D5.5 Final Report on Hardware-Assisted Schemes

Document Identification			
Status	Final	**Due Date**	30/09/2020
Version	1.0	**Submission Date**	30/09/2020

Related WP	WP5	**Document Reference**	D5.5
Related Deliverable(s)	D5.1, D5.3, D5.4, D5.7, D5.9	**Dissemination Level(*)**	PU
Lead Participant	UH	**Lead Author**	Kimmo Järvinen (UH)
Contributors	UH	**Reviewers**	Svetla Nikova (KU Leuven) Miha Stopar (XLAB)

Keywords:
Functional encryption, hardware, implementation, trust, trusted component, FPGA, ARM TrustZone

This document is issued within the frame and for the purpose of the FENTEC project. This project has received funding from the European Union's Horizon2020 under Grant Agreement No. 780108. The opinions expressed and arguments employed herein do not necessarily reflect the official views of the European Commission.

(*) Dissemination level.-PU: Public, fully open, e.g. web; CO: Confidential, restricted under conditions set out in Model Grant Agreement; CI: Classified, Int = Internal Working Document, information as referred to in Commission Decision 2001/844/EC.

Document Information

List of Contributors	
Name	Partner
Kimmo Järvinen	UH
Aymeric Genêt	Kudelski

Document History			
Version	Date	Change editors	Changes
0.1	22/04/2020	Kimmo Järvinen (UH)	Preliminary version
0.2	31/05/2020	Kimmo Järvinen (UH)	Updated version
0.3	28/08/2020	Aymeric Genêt (Kudelski)	TrustZone text
0.4	23/09/2020	Aymeric Genêt (Kudelski)	Updated TrustZone text
0.5	24/09/2020	Kimmo Järvinen (UH)	Review version
1.0	29/09/2020	Kimmo Järvinen (UH)	Final version

Quality Control		
Role	Who (Partner short name)	Approval Date
Deliverable Leader	Kimmo Järvinen (UH)	30/09/2020
Technical Manager	Michel Abdalla (ENS)	30/09/2020
Quality Manager	Diego Esteban (ATOS)	30/09/2020
Project Coordinator	Francisco Gala (ATOS)	30/09/2020

Table of Contents

List of Figures

List of Acronyms

Abbreviation / acronym	Description
ASIC	Application Specific Integrated Circuit
BN	Barreto-Naehrig (curves)
CPU	Central Processing Unit
DPA	Differential Power Analysis
FE	Functional Encryption
FPGA	Field Programmable Gate Array
HW	Hardware
MAC	Message Authentication Code
MIFE	Multi-Input Functional Encryption
OP-TEE	Open Portable TEE
PCP	Pairing Cryptography Processor
PRNG	Pseudo Random Number Generator
REE	Rich Execution Environment
ROM	Read Only Memory
RPMB	Replay-Protected Memory Block
SoC	System-on-Chip
SW	Software
TA	Trusted Application
TEE	Trusted Execution Environment
TPM	Trusted Platform Module
TRNG	True Random Number Generator

Executive Summary

In this deliverable D5.5 “Final Report on Hardware-Assisted Schemes”, we give a description of work that has been done in FENTEC for developing hardware-assisted schemes for FE. This deliverable is specifically about the work in Task 5.3. D5.5 extends D5.4 “Preliminary Report on Hardware-Assisted Schemes” from June 2019. As defined in D5.1 “Security and Trust Models”, we assume partial trust to the computing platform for the hardware-assisted schemes. Specifically, we assume that there exists a trusted part in an otherwise untrusted platform. Untrusted in this context means that it can be a subject for implementation attacks and that an adversary can potentially compromise it by utilizing weaknesses of the system. On the other hand, compromising a trusted component is assumed to be out of the adversary’s capabilities. We consider two different cases of hardware-assisted schemes in this deliverable: (1) the entire HW part of the computing platform (e.g., an FPGA) is trusted and (2) there exists only a small trusted component (e.g., a commercial TPM chip or a TEE in the main CPU) in an otherwise untrusted platform. We consider a specific multi-input inner-product FE scheme based on Paillier encryption and cryptographic pairings (optimal ate pairings on Barreto-Naehrig curves) as examples and show how they map to these two cases. In addition to these discussion, we also present a prototype FE implementation under the HW-assisted trust model where the trusted component is implemented with ARM TrustZone. Finally, we identify certain directions for future research.

1 Introduction

1.1 Purpose of the document

This deliverable D5.5 "Final Report on Hardware-Assisted Schemes" gives a description of work in WP5 of FENTEC and, in particular, in Task 5.3 of WP5. The deliverable provides details about research on HW-assisted schemes for FE, provides further discussion on what are interpreted as HW-assisted schemes, presents results obtained on this topic in FENTEC, and describes plans for continuation work after FENTEC.

This deliverable discusses the issues related to HW-assisted schemes, where only a part of the system can be trusted, compared to the cases of HW-optimized and HW-operated schemes, where the whole computing platform is either trusted or untrusted, respectively. This deliverable discusses these issues mainly on theoretical level but presents how issues arising from the HW-assisted scheme could be addressed also on practical level. FENTEC has implemented a prototype HW-assisted scheme for ARM TrustZone and this implementation will be more closely discussed in Section 4.1. Also the results that were produced for HW-optimized schemes and discussed more closely in D5.3 will be discussed again in this deliverable under the assumption that only the HW side is trusted in contrast to the fully trusted scenario of D5.3.

1.2 Structure of the document

This deliverable is structured as follows. Section 2 revisits the trust models defined in D5.1 with a focus on the HW-assisted trust model and presents two specific cases that are used in this deliverable. Section 3 considers the first case where the schemes are implemented in a system where the entire HW is trusted. Section 4 discusses the second case where there is a specific trusted component in an otherwise untrusted system. This section also presents the details of the FE implementation with ARM TrustZone. Section 5 ends the deliverable with conclusions.

2 Trust model

The deliverable D5.1 defined three different trust models for computing platforms that are considered in FENTEC. Fig. 1 shows these trust models as a recap from D5.1. This deliverable focuses on HW-assisted schemes shown in the middle and, thus, relates mainly to the work done in Task 5.3 of WP5.

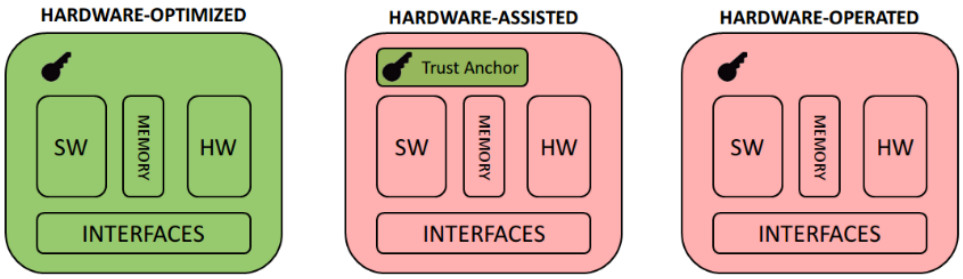

Figure 1: Underlying security models for the different tasks in WP5. Green (trusted environment), red (untrusted environment).

2.1 Hardware-assisted trust model

The trust model of HW-assisted schemes assumes partial trust in the computing system. To be more exact, it assumes that the computing platform includes a trusted component (also referred to as the trust anchor) in an otherwise untrusted system. The component is such that it can be assumed to be tamper-proof and to leak no information even if the untrusted system is compromised. This is in contrast with the fully trusted setting of the HW-optimized model studied in Task 5.2 of WP5 and the fully untrusted setting of the HW-operated model studied in Task 5.4 of WP5.

In D5.1, the definition of the trusted component was left open for more detailed definitions that correspond to the most appropriate definition of a specific application. In this deliverable, we will discuss two specific cases: (1) the entire HW part is trusted (leaving SW, memory and interfaces untrusted) and (2) there is a small trusted component in the system either as a part of the HW (e.g., a separate TPM chip or secure module in an SoC) or as a TEE for the SW. These two cases are discussed more in details below.

2.2 Case 1: Trusted hardware

In this case, we assume that the whole HW is trusted. The HW could be, e.g., an FPGA device or an ASIC-based coprocessor for cryptography. The HW is assumed to be fast and to act primarily as an accelerator for cryptographic computations. An adversary model that supports this kind of a trust assumption is such that the adversary is remote and does not have a physical access to the device. Thus, the adversary is able to launch attacks against the SW side to compromise the system over the network. E.g., the adversary can launch timing attacks against security-critical operations, try to exploit bugs in the OS, etc. However, the HW is isolated from the SW with

an appropriately well designed interface that prevents the adversary from compromising any of the security related operations that are carried out in the HW. In order for this assumption to hold, all private keys need to be stored in the HW so that they cannot be accessed directly from the SW. Also all cryptographic operations in the HW that use these keys must be constant-time and protected from all remote attacks (e.g., certain fault attacks). However, protection against more powerful attacks requiring physical proximity (e.g., power or electromagnetic radiation based attacks such as differential power analysis (DPA)) are out of the scope of this model, as mentioned above, because the adversary does not have physical access to the computing platform. These elaborated attacks and protections against them will be discussed in D5.7 on HW-operated schemes.

It directly follows from the above that a coprocessor in the HW, which has protections against remote side-channel attacks and an interface that prevents (also indirect) access to secrets, can be trusted and it can act as the trusted component. The main issues in this case are related to SW/HW codesign. E.g., which operations should be delegated to the HW side for performance reasons and which can be computed in the SW without a significant performance penalty. In addition to this, it is also important to ensure that all security-critical operations are done in the trusted HW.

This case will be discussed with more details in Section 3.

2.3 Case 2: Trusted component

In this case, we assume that there is a trusted component in the system as a part of either (a) the processor running SW (e.g., TrustZone) or (b) the HW (e.g., a TPM or other security module). The rest of the system is untrusted. This is exactly the HW-assisted trust model from D5.1 that is depicted in Fig. 1 in the middle. In this case, we allow a much stronger adversary who can have a lot more capabilities. These capabilities may include also physical access and ability, e.g., for DPA, if the trusted component claims protection against such attacks[1]. The main problem is to identify the security-critical operations that are delegated to the trusted component. The objective is to minimize the number and complexity of operations that must be performed in the trusted component because protecting operations against powerful side-channel attacks is expensive and trusted components may have very limited computing power.

Consequently, the main issue is to identify the minimum set of security-critical operations of the FE schemes and study how they can be securely and efficiently executed in the trusted component. On the other hand, it is of interest to study whether existing FE schemes could be modified or extended so that they would reduce the number and complexity of operations that need to be executed in a trusted component.

This case will be discussed with more details in Section 4.

[1] All trusted components do not claim protection against side-channel attacks. One notable example is Intel SGX [11].

3 Software/hardware co-design

In this section, we will focus on Case 1 from Section 2, where the entire HW side is assumed to be trusted. Hence, all operations that are performed in HW are assumed to be safe from the adversary. Because the other parts of the system are untrusted, they can be potentially compromised by the adversary and, therefore, the interfaces of the HW side must prevent security-critical information from leaking out of HW. For example, keys that are stored in HW must be protected so that they cannot be accessed from the SW side or they cannot be implicitly derived from the responses of the HW (e.g., the HW operations must be constant-time in order to prevent leakage via timing attacks). However, side-channel attacks requiring physical access (e.g., DPA) are assumed to be impossible for the adversary.

This section is structured so that, in Section 3.1, we first present the multi-input inner-product FE scheme based on Paillier encryption from [1, 2] and optimal ate pairings over Barreto-Naehrig (BN) curves that were considered already in D5.3 and that will be the main focuses also in this deliverable. Notice that although cryptographic pairings are not actual FE schemes, they are central building blocks in many advanced FE schemes (e.g., for more complicated functionalities than linear functions) and, consequently, they are very important primitives to be studied in FENTEC. In Section 3.2, we discuss the computation of these cryptosystems in a generic computing platform where the HW side is trusted, but the other parts are untrusted. Because both the multi-core architecture for FE and the flexible pairing cryptography processor (PCP) presented in D5.3 are implemented as HW/SW codesigns on a Xilinx Zynq SoC, they can be studied in the context of hardware-assisted schemes with trusted HW so that the HW side (i.e., the FPGA side of Zynq SoC) is assumed trusted while the other parts (i.e., particularly the SW side running on the ARM cores of Zynq SoC) are untrusted[2]. Having concrete cryptographic schemes and HW implementations help us to focus the discussion and to explain concepts clearly but, otherwise, this deliverable is generic and applies to many other FE schemes and implementations as well, and even more broadly to many other cryptographic schemes outside of the context of FE.

3.1 Algorithms

3.1.1 The FE scheme

We will discuss particularly the multi-input inner-product FE scheme based on Paillier encryption that was presented earlier in D5.3 because this is the FE scheme for which WP5 produced the most significant implementation results. In this scheme, there are n users who all encrypt a vector of m integers and send them to an evaluator who can compute the inner-product $\langle \mathbf{x}, \mathbf{y} \rangle$ where $\mathbf{x}$ are the users' inputs and $\mathbf{y}$ are the weights that define the inner-product to be computed. The decryption (inner-product computation) requires a decryption key $sk_{\mathbf{y}}$ that is generated by a trusted key authority. We refer the readers to D5.3 or to the original publications [1, 2] for more details but, for the readers' convenience, we give the encryption and decryption (inner-product computation) algorithms from D5.3 again in Fig. 2 and Fig. 3, respectively.

[2]Notice that in D5.3 both the FPGA and ARM were assumed trusted.

Input: The encryption key $sk_i = (N_i, g_i, \mathbf{h}_i, \mathbf{u}_i)$ for input i consisting of the composite modulus N_i, a generator $g_i \in \mathbb{Z}^*_{N_i^2}$, and two vectors $\mathbf{h}_i = (h_{i,0}, h_{i,1}, \ldots, h_{i,m-1})$ with $h_{i,j} \in \mathbb{Z}_{N_i^2}$ and $\mathbf{u}_i = (u_{i,0}, u_{i,1}, \ldots, u_{i,m-1})$ with $u_{i,j} \in \mathbb{Z}_L$; The plaintext vector $\mathbf{x}_i = (x_{i,0}, x_{i,1}, \ldots, x_{i,m-1})$ with $x_{i,j} \in \mathbb{Z}_\ell$

Output: Ciphertext $\mathbf{c}_i = (c_{i,0}, c_{i,1}, \ldots, c_{i,m})$ where $c_{i,j} \in \mathbb{Z}_{N_i^2}$

$\mathbf{w} = (w_0, w_1, \ldots, w_{m-1}) \leftarrow (x_{i,0} + u_{i,0}, x_{i,1} + u_{i,1}, \ldots, x_{i,m-1} + u_{i,m-1}) \pmod{L}$

$r \leftarrow_R \{0, 1, \ldots, \lfloor N_i/4 \rfloor\}$

$c_{i,0} \leftarrow g_i^r$

for $j = 0$ **to** $m - 1$ **do**

  $c_{i,j+1} \leftarrow (w_j N_i + 1) h_{i,j}^r$

return $\mathbf{c}_i = (c_{i,0}, c_{i,1}, \ldots, c_{i,m})$

Figure 2: Encryption for the input i of the multi-input FE for inner-products based on the Paillier encryption [1, adapted from Figs. 1, 3 and 9]

Input: The decryption key $sk_{\mathbf{y}} = (\mathbf{N}, \mathbf{y}, \mathbf{d}, z)$ for inner-product $\langle \mathbf{x}, \mathbf{y} \rangle$ consisting of the composite moduli $\mathbf{N} = (N_0, N_1, \ldots, N_{n-1})$, the weight vectors $\mathbf{y} = (\mathbf{y}_0, \mathbf{y}_1, \ldots, \mathbf{y}_{n-1})$ with $\mathbf{y}_i = (y_{i,0}, y_{i,1}, \ldots, y_{i,m-1})$ and $y_{i,j} \in \mathbb{Z}_\ell$, the decryption keys for individual inputs $\mathbf{d} = (d_0, d_1, \ldots, d_{n-1})$ with $d_i \in \mathbb{Z}_{\phi(N_i)}$, and the decryption key for inner-product $z \in \mathbb{Z}_L$; The ciphertexts $\mathbf{c} = (\mathbf{c}_0, \mathbf{c}_1, \ldots, \mathbf{c}_{n-1})$ where $\mathbf{c}_i = (c_{i,0}, c_{i,1}, \ldots, c_{i,m})$ and $c_{i,j} \in \mathbb{Z}_{N_i^2}$

Output: Inner-product $p \in \mathbb{Z}_L$

for $i = 0$ **to** $n - 1$ **do**

  $p_i \leftarrow c_{i,0}^{-1}$

  $p_i \leftarrow p_i^{d_i}$

  for $j = 0$ **to** $m - 1$ **do**

    $p_i \leftarrow p_i \cdot c_{i,j+1}^{y_{i,j}}$

  $p_i \leftarrow \frac{p_i - 1 \bmod N_i^2}{N_i}$

$p \leftarrow \sum_{i=0}^{n-1} p_i - z \bmod L$

return p

Figure 3: Decryption (inner-product computation) of the multi-input functional encryption for inner products based on the Paillier encryption [1, adapted from Figs. 1, 3 and 9]

Input: $P \in \mathbb{G}_1$ and $Q \in \mathbb{G}_2$.
Output: $a_{\text{opt}}(Q, P) = f$, where $f \in \mathbb{F}_{p^{12}}$.
Constant: $t \in \mathbb{Z}$ so that $p = 36t^4 + 36t^3 + 24t^2 + 6t + 1$ and $r = 36t^4 + 36t^3 + 18t^2 + 6t + 1$ are primes; and $s = 6t + 2 = \sum_{i=0}^{L-1} s_i 2^i$, where $s_i \in \{-1, 0, +1\}$.

1 $T \leftarrow Q$, $f \leftarrow 1$
2 **for** $i = L - 2$ **to** 0 **do**
3 $\quad f \leftarrow f^2 \cdot l_{T,T}(P)$; $T \leftarrow 2T$
4 $\quad$ **if** $s_i \neq 0$ **then**
5 $\quad\quad f \leftarrow f \cdot l_{T,s_iQ}(P)$; $T \leftarrow T + s_iQ$
6 $Q_1 \leftarrow \pi_p(Q)$; $Q_2 \leftarrow -\pi_{p^2}(Q)$
7 $f \leftarrow f \cdot l_{T,Q_1}(P)$; $T \leftarrow T + Q_1$
8 $f \leftarrow f \cdot l_{T,Q_2}(P)$; $T \leftarrow T + Q_2$
9 $f \leftarrow f^{(p^{12}-1)/r}$
10 **return** f

Figure 4: Optimal ate pairing over BN curves.

3.1.2 Cryptographic Pairing

A cryptographic pairing is a bilinear map $\mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_3$ where $\mathbb{G}_1$ and $\mathbb{G}_2$ are additive groups and $\mathbb{G}_3$ is a multiplicative group. Many types of cryptographic pairings and pairing-friendly elliptic curves have been proposed in the literature. Because of this, the PCP that was be described in D5.3 was designed so that it can be programmed to implement different types of pairings and parameters. Despite this programmability feature in the PCP, we will focus the discussion in this deliverable solely to optimal ate pairings on BN curves to keep the discussion simple and focused. However, we want to emphasize that the findings related to security of pairing implementations are generic and hold also for other types of pairings (at least after small modifications). For optimal ate pairings on BN curves, $\mathbb{G}_1$ and $\mathbb{G}_2$ are additive groups of points on elliptic curves $E(\mathbb{F}_p)$ and $E(\mathbb{F}_{p^k})$ and $\mathbb{G}_3$ is the multiplicative group of $\mathbb{F}_{p^k}$. The parameters must be chosen so that discrete logarithms in all three groups are infeasible; e.g., for approximately 128-bit security level, we need a 256-bit prime p and $k = 12$.

The algorithm for computing an optimal ate pairing over BN curves is given in Fig. 4. The two main operations in the algorithm are the Miller loop in lines 2–5 and the final exponentiation in line 9. The former consists of elliptic curve arithmetic in $E(\mathbb{F}_{p^2})$ and line evaluations in $\mathbb{F}_{p^{12}}$ that can be interleaved. The latter is an exponentiation in $\mathbb{F}_{p^{12}}$ that can be decomposed into $f^{(p^6-1)(p^2+1)(p^4-p^2+1)/r}$, of which the two first terms can be efficiently computed with Frobenius operators and conjugations. The last term is computationally the most demanding part and is called the hard part. We concentrate particularly on the parameters and algorithms from [5]. They used $t = 2^{62} - 2^{54} + 2^{44}$ that enables efficient computation of the Miller loop and the hard part of the final exponentiation while providing 126-bit security level. In [5], $\mathbb{F}_{p^{12}}$ is represented as a tower extension field where the operations are computed with a series of operations in $\mathbb{F}_p$, where p is a 254-bit prime.

3.2 FE computation with a trusted HW

3.2.1 Encryption

The encryption of a user's input with the algorithm of Fig. 2 involves a lot of sensitive values and operations and it must be executed almost completely in a trusted environment. In particular, the vector $\mathbf{u}_i$ and the random value r must remain secret from an adversary. The values of N_i, g_i, and $\mathbf{h}_i$ can be given to the adversary without sacrificing the security of the scheme. Unfortunately, all operations in Fig. 2 involve r or $\mathbf{u}_i$.

The bright side is that the exponentiations are the operations of Fig. 2 that are also the most computationally involving and, therefore, they are the first candidates to be accelerated with a crypto coprocessor. Hence, both the security and performance aspects are well aligned, i..e., the same operations should be computed in the trusted HW from security and performance reasons. The role of the untrusted SW in encryption is limited to control, interfacing, and communication with external systems.

The vast majority of the computation time of executing the algorithm of Fig. 2 is spent in the $m+1$ exponentiations to the power r. Hence, it is essential that these exponentiations are performed in constant time so that an adversary who can make timing measurements either over the network or directly from compromised SW cannot deduct information about the value of r based on timing information. There are well-known techniques to compute exponentiations in constant time (e.g., the square-and-multiply-always or Montgomery's ladder [14] with constant-time multiplications) and they should be used in computations involving r.

3.2.2 Decryption (inner-product computation)

The decryption with the algorithm of Fig. 3 offers more opportunities to distribute computations differently between the untrusted SW and the trusted HW than the encryption of Fig. 2 discussed above. Now, the objective of the adversary is to learn $sk_{\mathbf{y}}$ or, more precisely, to obtain the capability to compute $\langle \mathbf{x}, \mathbf{y} \rangle$ for arbitrary inputs from the users with a fixed functionality defined by $\mathbf{y}$. As shown in Fig. 3, $sk_{\mathbf{y}} = (\mathbf{N}, \mathbf{y}, \mathbf{d}, z)$, where $\mathbf{N}$ are the public moduli of the Paillier encryption, $\mathbf{y}$ define the function, $\mathbf{d}$ are the decryption keys for each particular user, and z is the final decryption key that removes the mask from the multi-input inner-product.

If the adversary learns (any of the values in) $\mathbf{N}$, it does not have an impact on the security of the multi-input FE scheme assuming that Paillier encryption is secure because they are the public parameters of the cryptosystem. We also assume that $\mathbf{y}$, the weights of the inner-product, are not sensitive (because if they were, then a function hiding scheme would probably be used to protect them from the evaluator, too). Hence, $\mathbf{N}$ and $\mathbf{y}$ can be given to the untrusted SW. The decryption keys $\mathbf{d}$ must be protected. The decryption key z, on the other hand, can be given to the untrusted SW because the partial results p_i are masked with $\langle \mathbf{u}_i, \mathbf{y}_i \rangle$ and do not leak information. This cannot be done in the opposite order (i.e., give all d_i to the untrusted SW and protect z) because, then, the untrusted SW would directly learn the value of z by $z = p - \sum p_i \bmod L$.

The operations in Fig. 3 include a modular inverse (in line 2), a modular exponentiation to a large exponent (in line 3), m exponentiations to small exponents and $m-1$ multiplications (in line 5), an integer division (in line 6), and a summation and subtraction with small operands (in line 7). Only the exponentiations in line 3 use the security-critical values and, thus, must be

computed in the trusted HW. The other operations are computed solely with public values, such as ciphertexts and inner-product weights. The untrusted SW can compute the inversion in line 2, the exponentations and multiplications in line 5 and the division in line 6 as well as line 7 without decreasing the security of the system. However, many of these operations are computationally involving and would benefit from HW acceleration.

The exponentiations in line 3 are arguably the most demanding operations of Fig. 3 almost on any computation platform. If there is a HW support for modular exponentiations in the trusted HW, then this accelerated operation is likely to increase the performance of inner-product computation considerably. The modular exponentiations with the inner-product weights are significantly faster in most cases because $y_{i,j}$ are typically much smaller values than d_i. Nevertheless, if the trusted HW includes an accelerator for modular exponentiations, then it most probably makes sense to use it also for computing these exponentiations[3]. The modular inversion in line 2 and integer division in line 7 are computationally demanding operations which might become the bottleneck if they are performed in the SW and the exponentiations are computed with a HW accelerator. Therefore, even lines 2 and 6 may also benefit from HW acceleration.

The algorithm in Fig. 3 requires in total n modular inversions and n integer divisions. If all moduli in $\mathbf{N}$ are different, then there is little that can be done for accelerating these computations, besides using HW acceleration. However, if all moduli are the same (i.e., $N_i = N$ for all i), then the computational requirements of these operations can be reduced significantly. The modular inversions in line 2 can be computed with the so-called Montgomery's trick [14, 18], which relies on the observation that x^{-1} and y^{-1} can be computed with a single inversion $(xy)^{-1}$ so that $x^{-1} = y \cdot (xy)^{-1}$ and $y^{-1} = x \cdot (xy)^{-1}$. In general, Montgomery's trick computes n modular inversions with one inversion and $3(n-1)$ multiplications, which is typically a good trade-off because inversions cost a lot more than multiplications. The integer divisions in line 6 are even simpler to optimize if the moduli are the same: One computes $p = \sum p_i - n \bmod N^2$ and then performs a single integer division p/N. The value N^{-1} can be precomputed and, hence, the cost reduces to a single multiplication: $p \cdot N^{-1}$.

3.2.3 FE computation with the multi-core architecture

A multi-core architecture for FE schemes based on large integer modular arithmetic was developed in WP5 and, particularly, in its Task 5.2. This architecture and its implementation on Xilinx Zynq-7020 reconfigurable SoC was discussed with more details in D5.2. Recently, also a journal paper was published about these results [4]. The implementation is such that SW running on an ARM core controls the multi-core architecture on the FPGA, which is designed particularly for fast modular multiplications and exponentiations. The implementation was developed particularly for the HW-optimized trust model where also the SW side was assumed trusted but, in the following, we shall discuss how the architecture would map into the HW-assisted use case and what kind of additional assumptions must be made and what alterations are required to the implementation if the SW running on the ARM is not trusted.

The operations of the multi-core architecture discussed in D5.2 were not all constant-time. Modular multiplications are, in fact, computed in constant time, but a modular exponentiation scans through the exponent and utilizes the so-called square-and-multiply algorithm which computes

[3] At least one exception for this rule would be if the interface between the untrusted SW and the trusted HW is slow making the operation slower than direct computation in the SW.

a squaring for each bit of the exponent, but an additional multiplication is computed only if an exponent bit is one. Hence, the computation time depends on the Hamming weight of the exponent. If the untrusted SW can accurately measure the latency of exponentiation, then it learns the number of ones in the exponent. This leakage can be fully prevented by using a constant-time exponentiation algorithm such as square-and-multiply-always or Montgomery's ladder [14] and they can be easily implemented for the multi-core architecture by updating its microcode. This would introduce a moderate performance penalty because the number of multiplications required for an exponentiation increases approximately by 33 % (from about 1.5 times the bit length of the exponent to 2 times). In the case of HW-assisted trust model, constant-time exponentiations should be used for both encryptions and decryptions.

The memory interface of the implementation of the multi-core architecture from D5.3 gives the SW a full access to the internal values of the crypto cores. Under the HW-assisted trust model, this access could be used by the untrusted SW either to read security-critical values directly or some intermediate values that would give indirect information about the security-critical values. Hence, the interface should be changed so that the SW can access only a part of the memory in the FPGA. This memory region should be used for communication between the SW and HW. All security-critical values and intermediate values of computations should be stored into a secure memory region that cannot be access from the SW. The separation of the memory regions could be implemented on the hardware level, e.g., by not allowing the SW to control the most significant bits of the address signal[4] or by using different memory blocks (BlockRAMs) inside the FPGA for the two memory regions.

The architecture discussed in D5.3 is based on microcodes. The SW loads microcodes (sequencies of instructions) into the HW side that determine how the HW side operates. A compromised SW side could, therefore, get control of the HW side (e.g., to write all secret values into the insecure memory regions). Hence, it is essential that the microcodes are not controlled by the SW side. This could be arranged, e.g., so that the HW side extended with microcode ROMs that store the secure microcodes.

We will next discuss the specific additional requirements for the implementation of encryptions with the algorithm of Fig. 2. The masks of the multi-input FE scheme, $\mathbf{u}$, must remain secret and should, therefore, be stored securely in the FPGA. In practice, this means that these values can be embedded into the configuration bit-file that is used for configuring the FPGA at power up. Modern FPGAs (also Xilinx Zynq-7020) support bit-file encryption that prevents these values from leaking to an adversary who analyses the bit-file. The decryption key for the encrypted bit-file is hard-wired into the FPGA. An encrypted bit-file can be decrypted only with a (set of) specific FPGA(s) that have the corresponding hard-wired decryption key and, therefore, the design can be bound to (a) specific FPGA(s).

The requirement of not having any of the security-critical values in the SW mandates that, in order to securely compute encryptions with the algorithm of Fig. 2, the architecture implemented on the trusted FPGA must be able to generate random numbers that are needed for the random r in Fig. 2. Hence, the multi-core architecture should be extended with a true random number generator (TRNG). A pseudo-random number generator (PRNG) initialized with a secret seed embedded into the bit-file (similarly to $\mathbf{u}$) would be problematic because one would need to be able to keep the state between power-ups in order to prevent the same randomness from being

[4] E.g., for a 12-bit address signal, the SW could control only the lowest 8 bits, consequently, restricting its access to the addresses 0x000...0x0ff and leaving the addresses 0x100...0xfff for the secure region.

generated multiple times. This is difficult because most commercial FPGAs are volatile and, hence, having a TRNG in the FPGA appears to be a better choice.

Finally, we discuss the specific additional requirements for decryptions with Fig. 3. As discussed above, the decryption operation uses secret keys d_i for $i = 0, \ldots, n-1$ and z. These values should be embedded into the architecture similarly to $\mathbf{u}$ in the case of encryption. Otherwise, we have not identified specific additional requirements for computing decryptions of Fig. 3 with the multi-core architecture from D5.3 under the HW-assisted trust model.

3.3 Pairing computations with a trusted HW

3.3.1 Pairing algorithm

Fig. 4 takes two inputs P and Q, of which typically one is the security critical parameter that needs to be protected and, thus, isolated from the untrusted SW. As can be seen in Fig. 4, neither P or Q affects the high-level control flow of the algorithm as the for-loops and if-clauses are controlled by fixed (public) parameters. This is a notable difference compared to many other algorithms used in public key cryptography such as exponentiations (e.g., the square-and-multiply algorithm) and elliptic curve scalar multiplications (e.g., the double-and-add algorithm). Because there is no high-level dependency on the secret values, the side-channel leakage is also smaller than in the aforementioned algorithms, where the secret key bit directly determines whether multiplication or point addition are computed or not in one iteration of the algorithm.

Nevertheless, side-channel attacks have been introduced against pairing algorithms. Most of them require physical proximity as they need ability to perform power measurements (e.g. [12, 6, 10]) or to induce deliberate faults into the computation (e.g., [15]). However, timing attacks have been also introduced (e.g., [12]) and it is evident that pairing computations must be constant-time in order to be secure under the HW-assisted model.

Constant-time pairing computations can be realized with constant-time modular operations in different hierarchical levels of the tower field arithmetic in $\mathbb{F}_{p^{12}}$. In HW implementations this typically means that the underlying $\mathbb{F}_p$ must be constant-time because higher hierarchical levels are implemented via fixed sequencies of $\mathbb{F}_p$ operations. In the first computationally involved part of pairing computations, namely, the Miller loop, the elliptic curve arithmetic and line evaluations in lines 3, 5, 7, and 8 of the algorithm in Fig. 4 are computed via fixed sequencies of field arithmetic operations. An interesting observation is that if the result f of a pairing computation is not security-critical, then the final exponentiation, which is the other computationally involved part of pairing computations, is not security-critical either because the final exponentiation is easy to invert. If f and the final exponentiation are security-critical, then constant-time $\mathbb{F}_{p^{12}}$ arithmetic suffices for a constant-time implementation of the final exponentiation. Consequently, realizing constant-time pairing computations in HW implementations boils down to constant-time $\mathbb{F}_p$ arithmetic.

To summarize, in a HW/SW codesign, where the SW side is untrusted, the lines 1–9 of the algorithm in Fig. 4 should be executed in the trusted HW in constant time to protect a secret Q or P. The line 10 can be computed in the untrusted side (assuming that f is not security-critical).

3.3.2 Pairing computations with the pairing cryptography processor

The requirements for mapping the PCP, which was discussed in D5.3 and originally developed for the HW-optimized trust model, to the HW-assisted trust model are very similar to those for the FE multi-core architecture discussed in Section 3.2.3.

As mentioned above, an implementation under the HW-assisted trust model must be constant-time. The PCP is already fully constant-time so this requirements is filled already based on the version prepared under the HW-optimized trust model. The part that requires most significant revisions is the interface with the SW side. In the HW-optimized version, the SW side is responsible of loading specific microcodes into the PCP during a pairing computation. A compromised SW could load malicious microcodes and obtain the security-critical values with them. Hence, all microcodes that are required in a pairing computation should be hardwired into the PCP, they should be stored in a secure ROM inside the trusted component, or microcodes loaded by the untrusted SW should be authenticated within the PCP. All these require additional resources in the trusted component besides the plain PCP core including a larger IMEM, an additional ROM component, or means to authenticate the microcodes (e.g., a message authentication code (MAC) core). The current interface also allows accessing the entire memory space of the PCP and this should be changed so that the memory is split into two memory regions, of which only one can be accessed by the SW side, similarly as for the FE implementation discussed above.

Again, the security-critical values (either P or Q) should be securely interfaced into the PCP. The approach proposed for the FE implementation, where $sk_{\mathbf{y}}$ is hardwired into the bit-file, is most likely not a viable option in most applications, because the value must be changed frequently. Hence, there must exist a way to securely input new security-critical values into the PCP. This can be done either (a) via a secure port that connects directly to the part of the trusted component that is responsible for generating this value or (b) via the untrusted SW side so that the values are encrypted and decrypted within the PCP. The option (a) may be the most viable option in the context of advanced FE schemes where the trusted component must include also other security-critical operations in addition to the pairing computations and, then, it is natural to communicate these values to the PCP via an additional secure port that connects these two parts inside the trusted component without the need to use the untrusted SW side as a middle man.

It was mentioned above that the final exponentiation does not need to be computed within the PCP. However, it is probably advantegous to do so because the PCP is an accelerator and the performance will likely increase if the final exponentiation is also accelerated with the PCP.

The results of the pairing implementation have been published also as a conference article in [3].

4 Trusted component

In this section, we will focus on Case 2 from Section 2, where the system has a trusted component either (a) as a part of the processor running SW (e.g., TrustZone) or (b) as an isolated part of the HW. In the latter case, the trusted component may be a separate chip (e.g., a commercial TPM) or a specific security module integrated into a larger HW-based system with other modules (e.g., in an FPGA). The rest of the system (SW, HW, memory and interfaces) are untrusted in the sense that they can potentially be compromised by an adversary. The trusted component is expected to be protected from the adversary because it is assumed to have sufficiently strong protections against all types of implementation attacks that the adversary can launch against the system. These can be significantly stronger assumptions than those used in Section 3 and may include, e.g., attacks that require physical proximity (e.g., DPA). While these assumptions may not (fully) hold for all practical trusted component instantiations (e.g., Intel SGX does not claim protection against side-channel attacks [11]), in this deliverable we assume them to hold and focus on how to implement FE schemes under the HW-assisted trust model that includes a strong trusted component.

Typically trusted components (e.g., commercial TPMs) have a very limited set of supported functions and their performance is low compared to SW and, in particular, to HW-based cryptography accelerators. Therefore, it is of interest to minimize the number and complexity of operations that must be computed in the trusted component. All operations of FE schemes that are not security-critical should be computed in the untrusted side (either SW or HW) and only the ones that are essential for the security of the system should be executed in the trusted component.

If the trusted component is used for FE computations in an otherwise completely compromised system, then the adversary naturally has already gained a lot. For example, even if the FE encryptions or decryptions (inner-product computations) are fully computed inside the trusted component, the adversary has obtained an access to an encryption and/or decryption oracle and may be able to cause many types of problems in the larger system. Nevertheless, the adversary is considered successful only if he or she is able to break the security guarantees of the FE scheme implemented using the trusted component. In this case the adversary is successful if he or she obtains the security-critical values of the FE and, consequently, gains the ability to perform FE encryptions or decryptions even without the trusted component.

Having a trusted component in the system changes the trust model of general FE schemes. In a general FE scheme anyone who has sk_i is able to encrypt and anyone with sk_f is able decrypt. E.g., the evaluator having sk_f can use the same key in different computing platforms to compute the function f or even give the same ability to another party simply by handing out sk_f. Introducing a trusted component into the system binds the ability to encrypt or decrypt to a specific piece of hardware. This may open up new types of applications in the future, but these possible new applications are not discussed further in this deliverable.

The only related work on using trusted components for FE that we are aware of was presented by Fisch et al. in [7] and considers using Intel SGX for implementing FE functionalities. Their solution relies on decrypting a ciphertext, which has been encrypted with traditional public-key encryption, in a secure enclave in Intel SGX and, then, computing the function f over the plaintext in the enclave. I.e., they do not have a FE scheme per se; they simply rely on the security guarantees offered by Intel SGX to implement functionalities comparable to FE. If the adversary is capable of breaking Intel SGX, then he or she may have access to the full plaintext. As already

explained above, we take another approach in FENTEC: We aim to build a true FE system where f is not computed over the plaintext and, thus, plaintext leakage from the evaluator side is not possible even if the trusted component is broken (unless the FE scheme is mathematically broken). In our setting, the trusted component is used for protecting the keys sk_i and sk_f for encryption and decryption, respectively, and for binding the corresponding capabilities to a specific piece of hardware. A successful adversary would be able to break this binding and gains the capability to perform these operations also without the specific piece of hardware.

4.1 Case study: MIFE with ARM TrustZone

Next, we shall have a closer look on how a trusted component, more specifically ARM TrustZone for ARMv7-A and ARMv8-A, should be used in the case of the multi-input inner-product FE scheme based on Paillier encryption from [1, 2] that was discussed in Section 3.1.1 and in D5.3.

ARM TrustZone is an embedded security technology built into the ARM CPUs that partitions the execution domain into two environments: a TEE and a "normal" environment (i.e., *non*-trusted), also referred to as the REE. This enforces a separation of trusted and non-trusted programs at the hardware level and enables many security features, such as process isolation.

In order to leverage ARM TrustZone capabilities on, for instance, a Linux system, a companion to the kernel (typically OP-TEE OS [13]) is required to interface between the two environments. With OP-TEE OS, the trusted programs consist of TAs that are securely pre-installed and digitally signed. The TAs are then called from an REE application through the API provided by OP-TEE client.

The goal of the following section is to show the benefits from the features of ARM TrustZone for MIFE, and explain how MIFE can be implemented as a TA.

4.1.1 Setup

In our case study, the implementation was intended to be useed by the NXP two-board solution of i.MX 8M Mini EVK [17] which includes four Cortex-A53 featuring ARM TrustZone. The motivation behind such a choice lies on the low power consumption and the video-processing capabilities of the board; both desirable qualities for a potential use by the IoT prototype in D7.7.

The operating system installed on the board is Ubuntu 18.04. The framework was provided by the NXP i.MX Yocto Project [16] (Revision 3.1) which contains a variant of OP-TEE OS (Version 3.8.0) optimized for the i.MX 8M.

The original source code for the Paillier-based MIFE scheme was based on the CiFEr library [19] which offers an implementation of FE primitives in C. In order for TAs to use functions from CiFEr, the GMP library [8] needed to be ported to OP-TEE OS (as done in [21]). The other third-party libraries normally required by CiFEr were disregarded and replaced by OP-TEE equivalents when available in the TEE Internal Core API. Most of the pre-existing code required to be modified in order to be compliant with the compilation toolchain.

4.1.2 Implementation

The goal is to provide an implementation of functional encryption for TrustZone as a trusted component in a hardware-assisted context. In particular, we discuss the advantageous features provided by OP-TEE in the case of MIFE to produce an implementation of functional encryption and decryption as TAs in a real-case scenario.

4.1.2.1 Functional Encryption

We model our functional encryption procedure as a TA designed for a single user that acts as a black-box encrypting software. As a result, security-critical values need to be secured within the software thanks to OP-TEE capabilities. As identified in Section 3.2, the procedure relies on two security-critical values: a random nonce r and the encryption key $\mathbf{u}_i$.

- The random nonce $r \in \{2, \ldots, \lfloor N_i/4 \rfloor\}$ can be generated inside the TEE using the secure random number generation of OP-TEE [9]. This is done by calling the function `TEE_GenerateRandom` which fills a given array with random uniform bytes. The generated bytes simply need to be converted into a GMP multi-precision variable after checking for a potential modulo bias.
- The encryption key $\mathbf{u}_i = (u_{i,0}, u_{i,1}, \ldots, u_{i,m-1})$ with $u_{i,j} \in \mathbb{Z}_L$ is secretly known by user i and does not change from one execution to another. Consequently, such information can be securely stored at rest with OP-TEE secure storage [20] which, depending on the needs, either relies on the REE filesystem or on a RPMB file partition:
  - In the secure storage based on the REE filesystem (i.e., the Linux filesystem), the content of a file owned by a TA is encrypted with a key only accessible by said TA. Such practice ensures confidentiality and isolation. However, these files are easily erasable and editable with an elevated privilege escalation, so there is no protection against denial of service nor replay attacks.
  - To address the lack of protection against replay attacks, OP-TEE also supports an RPMB filesystem in case the board uses an embedded Multi-Media Controller (eMMC) which offers such a feature. In this case, the TAs can detect replay attacks through OP-TEE OS and fail safely.

Furthermore, many ARM devices feature a cryptographic engine which provides a suite of hardware-accelerated cryptographic services that could be used in TrustZone to gain both in performance and security. As a result, since exponentiations form the majority of the workload in encryption, the Paillier-based functional encryption procedure would benefit from a cryptographic unit by relegating these computations to the module, in case such operation is supported.

Trusted application.

In the context of our use-case implementation, we wrote a programmable TA that can switch easily between instances of MIFE. In particular, the keys are not held in the secure storage, as recommended in our real-case scenario analysis of Section 4.1.2.1. The TA offers three commands: `FE_CMD_KEYGEN`, `FE_CMD_ENCRYPT`, and `FE_CMD_FREE`.

- The `FE_CMD_KEYGEN` command initializes a Paillier-based MIFE instance with the parameters provided by the client. Concretely, the function receives N, m, and Y as parameters, as

well as a bound for the inputs[5]. The encryption key and the corresponding decryption key are generated and held inside the trusted application as global variables.

- The `FE_CMD_ENCRYPT` command encrypts the input provided by the client with the initialized instance and returns the ciphertext to the client. Obviously, such command can only be called if `FE_CMD_KEYGEN` was called beforehand.
- The `FE_CMD_FREE` command releases the encryption and decryption keys generated by the `FE_CMD_KEYGEN` command so a new instance can be initialized.

4.1.2.2 Functional Decryption Similarly to functional encryption, we model our functional decryption procedure as a TA that acts as a black-box decrypting software intended to be used only when all the ciphertexts were computed by all users.

The decryption key $d_0, \ldots, d_{n-1}$ is the only security-critical value in the procedure that need to be securely stored at rest, using either of the secure storage solutions described in Section 4.1.2.1 since, as discussed in Section 3.2, z could be outsourced to the untrusted domain, because the partial results p_i in Fig. 3 do not reveal information about the actual partial or final inner-products and already having all d_i in the trusted component prevents the untrusted domain from computing MIFE decryptions without the trusted component.

Also, since functional decryption also consists mostly of modular exponentiations, a cryptographic engine which supports such an operation will improve the performance and security of the procedure.

Trusted application.

For simplicity, the TA used here is a direct extension of the TA from the functional encryption trusted-world benchmark. As a result, the three commands `FE_CMD_KEYGEN`, `FE_CMD_ENCRYPT`, and `FE_CMD_FREE` are still available with the addition of the new command: `FE_CMD_DECRYPT`. The secure storage solution was still unused to store the decryption keys.

- The `FE_CMD_DECRYPT` command functionally decrypts the ciphertexts provided by the client with the decryption key and returns the product of all inputs. Again, `FE_CMD_KEYGEN` requires to be called beforehand.

4.1.3 Results and Discussion

The resulting TA source code for both functional encryption and decryption is available in FENTEC GitHub repository[6].

In this case-study, we discussed the interesting features that OP-TEE and TrustZone could provide for functional encryption primitives. In particular, we explained the implementation details of MIFE based on Paillier as a TA for OP-TEE, which we have shown as a Proof of Concept available for the public.

It is also important to note that the key distribution from the key authority to the trusted component should be done securely because the untrusted SW acts as a man in the middle in this communication. Therefore, the future research will also need to consider how the key distribution

[5] The input bound was actually hard coded in the trusted application source code.
[6] https://github.com/fentec-project/optee-MIFE/

is implemented securely. One option to securely distribute keys for the trusted component is to build a secure connection between the key authority and the trusted component (e.g., with TLS). In the case of FPGA-based trusted components, the key authority could also provide an encrypted configuration bit-file for the trusted component that embeds the secret values.

In the future, an analysis of the underlying cost in terms of performance with and without TrustZone could be undertaken. This performance could also be accelerated with the use of the crypto engine, whose actual results still need to be derived. Finally, the programmed TA could be borrowed in a real-case prototype, such as D7.7.

5 Conclusions

In this deliverable, we discussed how FE schemes can be implemented under the HW-assisted trust model where a part of the system is trusted but the other parts are untrusted in the sense that they can be compromised by an adversary. We focused particularly on two different assumptions about the trusted part: the first case, where the entire HW is trusted, and the second case, where there is only a specific trusted component. The first case was motivated by an assumption that the adversary is remote and can possibly exploit SW bugs etc. in order to compromise the SW, but the HW still remains secure. In this case, the HW was primarily used as an accelerator but because of this security advantage, it also plays the role of a trusted component in the system. The second case assumed a more powerful adversary, who may have physical access to the computation platform and can launch elaborated side-channel attacks to compromise also the HW. However, the second case assumed that the computation platform has a trusted component (e.g., a commercial TPM chip or a TEE in the processor) that includes strong protections against even such powerful adversaries. Such a trusted component can, thus, provide trust into the system and to the FE computations performed in it. The performance offered by such a strongly protected component was assumed low. This assumption typically holds in practice because commercial trusted components often have only limited computational power.

We studied these two cases particularly in the context of the multi-input inner-product FE scheme based on Paillier encryption from [1, 2] but we considered also cryptographic pairings, which are used as building blocks for FE schemes. We identified the security-critical operations and values in this FE scheme and showed how it can be mapped efficiently and securely into the two cases mentioned above. Finally, this deliverable presented a prototype implementation of FE using ARM TrustZone.

The work done for this deliverable shows that FE schemes can be implemented so that they are compliant with the hardware-assisted trust model without excessive overheads. This increases the level of security of the FE schemes because the SW side is no longer security critical. Because the work was versatile in the sense that it considered different types of FE schemes (in particular, based on Paillier and pairings) as well as different types of trusted components (that is, the whole HW and ARM TrustZone), it has potential to be used for improving security of FE solutions with different kinds of setups and systems.

References

[1] Michel Abdalla, Dario Catalano, Dario Fiore, Romain Gay, and Bogdan Ursu. Multi-input functional encryption for inner products: Function-hiding realizations and constructions without pairings. In *Advances in Cryptology—CRYPTO*, volume 10991 of *LNCS*, pages 597–627. Springer, 2018. (Pages 3, 9, 10, 18, and 22)

[2] Shweta Agrawal, Benoît Libert, and Damien Stehlé. Fully secure functional encryption for inner products, from standard assumptions. In *Advances in Cryptology—CRYPTO*, volume 9816 of *LNCS*, pages 333–362. Springer, 2016. (Pages 9, 18, and 22)

[3] Milad Bahadori and Kimmo Järvinen. Compact and programmable yet high-performance SoC architecture for cryptographic pairings. In *The 30th International Conference on Field-Programmable Logic and Applications (FPL 2020)*. IEEE, 2020. (Page 16)

[4] Milad Bahadori and Kimmo Järvinen. A programmable SoC based accelerator for privacy enhancing technologies and functional encryption. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(10):2182–2195, October 2020. https://doi.org/10.1109/TVLSI.2020.3010585. (Page 13)

[5] Jean-Luc Beuchat, Jorge E González-Díaz, Shigeo Mitsunari, Eiji Okamoto, Francisco Rodríguez-Henríquez, and Tadanori Teruya. High-speed software implementation of the optimal ate pairing over Barreto–Naehrig curves. In *Pairing-Based Cryptography — Pairing 2010*, volume 6487 of *LNCS*, pages 21–39. Springer, 2010. full version: https://eprint.iacr.org/2010/354. (Page 11)

[6] Johannes Blömer, Peter Günther, and Gennadij Liske. Improved side channel attacks on pairing based cryptography. In *Constructive Side-Channel Analysis and Secure Design (COSADE'13)*, pages 154–168. Springer, 2013. (Page 15)

[7] Ben Fisch, Dhinakaran Vinayagamurthy, Dan Boneh, and Sergey Gorbunov. IRON: Functional encryption using Intel SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*, pages 765–782. ACM, 2017. (Page 17)

[8] Free Software Foundation. The GNU MP Bignum Library, 2000–2020. Web page, https://gmplib.org/, (retrieved Sep. 20, 2020). (Page 18)

[9] GlobalPlatform Technology. *TEE Internal Core API Specification*, 2018. Version 1.1.2.50 (Target v1.2). (Page 19)

[10] Damien Jauvart, Nadia El Mrabet, Jacques JA Fournier, and Louis Goubin. Improving side-channel attacks against pairing-based cryptography. *Journal of Cryptographic Engineering*, 10:1–16, 2019. (Page 15)

[11] Simon Paul Johnson. Intel® SGX and side-channels, 2018. Web page, https://software.intel.com/en-us/articles/intel-sgx-and-side-channels, (retrieved Jun. 6, 2019). (Pages 8 and 17)

[12] Tae Hyun Kim, Tsuyoshi Takagi, Dong-Guk Han, Ho Won Kim, and Jongin Lim. Side channel attacks and countermeasures on pairing based cryptosystems over binary fields. In *Cryptology and Network Security (CANS'06)*, pages 168–181. Springer, 2006. (Page 15)

[13] Linaro Limited. Open portable trusted execution environment - OP-TEE, 2020. Web page, https://www.op-tee.org/, (retrieved Sep. 20, 2020). (Page 18)

[14] Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987. (Pages 12, 13, and 14)

[15] Dan Page and Frederik Vercauteren. A fault attack on pairing-based cryptography. *IEEE Transactions on Computers*, 55(9):1075–1080, 2006. (Page 15)

[16] Yocto Project. Yocto Project – it's not an embedded linux distribution – it creates a custom one for you, 2020. Web page, https://www.yoctoproject.org/, (retrieved Sep. 20, 2020). (Page 18)

[17] NXP Semiconductors. i.MX 8M Mini applications processor — Arm Cortex A53/M4 — 1080p display — NXP, 2020. Web page, https://www.nxp.com/products/processors-and-microcontrollers/arm-processors/i-mx-applications-processors/i-mx-8-processors/i-mx-8m-mini-arm-cortex-a53-cortex-m4-audio-voice-video:i.MX8MMINI, (retrieved Sep. 20, 2020). (Page 18)

[18] Hovav Shacham and Dan Boneh. Improving SSL handshake performance via batching. In *Topics in Cryptology—CT-RSA*, volume 2020 of *LNCS*, pages 28–43. Springer, 2001. (Page 13)

[19] Lewandowski Marco Tilen Marc, Hartman Jan. fentec-project/cifer: Functional encryption library in C, 2020. Web page, https://github.com/fentec-project/CiFEr, (retrieved Sep. 20, 2020). (Page 18)

[20] TrustedFirmware.org. Secure storage — OP-TEE documentation documentation, 2019–2020. Web page, https://optee.readthedocs.io/en/latest/architecture/secure_storage.html, (retrieved Sep. 20, 2020). (Page 19)

[21] Guita Vasco. vascoguita/optee_gmp: GMP library for OP-TEE OS, 2020. Web page, https://github.com/vascoguita/optee_gmp, (retrieved Sep. 20, 2020). (Page 18)