IoT4CPS – Trustworthy IoT for CPS

FFG - ICT of the Future

Project No. 863129

Deliverable D3.5
Guidelines and recommendations for the use of cryptography to build trustworthy IoT applications

The IoT4CPS Consortium:
AIT – Austrian Institute of Technology GmbH
AVL – AVL List GmbH
DUK – Donau-Universität Krems
IFAT – Infineon Technologies Austria AG
JKU – JK Universität Linz / Institute for Pervasive Computing
JR – Joanneum Research Forschungsgesellschaft mbH
NOKIA – Nokia Solutions and Networks Österreich GmbH
NXP – NXP Semiconductors Austria GmbH
SBA – SBA Research GmbH
SRFG – Salzburg Research Forschungsgesellschaft
SCCH – Software Competence Center Hagenberg GmbH
SAGÖ – Siemens AG Österreich
TTTech – TTTech Computertechnik AG
IAIK – TU Graz / Institute for Applied Information Processing and Communications
ITI – TU Graz / Institute for Technical Informatics
TUW – TU Wien / Institute of Computer Engineering
XNET – X-Net Services GmbH

© Copyright 2019, the Members of the IoT4CPS Consortium

For more information on this document or the IoT4CPS project, please contact:
Mario Drobics, AIT Austrian Institute of Technology, mario.drobics@ait.ac.at

Document Control

Title: Guidelines and recommendations for the use of cryptography to build trustworthy IoT applications
Type: Public
Editor(s): Sebastian Ramacher
E-mail: sebastian.ramacher@iaik.tugraz.at
Author(s): Sebastian Ramacher, Katharina Pfeffer, Martin Matschnig
Doc ID: D3.5

Amendment History

Version	Date	Author	Description/Comments
v0.0	21.01.2019	IAIK – TU Graz	Added initial structure and cryptographic primitives
v0.1	27.02.2019	SBA	Added guidelines for design of cryptographic APIs
v0.2	09.05.2019	IAIK – TU Graz	Extended cryptographic primitives
v0.3	27.05.2019	SAGÖ	Added guidelines for implementing cryptographic primitives in hardware
v1.0	29.05.2019	IAIK – TU Graz	Added introduction and executive summary
v1.1	31.05.2019	Stefan Jaksic (AIT)	Post-Review updates

Legal Notices
The information in this document is subject to change without notice.
The Members of the IoT4CPS Consortium make no warranty of any kind with regard to this document, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The Members of the IoT4CPS Consortium shall not be held liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

The IoT4CPS project is partially funded by the "ICT of the Future" Program of the FFG and the BMVIT.

Federal Ministry
Republic of Austria
Transport, Innovation
and Technology

FFG
Forschung wirkt.

Content

Executive Summary

This deliverable gives a short overview of cryptographic schemes that ensure confidentiality, authenticity and integrity for secure communication in IoT applications. This overview also includes a discussion of the typical APIs that cryptographic libraries offer users and are used during the National Institute of Standards and Technology (NIST) competitions to standardize the post-quantum secure digital signature, public-key encryption and key exchange schemes as well as lightweight authenticated encryption schemes.

For secure implementations on both on the software as well as the hardware side, this deliverable introduces guidelines and recommendations for implementing cryptographic schemes. For the software side, those guidelines focus on the design of the APIs of cryptographic libraries to make them easily accessible to application developers, but also to ensure their correct usage. On the hardware side, the deliverable introduces guidelines for the secure implementation of lightweight symmetric encryption algorithms on Field Programmable Gate Arrays (FPGAs), so that protocols for secure communication can also be deployed on resource-constrained devices and legacy hardware.

1. Introduction

Buildings trustworthy IoT applications relies on the capability of the IoT device to securely communicate with each other or also with central services. One of the major protocols that provide a secure communication channel such that both authenticity and confidentiality of the transferred data can be ensured is Transport Layer Security (TLS) [11]. Due to the general-purpose nature of the protocol, its state machine is rather complex. It also offers a large variety of choices for the selection of cryptographic primitives. Those observations combined often render the full protocol too expensive to implement on resource-constrained devices. Especially since the full protocol stack also involves public key infrastructure (PKI), which adds another layer of complexity on top, e.g. parsing and validating of X.509 certificates [14]. Yet, the security properties guaranteed by TLS are of paramount importance for building a trustworthy infrastructure.

Therefore, for the case of devices with very limited resources, we focus on the lower-level cryptographic primitives that are employed in TLS and the surrounding infrastructure. We derive guidelines for their secure implementation in hardware, especially on FPGAs. In particular, we focus on lightweight primitives including lightweight authenticated encryption schemes. Thereby we may also deploy new algorithms on legacy hardware without disrupting their runtime guarantees, but still improve trustworthiness of IoT applications that rely on resource-restricted hardware.

For devices, that are more powerful, and are capable of running a typical TLS stack in software, we focus on a new property of the protocol that was introduced in TLS 1.3: zero round trip time (0-RTT) key exchange. This feature enables TLS clients to immediately start sending encrypted data to the TLS server without completing the full handshake first. Thereby, the additional latency introduced by the handshake of TLS can be significantly reduced. However, in the current version of the protocol, 0-RTT key exchange is only possible after at least one successful connection between the client and the server and additionally requires them to store some state in the form of a shared secret. In an IoT setting, where we have servers powerful enough to enable TLS and potentially weak TLS clients, puncturable key encapsulation schemes facilitate the implementation of 0-RTT key exchange without storing shared secret data on the devices. For the successful integration of this technique in TLS, we provide an implementation of a puncturable encryption scheme as library. Thus, we also derive guidelines for the implementers of cryptographic schemes to ensure that they can safely be used by software developers and integrators.

The deliverable is structured as follows. In Section 2, we give an overview of cryptographic schemes that are useful to implement TLS or protocols with similar security properties as well as the surrounding infrastructure such as Public-Key-Infrastructure (PKI). Additionally, we also discuss typical APIs that are used to implement libraries implementing those schemes. In Section 3, we discuss usability aspects of those APIs and derive guidelines for implementers when designing APIs for cryptographic libraries. Finally, in Section 4, we focus on the aspects related to hardware implementation of cryptographic schemes, especially of lightweight authenticated encryption schemes. Similar to the usability aspects, we also derive guidelines for hardware implementers.

2. Overview of Cryptographic Schemes

In this section we give a short overview of cryptographic schemes such as signature schemes, public key encryption, key exchange as well as authenticated encryption schemes. We particularly focus on their syntax and APIs of their implementations as part of library providing access to concrete schemes.

Digital signatures allow anyone to publicly verify the authenticity of messages. Thereby, they are a standard tool for implementing, i.e. PKI or secure software distribution. Formerly, a digital signature scheme is a triple (KeyGen, Sign, Verify) of probabilistic polynomial-time algorithms, which are defined as follows:

- KeyGen(1^κ): This algorithm takes a security parameter κ as input and outputs a secret (signing) key sk and a public (verification) key pk with associated message space M.
- Sign(sk, m): This algorithm takes a secret key sk and a message m in M as input and outputs a signature σ.
- Verify(pk, m, σ): This algorithm takes a public key pk, a message m in M, and a signature σ as input, and outputs a bit b indicating whether the signature is valid.

We require a signature scheme to be correct: for all security parameters κ, for all key pairs (pk, sk) generated using KeyGen(1^κ), and signatures for any message m in M produced using sk verifies with pk, i.e. Verify(pk, m, Sign(sk, m)) = 1. Additionally, we require signature schemes to be unforgeable, meaning that even if an adversary can observe signatures on arbitrary messages with respect to some public key pk, the probability that the adversary can produce a signature on a new message that verifies is negligible.

In the currently ongoing post-quantum cryptography project by NIST [9], the API for digital signature schemes is defined in the following way:

```
#define CRYPTO_SECRETKEYBYTES /* size of secret keys */
#define CRYPTO_PUBLICKEYBYTES /* size of public keys */
#define CRYPTO_BYTES /* maximal size of signatures */

int crypto_sign_keypair(unsigned char* pk, unsigned char* sk);
int crypto_sign(unsigned char* sm,
             unsigned long long* smlen,
             const unsigned char* m,
             unsigned long long mlen,
             const unsigned char* sk);

int crypto_sign_open(unsigned char* m,
                     unsigned long long* mlen,
                     const unsigned char* sm,
                     unsigned long long smlen,
                     const unsigned char* pk);
```

Note that with this type of API that there is no place to specify the security parameter and hence the implementation uses a fixed parameter. The verification function is defined in terms of an open function which returns the message only if the signature verifies successfully. Consequently, the message needs to be embedded in the signature passed to the function. All functions return 0 on success, and a non-0 error code otherwise.

A public key encryption scheme enables senders to encrypt messages with respect to the recipient's public key. The message cannot be decrypted by anyone who does not possess the matching private key, who is thus presumed to be the owner of that key and the person associated with the public key, thereby ensuring the confidentiality of the message intended for the recipient. Formally, a public key encryption scheme is a triple (KeyGen, Enc, Dec) of PPT algorithms such that:

- KeyGen(1^κ): This algorithm takes a security parameter κ as input, and outputs the secret (decryption) key sk and public key pk with associated message space M.
- Enc(pk, m): This algorithm takes a public key pk, a message m in M as input, and outputs a ciphertext C.
- Dec(sk, C): This algorithm takes a secret key sk and a ciphertext C as input, and outputs a message m in M or a failure symbol.

We say that an encryption scheme is perfectly correct if for all security parameters κ, for all key pairs (pk, sk) generated using KeyGen(1^κ), and ciphertexts for any message m in M encrypted with respect to pk, decrypt with sk to m. It is called correct if a negligible decryption error may occur. For encryption schemes we require indistinguishability under chosen plaintext attacks, meaning that an adversary cannot decide which message is actually contained in a ciphertext even when allowed to choose two challenge messages.
The NIST PQC project requires the following API for public key encryption schemes:

```
#define CRYPTO_SECRETKEYBYTES /* size of secret keys */
#define CRYPTO_PUBLICKEYBYTES /* size of public keys */
#define CRYPTO_BYTES /* maximal size of ciphertexts */

int crypto_encrypt_keypair(unsigned char *pk, unsigned char *sk);
int crypto_encrypt(unsigned char *c,
                   unsigned long long *clen,
                   const unsigned char *m,
                   const unsigned long long mlen,
                   const unsigned char *pk);

int crypto_encrypt_open(unsigned char *m,
                        unsigned long long *mlen,
                        const unsigned char *c,
                        unsigned long long clen,
                        const unsigned char *sk);
```

Like in the previous case, the API fixes a security parameter.

For 0-RTT key establishment, a key-encapsulation mechanism (KEM) to transport a symmetric encryption key instead of full-blown public-key encryption suffices. A KEM is a triple (KeyGen, Enc, Dec) of PPT algorithms s.t.:

- KeyGen(1^κ): This algorithm takes a security parameter κ as input, and outputs the secret (decryption) key sk and public key pk with associated key space K.
- Enc(pk): This algorithm takes a public key pk as input, outputs a ciphertext C and a symmetric key K.
- Dec(sk, C): This algorithm takes a secret key sk and a ciphertext C as input, and outputs a symmetric key K or a failure symbol.

We require for KEMs the same security notions as for public-key encryption and do not repeat them. Also, the API defined by NIST is very similar:

```
#define CRYPTO_SECRETKEYBYTES /* size of secret keys */
#define CRYPTO_PUBLICKEYBYTES /* size of public keys */
#define CRYPTO_CIPHERTEXTBYTES /* size of ciphertexts */
#define CRYPTO_BYTES 32 /* size of symmetric keys */

int crypto_kem_keypair(unsigned char *pk, unsigned char *sk);
int crypto_kem_enc(unsigned char *c, unsigned char *k, const unsigned char *pk);
int crypto_kem_dec(unsigned char *k, const unsigned char *c, const unsigned char *sk);
```

Furthermore, we are interested in puncturable versions of KEMs, i.e. puncturable key-encapsulation mechanisms (PKEM) as introduced by Derler et al. [8]. KeyGen of PKEMs take additionally parameters m and k, and they have a fourth PPT algorithm defined as follows:

- Punc(sk, C): This algorithms takes a secret key sk, and a ciphertext C as input, and outputs an updated secret key sk'.

As long as unpunctured secret keys are used we expect perfect correctness. If punctured secret keys are used, i.e. sk' = Punc(sk, C), we allow an decapsulation error bounded by some function μ, i.e. Dec(sk', C') with $C \neq C'$ may fail with a probability bounded by $\mu(m, k)$.

In the style of the APIs defined by NIST, we define the following additional function for KEMs:

```
int crypto_kem_punc(unsigned char *sk, const unsigned char *c);
```

Finally, as symmetric primitive, we are interested in authenticated encryption (or authenticated cipher, or authenticated encryption with associated data). The goal of such schemes is to provide one primitive to achieve confidentiality, authenticity and integrity in the symmetric setting. They also provide the possibility to authenticated associated data, i.e. public data or data known by both the sender and the receiver, such as package headers. Important protocols such as TLS 1.3 [11] continuously replace ad-hoc usage of block ciphers with a mode of operation and old stream ciphers with authenticated encryption algorithms. Formally, authentication encryption is a pair (Enc, Dec) of PT algorithms defined as follows [12]:

- Enc(sk, n, m, d): This algorithm takes the secret key sk, a nonce n, the message m, and associated data d, and outputs a ciphertext C, and a tag t.

- Dec(sk, n, C, t, d): This algorithm takes a secret key sk, a nonce n, a ciphertext C, a tag d, and associated data d as input, and outputs the message m or a failure symbol.

Note that secret keys are sampled uniformly at random from the key space without requiring an additional KeyGen algorithm. As for public-key encryption we require a correctness notion in the same sense and do not repeat it. Similarly, the confidentiality notions try to achieve the same goals as discussed for public-key encryption schemes. However, adversaries are given access to encryption oracles on unique nonces. Additionally, similar to message authentication codes, authentication encryption also requires the tags to be unforgeable, but the notions need to be slightly adapted: valid forgeries consist either of a ciphertext plus the corresponding tag for a new, not previously queried message, or of a new ciphertext with tag that was not previously returned as a result to any query.

The API for authentication encryption is defined by the CAESAR competition [10] as follows:

```
#define CRYPTO_KEYBYTES /* size of the secret keys */
#define CRYPTO_NSECBYTES /* size of the secret bytes */
#define CRYPTO_NPUBBYTES /* size of the public bytes */
#define CRYPTO_ABYTES /* maximum gap between plaintext and ciphertext size */

int crypto_aead_encrypt(unsigned char *c,
                        unsigned long long *clen,
                        const unsigned char *m,
                        unsigned long long mlen,
                        const unsigned char *ad,
                        unsigned long long adlen,
                        const unsigned char *nsec,
                        const unsigned char *npub,
                        const unsigned char *k);

int crypto_aead_decrypt(unsigned char *m,
                        unsigned long long *mlen,
                        unsigned char *nsec,
                        const unsigned char *c,
                        unsigned long long clen,
                        const unsigned char *ad,
                        unsigned long long adlen,
                        const unsigned char *npub,
                        const unsigned char *k);
```

Note that the nonces are provides as `npub`, and ciphertext and tags are combined in c. The API also provides an additional parameter `nsec` which is unused.

3. Guidelines for Designing APIs for Cryptographic Libraries

The complexity, insecure defaults, and poor documentation of cryptographic APIs are among reasons for frequent developer-induced errors in applications that subsequently lead to security incidents. It has been shown by Fahl et al. [7] and Georgiev et al. [6] that poor usability of cryptographic APIs and permissive security settings lead to numerous applications validating TLS certificates incorrectly. As a result, these applications failed at properly verifying the authenticity of the communication partner. Moreover, Egele et al. [1] discussed that usability issues of cryptographic APIs for Android are responsible for incorrect implementations of encryption and authentication schemes leading to security flaws.

We address this problem by extensively examining cryptographic APIs with regards to their usability and extract guidelines for improvement. Therefore, we initially performed an extensive literature review of empirical evaluations of cryptographic libraries and evaluations of their usability and, afterwards, systematized the results. Based on our findings we formulate the following guidelines for a usable cryptographic API which are based on work by [1],[3],[4],[5]:

- Integrate cryptographic functionality into regular APIs so developers do not have to interact with cryptographic APIs in the first place: This means that secure options of function calls (e.g., storing a password) should be offered by the standard API so that the developer does not have to include an additional API for security reasons.

- Sufficiently powerful to satisfy both security and non-security requirements: If the API is not powerful enough to fulfill the developer's needs, then developers will likely come up with their own code, which often leads to security vulnerabilities. Thus, it is suggested to interview potential API users (developers) and ask them for frequent use cases while creating APIs.

- Easy to learn, even without cryptographic expertise: The functions should be designed in a way that no cryptographic expertise is needed to use them.

- Self-explanatory function and parameter names: The name of the functions and function parameters should be expressive. For example, *function func* is not a good function name whereas *function generatePrivateKey* is self-explanatory. Meaningless parameter names such as *int i, j, k* should be avoided.

- Don't break the developer's paradigm: Developers usually have a low understanding of cryptography which is limited to certain paradigms (such as secret key encryption, public key encryption, digital signatures, etc.). An API should construct functions based on these building blocks and hide further details from the developer.

- Easy to use, even without documentation: It has been shown that developers do not read through API manuals. Consequently, it is not a good practice to expect developers to have read the manual in order to understand crucial functionality such as that a return value of 0 of a specific function speaks for an error. Generally, it is recommended to design self-explanatory functions.

- Hard to misuse. Incorrect use should lead to visible errors: Security errors are often invisible and hard to find while testing without involving adversaries, it is important to not only highlight potential incorrect usage of an API, but also make it sufficiently hard for developers to proceed with incorrect implementations (e.g., in an TLS implementation, connect to an endpoint that has no validated certificate).

- Defaults should be safe and never ambiguous: Developers often tend to use defaults instead of specifying their own parameters of cryptographic functions. These defaults have to be secure. An example for a bad practice is a cipher class, where the user can specify only the cipher name (e.g., AES) and go with the default values, when the default block cipher mode is ECB, which is well known to be prone to information leakage.

- Testing Mode: Security mechanisms often decrease performance, developers commonly test their implementations with security features disabled (or weakened). However, it has been shown that developers frequently forget to enable these features again when deploying. Thus, it is recommended that security APIs offer support for developing and testing modes.

- Easy to read and maintain code that uses it/Updatability: Security algorithms and parameters change frequently due to newly encountered attacks or weaknesses, it is important that developers can easily update their implementations in such cases. Therefore, it is recommended to never hard code security parameters (such as the number of iterations in a hash algorithm).

- Assist with/handle end-user interaction: Security errors at runtime are usually displayed to the end-users as error or warning messages. These messages commonly must be designed by the developer using the API. However, most developers lack the cryptographic background to properly phrase messages and make them understandable to end-users, which often leads to wrong security-critical decisions on the end-users' side. To overcome this problem, security APIs should offer ready-made texts for warnings and errors.

- Simplicity does promote security (to a point): Studies by Acar et al. [2] showed that simpler APIs achieved better results than complicated ones.

- Features and documentation matter for security: The usability of API documentation influences the security of the code tremendously. In particular, it is recommended to give secure code examples. Therefore, it is necessary to consider a broad range of potential tasks developers might need to accomplish and give code examples for these.

- Easy combination of low-level features/easy to extend: Since it may be difficult or impossible to predict all tasks API users may want or need, it is suggested that where lower-level features are necessary, they should be intentionally designed to make combining them securely into more complex tasks as easy as possible.

Hard to change/override core functionality: It should be hard for the developer to change core functionality (to a potentially insecure setting).

4. Guidelines for Implementing Cryptographic Algorithms in Hardware

In the current era of pervasive computing there is an ever-increasing need for security solutions that can be applied in small and cheap devices such as secure RFID tags and smart sensors. Today, design needs include the inherent support for secure communication targeted at sensible applications. Due to the tight cost constraints of these mass products, very special cryptographic solutions that are optimized for resource-constrained systems are required. Hardware properties such as memory, power, performance, area, throughput, latency and others must be considered when security solutions are applied on resource-limited hardware.

Consequently, a wide range of specialized security solutions has been developed in the recent years and standardization in the area of "Lightweight Cryptography" is currently evolving. In 2013, NIST initiated a lightweight cryptography project to study the performance of the current NIST-approved cryptographic standards on constrained devices and to understand the need for dedicated lightweight cryptography standards, and if the need is identified, to design a transparent process for standardization [13].

Within the following a very specific use case for lightweight cryptography is in focus: Legacy hardware which is already applied in the field for several years and cannot be replaced by new hardware. The challenge is to enable secure communication on an existing FPGA-based legacy device without changing any hardware. A small challenge within the current system is that the underlying protocol must support multicast telegrams that can be decoded by all participants. The FPGA in use does not offer inherent reliable security support such as modern devices like the Xilinx Zync UltraScale, or latest Intel parts. The only feature that can be directly exploited is bitstream encryption. On top in the selected system only a very small Soft-CPU-Core is available, which is already heavily loaded with its current SW tasks. Consequently, the available FPGA area, minimum throughput and maximum latency are given. The remaining degree of freedom is the optimized implementation of a suitable crypto algorithm which must fit into the remaining programmable logic.

Especially for the above-mentioned situation a set of RTL design guidelines for security IPs has been elaborated. These general recommendations shall help to implement well-structured, optimized and reusable cryptographic IPs applicable to FPGA devices.

- Modular architecture:
  A clear and well separated internal IP Structure, reflecting the main functional blocks of the underlying cryptographic algorithm, shall be established. Interfaces should be defined in reasonable simple ways leading to easy understanding by design and verification engineers and making reuse easy. Main functions shall be integrated in separated modules connected by narrow Interfaces.

- Library elements:
  Security IPs should be handled as a generic library element with a set of configuration parameters. All sub-blocks can be reused in the same IP for multiple instances.

- Vendor independence:
  The source code has no dependency on any technology specific primitives. No vendor specific cells or synthesis directives are used within the source code. Following these rules enables the use of the IP on different hardware platforms without any changes (the vendor synthesis tool chooses the optimal mapping of the primitives).

- Internal interfaces:
  Internally, the security IP core should use a simple and fast register/memory interface (simple streaming interface), which is independent of any standard protocols or interface.

- External interfaces:

The security IP can easily be extended to a standard bus protocol via bus wrappers such as AXI (AXI stream). The external communication interface shall be strictly separated from the internal IP functionality.

- Small register set:
  In order to allow fast and efficient integration of the IP, the internal register space shall be as small as possible. For example, status flags shall be grouped in one register to be readable with one single access. Same is true for interrupt related registers. Another positive effect of this measure is the reduction of the overall IP size and consequently an increase of reliability.

- Design for verification:
  The internal IP structure and its external interfaces shall be defined with (Register Transfer Level) RTL verification in mind. Whenever a SW reference model is available the modular structure of the RTL code shall be aligned to the functional structure of the SW model.

- Design for synthesis:
  In many cases FPGA synthesis is as push-button solution and can be done without any knowledge of the internals of a security IP. In case of problems, e.g. when the target frequency cannot be met or there is a lack of resources, manual interference is needed. Expressive signal names and consequent adherence to coding style conventions ease the definition of synthesis constraints.

- Scalability:
  Dependent on the configuration parameter the security IP is scalable and adaptable for resource utilization. This means that customized requirements define the usage of hardware resources on the IoT platform with a trade-off between speed and area.

- Standard cryptographic algorithms:
  All implementations of cryptographic algorithms must fully comply to defined international standards, e.g. NIST. Here the emphasis is on the term ‘full compliance’, which means that no trade-offs like area or speed against padding options are allowed. Even small deviations from the standard can open severe attack vectors which cannot be foreseen. Alternatively, suitable lightweight algorithms shall be selected and applied without any deviations from the written standard.

- Low latency and fast data throughput:
  The latency and data throughput of the security IP is important for the system’s behavior. Depending on the applied hardware architecture (memory mapped or streaming), the security IP must guarantee a deterministic turnaround time for security operations.

- Small protocol overhead:
  Whenever secure communication is added to a legacy system, additional data must be transmitted amongst the entities. The additional information should be as small as possible, especially for small data packets. Furthermore, the increase of transferred data should not

impact the real-time behavior of the system. All timing deadlines and real-time guarantees must hold for the new communication scheme.

- Master keys stored inside devices:
  Modern devices like Xilinx Zync Ultrascale and recent Intel devices offer different variants of secure key storage. Integration of specialized Security Chips with protected memories often is the preferred way. Especially in legacy hardware and FPGAs without dedicated key storage or key generation options, picking a method and location for accessing and storing keys poses a challenge, and a trade-off must be accepted. One option is to hide a master key directly within the encrypted bitstream. In case the bitstream is decoded, the master key is "lost" as well. A better, but by far more complex, option is offered by physically unclonable functions (PUFs). There are soft IPs available that can be integrated inside the FPGA fabric. In any case a pre-shared master key stored inside HW shall only be used by the security IP directly. The master key is never exchanged between entities and is never readable from the outside.

- Relieve main CPU:
  Inside resource-constrained FPGA solutions for IoT devices usually only weak processor subsystem are available. To prevent an overload of the processor core computation-intensive security operations should be allocated to the hardware.

- DPA (Differential Power Analysis) side channel hardening of security core:
  Security IPs should be protected against DPA side channel attacks. Common DPA countermeasures such as masking or randomization usually introduce significant area overhead, which may not be available. It is preferred to apply DPA hardening at protocol level. For example, changing keys with each transfer is an option.

- Random numbers:
  Wherever random numbers are used in cryptography their entropy is the crucial parameter. Bad randomness can easily be exploited for effective attacks. Very recent FPGA devices offer certified True Random Number Generators (TRNGs) which can produce random numbers with very high entropy, but legacy devices usually don't offer this option. In all cases where no external source of randomness is available a trade-off must be accepted, and random generators may be placed inside the programmable FPGA logic. Ring oscillator based TRNGs are widely used for this purpose.

5. References

[1] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, “An Empirical Study of Cryptographic Misuse in Android Applications,” in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, New York, NY, USA, 2013, pp. 73–84.

[2] Y. Acar *et al.*, “Comparing the Usability of Cryptographic APIs,” in *2017 IEEE Symposium on Security and Privacy (SP)*, San Jose, CA, USA, 2017, pp. 154–171.

[3] M. Green and M. Smith, “Developers are Not the Enemy!: The Need for Usable Security APIs,” *IEEE Security & Privacy*, vol. 14, no. 5, pp. 40–46, Sep. 2016.

[4] M. Green and M. Smith, “Developers are users too: designing crypto and security APIs that busy engineers and sysadmins can use securely” Talk at the USENIX Summit on Hot Topics in Security (HotSec’15) p. 25.

[5] P. L. Gorski *et al.*, “Developers Deserve Security Warnings, Too,” *Fourteenth Symposium on Usable Privacy and Security ({SOUPS} 2018)*p. 17, 2018.

[6] M. Georgiev, R. Anubhai, S. Iyengar, D. Boneh, S. Jana, and V. Shmatikov, “The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software,” p. 12.

[7] S. Fahl, M. Harbach, T. Muders, M. Smith, and L. Baumg, “Why eve and mallory love android: an analysis of android SSL (in)security,” p. 12.

[8] D. Derler, T. Jager, D. Slamanig, C. Striecks. “Bloom Filter Encryption and Applications to Efficient Forward-Secret 0-RTT Key Exchange.” EUROCRYPT (3) 2018. pp. 425-455.

[9] NIST. “Post-Quantum Cryptography Standardization.” online: https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization

[10] “CAESAR: Competition For Authenticated Encryption: Security, Applicability, and Robustness.” online: https://competitions.cr.yp.to/caesar.html

[11] E. Rescorla. “The Transport Layer Security (TLS) Protocol Version 1.3.” RFC 8446, 2018.

[12] M. Eichlseder. “Differential Cryptanalysis of Symmetric Primitives.” PhD Thesis, 2018.

[13] K. McKay, L. Bassham, M. S. Turan, N. Mouha. “NISTIR 8114 - Report on Lightweight Cryptography.” National Institute of Standards and Technology (NIST), Gaithersburg, 2017.

[14] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, W. T. Polk. “Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile”. RFC 5280, 2008.