IoT4CPS – Trustworthy IoT for CPS

FFG - ICT of the Future
Project No. 863129

Deliverable D3.6.1
Prototype of cryptographic library implementation

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

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: Prototype crypto implementation

Type: Software

Editor(s): Sebastian Ramacher

E-mail: Sebastian.Ramacher@ait.ac.at

Author(s): Thomas Hinterstoisser (Siemens), Martin Matschnig (Siemens), Sebastian Ramacher (AIT), Raphael Schermann (IAIK)

Doc ID: D3.6.1

Amendment History

Version	Date	Author	Description/Comments
v0.0	07.10.2019	Sebastian Ramacher	Initial structure prepared
v0.1	11.11.2019	Raphael Schermann	ISAP implementation status
v0.2	21.11.2019	Thomas Hinterstoisser Martin Matschnig	Added content for ISAP on FPGA Demonstrator
v0.3	22.11.2019	Sebastian Ramacher	BFE library implementation status
v0.4	26.11.2019	Martin Matschnig	Review
v0.5	28.11.2019	Sebastian Ramacher	Integrated review comments
v1.0	29.11.2019	Sebastian Ramacher	Final version

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

Abbreviations

FPGA	Field-Programmable Gate Array
DPA	Differential Power Analysis
COTS	Component Off-The-Shelf
ASIC	Application-specific integrated circuit
BFE	Bloom Filter Encryption
TLS	Transport Layer Security
0-RTT	0 round trip time
DPA	Differential power analysis
IND-CCA	Indistinguishability under chosen-ciphertext attacks
IoT	Internet of Things

Executive Summary

This deliverable gives a progress overview on the prototype implementations of the cryptographic primitives and protocols used as part of IoT4CPS. It will cover two implementations: First, we discuss the progress of the implementation of forward-secure 0-round trip time key exchange protocols based on bloom filter encryption. The library implements the bloom filter encryption scheme which will later be integrated as part of TLS library. Thereby, we obtain an extension of the TLS protocol that allows resource constraint devices operating as clients to send data via a secure channel to a server without waiting for the server's reply. Latency is thus significantly reduced since it is no longer necessary to wait for the full handshake to be completed.

Secondly, the progress on the implementation of the authenticated encryption scheme ISAP for FPGAs and ASICs is reported. This scheme is especially useful for constraint devices, since ISAP allows on the one hand for fast software implementations, but more importantly in the context of IoT4CPS, also for fast and compact hardware implementations. Additionally, ISAP is designed to be resistant against passive side-channel attacks such as timing attacks, differential power analysis, and also against active attacks including fault attacks. This implementation will then later be integrated into a deployment consisting of legacy hardware which is already operating for many years and which cannot be replaced by new more powerful hardware. The ISAP core will be responsible for encrypting and decrypting data streams on the fly without requiring any changes to the remaining architecture and thus adds confidentiality and authenticity to the legacy system.

1. Introduction

Cryptographic solutions play a crucial role in enabling security in modern distributed systems, such as the Internet of Things (IoT). Most importantly, they ensure secure communication between all the participants of the system even when they have to communicate over the internet or other untrusted networks. Thereby it can be ensured that all the transmitted data stays confidential against an attacker and, furthermore, the authenticity of the data is also guaranteed. To obtain these security guarantees, multiple different types of cryptographic primitives need to be integrated into the system: authenticated encryption for confidential and authentic data transfer, key exchanges to establish the key material used for the encryption scheme, and digital signatures to verify identities of the participants. Besides the standard security notions (c.f. D3.5) these schemes have to fulfill, the increasing abilities of attackers require more sophisticated schemes. From a cryptographic protocol point of view, a secure communication channel is required to be forward-secret, meaning that even if an adversary records all the network traffic and at some later point learns the involved key material of one channel, the security of all other channels remains intact, though. This feature can be achieved by building protocols based on forward-secret key exchange protocols, i.e. they produce completely fresh and independent shared keys for every execution of the key exchange protocol. In practice, also implementation-specific features become important. For example, implementations of the cryptographic schemes may suffer from side-channel attacks, i.e. attacks that are able to break the security of a system using data from the implementation such as timing data, power usage, or errors due to injected faults. Therefore, implementations are required to be resistant to these types of attacks. For example, software implementations are required to be constant-time, i.e. their runtime does not depend on any secret keys, or hardware implementations need to be resistant to power analysis and fault attacks.

These additional features come at a cost. They are more expansive in runtime, require more memory, or their hardware implementation leads to a higher hardware utilization. Therefore, efficient implementations are of particular importance to show that the primitives, protocols and schemes developed in theory, are indeed useful for applications in practice. Yet, in certain use-cases, e.g. use-cases involving strict hardware constraints, a straight-forward reference implementation of the primitives is often not enough. Instead, the implementations have to be adapted to these constraints. This often implies that implementations have to be built from the bottom up as programming paradigms differ between a typical desktop computer, smartphone or embedded devices in an IoT scenario. For example, when considering application-specific integrated circuit (ASIC) or field-programmable gate array (FPGA) implementations, a higher level of parallelism might change the performance picture completely [1]. On the other hand, a large number of constants or temporary variables become problematic when memory is scarce such as on embedded platforms.

The deliverable is structured as follows. In Section 2, we give an overview on the implementation of the forward-secret 0-RTT key exchange protocol by Derler et al. [2]. We present the current progress on the implementation of a software library in the C programming language and outline its integration in the handshake of the Transport Layer Security (TLS) protocol. In Section 3, the focus lies on the lightweight authenticated encryption scheme ISAP [7]. There, we discuss the implementation progress of an ASIC/FPGA implementation and its integration in the demonstrator.

2. C Library for Forward-Secret 0-Round Trip Time Key Exchange

Key exchange protocols are essential for establishing secure communication channels over a (potentially untrusted) network. They enable two parties, that do not share a-priori established secret-key material, to establish a shared secret. This shared secret then serves as basis to derive secret keys for authentication encryption, providing confidential and authenticated communication between the two parties. Such key exchange protocols are then integrated into more complex protocols such as Transport Layer Security (TLS) version 1.3 [5] or Secure Shell (SSH). Currently, ephemeral Diffie-Hellman (EDH) key exchange is used in these

protocols to provide forward-secure session key establishment, which is then used to derive all other keys required by the protocol.

2.1 Forward-Secret Key Exchange in TLS 1.3

In TLS 1.3, keys are established during the so-called handshake which describes the initial phase of the protocol (see Figure 1). First, a client sends a ClientHello message announcing its supported TLS versions, algorithms, etc. This message already includes the client's key share of the Diffie-Hellman key exchange protocol. When the server receives the client's message, it checks the supported algorithms and selects the suitable one. It then replies with a ServerHello message announcing the choice. The ServerHello message also contains the server's key share. The server can already compute the shared secret from the DH key exchange. The client, however, must wait until it received the server's answer. Once both parties completed the DH key exchange, the remaining handshake is already performed in an encrypted fashion. At the end of the handshake, client and server can optionally derive a shared secret that they can store and re-use for the next TLS connection between the same two parties. If the client then decides to use this shared secret on the next connection as pre-shared secret key (PSK), TLS 1.3 provides the possibility to send encrypted application data already after the ClientHello message. In this case, the PSK is used to derive the secret keys required for encryption. If one of the two parties are no longer in possession of this key, they fall back to the normal handshake.

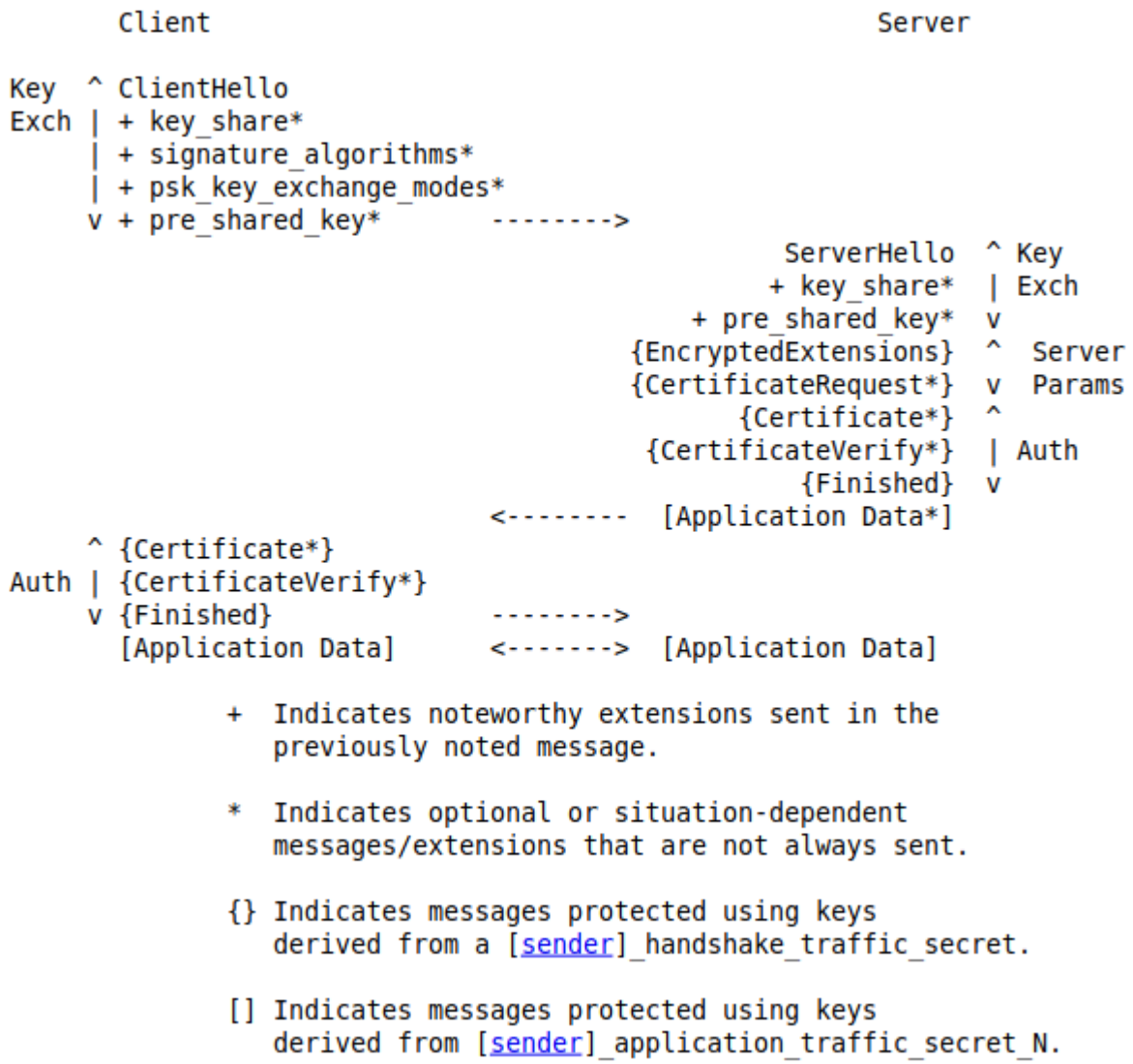

Figure 1: TLS 1.3 handshake [5]

As discussed above, EDH has the drawback that the key exchange requires one round-trip. Only after the two shares have been exchanged, the shared key can be derived and thus a secure channel be established. Recent work has explored alternatives to the PSK-based approach to reduce the number of round trips. In particular, the possibility to reduce the complexity to zero round trips using puncturable public-key encryption (PPE) or puncturable key encapsulation mechanisms (PKEM) has been investigated. In such a protocol, the client essentially encrypts a session key with respect to the public key of the server, and then sends it to the server. Therefore, the client can immediately start sending encrypted application data using the session key. The server decrypts the session key and can use the key as well. Note that, if such a protocol would be built trivially from public-key encryption, then the protocol would not provide forward secrecy. The latter is achieved by puncturing the secret key used to the decrypt the ciphertext in a way, that the server can no longer decrypt ciphertexts from past sessions. Besides providing forward-secrecy, puncturing the key also provides replay protection.

This new approach (see Figure 2) requires that the client already knows the public key of the server before establishing the connection. In the context of IoT, we observe that often clients communicate only with a pre-configured server. Therefore, the public key of the server can already be deployed during provisioning of the devices (cf. D3.4). In the case this is not possible, only during the first connection between a client and the server

a non-0-RTT key exchange has to be performed. The client will receive the public key during this connection and can store it for future use.

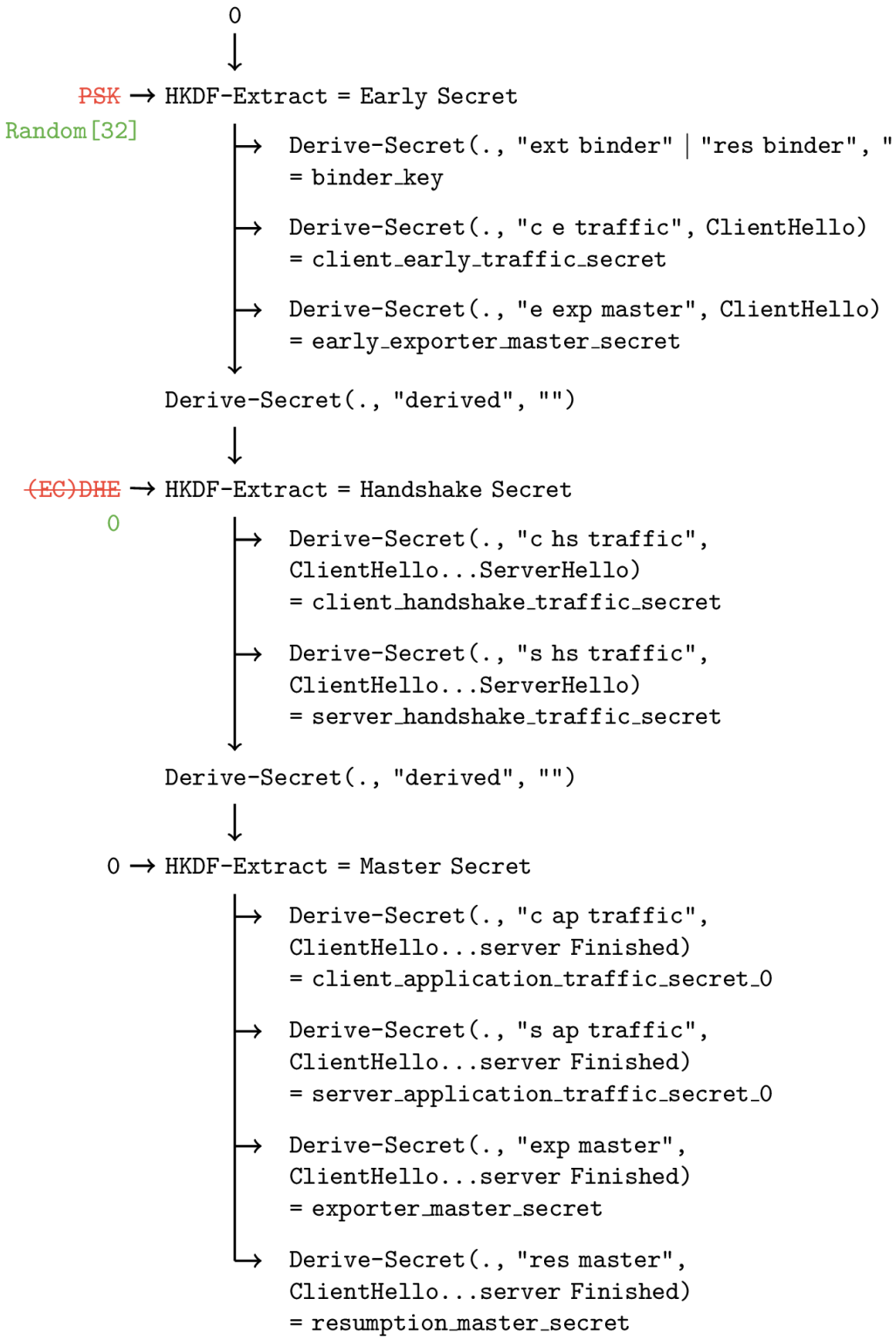

Figure 2: Potential modification of the key derivation in TLS when deploying BFE [19]

For the upcoming integration in TLS 1.3, we focus on a specific choice of such a forward-secret 0-RTT key exchange protocol, namely the one from Derler et al. [2]. Their protocol builds on top of bloom filter encryption (BFE). The idea there is that the ciphertexts have an attached tag which determines the key. The keys themselves are managed in a bloom filter. Once they are used and the corresponding bit is set in the bloom filter, the keys are removed and can no longer be used for decryption.

2.2 Bloom Filter Encryption Library

We provide an implementation of the IND-CCA secure version of BFE in the C programming language compatible with C11 [25].[1] The implementation explores the construction based on the Boneh-Franklin[2] identity-based encryption scheme [3] for BFE made CCA-secure using the Fujisaki-Okamoto transformation [13]. Derler et al. also provide time-based BFE built from Boneh-Boyen-Goh hierarchical identity-based encryption [6], which we indent to investigate at a later point.

The library includes a documentation produced with the help of doxygen [16]. The documentation also presents examples of the typical usage. The correctness of the implementation is tested with the unit testing framework Cgreen [17]. The library also includes an implementation of the interface for PKEMs as described in D3.5. In Figure 3, Figure 4, Figure 5 and Figure 6, the usage of the PKEM API is demonstrated.

```
// the public key
unsigned char pk[CRYPTO_PUBLICKEYBYTES];
// the secret key
unsigned char* sk = malloc(CRYPTO_SECRETKEYBYTES);

// generate key pair
if (crypto_kem_keypair(pk, sk)) {
  // handle error
}
```

Figure 3: Key generation

```
// the public key
unsigned char* pk;

// the ciphertext
unsigned char ct[CRYPTO_CIPHERTEXTBYTES];
// encapsulate a new key
unsigned char k[CRYPTO_BYTES];
if (crypto_kem_enc(ct, k, pk)) {
  // handle error
}
```

Figure 4: Key encapsulation

[1] The library is available in the bfe-1.0.tar.gz archive.

[2] As in [2], we implement hashed Boneh-Franklin in the type 3 bilinear pairings setting.

```
// the secret key
unsigned char* sk;
// the cipher text
unsigned char* ct;

// decapsulate key
unsigned char k[CRYPTO_BYTES];
if (crypto_kem_dec(k, ct, sk)) {
  // handle error;
}
```

Figure 5: Key decapsulation

```
// the secret key
unsigned char* sk;
// the cipher text
unsigned char* ct;

// puncture secret key with respect to a ciphertext
if (crypto_kem_punc(sk, ct)) {
  // handle error;
}
```

Figure 6: Puncturing of the secret key

Alternatively, the library also provides an API that is more suitable for the use on the server side. This API allows users to control serialization and deserialization of the key material and ciphertexts explicitly. Thereby, the server can keep the relatively large secret key in memory without having to serialize and deserialize it all the time. An additional advantage is that this API also allows users to provide their choice of parameters for the bloom filter. Usage of this API is demonstrated in Figure 7, Figure 8, and Figure 9.

```
bfe_secret_key_t sk;
bfe_public_key_t pk;

// generate new keys
bfe_init_secret_key(&sk);
bfe_init_public_key(&pk);
if (!bfe_keygen(&pk, &sk, 32, 1 << 19, 0.0009765625)) {
  // handle error
}

// serialize public key
uint8_t serialized_pk[bfe_public_key_size_bin()];
bfe_public_key_write_bin(serialized_pk, &pk);

// serialize secret key
uint8_t* serialized_sk =
      malloc(bfe_secret_key_size_bin(&sk));
bfe_secret_key_write_bin(serialized_sk, &sk);

// clean up keys
bfe_clear_secret_key(&sk);
bfe_clear_public_key(&pk);
```

Figure 7: Key generation with explicit choice of the parameters and serialization of the keys

```
// the serialized public key
const uint8_t* serialized_pk;
bfe_public_key_t pk;

// deserialize the public key
if (bfe_public_key_read_bin(&pk, serialized_pk)) {
  // handle error
}

// encaps a new key
bfe_ciphertext_t ciphertext;
bfe_init_ciphertext(&ciphertext, &pk);
uint8_t key[pk.key_size];
if (bfe_encaps(&ciphertext, K, &pk)) {
  // handle error
}

// serialize the ciphertext
const size_t csize = bfe_ciphertext_size_bin(&ciphertext);
uint8_t serialized_ct[csize];
bfe_ciphertext_write_bin(serialized_ct, &ciphertext);

// clean up
bfe_clear_ciphertext(&ciphertext);
bfe_clear_public_key(&pk);
```

Figure 8: Encapsulation of a new key and serialization of the ciphertext

```
// the serialized secret key
uint8_t* serialized_sk;
// the serialized public key
const uint8_t* public_key
// the serialized ciphertext
const uint8_t* serialized_ct;

// deserialize the secret key
bfe_secret_key_t sk;
if (bfe_secret_key_read_bin(&sk, serialized_sk)) {
  // handle error
}

// deserialize the public key
bfe_public_key_t pk;
if (bfe_public_key_read_bin(&pk, serialized_pk)) {
  // handle error
}

// deserialize the ciphertext
bfe_ciphertext_t ciphertext;
if (bfe_ciphertext_read_bin(&ciphertext, serialized_ct)) {
  // handle error
}

// decaps ciphertext
uint8_t key[pk.key_size];
if (bfe_decaps(key, &pk, &sk, &ciphertext)) {
  // handle error
}

// puncture secret key and serialized it again
bfe_puncture(&sk, &ciphertext);
bfe_secret_key_write_bin(seralized_sk, &sk);

// clean up
bfe_clear_ciphertext(&ciphertext);
bfe_clear_public_key(&pk);
bfe_clear_secret_key(&sk);
```

Figure 9: Decapsulation of a key and puncturing of the secret key including deserialization

We will briefly describe some implementation aspects of the concrete BFE scheme. As Boneh-Franklin is a pairing-based scheme, we require a pairing implementation. We have chosen the highly optimized open-source pairing library RELIC [12]. This library is good choice for multiple reasons: (1) it includes implementation of currently recommended pairing-friendly curves such as BLS12-381[3], (2) includes optimizations for desktop and server CPUs as well as embedded CPUs such as ARM and AVR processors, and (3) allows to configure memory handling suitable for the target platform. Note however, that a specific build of RELIC only supports one specific pairing-friendly elliptic curve, hence our implementation also does not provide any flexibility in choosing the curve.

Next, random oracles are required for applying the Fujisaki-Okamoto transformation. They are implemented via SHAKE [14]. Similarly, the hash functions used as part of the Bloom filter are implemented with SHAKE as well.[4] We have integrated the open-source implementation optimized for 64-bit systems [15]. The code package also includes many different implementations for other targets which can be selected depending on the target platform. When switching to a different platform, the BFE library just has to be built with a SHAKE implementation that is optimized for this platform to obtain the best results. No changes to the code-base of the library itself are required to adopt to this change.

Table 1 gives one choice of parameters for using the BFE library. The parameters are chosen in such a way that a TLS server can support one entirely new TLS connection per second over a timeframe of three months with an error probability of less than 2^{-10}. After this time period, a new key pair on the TLS server has to be generated. Note though, that the current TLS infrastructure is moving towards regenerating keys and reissuing certificates every three months due to the rise of Let's Encrypt [21].

Table 1: Potential choice of parameters for the BFE library

Parameter	Value
n (size of the Bloom filter)	524288
p (false-positive probability)	0.0009765625
Pairing-friendly elliptic curve	BLS12-381 (-DFP_PRIME=381 when building RELIC)

2.3 Benchmarks

Table 2 details the performance of the library using the parameters from Table 1. The benchmarks were performed on an Intel Core i7-8650U with 3 GHz and 16 GB of RAM. As expected, puncturing of the secret key is very efficient – it only requires removal of a specific subkey of the secret key. The key generation is slow, but it is never performed on a resource-constraint device. Decryption is little slower than encryption due to the nature of the Fujisaki-Okamoto transform.

Table 2: BFE library benchmark

Algorithm	Time (in ms)
KeyGen	941,857.77
Encrypt	3.66
Decrypt	4.94

[3] According to recent security estimations [18], this curve provides 120 bits of security.

[4] Technically, collision-resistance and pre-image resistance are not required. To have well distributed Bloom filter indices we erred on the safe side and chose SHAKE. More lightweight hash functions with a close to uniform distribution would work as well.

Puncture	0.01

In Table 3, we present sizes of the secret key, the public key and the ciphertexts for the parameters given in Table 1. Note that the secret key shrinks with each puncturing as parts of it get deleted and no longer need to be stored.

Table 3: Key and ciphertext sizes

Type	Size (in bytes)
Secret key	101,253,128
Public key	109
Ciphertext	453

2.4 Discussion

In the context of IoT4CPS, this choice of parameters leads to some constraints when deploying the library in practice. First, the approach is only suitable when resource-constraint devices connect to more powerful servers that can hold (and generate) the large secret keys used in this scheme. On the client side, one can actually reduce the requirements: while any PSK-based solution would require secure storage for the PSK, this is not necessary in our case. Only the server's public key needs to be stored on the client. Since the public key can be verified at any time via a certificate chain, no special secure storage is required to store the public key. The clients need to be able to receive new public keys if the server's key changes, however. For example, updates of the public key can be distributed by performing one classical TLS handshake and storing the received public key. Note though, this requirement does not infer an additional constraint: clients need to be prepared to handle new public keys in case keys need to be rolled over for other reasons, e.g., due to key compromise. Additionally, for deployments where a software implementation of the pairing might not be efficient enough, a co-processor providing elliptic curve operations and the pairing evaluation is an option to speed up the protocol even further [20]. Similarly, the SHAKE implementation can be replaced with a suitable co-processor or FPGA version [22,23,24]

3. ISAP for ASICs and FPGAs

In this section we put the focus on enabling secure communication for legacy devices with limited resources, which operate in the field for years and cannot be replaced easily with new hardware. The specific challenge is to apply authenticated encryption on an existing FPGA-based legacy device without changing any hardware or applying additional connectivity. Limited resources of FPGA-based legacy devices (such as the number of logic cells and power dissipation) are a constraint caused then by choosing highly cost-optimized hardware components. This poses another challenge for the new communication link that shall be resistant against simple Differential Power Analysis (DPA) attacks.

DPA attacks exploit data through power consumption of cryptographic devices. They use a broad number of power traces to analyze the power consumption at a fixed moment of time as a function of the processed data [21]. There exists a general attack strategy and consists of following five steps:

1. Select an intermediate result of the executed algorithm
2. Measuring the power consumption
3. Calculating hypothetical intermediate values

4. Mapping intermediate values to power consumption values
5. Comparing the hypothetical power consumption value with the power traces

A special requirement 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 by Xilinx 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 by software tasks. Consequently, the available FPGA area, minimum throughput and maximum latency are given. All these constraints together leave only one way open - designing an optimized implementation of a suitable algorithm that must fit into the remaining programmable logic.

A very suitable solution for the problem above is offered by the ISAP encryption scheme, which is shown in the following section. Within the course of the IoT4CPS project, an optimized ISAP module for FPGA is being developed and applied within a demonstrator design in order to enable detailed analysis and assessment.

3.1 ISAP Module

ISAP is a family of none-based authentication cipher with associated data designed with a focus on robustness against passive side-channel attacks. Therefore, it is very suitable for applications like firmware updates where robustness against power analysis and fault attack is central. It is a so-called lightweight authentication encryption algorithm and creates a small footprint in hardware, but performance is less critical. All ISAP family members are permutation-based designs that combine variants of the sponge-based ISAP mode with one of several published lightweight permutations. The main design goal of ISAP is to provide out-of-the-box robustness against certain types of implementation attacks while allowing to add additional defense mechanisms at a small cost. This is essential whenever cryptographic devices are deployed in locations that are physically accessible by potential attackers – a typical scenario in IoT applications [7].

The key features of ISAP are:

- Authenticated encryption using lightweight permutations
- Sponge-based mode of operation using well studied substitution-permutation-network (SPN) permutations
- Suitable for constrained devices: small state, simple permutation
- Side-channel resistance: Provably secure leakage-resilience for encryption and decryption
- Built-in hardening against fault attacks
- Easy to implement in software and hardware
- Compact in software: supports pipelined processing, bit-sliced 5-bit S-box
- Fast and compact in hardware
- Scalable for more conservative security or higher throughput
- Timing resistance: No table look-ups or additions
- Minimal overhead (ciphertext length = plaintext length)

ISAP recommended four permutation instances: ISAP-K(eccak)-128A, ISAP-A(scon)-128A, ISAP-K-128, ISAP-A-128. All instances are designed to provide 128-bit security against cryptanalytic attacks as well as inherent security against certain classes of side-channel attacks [7].

The implementation for the FPGA/ASIC is written in System Verilog, compatible with IEEE 1800-2005 [26].[5] It consists of three (sub)modules and one (top)module. The modules are:

- Isap_Enc: is responsible for the encryption part. The part is implemented and tested with a simple manual test.
- Isap_Mac: is responsible for the authentication functionality. This part is also implemented and tested with a simple manual test.
- ISAP: Is the (top) module of the ISAP. It uses the CAESAR Hardware API [8] to get the data loaded for the ISAP_Enc and ISAP_Mac functionality. Especially the lightweight implementation for Two-Pass algorithms [9]. It is not fully implemented and tested.
- Keccakf400 [10]: is required for the permutations. This part is implemented and manually/automatically tested.
- Ascon [11]: is not implemented yet.

The used target technology is the umc065LL (65nm low leakage) ASIC. However, the implementation in the register-transfer level (RTL) model can also be ported to an FPGA (e.g. Xilinx Kintex 7 Spartan) with slight adaptions for that platform. Furthermore, the implementation is a soft intellectual property (IP) core to provide maximum flexibility, reconfigurability, and to assure technology independency.

3.2 ISAP as part of the FPGA Demonstrator

For evaluation purposes the previously introduced ISAP module is integrated within a simplified demonstrator design, representing an exemplary distributed communication system. The design consists of one master node and several slave nodes, all located within one single large FPGA. Each individual node hosts instances of the ISAP module to encrypt and decrypt messages on-the-fly. The Demonstrator is built upon a Component-Off-The-Shelf (COTS) Eval Kit, which hosts a mainstream FPGA Device.

3.2.1 Demonstrator Architecture

The architecture of the demonstrator is based on smart nodes that communicate via an industrial standard interface (Figure). Each node consists of an independent processor subsystem, an encryption/decryption unit and additional peripheral elements. The master node can communicate with each slave node and vice versa.

For the demonstrator the Xilinx evaluation board ZCU102 with the Zynq Ultrascale+ MPSoC device is used. This FPGA platform provides an integrated hard-wired processor sub system (PS) together with a large programmable logic area (PL). For the sake of building a demonstrator that is easy to handle, the number of needed hardware devices is reduced by locating all nodes inside one big FPGA device. Internal communication is established with AXI stream interfaces instead of ethernet interfaces. Node addressing is handled with side channel information and is not part of the data transfer package. Optional external nodes can be connected via Ethernet (see optional external slave node in Figure 10). The external slave node is connected to the master node via a standard RJ45 (Ethernet) cable. Furthermore, each node has GPIO ports to signal status information via LEDs located on the FPGA board. Via UART port a terminal console on the Host PC can be used for standard input and output operations of the node CPUs.

[5] The implementation is available in the isap_src.zip archive.

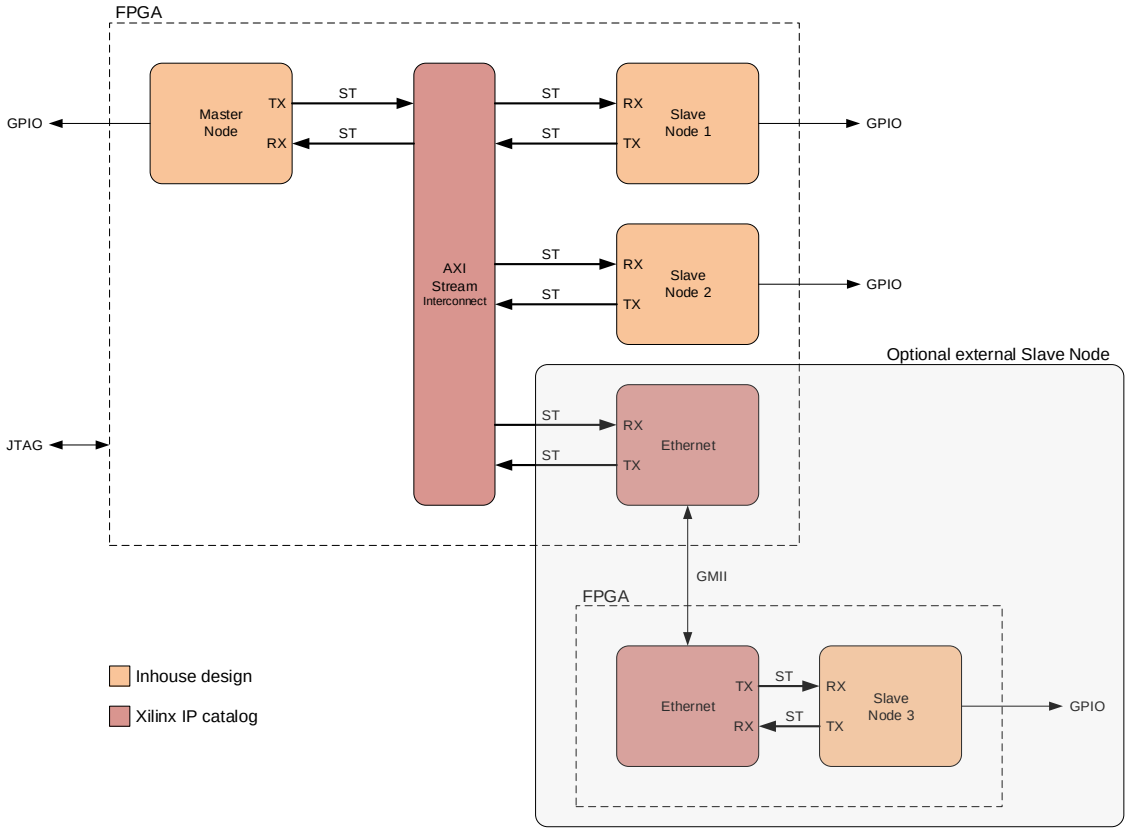

Figure 10: Architecture of demonstrator

3.2.2 Master Node

The master node is a completely independent CPU subsystem based on a hard-wired Quad-Core processor and a customized part inside the programmable logic. Standard components are used out of the Xilinx IP catalog, customized components are added manually to the Xilinx IP library for high reusability. Internally the AXI interconnect is used to connect memory mapped components that are visible to the processor in a defined address space. AXI streaming ports are applied for external data transfers that are handled by separate DMA units for each direction.

The streamed data packages are encrypted and decrypted on-the-fly by the ISAP module. Each outgoing package gets a unique session key for encryption. This session key and the transmitted nonce information for the slave node come from a generator unit, which derives the session key from the internally stored master key and a random number from the true random generator. For decrypting incoming packages, the received nonce value is used for regeneration of the session key.

Additionally, two partial reconfiguration blocks are used for individually checking the data stream before and after the ISAP module, and thus realize a hardware security checker. Different checking algorithms can be applied during normal operation. During the partial reconfiguration process the block will be decoupled from the rest of the logic to avoid undefined data on the streaming bus. Further details about the hardware checkers will be shown within Deliverable D4.2.

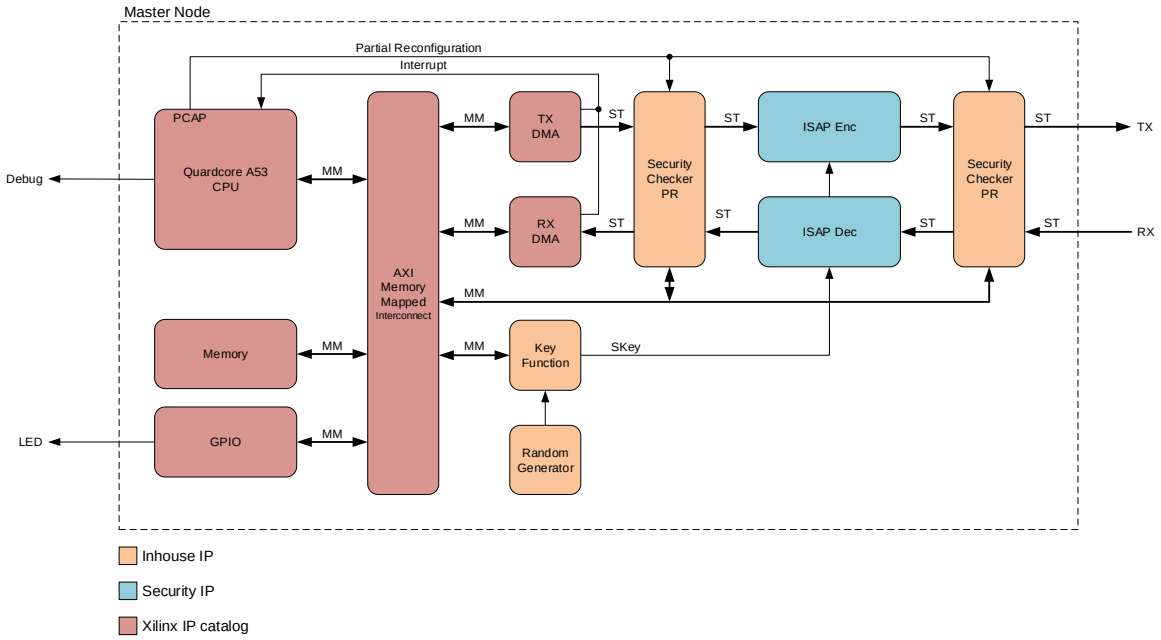

Figure 11: Master Node

The test application can either run as bare metal directly on one ARM core or under Linux on multi ARM cores. For the demonstrator software a simple program sends data packages to the slave nodes and waits for a response to check the correct behavior of data encryption and decryption between the master and the slave node. Visual observation is enabled by using the GPIO ports to show information on the board LEDs. Via the UART port an external terminal can be used for standard input and output interfaces. Loading the partial reconfiguration block for the security checker functionality is handled with predefined commands from the software library.

3.2.3 Slave Node

The slave node comprises the same AXI architecture as the master node, but with a reduced subset of components and functions (Figure 12). As processor core the Microblaze softcore is used, thus locating the complete slave node inside the programmable logic part. Furthermore, the slave node does not include partial reconfiguration blocks. All other components have the same functionality range as those within the master node (including the master key and key generator). In this way, equal timing behaviour of data processing for all nodes inside the system is guaranteed.

The slave node test application runs independently as bare metal software on the Microblaze processor. The program modifies the payload of the incoming data package and sends it back to the master node. With these changes inside the payload, the master node can check for correct behaviour of the secure communication. Blocking behaviour that might be caused by this cyclic data processing is avoided by an additional watchdog timer that resets the complete system after a defined timeout. Status information of the slave node can be displayed on the board LEDs. The UART interface can be used as debug port to show information on an external terminal.

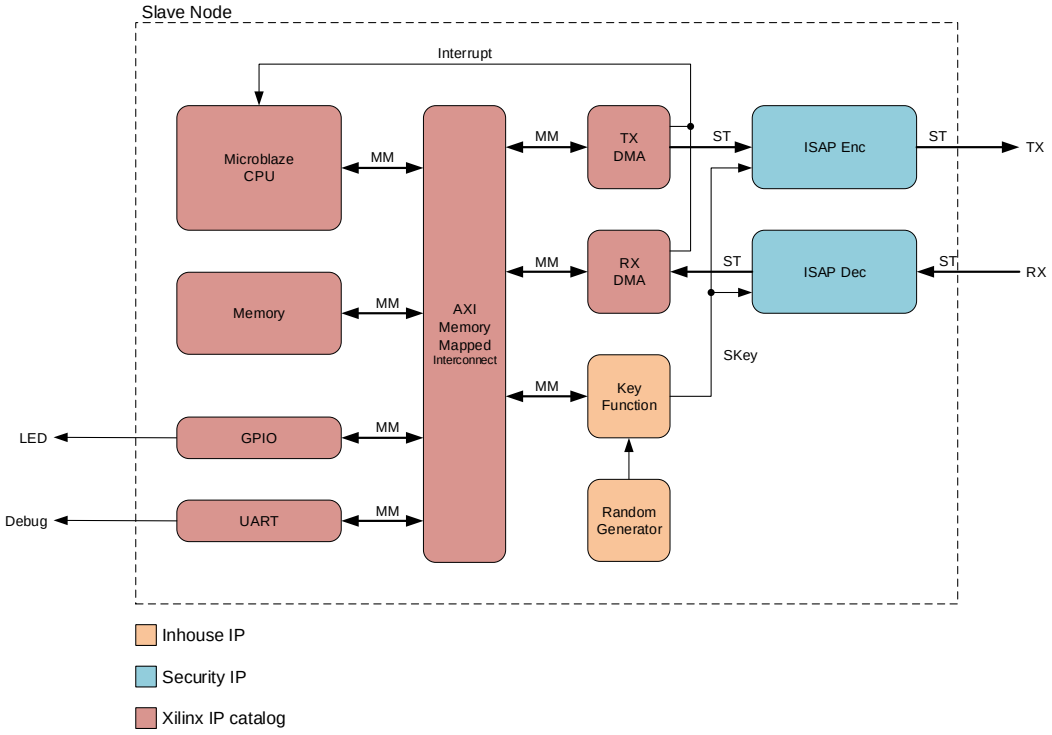

Figure 12: Slave Node

4. References

[1] D. Kales, S. Ramacher, C. Rechberger, R. Walch, „Efficient FPGA Implementations of LowMC and Picnic.”, CT-RSA (accepted), 2020

[2] 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.

[3] D. Boneh, M. Franklin. “Identity-based encryption from the Weil pairing.” CRYPTO 2001.

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

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

[6] D. Boneh, X. Boyen, E.-J. Goh. “Hierarchical identity based encryption with constant size ciphertext.”, EUROCRYPT 2005. pp. 440-456

[7] C. Dobraunig, M. Eichsleder, S. Mangard, F. Mendel, B. Mennink, R. Primas, T. Unterluggauer “ISAP v2.0 Submission to the NIST Lightweight Cryptography competition”, online: https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/round-1/spec-doc/ISAP-spec.pdf. 2019

[8] E. Homsirikamol, W. Diehl, A. Ferozpuri, F. Farahmand, P.Yalla, J. Kaps, and K. Gaj “Hardware API”, online: https://pdfs.semanticscholar.org/35a9/27d06d84d3f4079bde468cd74505235eb5e0.pdf

[9] E. Homsirikamol, P. Yalla, F. Farahmand,W. Diehl, A. Ferozpuri, J. Kaps, K. Gaj, “Implementer’s Guide to Hardware Implementations Compliant with the CAESAR Hardware API version 2.0“, online: https://cryptography.gmu.edu/athena/CAESAR_HW_API/CAESAR_HW_Implementers_Guide_v2.0.pdf

[10] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. “TheKeccak SHA-3 submission (Version 3.0)”.https://keccak.noekeon.org/Keccak-submission-3.pdf. 2011.

[11] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer. “Ascon v1.2”. Submission to CAESAR: Competition for Authenticated Encryption. Security, Applicability, and Robustness (Round 3 and Finalist). See: https://ascon.iaik.tugraz.at/ . 2016. url: https://competitions.cr.yp.to/round3/asconv12.pdf.

[12] D.F. Aranha, C. P. L. Gouvêa. “RELIC is an Efficient Library for Cryptography.” https://github.com/relic-toolkit/relic

[13] E. Fujisaki, T. Okamoto. “Secure Integration of Asymmetric and Symmetric Encryption Schemes.” CRYPTO 1999.

[14] NIST. “FIPS PUB 202: SHA3: Permutation-Based Hash and Extendable-Output Functions”, 2015

[15] “eXtended Keccak Code Package.” https://github.com/XKCP/XKCP

[16] D. van Heesch. “Doxygen: Generate documentation from source code.” http://www.doxygen.nl/

[17] „Cgreen: Unit Tests, Stubbing and Mocking for C and C++.” https://cgreen-devs.github.io/

[18] R. Barbulescu, S. Duquesne. “Updating Key Size Estimations for Pairings.”, Journal of Cryptology, 2019

[19] M. Krmpotić. “Implementation and Evaluation of Low-Latency Key-Exchange Protocols”, Master’s thesis, 2019

[20] T. Unterluggauer, E. Wenger. “Efficient Pairings and ECC for Embedded Systems.” CHES, 2014

[21] J. Aas, R. Barnes, B. case, Z. Durumeric, P. Echerskley, A. Flores-Lópes, J.A. Halderman, J. Hoffman-Andres, J. Kasten, E. Rescorla, S.D. Schoen, B. Warren. “Let’s Encrypt: An Automated Certificate Authority to Encrypt the Enture Web.”, ACM CCS, 2019

[22] H. Groß, D. Schaffenrath, S. Mangard. “High-Order Side-Channel Protected Implementations of KECCAK.” DSD, 2017

[23] H. Mestiri, F. Kahri, M. Bedoui, B. Bouallegue, M. Machhout. „High throughput pipelined hardware implementation of the KECCAK hash function.”, ISIVC, 2016

[24] T. Honda, H. Guntur, A. Satoh. „FPGA implementation of new standard hash function Keccak.” GCCE, 2014

[25] ISO. “ISO/IEC 9899:2011: Information technology – Programming languages – C.” https://www.iso.org/standard/57853.html

[26] IEEE. “IEEE 1800-2005 - IEEE Standard for SystemVerilog: Unified Hardware Design, Specification and Verification Language.” https://standards.ieee.org/standard/1800-2005.html