

Comparison of cryptographic implementations for embedded TLS

Dipl.-Phys. Andreas Walz, B.Sc. Oliver Kehret, Prof. Dr.-Ing. Axel Sikora

Institute for Reliable Embedded Systems and Communication Electronics
Offenburg University of Applied Sciences, Badstrasse24, D77652 Offenburg, Germany
{andreas.walz, axel.sikora}@hs-offenburg.de, okehret@stud.hs-offenburg.de

Abstract—The Transport Layer Security (TLS) protocol is a flexible and mature standard for securing communications over insecure networks. It is in widespread use in classical Internet applications and is increasingly considered for embedded and resource constrained applications. TLS ensures data confidentiality, integrity, and authenticity and for this purpose makes extensive use of cryptography. For an efficient and secure deployment of TLS a proper implementation of cryptographic services is crucial. Purely software-based solutions are flexible but might lack sufficient performance and security. Hardware-supported or fully hardware-based solutions reduce flexibility but potentially improve performance and security. We overview and compare various ways of implementing cryptography for embedded TLS nodes and give recommendations for corresponding design decisions.

Keywords—TLS; Embedded systems, cryptography, hardware acceleration

I. INTRODUCTION

The ever-growing pervasion of interconnected smart computer systems and Cyber Physical Systems (CPSs) in all areas of modern life requires secure communication protocols. As good candidates, the Transport Layer Security (TLS) and Datagram TLS (DTLS) protocols offer mature and standardized means for establishing secure end-to-end communication channels over insecure networks. To this end, both TLS and DTLS use strong cryptographic algorithms to achieve data confidentiality and integrity as well as to confirm the communication partner's authenticity.

However, security does not come for free and implementing (D)TLS along with the required cryptographic services is particularly challenging in resource constrained devices. This is true for at least two reasons: first, cryptography is computationally complex and can easily exhaust a small system's resources or make it unresponsive for an inadequately long time. Second, embedded systems tend to be physically exposed to potential attackers, thus opening attack vectors that can be more effective than typical network attacks.

In addition to the technical aspects of implementing security and cryptography, in many areas standardisation organisations and national legislation impose additional requirements on the

cryptographic algorithms to use and on the way they are implemented (see e.g. Smart Metering in Germany [1]).

With this paper we strive to provide an overview and a comparison of various options available for implementing cryptography for (D)TLS in small embedded systems. Our focus will be on systems whose resources are too short to run rich operating systems like Linux or popular libraries like OpenSSL. This corresponds to Class 1 devices and beyond as defined by the Internet Engineering Task Force (IETF) [11]. Popular examples in this category are small Internet-of-Things (IoT) entities like wireless sensor nodes for home and building automation, Smart Meters, etc.

The remainder of the paper is structured as follows: Chapter II introduces the TLS protocol family, while Chapter III overviews aspects related to cryptography in the context of (D)TLS. Chapter IV provides an overview of different generic types of cryptographic implementations in small embedded systems while existing implementations are overviewed and compared in Chapter V. We give design recommendations in Chapter VI and finally summarize the paper in Chapter VII.

II. THE TLS PROTOCOL FAMILY

The TLS protocol family comprises both TLS and DTLS. TLS requires reliable transport (e.g. via TCP), while DTLS [10] has been designed to support unreliable transport media (e.g. via UDP). In the following, "TLS" refers to both TLS and DTLS unless otherwise stated.

A. Protocol stack

TLS follows a client/server model with well-defined roles¹ and is composed of five sub-protocols. The fundamental protocol is the Record protocol which mainly ensures message confidentiality and integrity. Stacked on top of the Record protocol are the Handshake protocol, the Change Cipher Specification protocol, the Alert protocol, and the Application Data protocol. The first two are responsible for establishing a se-

¹ TLS (as opposed to DTLS) is connection-oriented, while DTLS is datagram-oriented. However, both TLS and DTLS establish security associations of indefinite lifetime between server and client.

cure session, while the latter two convey error messages and application data.

The specification of TLS does not fix cryptographic algorithms but introduces the concept of *cipher suites* to offer an extensible set of standardised combinations of cryptographic algorithms and schemes. Among other things, a cipher suite defines a mechanism for key establishment, authentication, as well as application data encryption and integrity.

B. Session establishment: the handshake

The first phase of a TLS connection is the handshake which typically constitutes the computationally most expensive phase. It serves the purpose of negotiating common session parameters, authenticating communication partners, and establishing shared cryptographic session keys. The latter are derived from a common *master secret* whose computation is based on a proprietary pseudo-random function (PRF) and random nonces contributed from both client and server. The PRF is based on the Keyed-Hash Message Authentication Code (HMAC) construction [2] using a hash function defined by the cipher suite.

Various options are available to establish a shared and authenticated master secret. However, the most common cipher suites use one of the following options:

- RSA is used to send an encrypted secret from the client to the server.
- The Diffie-Hellman algorithm [3] based on finite fields (DH) or elliptic curves (ECDH) [9] is used to obtain a shared secret. It is authenticated from server-side using RSA, the Digital Signature Algorithm (DSA), DSA based on elliptic curves (ECDSA), or pre-shared keys commissioned out-of-band [5][8].
- The master secret is derived from secret keys shared between the communication partners out-of-band (pre-shared keys, PSK).

Server authentication is implicit in the mechanisms described above, whereas optionally, client authentication is achieved using RSA, (EC)DSA, or PSK.

C. Application data protection

Once session keys are available from the handshake, application data can be encrypted and authenticated using symmetric cryptographic algorithms. To this end, the most common cipher suites use the AES block cipher with different key sizes and different block modes of operation. If the Cipher Block Chaining (CBC) mode is used typically a MAC based on the HMAC construction is added for making up the lack of message authentication in the CBC block mode of operation. Most cipher suites define SHA-256 as the hash algorithm for HMAC.

However, authenticated encryption schemes [29] like the Counter Mode with CBC-MAC (CCM) [11] or the Galois Counter Mode (GCM) [7] are considered more appropriate and secure by now. For both CCM and GCM block modes of

operation corresponding cipher suites using AES are available for TLS.

III. CRYPTOGRAPHY FOR TLS

Cryptographic operations contribute the lion's share to TLS' computational complexity, and thus to communication latency and energy consumption of constrained nodes using TLS. Furthermore, the security objectives to be achieved by TLS fully rely on cryptography and, thus, an inappropriate implementation of cryptography can render any otherwise secure TLS implementation completely insecure. This might even be the case due to subtle and seemingly innocuous defects in the cryptographic implementation opening e.g. a timing side-channel exploitable remotely [17]. It is, therefore, a widely accepted fact that under all circumstances one should rely on existing state-of-the-art cryptographic libraries and modules rather than designing custom solutions [19].

In addition to performance and security, developers usually need to consider numerous other figures of merit, e.g. cost, flexibility, and compliance with standards and legal requirements. Therefore, the decision on how to implement cryptographic operations should be taken serious and made bearing in mind its delicate nature and potential consequences.

A. Cryptographic requirements of TLS

Depending on the TLS cipher suites to be supported, several categories of cryptographic primitives are required for operating a TLS endpoint:

- hash functions (for PRF, MACs, and signatures),
- Diffie-Hellman (for forward-secure key exchange),
- asymmetric ciphers (for key exchange),
- asymmetric signature algorithms (for authentication),
- symmetric ciphers (for encryption),
- cryptographically secure random number generator,
- secure storage of cryptographic keys.

What algorithms actually get used in each category is mostly a matter of the application's requirements. However, the common choice for modern setups is SHA-256 as hash function and AES as a symmetric cipher [6]. A typical selection of asymmetric algorithms is given in the following section.

It is worth noting that for an efficient operation of TLS, multiple stateful instances of hash algorithms and symmetric ciphers need to persist in parallel. That is, the cryptographic implementation must provide means for keeping the state of an algorithm despite interleaved invocations of independent instances of the same or another algorithm. This requirement limits the use of cryptographic implementations for TLS that are only capable of keeping the state of one primitive at a time. Unfortunately, this is the case for most hardware-based cryptographic implementations, such that in these cases additional measures are required to multiplex cryptographic resources.

B. An example scenario

As an example, we consider a scenario where two TLS endpoints establish a secure session using ephemeral ECDH (ECDHE) for key exchange and ECDSA for mutual authentication (cipher suites TLS_ECDHE_ECDSA_* [14]). We furthermore assume that ECDSA keys for signature verification are already known to the corresponding communication partner (“raw public keys”) [13], such that there is no need to validate certificates.

Performing a TLS handshake in the given setting requires on either side (not necessarily in the given order):

- generating a random nonce of 32 bytes,
- generating an ephemeral Diffie-Hellman key pair,
- generating an ECDSA signature,
- verifying the peer’s ECDSA signature, and
- computing the shared Diffie-Hellman secret.

For simplicity, only asymmetric operations and the generation of random numbers have been listed. Please note that generating an ECDSA signature also involves generating a large random number.

When discussing example implementations in Chapter V we will give timing and energy measurements for these operations using different cryptographic implementations.

IV. TYPES OF CRYPTOGRAPHIC IMPLEMENTATIONS

From a high-level technical point of view three types of cryptographic implementations can be distinguished: software-only implementations, software implementations with hardware support, and modules implementing cryptographic services in dedicated hardware. We elaborate on these three types of cryptographic implementations in the following. A general overview is given in Table 1.

Please note that on one system multiple types of cryptographic implementations might be used simultaneously to implement different cryptographic primitives.

A. Software-only implementations

Software-only implementations are usable on almost any general processor (CPU) or microcontroller (MCU) as no specific hardware resources or additional circuitry is used. Therefore, the system may (in principle) be ported to different hardware or software platforms with minimal or even no effort. Furthermore, if the system’s firmware can be updated in the field, support of longer keys or new cryptographic algorithms may be added in case current key lengths and algorithms become insecure or more efficient implementations are available.

However, software-only implementations generally fail when it comes to the generation of cryptographically secure random numbers. In fact, even high-grade pseudo-random number generators implemented in software need an unpredictable source of entropy typically drawn from uncorrelated physical input to generate random numbers suitable for cryptographic purposes [4].

Furthermore, software-only solutions have to rely on internal or external standard memory (i.e. Flash or SRAM) for storing sensitive data like secret private keys and trusted public keys. Private keys need to be kept confidential and should not be part of the code image unless the code memory provides some read-out protection. Public keys need to be protected from unauthorized manipulation as they represent a trust relation and should not be stored within the code image unless some integrity protection is available. However, even with read-out and integrity protection in place an attacker might still leverage flaws in the software to gain access to secret and trusted keys and thereby fully break the system’s security. If external memory is used, an adversary might also attack the communication bus between CPU/MCU and memory.

Finally, unless appropriate measures are implemented, software-only solutions block the CPU/MCU for the duration of a cryptographic operation, thus potentially making the system unresponsive to an extent significantly impairing the user experience. Countering this issue by increasing CPU/MCU performance might degrade the system’s cost and energy efficiency with regard to its other tasks.

B. Software implementations with hardware support

In contrast to software-only implementations, some cryptographic services or algorithms might additionally rely on specific hardware peripherals integrated into (e.g. a cryptographic co-processor) or offloaded from the main CPU/MCU to accelerate the execution or to support e.g. the secure storage of sensitive data. Hardware acceleration can be beneficial in at least three respects: first, for retaining CPU/MCU cycles for other tasks; second, for increasing performance by utilizing specifically optimized hardware, and, third, for reducing the memory footprint, i.e. the size of the code image and the RAM usage.

Hardware acceleration integrated into CPUs/MCUs can quite often be found for symmetric cryptographic algorithms like block ciphers (e.g. AES) or hash functions. However, it is only rarely integrated for asymmetric algorithms like RSA or elliptic curve cryptography (ECC), even though constrained devices would significantly benefit from accelerating asymmetric in addition to symmetric cryptographic algorithms.

Furthermore, some hardware peripherals might be used to support the generation of cryptographically secure random numbers, e.g. independent oscillators. However, software post-processing (e.g. de-skewing and mixing) of random input gathered from multiple physical channels should be used in any case to improve the quality of the random data [4].

C. Hardware-based cryptographic modules

Certainly, the most specialised solution is based on dedicated cryptographic modules or even Trusted Platform Modules (TPMs) [21] that typically come in the form of separate Integrated Circuits (ICs) or Smart Cards. Such systems allow a very high degree of security as these devices commonly provide a tamper-resistant storage of cryptographic material and – to the greatest possible extent – a side-channel-resistant algo-

rithm execution. If public/private key pairs are generated on-chip this setup allows keeping the private key safely inside the device for all times as operations requiring the key can be executed on the device directly. Furthermore, these devices commonly hold dedicated circuits to support the generation of truly random numbers.

Hardware-based cryptographic devices typically feature specialised internal oscillators and reset circuits to counter fault injection attacks as well as tamper detection sensors to prevent sophisticated hardware attacks [24]. These systems are assumed to withstand attacks requiring reasonable knowledge and equipment.

However, depending on the capabilities of the adversary the communication between the CPU/MCU and the cryptographic module becomes the weak link that might be attacked [22][23]. Many modules use a low-level interface like I²C or SPI and a proprietary communication protocol with minimal security features. Therefore, without additional measures, an adversary might eavesdrop on unsecured communication between the CPU/MCU and the cryptographic module or might manipulate the conveyance of the output of an algorithm, e.g. the answer to a signature verification request.

To counter such attacks, some communication standards and modules consider an authenticated and encrypted channel between the CPU/MCU and the cryptographic module [25]. This, however, in turn implies the need for additional cryptographic services available to the CPU/MCU to be able to mutually authenticate the CPU/MCU and the module and to encrypt and authenticate exchanged data.

TABLE 1: PROPERTIES OF DIFFERENT TYPES OF CRYPTOGRAPHIC IMPLEMENTATIONS: “++” (BEST), “+”, “o”, “-”, AND “- -” (WORST)

	Software-only	Software with hardware support	Hardware crypto modules
Flexibility	++	-	--
Cost	++	o	--
Performance	o	+	+
Energy efficiency	o	+	+
Security (against software attacks)	-	o	++
Security (against hardware attacks)	--	o	++

V. EXISTING CRYPTOGRAPHIC IMPLEMENTATIONS

For the sake of comparison and for guiding the selection we give an overview of suitable existing cryptographic implementations and – following that – present a more detailed evaluation of the performance and energy efficiency of some of them.

A. Overview

There is a plethora of cryptographic implementations and the following overview – though listing only implementations suitable for small embedded systems – by no means is exhaustive.

*LibTomCrypt*² is an open source software-based cryptographic implementation written in C. It is a mature library and supports a wide range of cryptographic algorithms.

*MicroECC*³ is a small and efficient open source implementation of ECDH and ECDSA for microcontrollers written in C and optionally offering platform-specific assembler optimizations. It supports four standard elliptic curves: SECP128R1, SECP192R1, SECP256R1, and SECP384R1 [18]. As it lacks support of symmetric cryptographic algorithms *MicroECC* needs to be supplemented for operating TLS.

The *STM32 Cryptographic Library* [28] has specifically been designed by STMicroelectronics for the use on the STM32 family of microcontrollers. Depending on the hardware platform it allows to make use of cryptographic hardware acceleration. However, the library is only available as pre-compiled object code.

Furthermore, there are TLS libraries that come with their own software-based implementation of cryptography, such as *mbed TLS*⁴ or *TinyDTLS*⁵. The first is maintained by ARM[®] and supports most cryptographic algorithms used in the context of TLS. *TinyDTLS* offers implementations of AES, SHA-256, ECDSA, and ECDH. Both libraries are available as open source.

An overview of software-based cryptographic implementations is given in Table 2.

TABLE 2: OVERVIEW OF SOFTWARE-BASED CRYPTOGRAPHIC IMPLEMENTATIONS AND THE ALGORITHMS THEY SUPPORT

Name	SHA-2	AES	RSA	ECDSA	ECDH
<i>LibTomCrypt</i>	Yes	Yes	Yes	Yes	Yes
<i>MicroECC</i>	No	No	No	Yes	Yes
<i>STM32 Cryptographic Library</i>	Yes	Yes	Yes	Yes	Yes
<i>mbed TLS</i>	Yes	Yes	Yes	Yes	Yes
<i>TinyDTLS</i>	Yes	Yes	No	Yes	Yes

To support RSA operations in configurable hardware a modular exponentiation core developed in VHDL [20] is available as a scientific open source project. As it only supports RSA additional cryptographic services are required for operating TLS. However, since configurable hardware is rarely available in small embedded systems, it is only considered here for the sake of comparison.

There is also quite a number of hardware-based cryptographic devices. We list two examples in the following.

² <https://github.com/libtom/libtomcrypt>

³ <http://kmackay.ca/micro-ecc>

⁴ <https://tls.mbed.org/>

⁵ <http://tinydtls.sourceforge.net/>

The *VaultIC 460* from Inside Secure⁶ is a FIPS 140-2 certified hardware security module in a small package supporting a broad range of cryptographic algorithms and services, including RSA as well as ECDSA over arbitrary elliptic curves. The *VaultIC 460* offers several communication interfaces like I²C, SPI, and USB for connecting to the host MCU. It furthermore enables a secure communication with the host MCU using the Secure Channel Protocols SCP02 and SCP03 [25].

The *CryptoAuth ATECC508A* from Atmel⁷ is a hardware-based cryptographic device in a small package with focused support of authentication services based on the elliptic curve NIST P-256 [26] (equivalent to SECP256R1). It mainly supports ECDSA and ECDH and is available with different communication interfaces.

An overview of hardware-based cryptographic implementations is given in Table 3.

TABLE 3: OVERVIEW OF HARDWARE-BASED CRYPTOGRAPHIC IMPLEMENTATIONS AND THE ALGORITHMS THEY SUPPORT

Name	SHA-2	AES	RSA	ECDSA	ECDH
<i>Inside Secure VaultIC 460</i>	Yes	Yes	Yes	Yes	Yes
<i>Atmel CryptoAuth ATECC508A</i>	No	No	No	Yes	Yes

B. Time and Energy Measurements

In the following we present measurements of both time and energy consumption for selected cryptographic operations using different implementations. These numbers should only

be considered an estimation as they are affected by many parameters. Results are summarised in Table 4. For a comprehensive survey of the energy efficiency of various cryptographic algorithms we refer the reader to [27].

MicroECC has been evaluated on two microcontrollers, an EFM32 Cortex-M3 (EFM32-LG990F256) clocked at 14 MHz as commonly used for low-power devices and an STM32 Cortex-M4 (STM32F407) clocked at 168 MHz. The GCC ARM cross compiler has been used without any specific optimization.

The modular exponentiation core for RSA has been implemented in a Xilinx Zynq FPGA and was clocked at 100 MHz.

The *VaultIC 460* has been operated using its SPI interface clocked at 2 MHz whereas the *CryptoAuth ATECC508A* has been connected via its I²C interface clocked at 40 kHz.

For hardware-based modules the performance is measured including communication overhead. The energy consumption is determined as the average over multiple runs of the same operation. For hardware modules the total current is considered, whereas for software implementations the current drawn by the MCU beyond its current drawn in sleep mode is used. All platforms were operated at a voltage of 3.3 V.

C. Discussion

Generally, for ECDSA and ECDH the energy consumption of different software-based implementations and the *VaultIC 460* hardware module is very similar and of the order of 10 mJ, even though execution times significantly differ. However, the *Atmel CryptoAuth ATECC508A* clearly outperforms other implementations with regards to performance and energy

TABLE 4: OVERVIEW OF THE PERFORMANCE AND ENERGY CONSUMPTION OF DIFFERENT CRYPTOGRAPHIC OPERATIONS USING DIFFERENT IMPLEMENTATIONS. T AND E REFER TO THE DURATION AND ENERGY CONSUMPTION OF THE CORRESPONDING OPERATION, RESPECTIVELY. “N/A” IS GIVEN IF THE CORRESPONDING OPERATION OR MEASUREMENT IS NOT APPLICABLE.

Implementation	ECDSA		ECDH		RSA	
	Sign	Verify	Generate key pair	Compute secret	Sign/Decrypt	Verify/Encrypt
<i>MicroECC</i> on EMF32 @ 14 MHz	Curve: SECP256R1 / NIST P-256					
	$T \approx 1263$ ms $E \approx 11.5$ mJ	$T \approx 1373$ ms $E \approx 12.0$ mJ	$T \approx 1076$ ms $E \approx 9.8$ mJ	$T \approx 1078$ ms $E \approx 9.6$ mJ	N/A	
<i>MicroECC</i> on STM32 @ 168 MHz	Curve: SECP256R1 / NIST P-256					
	$T \approx 114$ ms $E \approx 9.8$ mJ	$T \approx 128$ ms $E \approx 11$ mJ	$T \approx 97$ ms $E \approx 8.3$ mJ	$T \approx 99$ ms $E \approx 8.5$ mJ	N/A	
Modular exponentiation core	N/A		N/A		Key size: 1024 bit	
					$T \approx 33$ ms	$T \approx 1.7$ ms
					Key size: 1536 bit	
<i>Inside Secure VaultIC 460</i>	Curve: SECP256R1 / NIST P-256					
	$T \approx 309$ ms $E \approx 7.0$ mJ	$T \approx 428$ ms $E \approx 9.8$ mJ	N/A		$T \approx 74$ ms	$T \approx 2.8$ ms
<i>Atmel CryptoAuth ATECC508A</i>	Curve: SECP256R1 / NIST P-256					
	$T \approx 120$ ms $E \approx 1.0$ mJ	$T \approx 77$ ms $E \approx 0.8$ mJ	$T \approx 215$ ms $E \approx 2.0$ mJ	$T \approx 203$ ms $E \approx 2.0$ mJ	N/A	

⁶ <http://www.insideseecure.com/Products-Technologies/Secure-Solutions/VaultIC460>

⁷ <http://www.atmel.com/devices/ATECC508A.aspx>

efficiency by a factor of up to ten.

VI. DESIGN RECOMMENDATIONS

Clearly, there is no solution that is universally optimal. Selecting a cryptographic implementation means balancing performance, security, cost, flexibility, and other figures of merit. Every solution has drawbacks and weaknesses whose acceptability depends on the threats the system is expected to be facing. Therefore, a thorough analysis of potential threats to and attacks on the system along with an identification of possible attack vectors is an essential prerequisite to a sound standing solution.

For cost-sensitive systems with low or medium security requirements software-based implementations may be the appropriate choice. For highly security-sensitive applications, dedicated hardware-based cryptographic modules should be used. However, such devices also constitute an interesting option for systems where a powerful microcontroller would be needed for the mere purpose of boosting cryptographic performance.

We give some additional design recommendations regarding cryptography in the following:

- Do not use custom cryptographic implementations but rather rely on existing and mature cryptographic libraries and devices.
- Beware of incautious optimizations that affect the implementation of cryptography.
- Try to avoid storing sensitive cryptographic keys in general purpose memory.
- When trading in security features for resource and cost efficiency do it with great care.
- Define and use a software interface that decouples the cryptographic implementation from the modules using it. This will minimise interdependencies and simplify adaptations in the future.

VII. SUMMARY

The TLS protocol family can help securing network communication for sensitive applications using strong cryptography. Even though using cryptographic services in resource constrained systems is non-trivial, there are plenty of solutions and options. We presented an overview of various cryptographic implementations and gave measurements of their performance and their energy consumption.

ACKNOWLEDGMENT

We are grateful for the contributions from our students Lucas Oliver Wagner and Maurice Billmann.

REFERENCES

- [1] Bundesamt für Sicherheit in der Informationstechnik, “BSI TR-03109-1: Anforderungen an die Interoperabilität der Kommunikationseinheit eines intelligenten Messsystems”, Version 1.0
- [2] The Internet Engineering Task Force, “RFC2104: HMAC: Keyed-Hashing for Message Authentication”

- [3] The Internet Engineering Task Force, “RFC2631: Diffie-Hellman Key Agreement Method”
- [4] The Internet Engineering Task Force, “RFC4086: Randomness Requirements for Security”
- [5] The Internet Engineering Task Force, “RFC4279: Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)”
- [6] The Internet Engineering Task Force, “RFC5246: The Transport Layer Security (TLS) Protocol Version 1.2”
- [7] The Internet Engineering Task Force, “RFC5288: AES Galois Counter Mode (GCM) Cipher Suites for TLS”
- [8] The Internet Engineering Task Force, “RFC5489: ECDHE_PSK Cipher Suites for Transport Layer Security (TLS)”
- [9] The Internet Engineering Task Force, “RFC6090: Fundamental Elliptic Curve Cryptography Algorithms”
- [10] The Internet Engineering Task Force, “RFC6347: Datagram Transport Layer Security Version 1.2”
- [11] The Internet Engineering Task Force, “RFC6655: AES-CCM Cipher Suites for Transport Layer Security (TLS)”
- [12] The Internet Engineering Task Force, “RFC7228: Terminology for Constrained-Node Networks”
- [13] The Internet Engineering Task Force, “RFC7250: Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)”
- [14] The Internet Engineering Task Force, “RFC7251: AES-CCM Elliptic Curve Cryptography (ECC) Cipher Suites for TLS”
- [15] The Internet Engineering Task Force, “Draft: TLS/DTLS Profiles for the Internet of Things”
- [16] The Internet Engineering Task Force, “Draft: A Hitchhiker’s Guide to the (Datagram) Transport Layer Security Protocol for Smart Objects and Constrained Node Networks”
- [17] D. Boneh, D. Brumley: “Remote timing attacks are practical,” in proceedings of the 12th Usenix Security Symposium, 2003
- [18] Certicom Research, “SEC 2: Recommended Elliptic Curve Domain Parameter”, Version 1.0, 2000
- [19] European Union Agency for Network and Information Security, “Security and Resilience of Smart Home Environments”, 2015
- [20] G. Ottoy, B. Preneel, J.-P. Goemaere, L. De Strycker, “Flexible Design of a Modular Simultaneous Exponentiation Core for Embedded Platforms,” in Reconfigurable Computing: Architectures, Tools and Applications, pp. 115 – 121, 2013
- [21] Trusted Computing Group, “TPM Main Specification Level 2 Version 1.2”, 2011
- [22] K. Kursawe, D. Schellekens, B. Preneel, “Analyzing Trusted Platform Communication,” in ECRYPT Workshop, CRASH Cryptographic Advances in Secure Hardware, 2005
- [23] J. Winter, K. Dietrich: “A Hijacker’s Guide to Communication Interfaces of the Trusted Platform Module,” in Computers & Mathematics with Applications, 2013
- [24] D. Karaklajic, J.-M. Schmidt, I. Verbauwhede: “Hardware Designer’s Guide to Fault Attacks”
- [25] GlobalPlatform, “GlobalPlatform Card Technology, Secure Channel Protocol 03, Card Specification v 2.2 – Amendment D”, Version 1.1, 2009
- [26] National Institute of Standards and Technology: “Recommended Elliptic Curves for Federal Government Use”, 1999
- [27] D. Singelee, S. Seys, L. Batina, I. Verbauwhede: “The Energy Budget for Wireless Security: Extended Version”, Cryptology Eprint Archive, 2015
- [28] STMicroelectronics: “STM32 Cryptographic Library, User manual”
- [29] Shawn Fitzgerald, iSEC Partners Inc: “An Introduction to Authenticated Encryption”