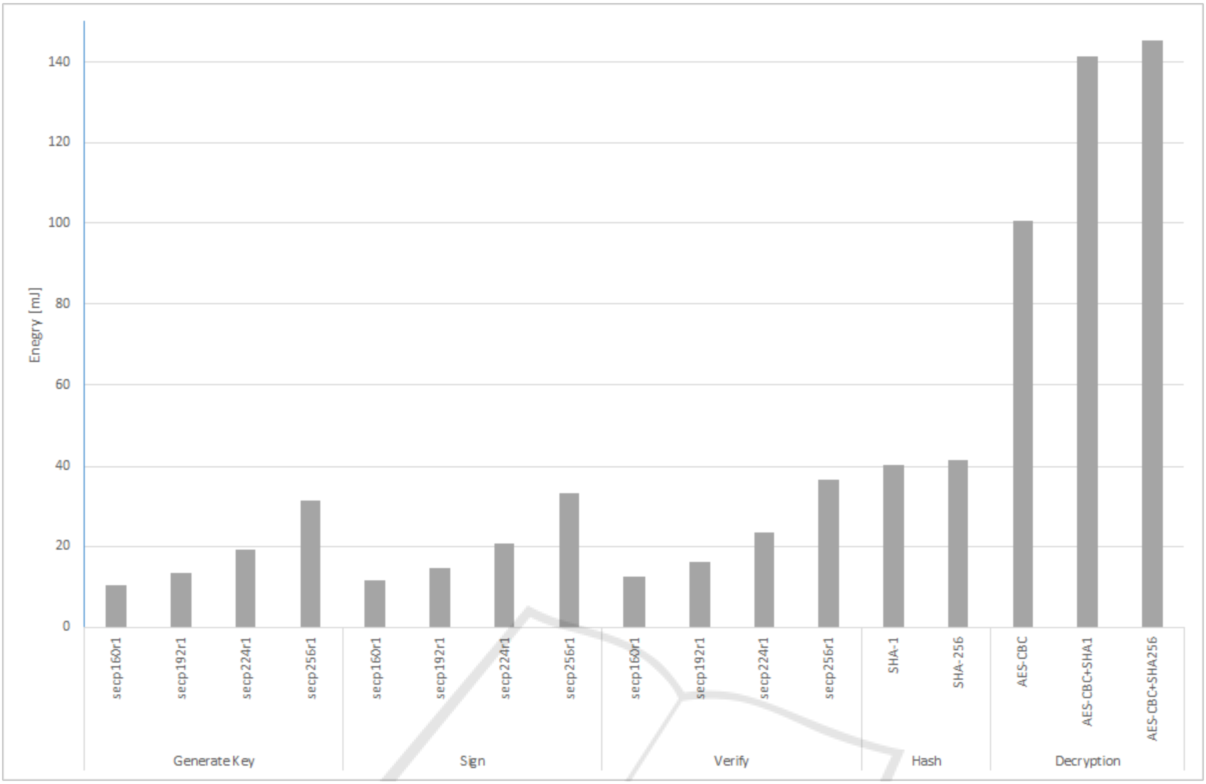

Figure 2: Energy consumption for various cryptographic operations.

5 CONCLUSIONS AND OUTLOOK

We were interested in the practical and formal evaluation of a process for remotely updating the firmware of IoT devices as easy and less intrusive as possible, requiring neither modification to COTS hardware, nor specialized update servers or secure connection tunnels. The developed bootloader is able to eliminate nearly 80% of the threats identified for this project. This result was achieved by designing a specialized update package format, by signing the header of the update package and by signing and encrypting the firmware (i.e. the payload of the update package).

The decision which cryptographic libraries to use for this purpose was dependent on various performance tests. Implementing our solution has a negligible impact on performance and power consumption (and thus battery lifetime) of the IoT device.

Future work will deal with further practical evaluations of this approach in the business context of the participating companies, as well as addressing the still open identified security threats.

ACKNOWLEDGEMENTS

This project was supported by the Vienna Business Agency.

Manuel Koschuch is being supported by the MA23 - Wirtschaft, Arbeit und Statistik - in the course of the funding programme “Stiftungsprofessuren und Kompetenzteams für die Wiener Fachhochschul-Ausbildungen”.

Gefördert von MA23 Wirtschaft, Arbeit Statistik StaDt Wien

vienna business agency

A service offered by the City of Vienna

REFERENCES

(2015). Atmel — SMART SAM R21. Technical report, Atmel Corporation, 1600 Technology Drive, San Jose, CA 95110 USA.

Brown, D. R. (2010). Recommended elliptic curve domain parameters. In *Standards for Efficient Cryptography 2 (SEC 2): Recommended Elliptic Curve Domain Parameters*. Certicom Research.

Choi, B. C., Lee, S. H., Na, J. C., and Lee, J. H. (2016). Secure firmware validation and update for consumer devices in home networking. *IEEE Transactions on Consumer Electronics*, 62(1):39–44.

Cui, A., Costello, M., and Stolfo, S. J. (2013). When firmware modifications attack: A case study of embedded exploitation. In *NDSS*. The Internet Society.

Desnitsky, V. and Kotenko, I. (2018). *Modeling and Analysis of IoT Energy Resource Exhaustion Attacks*, pages 263–270. Springer International Publishing, Cham.

Fuchs, A., Krauß, C., and Repp, J. (2016). *Advanced Remote Firmware Upgrades Using TPM 2.0*, pages 276–289. Springer International Publishing, Cham.

Hankerson, D., Menezes, A., and Vanstone, S. (2004). *Guide to Elliptic Curve Cryptography*. Springer Professional Computing. Springer-Verlag New York.

Idrees, M. S., Schweppe, H., Roudier, Y., Wolf, M., Scheuermann, D., and Henniger, O. (2011). *Secure Automotive On-Board Protocols: A Case of Over-the-Air Firmware Updates*, pages 224–238. Springer Berlin Heidelberg, Berlin, Heidelberg.

Jain, N., Mali, S. G., and Kulkarni, S. (2016). Infield firmware update: Challenges and solutions. In *2016 International Conference on Communication and Signal Processing (ICCSP)*, pages 1232–1236.

Jurkovic, G. and Sruk, V. (2014). Remote firmware update for constrained embedded systems. In *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1019–1023.

Kachman, O. and Balaz, M. (2017). Firmware update manager: A remote firmware reprogramming tool for low-power devices. In *2017 IEEE 20th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pages 88–91.

Kleidermacher, D. and Kleidermacher, M. (2012). *Embedded Systems Security: Practical Methods for Safe and Secure Software and Systems Development*. Elsevier.

Krasner, J. (2004). Using Elliptic Curve Cryptography (ECC) for Enhanced Embedded Security - Financial Advantages of ECC over RSA or Diffie-Hellman (DH).

Lee, B. and Lee, J.-H. (2017). Blockchain-based secure firmware update for embedded devices in an internet of things environment. *The Journal of Supercomputing*, 73(3):1152–1167.

Lee, Y., Lee, W., Shin, G., and Kim, K. (2017). *Assessing the Impact of DoS Attacks on IoT Gateway*, pages 252–257. Springer Singapore, Singapore.

Lenstra, A. K. and Verheul, E. R. (2001). Selecting Cryptographic Key Sizes. *Journal of Cryptology: the journal of the International Association for Cryptologic Research*, 14(4):255–293.

Meloni, N. (2007). New point addition formulae for ecc applications. In *WAIFI 2007. LNCS*, pages 189–201. Springer.

Montgomery, P. L. (1987). Speeding the pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264.

Potlapallyy, N. R., Raviy, S., Raghunathany, A., and Lakshminarayanaz, G. (2002). Optimizing Public-Key Encryption for Wireless Clients. In *Communications, 2002. ICC 2002. IEEE International Conference on*, volume 2, pages 1050 – 1056.

Ravi, S., Raghutan, A., and Potlapally, N. (2002). Securing Wireless Data: System Architecture Challenges. In *ISSS 02*.

Rico, J., Sancho, J., Díaz, Á., González, J., Sánchez, P., Alvarez, B. L., Cardona, L. A. C., and Ramis, C. F. (2015). *Low Power Wireless Sensor Networks: Secure Applications and Remote Distribution of FW Updates with Key Management on WSN*, pages 71–111. Springer International Publishing, Cham.

Rivain, M. (2011). Fast and regular algorithms for scalar multiplication over elliptic curves. iacr cryptology eprint archive.

Tweneboah-Koduah, S., Skouby, K. E., and Tadayoni, R. (2017). Cyber security threats to iot applications and service domains. *Wireless Personal Communications*, 95(1):169–185.

APPENDIX

The STRIDE threat model for the entire project comprises the following threats (the *Test passed:* results are from our implementation, as detailed in Section 3):

Table 7: STRIDE Threat Model.

	Threat	Damage	Countermeasure	Evaluation	Passed
Spoofing Identity	attacker acts as server and sends own software to the device.	device is useless since wrong software is executed.	signing the update package's payload.	sending an update package with a wrong signature.	YES
Tampering with Data	attacker intentionally manipulates the update package.				YES
	update package is damaged by accident.				YES
	incorrect transmission of the update package.			sending a manipulated update package.	YES
	incomplete transmission of the update package(due to server failure, border-router failure, node failure).				YES
	manual manipulation of the firmware due to physical access to the device.	undefined behavior of the device.	not part of this work.	programm the device via serial interface with a wrong firmware.	not part of this work.
Repudiation	the sender denies the sending of the update package.	not part of this work.	authentication of the server at the client.	send update package with wrong authentication.	not part of this work.
Information Disclosure	the version number of the firmware is manipulated,or read from the update header.	old versions of the firmware can be installed on the device (fallback).	signing the update package's header.	sending an outdated firmware.	YES
	the key IDs are manipulated or read from the update header.	conclusions about the keys can be drawn from key IDs; wrong keys could be used by the device, rendering signature and encryption useless	signing and encrypting the update package's header.	sending an update package with manipulated key IDs	YES
	the payload size is manipulated or read from the update header.	attackers can damage the update package.	signing the update package's header.	sending an update package with manipulated payload size value.	YES
	the payload is manipulated or read from the update header.	attacker retrieves data and information of customers, etc.	signing and encryption of the update package's payload.	sending an update package with a manipulated payload.	YES
	keys are being retrieved from key storage.	attacker retrieves keys for signature and encryption.	not part of this work.	not part of this work.	not part of this work.
Denial of Service	malware is sent instead of the update package.	device useless	signing the update package.	sending an update package with an invalid signature.	YES
	attacker intentionally manipulates the update package.	device useless.	signature.	sending manipulated update package	YES
	update package is manipulated or damaged by accident.			sending manipulated update package.	YES
	incorrect or incomplete data transmission.			interrupt or stop data transmission by server/client.	YES
	update package manipulated with the intention of DoS attacks.	device unreachable.	signing the update header.	sending faulty firmware.	YES
	manually deleting the firmware due to physical access to the device.	device useless.	not part of this work.	deleting the flash memory via debug interface.	not part of this work.