DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Implementation of Attribute-Based Encryption in Rust on ARM Cortex M Processors

Daniel Bücheler

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Implementation of Attribute-Based Encryption in Rust on ARM Cortex M Processors

Implementierung von Attributbasierter Verschlüsselung in Rust auf ARM Cortex M Prozessoren

Author:	Daniel Bücheler
Supervisor:	Prof. Dr. Claudia Eckert
Advisor:	Stefan Hristozov
Submission Date:	07.04.2021

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 07.04.2021 Daniel Bücheler

Acknowledgements

Abstract

Attribute-Based Encryption (ABE) enforces flexible access control by allowing implicit specification of intended recipients. Ciphertexts and users are described by attributes, and decryption is possible only if they match according to a given policy. The flexibility of ABE and expressiveness of its policies make it very useful for many scenarios, e.g. encrypted communication with multiple addressees, especially if the group of recipients is not predetermined.

Most ABE schemes are constructed with bilinear pairings of elliptic curves. These are computationally very expensive, and thus ABE has found limited use the IoT.

This thesis gives an overview of ABE and evaluates its applicability on ARM Cortex-M4 processors. The runtime, RAM usage and size of the executable are evaluated on a SoC with a 64 MHz Cortex-M4 CPU and 256 KB RAM. Two Key-Policy ABE schemes are implemented in a library for embedded systems; one of the schemes is a pairing-free scheme. While the security of pairing-based schemes is better, the pairing-free scheme provides significantly better performance. Encryption is much slower with the pairing-based scheme, but feasible with both schemes if if a delay of a few seconds is acceptable and the number of attributes is not too large. Decryption is only practical with the pairing-free scheme.

Contents

1. Introduction

Attribute-Based Encryption (ABE) promises to provide security while improving flexibility over standard cryptosystems. Its approach of combining attributes and policies describing admissible attribute combinations is closer to traditional notions of access control (e.g. RBAC) [12].

This increased flexibility is also desirable for IoT applications. However, the Internet of Things (IoT) is especially affected by the trade-off between cost and security: Because hardware must be cheap and has to run on batteries, it is often highly constrained. Many traditional security measures, even some standard cryptographic schemes, are not practical on IoT devices. Compared to standard cryptgraphy, most ABE schemes demand even more resources [43]. Thus, the feasibility of ABE on constrained IoT nodes is disputed [43, 3, 4, 19, 16].

The objective of this thesis is to assess to what extent ABE can be practically applied on ARM Cortex M4 processors. To this end, an ABE library is developed using the Rust programming language. This library is then tested and evaluated on the nRF52840 SoC with a 64 MHz ARM Cortex M4 processor and 256 KB of RAM. In addition, this thesis aims to give an easy-to-understand explanation of ABE and how it can be implemented.

For a real-world use-case of this library in the medical field, see Figure 1.1. It is assumed that the sensor (e.g. an ECG or blood glucose sensor) is unable to communicate with the internet directly. Therefore, the data is sent to a gateway by Bluetooth Low Energy (BLE) and then uploaded to the cloud.

The goal of this project is to enable end-to-end Attribute-Based Encryption: Measurements are encrypted on the sensor before they are transmitted over BLE and only decrypted when read by an authorized client (e.g. the attending doctor). In this system, the sensor exclusively needs to encrypt data; decryption, key generation and setup are not required on the constrained node, which is the case in many IoT applications.

The end-to-end ABE approach does not require trust in the gateway or cloud provider. Trusting them would be an unreasonable assumption in many scenarios.

An alternative to encryption with ABE on the sensor would be outsourcing the ABE operations to the gateway: The sensor and gateway share a secret key, which is used to secure the communication over BLE. The gateway then decrypts the symmetrically encrypted data and re-encrypts it using ABE. This approach does not require ABE on the sensor, but only on the gateway. The latter is assumed to be much more powerful

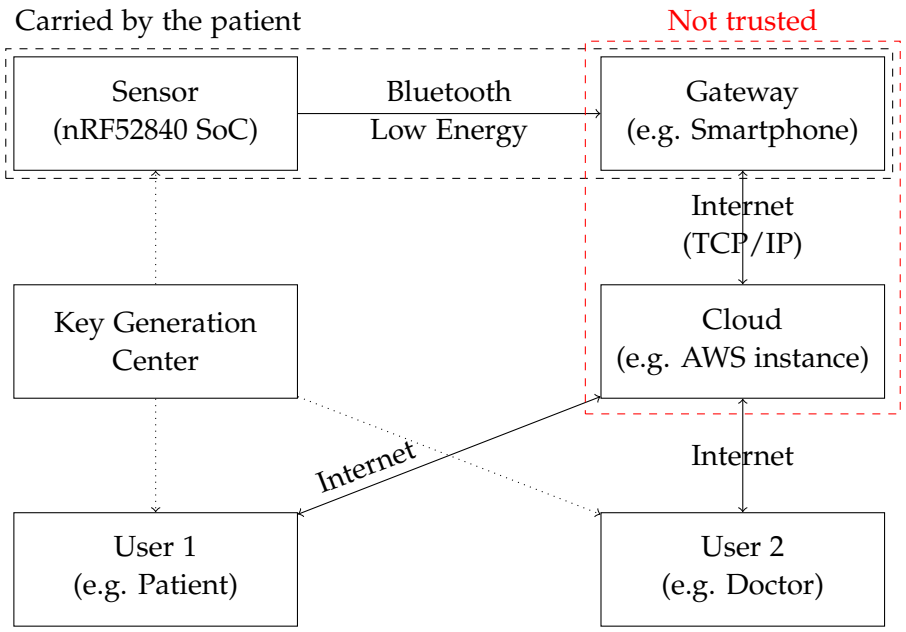


Figure 1.1.: Simplified use case for end-to-end Attribute Based Encryption with encryption on a constrained sensor MCU. The ABE library developed for this thesis runs on the sensor.

than the sensor. However, the gateway must be trusted in this scenario.

An alternative to ABE itself would be to let the cloud server enforce access control: All participants exchange secret keys with the cloud server. The sensor then uses this key to encrypt the data, which only the cloud server can read. When a user wants to read some data, they request it at the cloud server. If the cloud server grants their request, it re-encrypts the data with the respective user's key and transmits it to them. This approach removes trust in the gateway and requires only symmetric cryptography, which is much faster than ABE or asymmetric encryption schemes. The disadvantage is that the cloud server knows all data in plain text. It thus needs to be trusted and represents a single point of failure.

End-to-end ABE does not, however, remove the single point of failure: A malicious Key Generation Center (KGC) could issue keys that allow decryption of arbitrary data (see Section 2.2.1). However, the Key Generation Center (KGC) is only required once to setup the ABE system and issue the participant's keys. It is not involved in any encryption or decryption operations. The KGC does not need to be online during the use of the system and could e.g. be located on an air-gapped network. This greatly reduces the attack surface.

2. Preliminaries

This chapter introduces Attribute-Based Encryption and the relevant mathematical background for implementing it.

2.1. Confidentiality with Classic Symmetric and Asymmetric Cryptography

Today's conventional cryptography knows two main classes of cryptosystems: symmetric encryption schemes and asymmetric encryption schemes. See Figure 2.1 for an illustration of the differences.

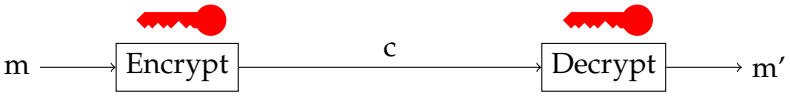
Consider n participants wanting to communicate securely (i.e. no user can read encrypted messages between two other users). Using a symmetric encryption scheme, each participant would need to agree on a unique key with every other participant, resulting in a total number of $\frac{n(n-1)}{2}$ keys. Using a asymmetric encryption scheme reduces the number of keys to only n, because each participant could obtain everyone else's public key and then send messages to them securely.

Another problem remains, however: Encrypting a single message to a large number of participants requires encrypting it with everyone's public key separately. For a large number of recipients, this is inefficient. So, for example, to encrypt a message for all students of a certain university, we'd need to obtain each student's public key and encrypt the message with each key separately.

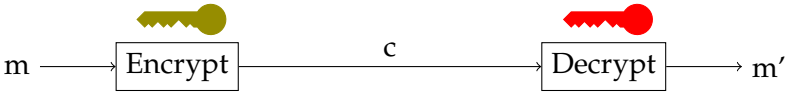
Even worse, what if we want to encrypt data for any student of said university, even if they *haven't joined the university yet*. In this case, our only option using classic asymmetric cryptography would be to have some trusted instance keep a plaintext copy and re-encrypt the data for any new student when they join the university. Attribute-Based encryption solves this problem.

2.2. Attribute-Based Encryption

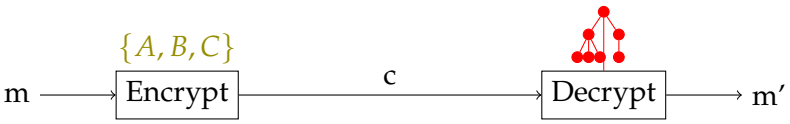
Attribute-Based Encryption (ABE) uses a combination of attributes to define a *group* of private keys that should be able to read encrypted data, instead of encrypting it for one



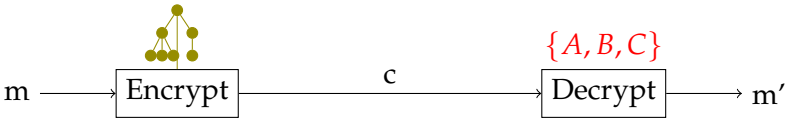
(a) Symmetric Encryption: Both keys are identical.



(b) Asymmetric Encryption: Different keys for encryption and decryption. Decryption succeeds if and only if the decryption key is exactly the counterpart to the encryption key.



(c) Key-Policy Attribute-Based Encryption: Attributes for encryption, access structure for decryption. Decryption succeeds if and only if the attributes of the ciphertext satisfy the policy embedded in the key.



(d) Ciphertex-Policy Attribute-Based Encryption: Access policy for encryption, attributes for decryption. Decryption succeeds if and only if the attributes of the key match the policy embedded in the ciphertext.

Figure 2.1.: Keys used for encryption and decryption in different classes of encryption schemes. Red information has to be kept secret, green information may be made publicly available. For the differences between the two types of ABE, see Section 2.2.3.

specific private key only (as in asymmetric encryption schemes). In Figure 2.1d, this is represented by a tree.

The combination of attributes may be as restrictive or permissive as needed. It is possible to create ciphertexts that can be read by almost all members of an ABE scheme, and ciphertexts that can be read by nobody except a few selected participants.

Figure 2.2 shows a small ABE system with the KGC initializing the system and issuing keys, and two users sharing an encrypted message.

2.2.1. Attributes and the Key Generation Center

In essence, attributes are strings describing certain characteristics or features of actors and objects. For example, a typical freshman student of informatics at TUM could be described by the attributes `"semester count 1"`, `"computer science"`, `"tum"`, `"is young"`, `"started degree in 2017"`.

These attributes themselves don't contain any information to which users or object they apply; instead this is a matter of interpretation. Some attributes may be very clearly defined, e.g. `"started degree in 2017"` from above. For others, it may be more difficult to decide whether they apply, e.g. the attribute `"is young"`: Until what age is a student young?

In any instance of ABE, there needs to exist an arbiter who decides whether an attribute applies to a certain user or object. This role is assumed by a trusted third party, the Key Generation Center (KGC). It has two main responsibilities: First, the KGC decides which attribute applies to which user. Second, it issues private keys corresponding to these attributes, and hands these to the users.

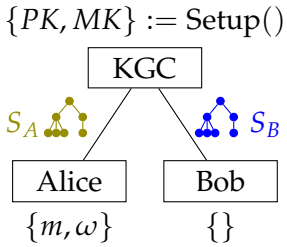
Without this KGC, there is no ABE. This differs from traditional public-key encryption schemes, where any user can independently create their own keypair.

Regarding the set of possible attributes (called the *attribute universe*), there are two possibilities: In a large universe construction, all possible strings can be used as attributes [20]. In a small universe construction, the universe of attributes is explicitly fixed when the system is instantiated, i.e. when the KGC runs the *Setup* algorithm (see below, section 2.2.2) [20]. With a small universe construction, the size of the public parameters usually grows with the size of the attribute universe [20].

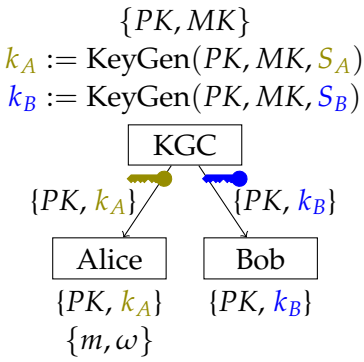
2.2.2. Formal definition of an ABE Scheme

We will define a KP-ABE scheme here, for the difference between CP-ABE and KP-ABE and formal definitions of Access Trees, see the next sections.

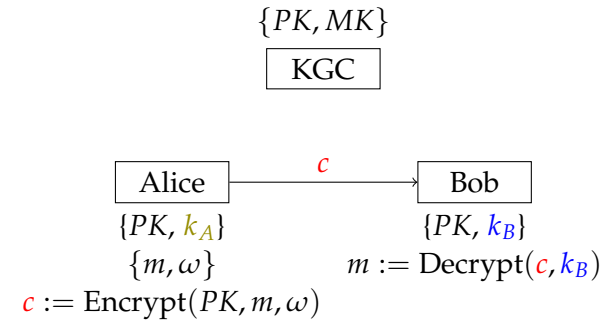
Definition 2.1. A (Key-Policy) Attribute-Based Encryption scheme consists of the following four algorithms: [20]



(a) The KGC sets up the system and obtains Alice's and Bob's desired access policies.



(b) If S_A and S_B are allowed access policies for Alice and Bob, respectively, the KGC creates their private keys and transmits them via a secure channel.



(c) Alice can now use the public parameters PK and ω to encrypt a message for Bob, which he can decrypt only if his access policy S_B is satisfied by ω. Note that the KGC plays no role in this step

PK	Public Key (same for all participants)
MK	Master Secret (known only to the KGC)
k_A, k_B	Private Decryption Keys
S_A, S_B	Access Structures
m	Plaintext message
ω	Set of attributes

Figure 2.2.: Alice wants to send an KP-ABE encrypted message m to Bob. She wants to encrypt the message under a set of attributes ω. Both Alice and Bob create a desired access policy, S_A and S_B, respectively. Note that the KGC will only issue a corresponding key if it deems that they should be allowed to obtain a key under the given access policy.

- *Setup*. Run once by the Key Generation Center (KGC). Sets up the system by generating public parameters PK and a private master key MK. The public parameters are shared with all participants, while the master key remains only known to the KGC.
- *KeyGen(PK, s, S)*. Input: public parameters PK, master secret s and access structure S.
  Run by the trusted authority once for each user to generate their private key. Returns a private key k corresponding to S.
- *Encrypt(PK, m, ω)*. Input: public parameters PK, plaintext message m and set of attributes ω.
  Run by any participant of the system. Encrypts m under ω and returns the ciphertext c.
- *Decrypt(c, k)*. Input: ciphertext c (output of *Encrypt*) and key k (output of *KeyGen*).
  Run by any participant holding a private key generated by *KeyGen*. Outputs the correctly decrypted message m' if and only if the set of attributes under which m was encrypted satisfies the access structure under which k was created.

The definition of a CP-ABE scheme is identical, except that *Encrypt(PK, m, S)* takes an access structure S and *KeyGen(PK, s, ω)* takes a set of attributes.

How exactly these algorithms work in concrete ABE schemes will be discussed in Chapter 4.

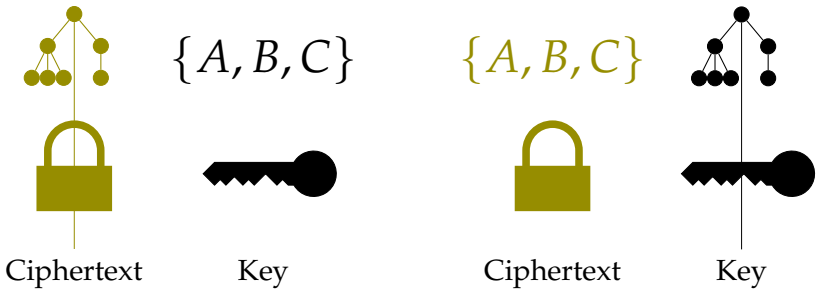
2.2.3. KP-ABE and CP-ABE

Two components are necessary to specify a group of keys that shall be able to decrypt a ciphertext: A number of attributes that are present, and a policy that defines a combination of required attributes. Each of these can either be associated with the ciphertext, or with the decryption key:

In Ciphertext-Policy ABE (CP-ABE), the key is associated with a set of attributes and the ciphertext is encrypted under an access policy. Key-Policy ABE (KP-ABE) works the other way around, so the ciphertext is associated with a set of attributes, and the key is associated with an access policy. See Figure 2.3 for illustration.

In both cases, a ciphertext can be decrypted if and only if the set of attributes specified in one part satisfy the access policy associated with the other part.

CP-ABE tends to be more intuitive because, when encrypting a plaintext, the encryptor controls rather explicitly who can decrypt their ciphertext: They set the access policy that defines which combinations of attributes are required from the users to successfully decrypt the ciphertext [12]. An example use case for CP-ABE in a hospital setting would



(a) CP-ABE (b) KP-ABE

Figure 2.3.: CP-ABE vs. KP-ABE: Association of key and ciphertext with Access Policy and set of attributes.

be sending an encrypted note about problems with a specific treatment to all doctors, patients that received that treatment and nurses of the department that administered the treatment. This could be specified by an access policy as (`hospital-name` AND (`doctor` OR (`patient` AND `received-treatment-x`) OR (`nurse` AND `department-y`)).

With KP-ABE, on the other hand, the encryptor doesn't have direct control over who will be able to access the data, except for the choice of attributes under which they encrypt the plaintext [12]. In the hospital setting from above, KP-ABE could be employed in a different use case: For encrypted storage of a patient's medical record, the patient's name could be used as an attribute in KP-ABE. If the patient sees a new doctor, they could simply have their key policy extended to include the patient's attribute. With CP-ABE, seeing a new doctor would require re-encrypting the entire data under a new access policy. Note that this scenario is very close to our use case from Figure 1.1.

With KP-ABE, instead of the encryptor, the Key Generation Center must be trusted with intelligently deciding which key to give to the decrypting party [12]. This property can be desirable: Consider a constrained IoT device as an encryptor, which can reliably transmit data, but not receive. If a new doctor must be given access to a patient's data, CP-ABE would require updating the policy on this device. With KP-ABE, policies are handled by the KGC, which is in general much more powerful and better-connected than encryptors might be.

2.2.4. Access Structures

Access structures formally determine which sets of attributes are required to reconstruct the ciphertext under ABE. This definition is adapted from [10] for our setting of allowed attribute sets, instead of allowed parties.

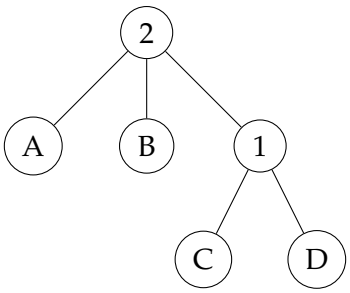


Figure 2.4.: Sample Access Tree over the attributes A, B, C, D.

Definition 2.2. Access Structure [10].
Let $U = \{A_1, \ldots, A_n\}$ be the universe of attributes. A set $\mathcal{A} \subseteq 2^U$ is monotone if for all $B \in \mathcal{A}$ and $C \supseteq B$, $C \in \mathcal{A}$. An access structure $\mathcal{A}$ is a non-empty subset of 2^U, i.e. $\mathcal{A} \in 2^U \backslash \{\emptyset\}$. A monotone access structure is an access structure that is monotone. The sets in $\mathcal{A}$ are called the *authorized sets*, those not in $\mathcal{A}$ are called the *unauthorized sets*.

Intuitively, the monotonicity of an access structure means that adding an attribute to an authorized set cannot result in an unauthorized set.

2.2.5. Access Trees

Explicitly specifying an access structure is not feasible, as its size may be exponential in the size of the attribute universe. Therefore, we will use the construction of *Access Trees* from Goyal et al. in [20]. Each leaf of this tree is labelled with an attribute, and each interior node is labelled with an integer, the threshold for it to be satisfied [20].

Figure 2.4 illustrates an example for an Access Tree. It is satisfied by any set of attributes that contains two of A, B and either C or D. That is, $\{A, B\}$ would satsify the tree, just as $\{B, D\}$ would, but $\{C, D\}$ would not be sufficient. For an attribute universe of $U = \{A, B, C, D\}$ this would realize the access structure $\mathcal{A} = \{\{A, B, C, D\}, \{A, B, C\}, \{A, B, D\}, \{A, C, D\}, \{B, C, D\}, \{A, B\}, \{A, C\}, \{A, D\}, \{B, C\}, \{B, D\}\}$

Definition 2.3. Access Tree [20].
An interal node x of an access tree is defined by its children and a threshold value d_x. If x has num_x children, then its threshold value satisfies $0 < d_x \leq num_x$.
A leaf node x is defined by an attribute and a threshold value $k_x = 1$.

[20] also defines the following functions for working with access trees: The parent of a node x in the access tree is denoted by $\text{parent}(x)$. If x is a leaf node, $\text{att}(x)$ denotes the attribute associated with x; otherwise it is undefined. The children of a node x are numbered from 1 to num_x. Then $\text{index}(y)$ denotes the unique index of y among the children of its parent node.

Definition 2.4. Satisfying Access Trees [20].
Let $\mathcal{T}$ be an access tree with root r and $\mathcal{T}_x$ the subtree with x as its root. If a set of attributes γ satisfies the access tree $\mathcal{T}_x$, we write $\mathcal{T}_x(\gamma) = 1$; otherwise $\mathcal{T}_x(\gamma) = 0$.
If x is a leaf node, then $\mathcal{T}_x(\gamma) = 1$ if and only if $\text{attr}(x) \in \gamma$.
If x is an internal node, then $\mathcal{T}_x = 1$ if and only if d_x or more of the children x' of x return $\mathcal{T}_{x'}(\gamma) = 1$.

The set of attribute sets that satisfy a tree $\mathcal{T}$ is then the access structure it represents: $\mathcal{A} = \{\gamma \in 2^U | \mathcal{T}(\gamma) = 1\}$. Note that $\mathcal{A}$ has to be monotone. It is not possible to specify that the *absence* of an attribute in the tree.

Using the threshold-gate construction, we can express *A AND B* as a node with two children A and B and threshold 2, and express *A OR B* as a node with two children A and B and threshold 1 [46].

2.3. Shamir's Secret Sharing

This secret sharing scheme based on polynomial interpolation was first introduced by Adi Shamir in 1979 [39]. It allows a secret s, which is generally just a number, to be shared among a number of n participants. The shares are computed such that s can be reconstructed if, and only if, at least k participants meet and combine their shares. Such a theme is then called a (k, n)-threshold scheme. [39]

2.3.1. Lagrange interpolation

Shamir's scheme makes use of a property of polynomials: A polynomial of degree d is unambiguously determined by $d + 1$ points (x_i, y_i). In other words, any polynomial of degree d can be unambiguously interpolated (reconstructed) from $d + 1$ distinct points.

To interpolate a polynomial of degree d from $d + 1$ given points $(x_1, y_1), \ldots, (x_{d+1}, y_{d+1})$, we can make use of the lagrange basis polynomials: [46]

Definition 2.5. Lagrange interpolation: Given a set of $d + 1$ points $(x_1, y_1), \ldots, (x_{d+1}, y_{d+1})$.

Then the polynomial

$$L(x) = \sum_{k=0}^{d} \Delta_{\omega, x_k}(x) \cdot y_k \tag{2.1}$$

is the lagrange interpolation polynomial for that set of points, where $\omega = \{x_1, \ldots, x_{d+1}\}$ and $\Delta_{\omega,k}(x)$ are the Lagrange basis polynomials:

$$\Delta_{\omega,k}(x) = \prod_{\substack{i \in \omega \\ i \neq k}}^{d} \frac{x - i}{k - i} \tag{2.2}$$

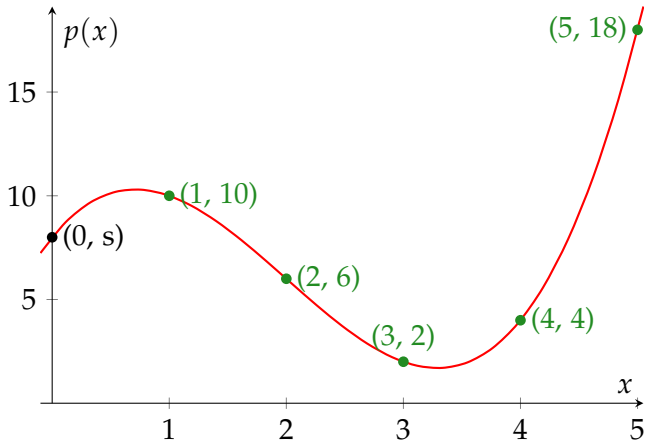


Figure 2.5.: Example for a $(5, 4)$-threshold scheme with $s = 8$ and $p(x) = 8 + 7x - 6x^2 + x^3$. The five green-colored points are distributed as the secret shares. Because $p(x)$ has degree three, at least four shares are required to reconstruct s.

This polynomial has degree d. If the points (x_i, y_i) lie on a d-degree polynomial, then the lagrange interpolation $L(x)$ is *exactly* that polynomial.

On the other hand, if there are less than $d + 1$ points of a d-degree polynomial known, there are infinitely many d-degree polynomials that pass through all given points. [39]

2.3.2. Secret sharing with polynomials

To share our secret, we now hide it in a polynomial and give out points on this polynomial as secret shares. Using the lagrange basis polynomials, we can then reconstruct $p(x)$ and thus the secret if we know enough shares [39].

Definition 2.6. Shamir's (k, n)-threshold secret sharing scheme [39]. To share a secret s among n participants such that s can be recovered if and only if k or more shares are combined, do:

1. Pick coefficients $a_1, ..., a_{k-1}$ at random
2. Set $a_0 = s$. This results in the polynomial $p(x) = a_0 + a_1 x + \cdots + a_{k-1} x^{k-1}$. Note that $p(0) = s$.
3. The secret shares are $(1, p(1)), (2, p(2)), \ldots, (n, p(n))$. Give one to each participant.

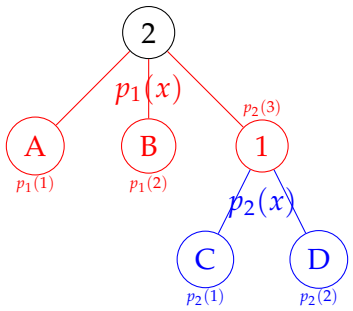


Figure 2.6.: Access Tree from Figure 2.4 showing how Shamir's Secret Sharing is employed recursively. $p_1(x)$ is a the polynomial of a $(2,3)$-threshold scheme, $p_2(x)$ of a $(1,2)$-threshold scheme. Shown in small are the secret shares embedded into each node.

To reconstruct the secret from any subset of k shares, interpolate the polynomial $p(x)$ and evaluate $p(0) = s$.

See also Figure 2.5 for illustration. In practice, the numbers would be far bigger and calculations wouldn't be performed over the real numbers, but rather a finite field modulo a prime [39].

2.3.3. Secret Sharing in Attribute Based Encryption

To realize an Access Tree that „gives away" a secret if and only if it is satsified by a set of attributes, we can recursively use Shamir's Secret Sharing scheme:

We use a secret-sharing polynomial on each internal node of the Access Tree: For a node x with threshold d_x and num_x children, we define a (d_x, num_x)-threshold scheme and embed one share of the secret in each child. Begin in the root, and set s as the secret we want to embed in the tree. For all other nodes, set s as the secret share received from the parent node.

If the child is a leaf, we modify the share such that it can only be used if the relevant attribute is present (how exactly this is done differs between CP-ABE and KP-ABE).

Now, let ω be a set of attributes. We have built our tree in such a way that the share embedded in a leaf node u can be used only if $\text{attr}(u) \in \omega$. That means, a leaf node's secret share can be used if and only if the set of attributes satisfies this leaf node.

For the internal nodes x, the use of a (d_x, num_x)-threshold scheme ensures that the secret embedded in x can be reconstructed if and only if the secret shares of at least d_x child nodes can be used, i.e. at least d_x child nodes are satisfied. Following this recursive definition up to the root, we can see that our secret s embedded in the root

can be reconstructed exactly if ω satisfies the Acess Tree.

See Figure 2.6 for an illustration with the tree from Figure 2.4: Two (k, n)-threshold schemes are employed, one for each internal node of the access tree. $p_1(x)$ is the polynomial of the root's $(2, 3)$-threshold scheme, sharing s, the secret to be embedded in the tree (i.e. $p_1(0) = s$). $p_2(x)$ is the polynomial for the $(1, 2)$-threshold scheme belonging to the node labelled "1" and shares the value $p_1(3)$ that it received from the $(2, 3)$-threshold scheme of the layer above (i.e. $p_2(0) = p_1(3)$).

2.4. Revocation

So far, it is not possible to take away privileges from a user: Once the private key has been issued, it can not be taken back. A user's capabilities can only be extended (e.g. by issuing a key with additional attributes or with a more permissive policy). This is is an problem, e.g. if their private key is compromised [14].

The simplest approach is to simply renew the keys of valid users from time to time [14]. When a user is revoked, their key will not be updated any longer. Thus any ciphertexts encrypted after the next key update will not be readable for the revoked user [14]

This approach requires the KGC to update or re-issue one key per valid user and requires a secure channel to the KGC [14].

Attrapadung and Imai [8] differentiate between *direct* and *indirect revocation*: With direct revocation, the list of revoked users is directly specified by the encrypting party (i.e. the encryption takes a "black list" of revoked users). Indirect revocation achieves revocation by means of updating the keys of valid users, as described in the naive approach above.

Direct revocation requires the encryptor to know the list of revoked users [8] (i.e. the encrypting party is responsible for correct revocation of the users on the revocation list). This can be a major drawback, especially in large systems or when encryptors are severely constrained (as in our case, Internet of Things (IoT) devices). With indirect revocation, the encryptor does not need to do anything except use the most recent version of the public parameters [8].

The advantage of direct revocation, however, is that it works instantly: As soon as an encryptor knows about the revocation of a user, they will include them on their revocation list for future encryptions. With indirect revocation, a revoked user can decrypt all ciphertexts created until the next key update is distributed by the KGC [8].

In either case, ciphertext encrypted before a user's revocation becomes effective, remains readable using the revoked key. Some schemes include a proxy-reencryption mechanism that allows an untrusted third party to update ciphertexts such that they

cannot be decrypted by revoked users [29].

2.5. Elliptic Curves

The mathematics of modern cryptosystems (including, but not limited to ABE) work any group that satisfies the axioms (see below), and elliptic curves are just one of them. Because Elliptic Curves allow for shorter key lengths than, e.g. groups modulo a prime, they have become very popular for use in cryptography. Exact definitions and notations differ, these are taken from the textbook *Introduction to Modern Cryptography* by Katz and Lindell [24].

2.5.1. Group Axioms

Definition 2.7. [24]. A *Group* $\langle \mathbb{G}, \circ, e \rangle$ consists of a set $\mathbb{G}$ together with a binary operation $\circ$ for which these four conditions hold:

- Closure: For all $g, h \in \mathbb{G}$, $g \circ h \in \mathbb{G}$.
- Existence of identity: There is an element $e \in \mathbb{G}$, called the *identity*, such that for all $g \in \mathbb{G}$, $g \circ e = g = e \circ g$.
- Existence of inverse: For every $g \in \mathbb{G}$ there exists an *inverse* element $h \in \mathbb{G}$ such that $g \circ h = e = h \circ g$.
- Associativity: For all $g_1, g_2, g_3 \in \mathbb{G}$, $(g_1 \circ g_2) \circ g_3 = g_1(\circ g_2 \circ g_3)$.

When $\mathbb{G}$ has a finite number of elements, the group $\mathbb{G}$ is called finite and $|\mathbb{G}|$ denotes the order of the group.

A group $\mathbb{G}$ with operation $\circ$ is called *abelian* or commutative if, in addition, the following holds:

- Commutativity: For all $g, h \in \mathbb{G}, g \circ h = h \circ g$.

When the binary operation is clear from context, we simply use $\mathbb{G}$ to denote the group.

We also define *Group Exponentiation*: $g \in \mathbb{G}, m \in \mathbb{N}^+$, then $mg = \underbrace{g \circ \cdots \circ g}_{m \text{ times}}$.

Usually, the symbol used to denote the group operation is not the $\circ$ from above, but either $+$ or $\cdot$. These are called *additive* and *multiplicative* notation, respectively. It is important to remember, though, that the group operation might be defined completely differently!

In multiplicative notation, the group exponentiation of $g \in \mathbb{G}$ with $m \in \mathbb{N}^+$ is written as g^m, in additive groups it is written as $m \cdot g$.

An example for an additive group is $\langle \mathbb{Z}_N, +_N, 0 \rangle$ with $\mathbb{Z}_N = \{0, 1, \ldots, N-1\}$ and $+_N$ the normal addition modulo N. This group is cyclic with generator 1. If p is prime, then $\langle \mathbb{Z}_p^*, \cdot_p, 1 \rangle$ is a group with $\mathbb{Z}_p^* = \{k \in \mathbb{Z}_p | \gcd(k, p) = 1\}$ and $\cdot_p$ the normal multiplication modulo p.

2.5.2. Elliptic Curves

Definition 2.8. Given a prime $p \geq 5$ and $a, b \in \mathbb{Z}_p$ with $4a^2 + 27b^2 \neq 0 \bmod p$, the Elliptic Curve over $\mathbb{Z}_p$ is: [24]

$$E(\mathbb{Z}_p) := \{(x, y) \mid x, y \in \mathbb{Z}_p \text{ and } y^2 = x^3 + ax + b \bmod p\} \cup \{\mathcal{O}\} \tag{2.3}$$

a and b are called the curve parameters, and the requirement that $4a^2 + 27b^2 \neq 0 \bmod p$ makes sure that the curve has no repeated roots [24]. The curve is simply the set of points $(x, y) \in \mathbb{Z}_p \times \mathbb{Z}_p$ that satisfy the curve equation $y^2 = x^3 + ax + b \bmod b$. One special point is added, the *point at infinity* denoted by $\mathcal{O}$. This will help define the point addition as a group operation in the next paragraph. [24]

2.5.3. Point Addition

Now, it is possible to show that every line intersecting a curve $E(\mathbb{Z}_p)$ intersects it in exactly three points, if you (1) count tangential intersections double and (2) count any vertical line as intersecting the curve in the point at infinity $\mathcal{O}$ [24]. Therefore, $\mathcal{O}$ can be thought of as sitting "above" the end of the y-axis [24]. Figure 2.7 shows all four different combinations, feel free to convince yourself that this statement indeed makes sense for the plotted curve.

Using this intersecting line, we can define an operation on curve points:

Definition 2.9. Given an Elliptic Curve $E(\mathbb{Z}_p)$, we define a binary operation called *(point) addition* and denoted by $+$: [24]
Let $P_1, P_2 \in E(\mathbb{Z}_p)$.

- For two points $P_1, P_2 \neq \mathcal{O}$ and $P_1 \neq P_2$, their sum $P_1 + P_2$ is evaluated by drawing the line through P_1 and P_2. This line will intersect the curve in a third point, $P_3 = (x_3, y_3)$. Then the result of the addition is $P_1 + P_2 = (x_3, -y_3)$, i.e. P_3 is reflected in the x-axis (Figure 2.7-1). If $P_3 = \mathcal{O}$, then the result of the addition is $\mathcal{O}$ (Figure 2.7-3).
- If $P_1, P_2 \neq \mathcal{O}$ and $P_1 = P_2$, as above but draw the line as tangent on the curve in P_1 (Figure 2.7-2 and -4).

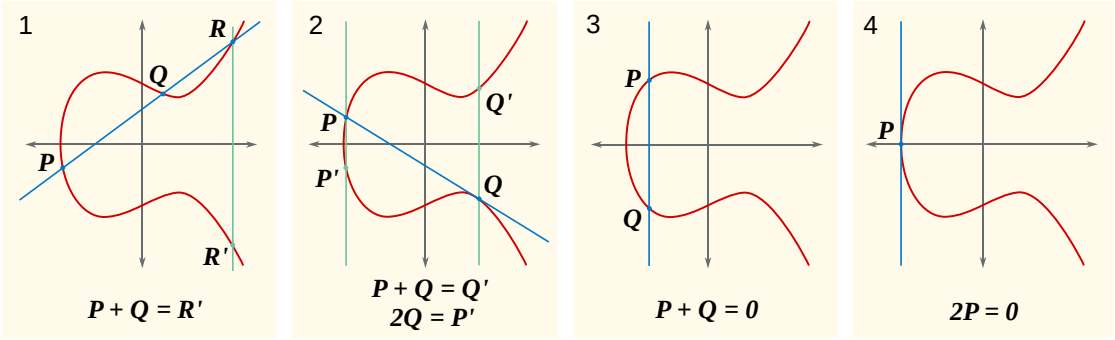


Figure 2.7.: Elliptic Curve point addition
(Image by SuperManu, licensed under Creative Commons.)

- If $P_1 = \mathcal{O}$, then $P_1 + P_2 = P_2$ and vice-versa.

We will be adding points to themselves a lot. Therefore, we define for ease of notation:

Definition 2.10. Point-Scalar multiplication: Given a point $P \in E(\mathbb{Z}_p)$ and a scalar $d \in \mathbb{N}$:

$$d \cdot P = \underbrace{P + P + \cdots + P}_{d \text{ times}} \tag{2.4}$$

That is exactly the definition of group exponentiation, applied to our additive Elliptic Curve group. Note that the product of a scalar with a point is again a point on our curve.

2.5.4. Groups on Elliptic Curves

Theorem 2.1. *The points of an Elliptic Curve $E(\mathbb{Z}_p)$ plus the addition law as stated in Definition 2.9 forms an abelian (commutative group) [24, 44]:*

Proof. A formal proof is outside the scope of this thesis, but here's some informal reasoning about the group axioms:

- Existence of Identity: $P + \mathcal{O} = P$ (as per definition)
- Commutativity: For all $P_1, P_2 \in E(\mathbb{Z}_p)$, $P_1 + P_2 = P_2 + P_1$ (obvious, because the line through P_1 and P_2 will be the same)
- Unique inverse: For any point $P = (x, y) \in E(\mathbb{Z}_p)$, the unique inverse is $-P = (x, -y)$ (obvious).

- Associativity: For all $P_1, P_2, P_3 \in E(\mathbb{Z}_p)$, $(P_1 + P_2) + P_3 = P_1 + (P_2 + P_3)$ (much less obvious, see e.g. [44, Chapter 2.4] for a proof).

□

Of particular interest to cryptography are *cyclic* groups on elliptic curves:

Definition 2.11. A (multiplicative) group $\mathbb{G}$ is cyclic if there is an element $g \in \mathbb{G}$ that generates $\mathbb{G}$, i.e. $\mathbb{G} = \langle g \rangle = \{g^n | n \in \mathbb{Z}\}$.

Translated to our (additive) groups on elliptic curves, this means that there is a generator point $P \in E(\mathbb{Z}_p)$, such that every point $Q \in E(\mathbb{Z}_p)$ can be written as $Q = nP$ with some $n \in \mathbb{N}$.

Theorem 2.2. *[24] Let $\mathbb{G}$ be a finite group of order n, i.e. $|\mathbb{G}| = n$. Let $g \in \mathbb{G}$ be an element of $\mathbb{G}$ with order k, i.e. $k = |\langle g \rangle|$*
Then $k|n$, i.e. the order of g divides the group order n.

Proof. See [24, Proposition 8.54]. □

There is an important consequence to this fact: If a group has prime order, all points except the identity are generators. This stems from the fact that a prime number has exactly two divisors: One (the order of the identity) and itself (the order of all other points).
This follows from the fact that for any point $P \in E(\mathbb{Z}_p)$, its order $\text{ord}(P) = |\langle P \rangle|$ must divide the group order. A prime has exactly two divisors: One (the order of $\mathcal{O}$) and itself (the order of all other points).

Again, translated to Elliptic Curves this means that if the number of points $\#E(\mathbb{Z}_p)$ on a curve is prime, all points except $\mathcal{O}$ are generators. These cyclic elliptic curve groups (or, cyclic subgroups of non-cyclic elliptic curves) are exactly the groups we are interested in for doing actual cryptography. For a detailed description why, see [24, p. 321].

2.5.5. Bilinear Pairings

Definition 2.12. Bilinear pairing [25].
Let $\mathbb{G}_1$ and $\mathbb{G}_2$ denote cyclic groups with prime order n. Let $\mathbb{G}_T$ be another cyclic group of the same order n. $\mathbb{G}_1$ and $\mathbb{G}_2$ are written additively, $\mathbb{G}_T$ is written using multiplicative notation.
A *bilinear pairing* then is a function $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ with the following properties:

- *Bilinearity.* For all $P_1, P_2 \in \mathbb{G}_1, Q_1, Q_2 \in \mathbb{G}_2$

  - $e(P_1 + P_2, Q_1) = e(P_1, Q_1) \cdot e(P_2, Q_1)$
  - $e(P_1, Q_1 + Q_2) = e(P_1, Q_1) \cdot e(P_1, Q_2)$
- *Non-Degeneracy.*
  - for each $P \in \mathbb{G}_1, P \neq 0$ there is a $Q \in \mathbb{G}_2$ with $e(P, Q) \neq 1$
  - for each $Q \in \mathbb{G}_2, Q \neq 0$ there is a $P \in \mathbb{G}_1$ with $e(P, Q) \neq 1$
- *Computability.* There is an algorithm that computes e efficiently.

If $\mathbb{G}_1 = \mathbb{G}_2$, the pairing is called a *symmetric pairing*, otherwise it is an *asymmetric pairing*.

There are a few different concrete pairing functions, e.g. the Weil pairing, Tate pairing and the Ate pairing [25]. Usually the source groups $\mathbb{G}_1$ and $\mathbb{G}_2$ are subgroups of certain elliptic curves [25] and the target $\mathbb{G}_T$ is a finite field (*not* another point on a curve) [13].

3. Related Work

This chapter shall give an overview of the state of research in ABE. The first section deals with abstract constructions for ABE scheme and their different properties without concerning with the implementation or perfomance numbers. The second section gives a brief overview of ABE implementations on regular PC hardware. The third section is then most closely related to the topic of this thesis and deals with implementations of pairings and ABE on resource-constrained devices.

3.1. Theoretical work on ABE schemes

Attribute-Based Encryption was introduced by Sahai and Waters in 2005 [35]. They proposed a new type of identity-based encryption where identities are a set of attributes. Their so-called *fuzzy* identity-based encryption scheme allows a user to decrypt a ciphertext even if their identity doesn't exactly match the identity specified at the time of encryption [35]. Instead, an overlap larger than some threshold value between the attributes in the ciphertext's identity with the attributes of the key's identity is sufficient [35]. This property is realized by means of a (k, n)-threshold secret sharing scheme.

Sahai and Water's construction can already be seen as an ABE scheme with very limited expressiveness, i.e. it only works with "k-out-of-n" access structures [20].

In 2006, Goyal, Pandey, Sahai and Waters [20] extended this into the first expressive KP-ABE scheme using the access tree construction described in Chapter 2. Their main construction uses access trees and a small attribute universe, but they also give constructions with a large attribute universe and for linear secret sharing scheme (LSSS) access structures, respectively.

The first expressive CP-ABE scheme was proposed by Bethencourt, Sahai and Waters in [12]. It is also a large-universe construction and uses access trees. Waters [45] later also gives the first CP-ABE schemes with a security proof in the standard model, not only in the generic group model (the distinction is not relevant for this thesis).

Both the schemes in [20] and in [12] only support monotonic access structures.

In [20], an inefficient realization of general (non-monotonic) access structures is proposed, which is to simply represent the absence of an attribute as a separate attribute. This is inefficient because it doubles the total number of attributes in the

system [20]. Non-monotonic access structures over a universe of n attributes are represented by monotonic access structures over a universe of $2n$ attributes. It also requires every ciphertext to be associated with n attributes (i.e. either with their positive or negated of a corresponding attribute). Note that the size of ciphertexts or keys is usually linear in the number of attributes in expressive ABE schemes.

The first efficient construction for non-monotonic access structures was given in [33]. However, this construction leads to large private keys. More specifically, the size is $\mathcal{O}(t \log(n))$, where t is the number of leaf nodes in the key's access tree and n a system-wide bound on the number of attributes a ciphertext may have [26].

In [26] direct revocation is related to the realization non-monotone access structures and a scheme with efficient direct revocation is presented. The authors also present an efficient construction for non-monotone access structures with keys of size $\mathcal{O}(t)$ [26], where t is again the number of leaf nodes in the key's access tree.

The difference between direct and indirect revocation is introduced in [8], and a *Hybrid Revocable* ABE scheme is given. It allows the encryptor to choose the revocation mode separately for every message [8].

All of these schemes are built using a bilinear pairing as introduced in Section 2.5.5. A pairing-free KP-ABE scheme was proposed by Yao, Chen and Tian [46] in 2015. Their scheme only uses a single group and no bilinear pairing. Instead of encrypting a group element that encodes a message, their scheme yields a random group element which is then used as a key for a symmetric encryption algorithm [46].

In [41] a cryptanalysis of the scheme in [46] is performed. It is shown that the scheme is not secure, but the authors propose an effective fix and prove its security. They also extend the scheme to allow for key delegation (i.e. a hierarchical KP-ABE scheme) [41].

[40] presents a pairing-free ABE scheme with indirect revocation. It is an adaptation of the schemes in [46, 41], see also [22].

All three of these schemes were attacked by Herranz in [22] (one attack for all three schemes is given, as they are very similar). [22] argues that it is not possible to build secure ABE schemes in the (non-bilinear) discrete-logarithm setting (i.e. on elliptic curves without bilinear pairings). For this reason, the security of pairing-free schemes like [46, 40, 41] remains questionable, even if further improved.

3.2. Evaluation on unconstrained hardware

One of the major factors for the performance of the implementation of an ABE scheme is the underlying pairing computation (except for [46] and its derivatives, of course). Not only ABE is based on bilinear pairings, but a large variety of cryptographic schemes, e.g.

a three-party Diffie-Hellman Key Exchange [23] or short digital signature schemes [15]).

Therefore, a fast implementation of the bilinear pairing is vital. Comparing different pairing implementations is difficult because the performance greatly depends on the security level, the concrete pairing implemented, the choice of elliptic curves and of course the speed of the hardware and architecture used. Furthermore, many implementations are not portable due to hand-optimized assembly code or the use of architecture-specific instructions.

One of the first notable implementations was the *Pairing-Based Cryptography Library (PBC)* [28, 27]. The efficiency improvements implemented by the PBC library were first described by its author, Ben Lynn, in [27]. This implementation runs sufficiently fast on standard PC hardware, e.g. it takes 20.5ms to compute a pairing on a 224-bit MNT curve on a 2.4 GHz Intel Core i5 processor [2].

Implementations of ABE on standard PC hardware are well-studied [12, 1, 21]; for an overview see [48].

In [2], a pairing-based ABE scheme is evaluated on a standard computer and an ARM-based smartphone (iPhone 4). On the smartphone, only decryption is implemented because encryption is not needed in their scenario. This implementation uses the PBC library and 224-bit MNT curve from [27]. They conclude that for policies with less than 30 leaves, decryption on a smartphone is feasible (taking around 2 to 7 seconds, depending on the scheme) [2].

In [36], a pairing library and ABE scheme is implemented using the NEON instructions, a set of SIMD vector instructions for ARM processors. They evaluate their implementations on several ARM Cortex A9 and A15 processors with clock frequencies between 1GHz and 1.7GHz. The use of NEON improves performance by 20-50%, depending on the chip. Note that the NEON instruction set is not available on our SoC.

In [43], CP- and KP-ABE are evaluated for different security levels on an Intel Atom-based smartphone using a Java implementation. They conclude that ABE on smartphones is not fast enough to be practical. This is subsequently challenged in [4], where a C implementation also using the PBC library from [27] is evaluated on another smartphone with a 1.2GHz ARM Cortex A9 CPU. This implementation is significantly faster than the one in [43] at comparable security levels. As such, the authors conclude that ABE is indeed feasible on smartphones.

3.3. Evaluation on constrained devices

Despite pairing computation being very computationally demanding, there exist libraries even for the smalles microcontrollers: For example, the *TinyPBC* library [7]. It

takes a minimum of 1.9 s to compute a pairing on a 7 MHz ATmega128L processor with optimized assembly code [32]. Their choice of elliptic curves, however, only provides a security level of 80 bits. This is significantly lower than the security level of the 224-bit MNT curve from the PBC library is (around 128 bits) [2] and the curves in this project (around 100 bits).

Scott [37] provides a fast implementation of the 254-bit BN curve (the same as used in this project) in the *MIRACL Core Cryptographic Library* [38]. They also evaluate their library on the same SoC as used in this thesis (nRF52840, 64 MHz ARM Cortex M4 CPU) and compute a pairing of the 254-bit BN curve in 635 ms [37, Table 4]. Only the pairing implementation is tested and evaluated, the authors do not implement an ABE scheme.

The authors of [4] test their ABE implementation on IoT devices in [3]. They evaluate the performance of the same library on full-fledged IoT devices (among others, on a Raspberry Pi Zero with 1 GHz ARM11 CPU) and conclude that ABE is feasible on these devices, too. However, they note that especially lower security levels are suitable and that the penalty for increasing the security level is very high (e.g. increasing the security level from 80 to 128 bits without increasing the encryption time requires reduction of the number of attributes by a factor of 10) [3]. In contrast to the setting in this thesis, their hardware is significantly more powerful and runs a full operating system.

The setting in [16] is much closer to ours: ABE is implemented bare-metal (i.e. without operating system) on a sensor equipped with an STM32L151VCT6 SoC with a maximum clock frequency of 32 MHz. They use the pairing library *RELIC Toolkit* [6] at a security level of 128 bits and evaluate a C implementation of the CP-ABE scheme in [45]. Only decryption is evaluated; decryption is not implemented on the SoC. The author again concludes that ABE encryption on the sensor is feasible if the policy size is rather small and the runtime of several seconds is acceptable [16]. In this case, the encryption time is over 10 s already for just six attributes [16]. In contrast to our work, the hardware is slightly more constrained and the evaluated scheme is CP-ABE

[19] provides a similar analysis for the slightly faster ESP32 board (240 MHz Xtensa LX6 processor). They also test the pairing-free YCT scheme [46] and evaluate the energy consumption of ABE operations. The authors port existing ABE libraries to the ESP32 platform and use curves with a security level of 80 bits. Due to a bug in the library for the YCT scheme, their evaluation only considers its performance with five and ten attributes (the other schemes are evaluated with up to 50 attributes). The conclusion of [19] is similar to that of [16]. In contrast, our SoC is more constrained and we evaluate the YCT scheme for larger and more meaningful policy sizes.

4. Evaluated ABE schemes

This chapter will describe two ABE schemes in detail; those were implemented in this thesis. In addition to a detailed description of the schemes, any modifications from the original papers are made clear.

Both implemented schemes are KP-ABE. This choice was made because KP-ABE is better suited to our use case from Figure 1.1 (see Section 2.3). Also, encryption tends to be more efficient than with CP-ABE.

The GPSW scheme was chosen because it was the first expressive KP-ABE scheme. It is also considered a rather efficient scheme, compared to others that use bilinear pairings [19]. The YCT scheme was chosen for its unique approach without bilinear pairings. Because pairings are computationally expensive, this promises better efficiency.

4.1. Goyal, Pandey, Sahai and Waters, 2006

This scheme was the first ABE scheme with expressive access policys. Policies are associated with the key (KP-ABE). It was described by Goyal, Pandey, Sahai and Waters [20] in 2006. This scheme will be referred to as GPSW.

Goyal *et. al.* extend the earlier work from Sahai and Waters [35] to allow arbitrary access structures expressed by access trees, not just a "k-out-of-n" attributes. They are the first to use Shamir's Secret Sharing hierarchically in the access tree as described in Section 2.3.3.

The GPSW scheme encrypts a message represented by a point of the bilinear pairing's target group $\mathbb{G}_T$. The main construction follows the small universe approach, but a construction allowing arbitrary attributes is also given. The construction described here ist the small universe construction.

The construction given here is exactly as implemented; it differs from the original construction in the use of an asymmetric pairing ($e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$) instead of a symmetric pairing ($e : \mathbb{G}_1 \times \mathbb{G}_1 \rightarrow \mathbb{G}_T$).
In the GPSW construction, the pairing is only evaluated during decryption phase for leaf nodes (see below). There, the curve point on one side comes from the ciphertext, and the point on the other side from the key. Originally, a symmetric pairing is used, so their order can be swapped freely. As we want to improve the speed of the encryption,

we use the shorter elements of $\mathbb{G}_1$ for everything that has to do with the ciphertext, because that way only elements of $\mathbb{G}_1$ need to manipulated during encryption.

To speed up encryption and decryption, the plaintext is not encrypted with the GPSW ABE scheme directly. Instead, a random group element is chosen and encrypted under GPSW (i.e. a $k \in \mathbb{G}_T$). This element is hashed to obtain an symmetric key, which is then used to encrypt the plaintext with AES-GCM (an AEAD mode of operation). The ciphertext now consists of the GPSW-encrypted group element plus the AES-GCM ciphertext.

Let $\mathbb{G}_1$ and $\mathbb{G}_2$ be bilinear groups of prime order q. Let P be a generator of $\mathbb{G}_1$ and Q be a generator of $\mathbb{G}_2$. Let $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ be a bilinear map. Note that $\mathbb{G}_1$ and $\mathbb{G}_2$ are written additively, but $\mathbb{G}_T$ is written using multiplicative notation.

Setup [20]. The attribute universe is defined as $\mathrm{U} = \{1, 2, \ldots, n\}$ and is fixed. For every attribute $i \in U$, choose uniformly at random a secret number $t_i \in \mathbb{Z}_q$. Then the public key of attribute i is $T_i = t_1 \cdot P$. Also, choose uniformly at random the master private key $y \in \mathbb{Z}_p$, from which the master public key $Y = e(P, Q)^y$ is derived.

Publish $Params = (Y, T_1, \ldots, T_n)$ as the public parameters, privately save $MK = (y, t_1, \ldots, t_n)$ as the master key.

KeyGen(Γ, MK) [20]. Input: access tree Γ and master key MK.

For each node u in the access tree Γ, recursively define polynomials $q_u(x)$ with degree $(d_u - 1)$, starting from the root.

For the root r, set $q_r(0) = s$ and randomly choose $d_r - 1$ other points to determine the polynomial $q_r(x)$. Then, for any other node u, including leaf nodes, set $q_u(0) = q_{\mathrm{parent}(x)}(\mathrm{index}(x))$ and choose $d_u - 1$ other points at random to define the polynomial. For all leaf nodes u, create a secret share $D_u = q_x(0) \cdot t_i^{-1} \cdot Q$ where $i = \mathrm{att}(x)$.

The set of these secret shares is the decryption key $D = \{D_u | u \text{ leaf node of } \Gamma\}$.

Encrypt(M, ω, Params) [20]. Input: Message $M \in \mathbb{G}_T$, set of attributes ω and public parameters $Params$.

Choose $s \in \mathbb{Z}_q$ at random and compute $E' = M \cdot Y^s$. For each attribute $i \in \omega$ compute $E_i = s \cdot T_i$.

Return the ciphertext as $E = (\omega, E', \{E_i | i \in \omega\})$

Decrypt(E, D) [20]. Input: Ciphertext E and decryption key D.

First, define a recursive procedure $\mathrm{DecryptNode}(E, D, u)$ which takes as inputs a ciphertext $E = (\omega, E', \{E_i | i \in \omega\})$, the decryption key D and a node x of the access tree

associated with the decryption key. It outputs either en element of $\mathbb{G}_T$ or $\perp$.

If u is a leaf node, then $i = \text{att}(x)$ and

$$\text{DecryptNode}(E, D, u) = \begin{cases} e(E_i, D_u) = e(s \cdot t_i \cdot P, q_u(0) \cdot t_i^{-1} \cdot Q) = e(P, Q)^{s \cdot q_u(0)} & i \in \omega \\ \perp & i \notin \omega \end{cases} \tag{4.1}$$

If u is not a leaf node, instead call $\text{DecryptNode}(E, D, v)$ for all child nodes v of u and store the result in F_v. Let S_u be an arbitrary d_u-sized subset of child nodes v with $F_v \neq \perp$. If no such set exists, the node was not satisfied. In this case return $\perp$. Then compute with $i = \text{index}(z)$ and $S'_u = \{\text{index}(z) | z \in S_u\}$.

$$\begin{aligned} F_u &= \prod_{z \in S_u} F_z^{\Delta_{i,S'_u}(0)} \\ &= \prod_{z \in S_u} \left(e(P, Q)^{s \cdot q_z(0)}\right)^{\Delta_{i,S'_u}(0)} \\ &= \prod_{z \in S_u} \left(e(P, Q)^{s \cdot q_{\text{parent}(z)}(\text{index}(z))}\right)^{\Delta_{i,S'_u}(0)} \\ &= \prod_{z \in S_u} e(P, Q)^{s \cdot q_u(i) \cdot \Delta_{i,S'_u}(0)} \\ &\overset{(*)}{=} e(P, Q)^{s \cdot q_u(0)} \end{aligned} \tag{4.2}$$

The equality $(*)$ holds because, in the exponent, the product becomes a sum: $\sum_{i \in S'_u} s \cdot q_u(i) \cdot \Delta_{i,S'_u}(0)$ is exactly the lagrange interpolation of $s \cdot q_u(0)$.

Let the root of the access tree be r, then the decryption algorithm simply calls $\text{DecryptNode}(E, D, r) = e(P, Q)^{s \cdot y} = Y^s$, if the ciphertexts's attributes satisfy the access tree. If they don't, then $\text{DecryptNode}(E, D, r) = \perp$.

To retrieve the message from $E' = M \cdot Y^s$, simply calculate and return $M' = E' \cdot (Y^s)^{-1}$.

Of course, it is rather difficult (and slow) to encode the full plaintext as a group element of $\mathbb{G}_T$. Therefore, it is advisable to simply generate a random $K \in \mathbb{G}_T$ and encrypt the plaintext using a secure symmetric cipher with key $k = \text{KDF}(K)$, where KDF is a key derivation function. Then encrypt the point K using the GPSW scheme and attach its ciphertext to the symmetric ciphertext. Correct decryption of $K \in \mathbb{G}_T$ then allows a receiver to decrypt the actual payload.

4.2. Yao, Chen and Tian 2015

This scheme was described by Yao, Chen and Tian [46] in 2015. In 2019, Tan, Yeow and Hwang [41] proposed an enhancement, fixing a flaw in the scheme and extending it to be a hierarchical KP-ABE scheme.

Yao, Chen and Tian's ABE scheme (hereafter written just YCT) is a KP-ABE scheme that does not use any bilinear pairing operations. Instead, the only operation performed on Elliptic Curves are point-scalar multiplication [46]. This makes it especially useful for our resource-constrained context, as bilinear pairings are significantly more costly in terms of computation and memory.

As opposed to other ABE schemes based on pairings, YCT uses a hybrid approach similar to Elliptic Curve Integrated Encryption Standard (ECIES): The actual encryption of the plaintext is done by a symmetric cipher, for which the key is derived from a curve point determined by the YCT scheme [46]. If a key's access structure is satisfied by a certain ciphertext, this curve point and thus the symmetric encryption key can be reconstructed, allowing for decryption [46].

The original description of this scheme uses the x- and y-coordinates as keys for separate encryption and authentication mechanisms. Instead, my implementation uses a combined AEAD scheme (more specifically, AES-256 in CCM mode). This uses a single key (derived by hashing the curve point) to ensure confidentiality and integrity of the data.

The description below includes the fix proposed in [41], for which an additional PRF is used to randomize the value of the index$(\cdot)$ function for nodes of the access tree. For this, instead of index$(\cdot)$, the modified index$'(\cdot) = \mathrm{PRF}(r_l, index(\cdot))$ is used [41]. r_l is a random seed value that differs for each layer l of the access tree [41]. In our implementation, HMAC-SHA512 is used as the PRF.

The four algorithms of the YCT scheme are defined as follows:

Setup [46]. The attribute universe is defined as $\mathrm{U} = \{1, 2, \ldots, n\}$ and is fixed.

For every attribute $i \in U$, choose uniformly at random a secret number $s_i \in \mathbb{Z}_q^*$. Then the public key of attribute i is $P_i = s_i \cdot G$ (i.e. a curve point).

Also, choose uniformly at random the master private key $s \in \mathbb{Z}_q^*$, from which the master public key $PK = s \cdot G$ is derived.

Publish $Params = (PK, P_1, \ldots, P_n)$ as the public parameters, privately save $MK = (s, s_1, \ldots, s_n)$ as the private master key.

KeyGen(Γ, *MK)* [46]. Input: access trees Γ and master key MK.

For each layer $l = 0, 1, \ldots$ of the access tree, generate a random seed value $r_l \in \mathcal{K}_{PRF}$ from the PRF's key space.

For each node u in the Access Tree Γ, recursively define polynomials $q_u(x)$ with degree $(d_u - 1)$, starting from the root.

For the root r, set $q_r(0) = s$ and randomly choose $(d_r - 1)$ other points to determine the polynomial $q_r(x)$. Then, for any other node u (including leafs), set $q_u(0) = q_{\text{parent}(u)}(\text{index}'(u))$ and choose $(d_u - 1)$ other points for q_u, similar to above.

Whenever u is a leaf node, use $q_u(x)$ to define a secret share $D_u = \frac{q_u(0)}{s_i}$; where $i = \text{attr}(u)$, s_i the randomly chosen secret number from *Setup* and s_i^{-1} the inverse of s_i in $\mathbb{Z}_q^*$.

Return the generated key as $D = (\{D_u | u \text{ leaf node of } \Gamma\}, \{r_0, r_1, \dots\})$.

Encrypt(m, ω, Params) [46]. Input: Message m, set of attributes ω and public parameters *Params*.

Randomly choose $k \in \mathbb{Z}_q^*$ and compute $C' = k \cdot PK$. If $C' = \mathcal{O}$, repeat until $C' \neq \mathcal{O}$. $C' = (k_x, k_y)$ are the coordinates of the point C'. k_x is used as the encryption key and k_y as the integrity key.

Then compute $C_i = k \cdot P_i$ for all attributes $i \in \omega$.

Encrypt the actual message as $c = \text{Enc}(m, k_x)$, generate a Message Authentication Code $\text{mac}_m = \text{HMAC}(m, k_y)$.

Return the ciphertext $CM = (\omega, c, \text{mac}_m, \{C_i | i \in \omega\})$

Decrypt(CM, D, Params) [46]. Input: Ciphertext CM, decryption key D and public parameters *Params*.

Decryption is split into two phases: Reconstructing the curve point C' to get the encryption and integrity keys, and actual decryption of the ciphertext.

First, define a recursive decryption procedure for a node u: $\text{DecryptNode}(CM, D, u)$. For leaf nodes with $i = \text{attr}(u)$:

$$\text{DecryptNode}(CM, D, u) = \begin{cases} D_u \cdot C_i \overset{(*)}{=} q_u(0) \cdot k \cdot G & i \in \omega \\ \perp & i \notin \omega \end{cases}$$

Where the equality $(*)$ holds because s_i and s_i^{-1} cancel out:

$$D_u \cdot C_i = q_u(0) \cdot s_i^{-1} \cdot k \cdot P_i = q_u(0) \cdot s_i^{-1} \cdot k \cdot s_i \cdot G = q_u(0) \cdot k \cdot G$$

For an internal node u on layer l, call $\text{DecryptNode}(CM, D, v)$ for each of its childen v. If for less than d_u of the child nodes $\text{DecryptNode}(CM, D, v) \neq \perp$, return $\text{DecryptNode}(CM, D,) = \perp$. Then let ω_u be an arbitrary subset of d_u child nodes of u, where for all $v \in \omega_u$, $\text{DecryptNode}(CM, D, v) \neq \perp$. Then $\text{DecryptNode}(CM, D, u)$

is defined as follows, where $i = \text{index}(v)$, $\omega'_u = \{\text{index}(v) | v \in \omega_u\}$.

$$\begin{aligned}
&\text{DecryptNode}(CM, D, u) \\
&= \sum_{v \in \omega_u} \Delta_{\omega'_u, i}(0) \cdot \text{DecryptNode}(CM, D, v) \\
&= \sum_{v \in \omega_u} \Delta_{\omega'_u, i}(0) \cdot q_v(0) \cdot k \cdot G \\
&= \sum_{v \in \omega_u} \Delta_{\omega'_u, i}(0) \cdot q_{\text{parent}(v)}(\text{index}(v)) \cdot k \cdot G \\
&= \sum_{v \in \omega_u} \Delta_{\omega'_u, i}(0) \cdot q_u(i) \cdot k \cdot G \\
&\overset{(*)}{=} q_u(0) \cdot k \cdot G
\end{aligned}$$

The equality $(*)$ holds because $\sum_{v \in \omega'_u} \Delta_{\omega'_u, i}(0) \cdot q_u(i) = q_u(0)$ is exactly the lagrange interpolation polynomial $q_u(x)$ at $x = 0$ with respect to the points $\{(index(v), q_v(0)) | v \in \omega_u\}$.

This means for the root r of the access tree Γ, we have

$$\text{DecryptNode}(CM, D, r) = q_r(0) \cdot k \cdot G = s \cdot k \cdot G = (k'_x, k'_y)$$

With k'_x the decryption key for m and k'_y the integrity key. Therefore now decrypt $m' = \text{Dec}(c, k'_x)$.

Now check if $\text{HMAC}(m', k'_y) = \text{mac}_m$. If yes, the ciphertext has been correctly decrypted and was not tampered with. Return m', otherwise return $\perp$.

5. Implementation

This chapter describes how the schemes from Chapter 4 were implemented on a lower level. It shall make clear what challenges had to be overcome to run ABE on the sensor.

5.1. Hardware

The main goal of this project was to implement an ABE scheme on a constrained embedded ARM processor. More specifically, the chip used was a Nordic Semiconductor nRF52840 with a 64 MHz Cortex M4 CPU, 256 KB of RAM and 1 MB of flash storage. For the detailed specifications, see [31]. This SoC will be referred to simply as 'the SoC'.

For reference, the implementation was also tested on a standard laptop, referred to as 'the Laptop'. More specifically, this system has a 2.7 GHz Intel i7-7500U CPU and 16 GB of RAM. It runs a Linux-based operating system.

5.2. Programming language and libraries

Rust

Rust was chosen as programming language for this project. First, it is a compiled language and thus incurs little overhead at runtime. Its speed is comparable to that of C/C++. Second, it provides much stronger memory safety guarantees than other compiled languages (especially C/C++ where extreme care is required to avoid introducing exploitable vulnerabilities). This is especially attractive for security-critical components like an encryption library.

The `rabe-bn` library

A Rust-only implementation of elliptic curves and a pairing is provided by the open-source library `rabe-bn` [17], a derivative of the `bn` library by Zcash [17]. It implements a concrete pairing on 256-bit *BN curves*. BN curves are a family of pairing-friendly elliptic curves proposed by Barreto and Naehrig [9].

The 256-bit modulus of the BN curve used in `rabe-bn` was originally believed to provide a security level of 128 bits [11]. Due to the discovery of better attacks on the

underlying cryptographic assumptions, the estimate for the security level has been revised down to 100 bits [47].

The library provides four structs: `G1`, `G2` and `Gt`, elements of the groups $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$, respectively. Let their orders be r, then `Fr` represents an element of the field $\mathbb{F}_r$.

For the elliptic curve groups (structs `G1` and `G2`), additive notation is used and the `*` operator is conveniently overloaded. The target group (struct `Gt`) uses multiplicative notation. For this reason, the description of the schemes in Chapter 4 has also been adapted to use compatible notation.

nRF52840 HAL crate

For easier access to the peripherals of the SoC, the hardware abstraction layer (HAL) crate `nrf52840-hal` was also used. It provides simplified access to the hardware random number generator (RNG) and the timers. Strictly speaking, these were not necessary to build a Rust library (timers are only needed for evaluation and the library interface allows the caller to pass their desired random number generator). However, for testing and actual use of the library bare-metal on the SoC, both the RNG and the timers were needed.

`heapless` crate

The `heapless` [5] crate provides stack-allocated versions of some of the data structures from `std::collections`. Most important were `heapless::Vec` (replaces `std::vec::Vec`) and `heapless::FnvIndexMap` (replaces `std::collections::HashMap`). These data structures are statically allocated and expect their desired capacity as an additional generic type parameter.

5.3. Porting `rabe-bn` to the SoC

The implementation of `rabe-bn` unfortunately relied on the standard library (mostly through the use of heap-allocated dynamic vectors, i.e. `std::vec::Vec`) and is thus not suited for bare-metal applications. Rust provides the dependency-free and platform-agnostic `core` library as an alternative to the standard library. This library does not depend on an operating system or dynamic memory allocation, and thus does not include heap-allocated data structures (like `std::vec::Vec`).

Therefore, I rewrote the `rabe-bn` library to introduce a cargo-feature `std` which controls the inclusion of the standard library and is enabled by default. If this feature is disabled, the `core` library and stack-allocated collections of fixed size from the `heapless` crate are used instead.

Some further modifications were necessary to implement the `core::fmt::Display` trait for the `Gt` struct in a bare-metal compatible manner. The implementation of this trait was used in conjunction with SHA-3 as a key derivation function to create an AES key from curve points. The behavior of the `core::fmt::Display` implementation stayed exactly the same to ensure interoperability with the original `rabe-bn` library.

With these modifications, the `rabe-bn` library runs on the SoC.

5.4. Random Number Generation

Regular Rust programs use the `rand` crate's `ThreadRng` struct to generate random numbers. `ThreadRng` is cryptographically secure [42], but it relies on the operating system randomness pool for seeding.

Therefore, this generator is unavailable on the SoC. Instead we use the hardware RNG. The `nrf52840-hal` crate directly implements the trait `rand::RngCore` for the hardware RNG, which makes it relatively easy to use. This generator, however, is quite slow and speed can differ greatly: With bias correction enabled (required for uniform distribution of the generated data), typically around 120μs per byte [31].

To alleviate this, the hardware RNG is only used to seed a ChaCha20 pseudorandom number generator (crate `rand_chacha`). This is essentially the same construction as the current implementation of `ThreadRng` [42].

5.5. Aspects common to both ABE schemes

These aspects of the implementation are common to both ABE schemes and therefore outsourced into a separate `abe_utils` crate.

5.5.1. Representation of Access Trees

The Rust type system is very well suited to represent the type of tree structures we need for Access Trees. A simple implementation might look like the one in Listing 5.1.

Listing 5.1: Simple Implementation of Access Trees (using the standard library)

```
enum AccessTree<'a> {
    // threshold, vector of children
    Node(u64, std::vec::Vec<AccessTree<'a>>),
    // reference to the attribute label
    Leaf(&'a str),
}
```

This, however, does not work when the `std::vec::Vec` is replaced by a stack-allocated `heapless::Vec`: The `std::vec::Vec` is allocated on the heap and thus only a pointer to the vector needs to be stored in the `AccessTreeNode`. This pointer has constant size.

A `heapless::Vec` is not located on the heap, but directly inside the `AccessTreeNode`. Even if there is a limit on the number of children a single inner node might have, there is no limit to the depth of the access tree. Therefore, an `AccessTreeNode` might be arbitrarily large because the `heapless::Vec` might need to hold an arbitrary number of child nodes.

Because of this, Access Trees were implemented as a flat slice of nodes as in Listing 5.2. The vector of children doesn't hold references to the children themselves, but only their index within the vector of Access Tree nodes. This again introduces an indirection (like the heap pointer in the simple implementation) and allows the enums to have constant size.

Listing 5.2: Refined implementation of Access Trees (works without standard library)

```
type AccessTree<'a, 'b> = &'b [AccessNode<'a>];
enum AccessNode<'a> {
    // threshold, vector of child indexes
    Node(u64, heapless::Vec<u8, consts::U16>),
    // reference to the attribute label
    Leaf(&'a str),
}
```

Listing 5.3 shows the access tree from Figure 2.4 in this representation.

Listing 5.3: Sample access tree in the heapless Rust representation

```
let access_tree: AccessTree = &[
    AccessNode::Node(2, Vec::from_slice(&[1, 2, 3]).unwrap()),
    AccessNode::Leaf("A"),
    AccessNode::Leaf("B"),
    AccessNode::Node(1, Vec::from_slice(&[4, 5]).unwrap()),
    AccessNode::Leaf("C"),
    AccessNode::Leaf("D"),
];
```

5.5.2. Hybrid Encryption

Both schemes encrypt a given plaintext using a hybrid approach: Instead of encrypting the data with ABE directly, the plaintext is encrypted with AES and the key for AES is encrypted under ABE. This results in faster encryption and decryption for long ciphertexts because AES is significantly faster than ABE.

Key derivation

Both schemes output a randomly generated curve point to use as symmetric key. The binary representation of these curve points is much longer than the 256-bit key AES expects. Therefore, the curve point is run through a key derivation function, which can be seen in Listing5.4 in the string representation of the curve point is hashed using SHA-3:

Listing 5.4: Hash-based key derivation Function for curve points

```
struct Wrapper<W: sha3::Digest>(pub W); // newtype for sha3::Digest
impl<W: sha3::Digest> core::fmt::Write for Wrapper<W> {
    fn write_str(&mut self, arg: &str) -> fmt::Result {
        self.0.update(arg);
        Ok(())
    }
}

fn kdf<G: core::fmt::Display>(inp: &G) -> GenericArray<u8, consts::U32> {
    let mut hasher = Wrapper(Sha3_256::new());
    write!(&mut hasher, "{}", inp).unwrap();
    hasher.0.finalize()
}
```

The `kdf()` function takes any struct that implements the `core::fmt::Display` trait and outputs a 256-bit byte array (exactly the size needed for an AES key). `Display` is a formatting trait for user-facing output, i.e. it would be used to format a struct when something is printed to the console. Normally, `Display` is used to turn a struct into a `String`, but those are implemented in the standard library and thus not available on the SoC. Therefore, the naive approach of using `Display` to turn our curve point into a `String` and then hashing the `String` does not work.

To circumvent this problem, I implemented the `core::fmt::Write` trait for the SHA-3 hasher using a newtype pattern. This trait represents a sink for formatted text, like that produced by structs implementing the `core::fmt::Display` trait. Then it is possible

to simply use the `write!()` macro to write the formatted curve points (or any other struct) into the SHA-3 hasher.

Symmetric Encryption

The 256-bit key obtained from `kdf()` is then used as the key to encrypt the actual plaintext. This is done by AES-256 in CCM mode (Counter Mode Encryption with CBC-MAC). CCM is an AEAD mode, i.e. it secures both confidentiality and integrity of the data.

The pure-Rust implementations of AES, the CCM mode and SHA-3 (for `kdf`) in the crates `aes`, `ccm` and `sha3` by the RustCrypto organization are used [34].

Listing 5.5: Symmetric Ciphertext struct

```
pub struct Ciphertext<'data> {
    data: &'data mut [u8],
    nonce: [u8; 13],
    mac: ccm::aead::Tag<ccm::consts::U10>,
}
```

Listing 5.5 shows the result of encryption with AES-CCM. Because dynamic allocation of additional memory is not possible, the plaintext is encrypted in-place. In addition to a reference to the encrypted data itself, the ciphertext stores the nonce and the authentication tag.

The `Ciphertext` struct allows a CCM-encrypted ciphertext to be reconstructed and checked for unauthorized modifications if the key is known. In combination with the ABE ciphertext, it forms a hybrid ciphertext. This combined ciphertext can then be decrypted with a valid ABE key.

5.5.3. Random polynomials over $\mathbb{F}_r$

Both schemes employ Shamir's secret sharing with polynomials for key generation, see Section 2.3.3. To facilitate this, a `Polynomial` struct was written, see Listing 5.6. It represents a polynomial over $\mathbb{F}_r$ as a vector of coefficients from $\mathbb{F}_q$. Polynomial evaluation, random generation and Lagrange interpolation are implemented. For reasons of simplicity, the latter is only implemented for $x = 0$, i.e. the interpolation is immediately evaluated at $x = 0$. Evaluation at any other x is not needed for the ABE implementation.

Listing 5.6: Implementation of polynomials over $\mathbb{F}_r$

```
struct Polynomial(Vec<F, cons>);
```

```
impl Polynomial {
    /// Evaluates the polynomial p(x) at a given x
    fn eval(&self, x: F) -> F {
        self.0.iter().rev().fold(F::zero(), |acc, c| *c + (x * acc))
    }
    /// Generates a random polynomial p(x) of degree `coeffs`,
    /// where p(0) = `a0`
    fn randgen(a0: F, coeffs: u64, rng: &mut dyn RngCore) -> Polynomial {
        // [...]
    }
    /// Calculates the langrage base polynomials l_i(x) for given set of
    /// indices omega and the index i. As we only ever need to
    /// interpolate p(0), x=0 is hard-coded.
    fn lagrange_of_zero(i: &F, omega: &Vec<F, S>) -> F {
        // [...]
    }
}
```

5.6. Implementation of the four ABE algorithms

5.6.1. Setup

The implementations of the setup algorithm are fairly close to the descriptions given in Chapter 4. The private and public attribute keys are stored in `FnvIndexMaps`.

See Listing 5.7 for the structure representing the public and private parameters generated by Setup of GPSW. For YCT, the generators are not required and the attribute public keys are of type `G1` instead of `G2`.

Listing 5.7: Private and public system parameters structs for GPSW

```
/// Private parameters, known only to KGC
pub struct GpswAbePrivate<'attr, 'own> {
  atts: &'own FnvIndexMap<&'attr str, F, S>, // att private keys
  master_secret: F,
}
/// Public parameters, known to all participants
pub struct GpswAbePublic<'attr, 'own> {
  g1: G1, // generator of G1
  g2: G2, // generator of G2
```

```
  atts: &'own FnvIndexMap<&'attr str, G2, S>, // att public keys
  pk: Gt,
}
```

5.6.2. Encrypt

As mentioned before, encryption is done in a hybrid manner. The implementations of both schemes provide a struct containing the ABE ciphertext according to Chapter 4. Combined with the symmetric ciphertext, this forms a hybrid ciphertext. See Listing 5.8 for the GSPW version of the ABE ciphertext (`GpswAbeGroupCiphertext`) and the combined hybrid ciphertext (`GpswAbeCiphertext`).

Listing 5.8: ABE ciphertext structure for GPSW

```
struct GpswAbeGroupCiphertext<'attr> {
    e: Gt,
    e_i: FnvIndexMap<&'attr str, G2, S>,
}
pub struct GpswAbeCiphertext<'attr, 'data>(GpswAbeGroupCiphertext<'attr>,
     kem::Ciphertext<'data>);
```

The GPSW scheme as described in Chapter 4 encrypts a message encoded as an element of $\mathbb{G}_T$. The hybrid encryption function therefore just samples a random element from `Gt`, encrypts it with GPSW and then encrypts the ciphertext using AES with the key derived from the random `Gt`.

The YCT scheme itself only generates a random curve point; its formal definition relies on a symmetric encryption scheme to encrypt the data. Therefore, the curve point generated by YCT (C' in the formal definition) is simply used as the symmetric key to encrypt the payload.

5.6.3. KeyGen

In both schemes, key generation is defined recursively: Starting from the root, random polynomials are formed for each inner node of the access tree. The actual decryption key is a set of secret shares, which are created whenever the key generation algorithm hits a leaf node.

To realize this, a recursive function `decrypt_node()` was implemented. Listing 5.9 shows its signature for the GPSW scheme.

Listing 5.9: Function signature of recursive key generation

```
fn keygen_node (
    &self, // private parameters (MK)
    pubkey: &GpswAbePublic,
    tree_arr: AccessStructure<'key, 'key>,
    tree_ptr: u8,
    parent_poly: &Polynomial,
    index: F,
    rng: &mut dyn RngCore,
  ) -> Vec<(u8, G1), consts::U30>;
```

The argument `tree_ptr` is the position of the currently visited access tree node within `tree_arr`. `parent_poly` is the parent node's secret sharing polynomial and is evaluated at the index of the current node (argument `index`). The function returns a list of secret shares created by the current node and its children. These secret shares consist of an element of `G1` (the curve element) and a `u8`, which is the index of the leaf node that generated the secret share.

The public-facing `keygen()` function calls `decrypt_node()` on the root and combines the result with a reference to the access tree to form the actual decryption key. See Listing 5.10 for the structure representing a decryption key in GPSW. The decryption key for YCT only differs in the type of secret shares contained in the `FnvIndexMap` (`F` instead of `G1`).

Listing 5.10: Decryption key struct

```
pub struct PrivateKey<'attr, 'own>(AccessStructure<'attr, 'own>,
    FnvIndexMap<u8, G1, consts::U32>);
```

5.6.4. Decrypt

Like key generation, decryption is defined recursively for both schemes. Again, this is implemented using a recursive function `decrypt_node()` that is called by the public-facing `decrypt()` function.

See Listing 5.11 for the signature of `decrypt_node` with GPSW.

Listing 5.11: Function signature of recursive decryption

```
fn decrypt_node(
    tree_arr: AccessStructure<'attr, 'key>,
    tree_ptr: u8,
    secret_shares: &FnvIndexMap<u8, G1, S>,
    att_es: &FnvIndexMap<& 'attr str, G2, S>
```

```
) -> Option<Gt>;
```

The arguments `tree_arr` and `tree_ptr` are like with Listing 5.9. In `secret_shares`, the secret shares corresponding to the leafs of the access tree are passed. `att_es` contains the curve elements corresponding to the attributes associated with the ciphertext.

`decrypt_node()` returns an `Option<Gt>`. This is an optional type because decryption will fail if the access tree is not satisfied by the attributes associated with the ciphertext.

The public-facing `decrypt()` function receives both the combined ABE and symmetric ciphertext. It calls `decrypt_node` on the the root of the access tree to obtain the secret curve point. This is then run through `kdf()` to obtain the AES key. The plaintext is decrypted in-place with AES-CCM and a reference to the (now successfully decrypted) data is returned.

5.6.5. Randomization of index (only YCT)

To fix a flaw in the original YCT scheme, the index of nodes within their parent is additionally randomized. See Section 4.2 for a detailed description of this fix.

The fix is implemented using an additional function `index_prf()` which uses an HMAC-SHA3-512 pseudorandom function to randomize the index. On every layer of the access tree, a different random seed is used. These seeds are included in the private decryption key and passed to the YCT versions of `keygen_node` and `decrypt_node`.

6. Evaluation

This chapter shall present the results of my study and discuss the implications of what was observed.

The performance of ABE greatly depends on the underlying implementation of the elliptic curves and pairing because those are the most expensive operations performed by the schemes presented in Chapter 4. Therefore, the performance of the ported `rabe-bn` library will be evaluated in Section 6.1 before turning to the performance of the actual ABE schemes in Section 6.2.

6.1. Performance of `rabe_bn`

See Table 6.1 for performance measurements of random element sampling, group-scalar exponentiation and the pairing operation. The times have been measured using randomly sampled elements and averaged over 100 calls each.

Operation	SoC [ms]	Laptop [ms]
Sample from $\mathbb{F}_r$	2.164	0.003
Sample from $\mathbb{G}_1$	152.852	0.600
Sample from $\mathbb{G}_2$	626.686	2.398
Sample from $\mathbb{G}_T$	2427.512	9.325
Exponentiation in $\mathbb{G}_1$	149.973	0.576
Exponentiation in $\mathbb{G}_2$	641.408	2.462
Exponentiation in $\mathbb{G}_T$	1414.849	5.279
Pairing	1641.741	6.412

Table 6.1.: Execution times for various operations on the SoC and the laptop

It is evident that the the cost of the operations differs greatly between the groups. Sampling a random element takes about the same time as group exponentiation for $\mathbb{G}_1$ and $\mathbb{G}_2$, but is significantly more costly for $\mathbb{G}_T$. Looking at the implementation, the reason for this is obvious: Sampling from $\mathbb{G}_1$ and $\mathbb{G}_2$ simply generates a random $z \in \mathbb{F}_r$ and returns the group element $z \cdot G$ for G a generator. Sampling from $\mathbb{G}_T$ is

done by generating random elements of $\mathbb{G}_1$ and $\mathbb{G}_2$ and computing their pairing. This is reflected in the measured timings.

Interestingly, RAM size seems to be a limiting factor when computing pairings on embedded devices. During development, I also tested the ported `rabe-bn` library on the nRF52832 SoC, which has 64 KB of RAM (vs. 256 KB in the nRF52840). On this chip, a pairing could be computed successfully only if the library was built *without debug symbols*. With debug symbols, there was not enough RAM available and the pairing computation failed. This suggests that the memory use during pairing computation is close to 64 KB, which would still be a quarter of the RAM on the nRF52840 SoC. While this memory is not consumed permanently, it still needs to be available when the pairing function is called.

Measurement of the RAM use with Valgrind on the laptop shows that one pairing computation needs 58,552 KB of RAM. This is indeed close to the 64 KB and thus supports our assumption that RAM use on the laptop is representative for RAM use on the SoC.

6.2. Evaluation of the ABE library

In this section, the performance of the four algorithms Encrypt, Decrypt, Setup and KeyGen of the two ABE schemes GPSW and YCT will be evaluated.

For the results of the runtime analysis, see Figures 6.1 and 6.2. For the analysis of RAM use, see Figure 6.3; for flash use see Figure 6.4.

6.2.1. Methods of measurement

This section describes how the presented result were obtained.

Time measurements

All times were measured using a hardware timer on the nRF52840 SoC.

For Setup, the number of attributes refers to the total number of attributes in the system. For Encrypt, the number of attributes refers to the number of attributes under which a ciphertext is encrypted.

For KeyGen, the number of attributes refers to the number of leaf nodes in the used access policy. The access trees consists only of a root node and the given number of children, i.e. the root node has up to 30 children. To ensure that all leaves have to be evaluated, the threshold is set to the number of children (the root acts as an *AND* node). This approach is also found in other evaluations of ABE [19].

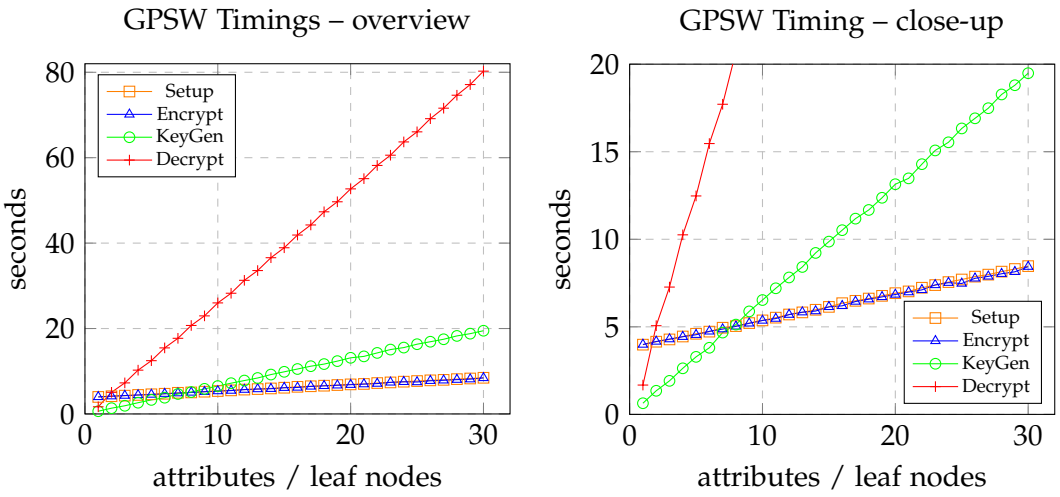


Figure 6.1.: Timings of GPSW operations on the SoC.

For Decrypt, the same access policies from KeyGen are used to decrypt a ciphertext encrypted with all system attributes. This ensures that the ciphertext can always be decrypted, and does not influence the decryption speed.

For reference, the same measurements were made on the laptop. These timings are included in Appendix A.

RAM use measurements

RAM use is not presented for encryption and key generation because those algorithms use a constant amount of memory. The RAM use of key generation and decryption depends only on the depth of the access tree. For this reason, the flat policies from the timing evaluation could not be re-used for RAM evaluation. Instead, access trees in the shape of perfect binary trees were employed.

The values were obtained by running decryption on the laptop and measuring the RAM use using Valgrind's `Massif` tool [30]. Valgrind is a tool suite for analyzing memory use and memory management of programs. Massif itself is originally a heap profiler, but can also be used to measure the stack's memory use with the option `--stacks=yes`. Heap use is not profiled because our library is specifically written to work without the heap.

Massif outputs a number of measurements for different instants during the runtime of the program. The presented values represent the maximum stack memory use during the entire runtime.

It is assumed that RAM use on the laptop is representative for RAM use on the SoC.

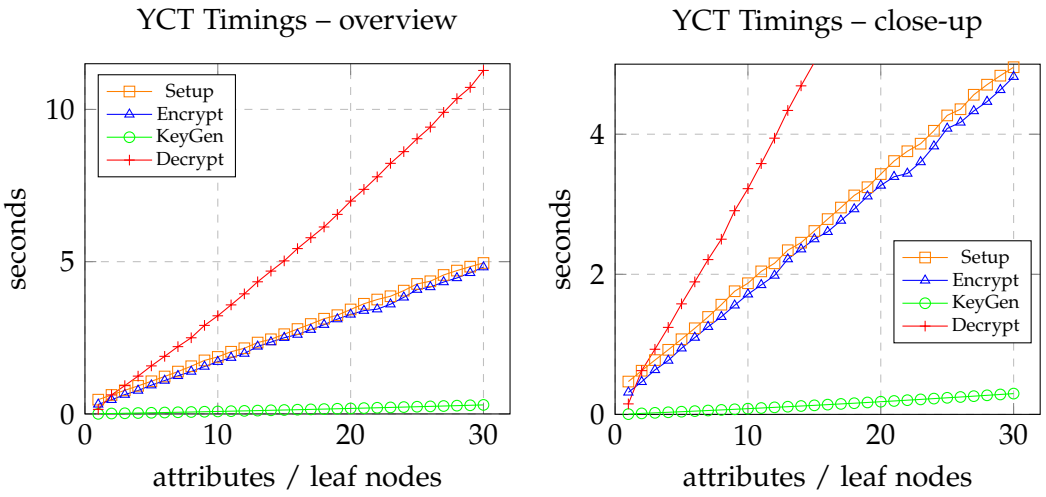


Figure 6.2.: Timings of YCT operations on the SoC.

This is in line with other evaluations of ABE [16]. In addition, decryption with the same policies on the SoC confirmed this assumption: With three- and four-layer policies, decryption failed on the SoC, but it worked with one- and two-layer policies. The RAM use on the laptop is below 256 KB (the RAM size of the SoC) for the two-layer policy, but above 256 KB for the three-layer policy.

Flash size measurements

The sizes of the executable binaries were measured by calling `cargo size -- -A` with optimizations on a binary crate that contains a minimal code snippet calling the respective ABE library. `cargo size` is part of `cargo-binutils` [18], which provides access to the LLVM tools in the Rust toolchain. The presented sizes are the combined size of the `text`, `rodata` and `bss` segments.

6.2.2. Results

The pairing-free YCT scheme is significantly faster than the pairing-based GPSW scheme in all four algorithms. For both schemes, Setup takes about the same time as Encryption for the same number of attributes in the system or ciphertext, respectively. The runtime of Setup and Encrypt with GPSW is about 3.5 s longer than with YCT, independent on the number of attributes. The runtimes increase linearly in both, with each additional attribute adding about 150 ms.

For the KeyGen algorithm, difference is even larger: The runtime of GPSW increases

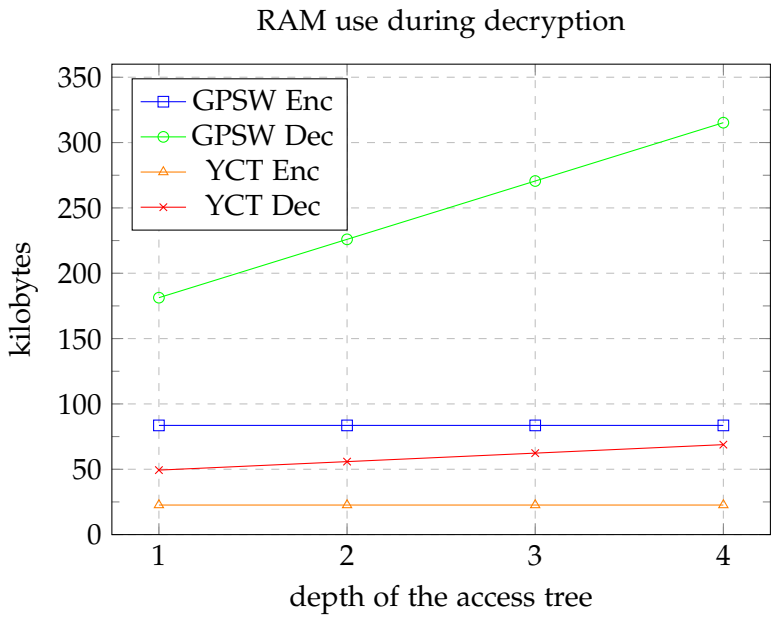


Figure 6.3.: RAM use of ABE encryption and decryption on the laptop. The access trees used for decrytion are perfect binary trees of the given depth. A depth of one means that the tree consists of one root with two children.

considerably with larger policies. With a single-attribute policy, KeyGen takes about 650 ms. With 25 attributes, it already takes more than 16 seconds. The runtime of YCT only increases slightly from 35 ms for one attribute to 210 ms for 25 attributes.

Decryption time is also linear with both schemes. Again, YCT is much faster than GSPW with YCT taking 12 s and GPSW taking 80 s for the largest policy.

The RAM use also shows a large advantage for YCT: With GPSW, a single-level policy already uses up more than three times more RAM than YCT; this difference only increases with more policy levels. With three or more levels, decryption fails entirely on the SoC because there is not enough RAM. Assuming RAM use on the laptop is representative for the SoC, the single-level policy already requires more than two thirds of the total RAM on the sensor. YCT does not have this problem: Even with a four-level policy, it uses only the equivalent of about a quarter of the total RAM.

The difference in flash size is not as big, but the GPSW scheme still requires about 70 % more flash storage than the YCT scheme.

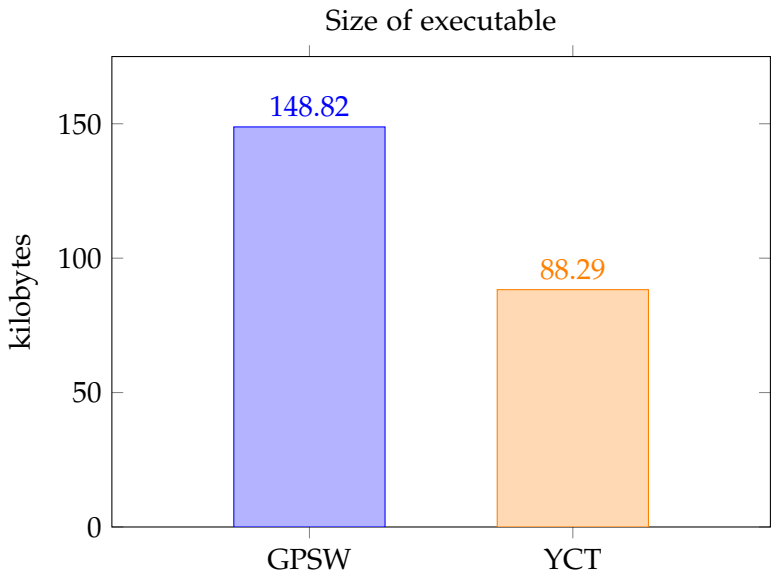


Figure 6.4.: Comparison of the size of the ABE library when using the two schemes.

6.3. Discussion

The timings of Setup and Encrypt are in line with the time-consuming operations performed by the algorithms: In both schemes, each additional attribute results in one additional exponentiation in $\mathbb{G}_1$. This takes about 150 ms as per Table 6.1, which is exactly the additional time per attribute. The constant overhead of about 3.5 s with GPSW is a result of the pairing computation and exponentiation in $\mathbb{G}_T$ (for Setup) and the sampling and exponentiation in $\mathbb{G}_T$ (for Encrypt).

This contradicts the result from [19], where the encryption performance of GPSW and YCT are explicitly noted to be equivalent. However, their conclusion is based on very small numbers of attributes for YCT due to a bug in the evaluated library. In addition, their SoC is significantly faster than ours (240 MHz vs. 64 MHz), thus the constant overhead of GPSW is smaller.

For the KeyGen algorithm, the speed difference between YCT and GSPW is especially striking: The YCT algorithm is extremely fast for a small number of attributes (below 100 ms) and only increases slightly as more attributes are added. GPSW is not only slower already for a few attributes, but its runtime also increases much more quickly when increasing the policy size. Again, looking at the schemes, the reason becomes evident: YCT's KeyGen only works on elements of $\mathbb{F}_r$, which are small and easy to calculate with. GPSW uses secret shares from $\mathbb{G}_2$, for which operations take considerably

more time. This is also the case for decryption: YCT only requires exponentation and point addition in $\mathbb{G}_1$, whereas GPSW performs pairings, multiplications and exponentiations in $\mathbb{G}_T$.

The RAM consumption of decryption with GPSW is a limiting factor on the SoC: Decryption with policies of more than two layers fails because of too little memory. Unlike the timing results, this poses a hard limit on the size of access policys. Also, in our evaluation, the ABE library was able to use all of the available RAM on the SoC. In a real-world use case, the application employing ABE might already occupy a considerable portion of the available RAM, leaving too little space even for decryption with small access trees. This problem is much less pronounced with the pairing-free YCT scheme.

If some latency is acceptable, doing encryption with ABE on the SoC is feasible if the number of attributes is not very large. The pairing-free YCT scheme performs better, but the pairing-free GPSW scheme is still good enough for encryption.

With decryption, the YCT scheme is to be preferred: With GPSW, decryption takes considerably more time and uses a large amount of RAM. This results in the mentioned hard limit on the policy depth, which is not present with YCT.

However, as the security of pairing-free ABE scheme is questioned (see [22]), a pairing-based scheme might be preferable. Therefore, in applications where only encryption is necessary on the SoC, the GPSW scheme might be preferred over YCT.

The other two algorithms, Setup and KeyGen, are run only by the KGC. Therefore, it is reasonable to assume that these don't need to be run on a constrained node in real-world scenarios. The KGC would probably be a specially protected PC or at least a powerful IoT node (e.g. Raspberry Pi).

6.4. Further Improvements / Future Work

The implementations evaluated in this thesis offer room for improvement in many ways. For one, interoperability between the "light" ABE library presented here and a potential "full" counterpart (e.g. for the KGC) was not a priority.

Regarding the results on the SoC, improvements could be made by improving the underlying pairing and elliptic curve implementation of the `rabe-bn` library. Even though the operations remain computationally expensive, other implementations do significantly better: In [37], the *MIRACL Core* library was tested on the same SoC as ours using the same type of curve (256-bit BN curve). This library evaluates a pairing in about 600 ms; our library takes 1600 ms. Exponentiation in $\mathbb{G}_T$ takes about 300 ms with their library and about 1400 ms with ours. Both libraries are high-level implementations (i.e. no optimized assembly code), and thus it is likely that the optimizations from

MIRACL Core could be carried over to Rust. As the runtime of ABE is dominated by these expensive curve operations, an improved pairing and curve implementation offers considerable potential for speedup. These improvements, however, are clearly outside the scope of this thesis.

Runtimes in the order of several seconds might still be too long, even if improved by a better pairing library. To aid this, the symmetric key used to encrypt the actual payload may be reused: In both implemented schemes, the data is not encrypted directly but rather using a hybrid approach, i.e. a random key is generated and encrypted under ABE (so-called key encapsulation). This key is then used to encrypt the actual payload using AES (so-called data encapsulation). This approach allows fast encryption and decryption of messages of arbitrary size and is standard practice with asymmetric encryption schemes.

Usually, a new key is generated and encrypted for every message, but this is not strictly necessary: By saving both the symmetric key and its ABE-encrypted version, more than one message can be encrypted with the same symmetric key. A single ABE encryption operation can then be used for many messages. The encyptor may periodically generate and encrypt a new symmetric key (e.g. once per day). If the messages should be decryptable on their own, the ABE-encrypted symmetric key can be copied into each encrypted message. The downside is that all messages encrypted with the same symmetric key naturally have the same attributes or access policy attached (i.e. reduced granularity). Also, if the symmetric key is compromised (possibly while it is stored to encrypt the next message), all messages encrypted with that key will be compromised.

This approach does not reduce the ciphertext size because the ABE-encrypted key is copied into each ciphertext. If this is an issue (e.g. due to low-power wireless transmission), the ABE-encrypted key may be transmitted separately from the symmetrically encrypted messages. Then, both the symmetric ciphertext of and the respective ABE-encrypted key must be present to decrypt a message.

This symmetric key-caching approach was implemented for use in a system similar to that presented in Figure 1.1. However, it was not evaluated for this thesis, because it doesn't represent "true" Attribute-Based Encryption.

The two schemes implemented in the library are KP-ABE schemes, currently no CP-ABE schemes are implemented. The implemented schemes also don't support key revocation, key delegation or a large universe of attributes. Implementation and evaluation of such schemes is left for future projects.

7. Summary and Conclusion

In this thesis, I presented ABE and evaluated its practical feasibility on constrained IoT nodes. More specifically, I implemented an Attribute-Based Encryption (ABE) library with two schemes and evaluated it on an ARM Cortex M4 SoC. To this end, I had to modify the underlying elliptic curve and pairing library `rabe-bn` to run on embedded systems. For the evaluation of the library, all four ABE algorithms (Encrypt, Decrypt, Setup and KeyGen) were tested on the embedded SoC.

Computing the underlying primitives for ABE on the SoC is feasible, but comes at a rather high price. Comparison with other pairing implementations shows that there is some room for improvement here, but the the operations remain expensive. In addition to computation time, memory size is a major bottleneck for pairing-based implementations.

In the use case presented in the introduction, the constrained node only needs to encrypt data with ABE. For a small or medium number of attributes, this definitely is feasible: RAM size was sufficient even with GPSW and large policies. The runtime is in the order of a few seconds and only increases linearly as more attributes are added. For use cases where such a delay is not practical, re-use of the encapsulated symmetric key might mitigate the issue.

With decryption, the case is different: No problems were encountered with the pairing-free YCT scheme. Indeed, decryption was faster than encryption for the randomly chosen policies in the test set. With GPSW, however, encryption failed for a larger number of attributes because of insufficient RAM. Also, runtimes were longer than for encryption and increased much more significantly with growing policy size. While long runtimes might sometimes be acceptable, the RAM limitation poses a hard limit.

If decryption with large policies is necessary on the SoC, it is therefore advisable to choose YCT over GPSW. The security of pairing-free ABE schemes, however, remains questionable, see [22]. Further research in this area is needed.

In short, Attribute-Based Encryption (ABE) seems feasible on the considered hardware, especially if only encryption is needed and the number of attributes isn't too large. Still, ABE comes at a high price in runtimes and memory consumption. Especially decryption with the pairing-based scheme is infeasible if large policies are used.

A. Timing evaluation on the laptop

These measurements were obtained using the operating system timer and averaged over ten calls each.

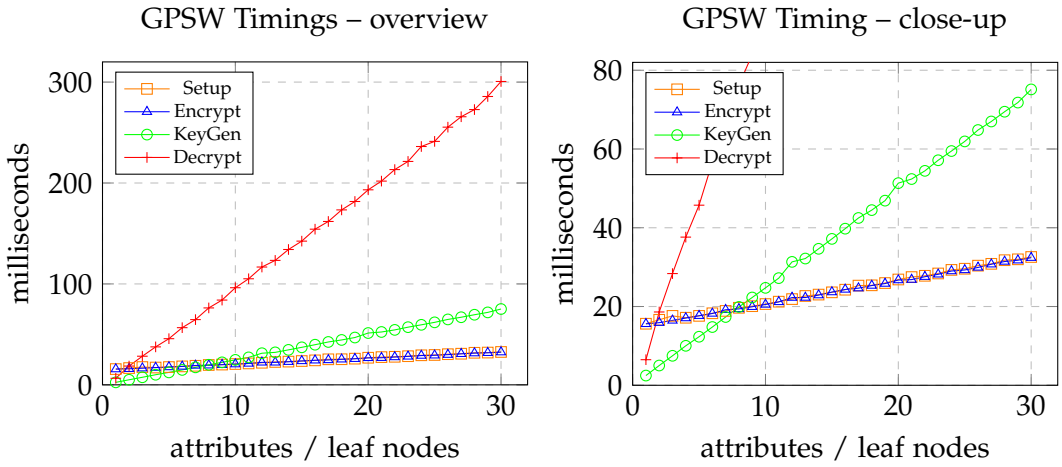


Figure A.1.: Timings of GPSW operations on the laptop.

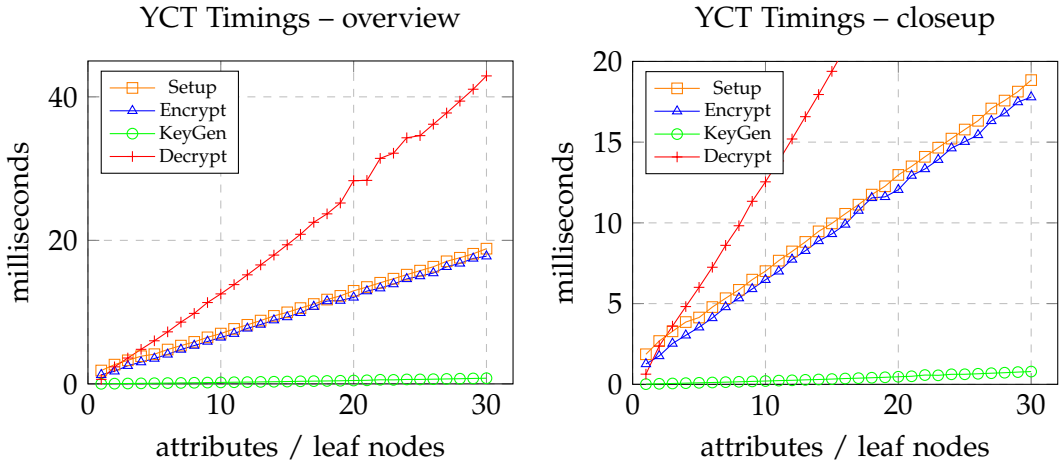


Figure A.2.: Timings of YCT operations on the laptop.

List of Figures

List of Tables

Glossary

access policy A policy that defines what combination of attributes shall be required to access data. Formalized by an access structure and usually realized by an access tree, see Section 2.2.4 plural. 1, 5, 7–9, 24, 46, 52, 53

access structure defines the attribute combinations that are required and sufficient to decrypt a ciphertext. See Section 2.2.4 and Section 2.2.5. 8, 10, 20, 21, 27

access tree construction to realize (monotone) access structures. See Section 2.2.5. 20, 21, 24–27, 33, 37–39, 41, 42, 44, 46

asymmetric encryption scheme type of encryption scheme where different keys are used for encryption and decryption. The encryption key may be made public, while the decryption key is kept private.. 2, 4, 6, 47

attribute Property of an actor or object, e.g. “is student” or “has blonde hair”. 1, 25, 52, 53

attribute universe set of possible attributes. 6

ciphertext-policy ABE Variant of ABE where the key is associated with an access policy and the ciphertext is associated with a set of attributes. 5, 8, 22, 55

crate bundle of Rust code that groups some related functionality together. Can be published to `crates.io` to share with other developers. 31, 32

diffie-hellman key exchange key agreement protocol that allows two parties A and B to agree on a shared secret value over an insecure channel. A and B can derive the same secret value, while any adversaries cannot (as long as they only passively eavesdrop, but not modify, the information exchanged between A and B). 22

digital signature scheme asymmetric cryptographic scheme/protocol for ensuring message authenticity and integrity. 22

elliptic curve Algebraic structure that forms a group, see Section 2.5. 15, 17

generic group model formal security model assuming that the attacker only has oracle access to the group operation (provides less formal security than the standard model). 20

group A set together with a binary operation that satisfies the group axioms, see Section 2.5.1. 52

identity-based encryption type of encryption where data is encrypted using a unique identity (e.g. an email address or phone number), and only the participant holding the secret key corresponding to that identity is able to decrypt the ciphertext.. 20

key derivation function function that derives a suitable cryptographic key from some other data, that may be too long or not in the right format to serve as a key. Usually, hash functions are used as KDF. 26, 32, 34

key generation center Trusted central authority that sets up an ABE scheme and generates keys for users of an ABE scheme . 2, 6, 8, 9, 55

key-policy ABE Variant of Attribute-Based Encryption (ABE) where the ciphertext is associated with an access policy and the key is associated with a set of attributes. 5

large universe type of ABE construction where any string can be used as an attribute. 6, 47

linear secret sharing scheme secret sharing scheme in which the share generation can be described by a matrix. See [10]. 20, 55

monotone span program linear algebraic computation model that is equivalent to LSSS. See Section **??**. 55

perfect binary tree binary tree in which all inner nodes have exactly two children and all leaves have the same depth. 42, 44

security level measure of the strength of a cryptographic scheme, usually given in bits. A security level of n bits means that the most efficient attack needs to perform at least around 2^n operations to break the scheme. Note that this does not directly translate to the size of the used parameters: to guarantee a security level of n bits, usually the the size of the field underlying our elliptic curve (i.e. the number of bits of its modulus) needs to be *at least* $2n$, sometimes much larger. 22

small universe type of ABE construction where the possible attributes have to be fixed when the system is instantiated. 6, 24

standard model formal security model that imposes no restrictions on the attacker, except for the limit on the complexity of their computations. 20, 53

symmetric encryption scheme type of encryption scheme where the same key is used for encryption and decryption. This means that the key has to be shared among all parties via some secure channel (e.g. a personal meeting).. 4, 37

Acronyms

ABE Attribute-Based Encryption. 1, 2, 4–6, 8, 9, 20–24, 30, 34, 35, 37, 40, 41, 43, 44, 46–48, 52–55

ABE scheme Attribute-Based Encryption scheme. 1, 6, 20–25, 40, 46, 48, 50, 53

AEAD authenticated encryption and associated data. 25, 27, 35

BLE Bluetooth Low Energy. 1

CP-ABE Ciphertext-Policy ABE. 6, 8, 9, 13, 20, 23, 24, 47, 50, *Glossary:* ciphertext-policy ABE

HAL hardware abstraction layer. provides abstract access to chip-specific peripherals. 31

IoT Internet of Things. 1, 14, 23, 46, 48

KGC Key Generation Center. 2, 6–9, 14, 46, *Glossary:* Key Generation Center

KP-ABE Key-Policy ABE. 6–9, 13, 20–22, 24, 47, 50, *Glossary:* key-policy ABE

LSSS linear secret sharing scheme. 20, 53, *Glossary:* linear secret sharing scheme

MSP monotone span program. *Glossary:* monotone span program

PRF pseudorandom function. Function that has a (seemingly) random relation between its in- and outputs. Usually hash functions are used as PRF. 27

RNG random number generator. 31, 32

SHA-3 Secure Hash Algorithm 3. Cryptographic hash function standardized by NIST. 32, 34, 35

SIMD single instruction, multiple data. Vectorized CPU instruction that processes several pieces of data at once. 22

Bibliography

[1] J. A. Akinyele, C. Garman, I. Miers, M. W. Pagano, M. Rushanan, M. Green, and A. D. Rubin. "Charm: a framework for rapidly prototyping cryptosystems." In: *Journal of Cryptographic Engineering* 3.2 (2013). Publisher: Springer-Verlag, pp. 111–128. ISSN: 2190-8508. DOI: 10.1007/s13389-013-0057-3. URL: http://dx.doi.org/10.1007/s13389-013-0057-3.

[2] J. A. Akinyele, C. U. Lehmann, M. D. Green, M. W. Pagano, Z. N. J. Peterson, and A. D. Rubin. "Self-Protecting Electronic Medical Records Using Attribute-Based Encryption." In: (2010). Published: Cryptology ePrint Archive, Report 2010/565. URL: https://eprint.iacr.org/2010/565.

[3] M. Ambrosin, A. Anzanpour, M. Conti, T. Dargahi, S. R. Moosavi, A. M. Rahmani, and P. Liljeberg. "On the Feasibility of Attribute-Based Encryption on Internet of Things Devices." In: *IEEE Micro* 36.6 (Dec. 2016), pp. 25–35. ISSN: 1937-4143. DOI: 10.1109/MM.2016.101.

[4] M. Ambrosin, M. Conti, and T. Dargahi. "On the Feasibility of Attribute-Based Encryption on Smartphone Devices." In: *Proceedings of the 2015 Workshop on IoT Challenges in Mobile and Industrial Systems*. IoT-Sys '15. event-place: Florence, Italy. New York, NY, USA: Association for Computing Machinery, 2015, pp. 49–54. ISBN: 978-1-4503-3502-7. DOI: 10.1145/2753476.2753482. URL: https://doi.org/10.1145/2753476.2753482.

[5] J. Aparicio and E. Fresk. *heapless - static friendly data structures that don't require dynamic memory allocation*. URL: https://crates.io/crates/heapless.

[6] D. F. Aranha, C. P. L. Gouvêa, T. Markmann, R. S. Wahby, and K. Liao. *RELIC is an Efficient LIbrary for Cryptography*. URL: https://github.com/relic-toolkit/relic.

[7] D. F. Aranha, C. P. Gouvêa, J. Lopez, L. B. Oliveira, M. Scott, and R. Dahab. *TinyPBC*. URL: https://sites.google.com/site/tinypbc/.

[8] N. Attrapadung and H. Imai. "Attribute-Based Encryption Supporting Direct/Indirect Revocation Modes." In: *Cryptography and Coding*. Ed. by M. G. Parker. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 278–300. ISBN: 978-3-642-10868-6.

[9] P. S. L. M. Barreto and M. Naehrig. "Pairing-Friendly Elliptic Curves of Prime Order." In: *Selected Areas in Cryptography*. Ed. by B. Preneel and S. Tavares. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 319–331. ISBN: 978-3-540-33109-4.

[10] A. Beimel. "Secure Schemes for Secret Sharing and Key Distribution." en. Ph.D. thesis. Haifa: Technion - Israel Institute of Technology, 1996. URL: https://www.iacr.org/phds/index.php?p=detail&entry=548 (visited on Feb. 8, 2021).

[11] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. *Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture*. Published: Cryptology ePrint Archive, Report 2013/879. 2013. URL: https://eprint.iacr.org/2013/879 (visited on Mar. 12, 2021).

[12] J. Bethencourt, A. Sahai, and B. Waters. "Ciphertext-Policy Attribute-Based Encryption." en. In: *2007 IEEE Symposium on Security and Privacy (SP '07)*. Berkeley, CA: IEEE, May 2007, pp. 321–334. ISBN: 978-0-7695-2848-9. DOI: 10.1109/SP.2007.11. URL: http://ieeexplore.ieee.org/document/4223236/ (visited on Dec. 2, 2020).

[13] I. Blake, G. Seroussi, and N. Smart, eds. *Advances in elliptic curve cryptography*. Lecture note series. Cambridge [u.a.]: Cambridge University Press, 2005. ISBN: 0-521-60415-X.

[14] A. Boldyreva, V. Goyal, and V. Kumar. "Identity-Based Encryption with Efficient Revocation." In: *Proceedings of the 15th ACM Conference on Computer and Communications Security*. CCS '08. event-place: Alexandria, Virginia, USA. New York, NY, USA: Association for Computing Machinery, 2008, pp. 417–426. ISBN: 978-1-59593-810-7. DOI: 10.1145/1455770.1455823. URL: https://doi.org/10.1145/1455770.1455823.

[15] D. Boneh, B. Lynn, and H. Shacham. "Short Signatures from the Weil Pairing." In: *Advances in Cryptology — ASIACRYPT 2001*. Ed. by C. Boyd. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 514–532. ISBN: 978-3-540-45682-7.

[16] J. Borgh. "Attribute-Based Encryption in Systems with Resource Constrained Devices in an Information Centric Networking Context." MA thesis. Uppsala: Uppsala University, 2016. URL: http://www.diva-portal.org/smash/get/diva2:945208/FULLTEXT01.pdf (visited on Nov. 19, 2020).

[17] S. Bowe. *bn - Pairing cryptography in Rust*. 2016. URL: https://electriccoin.co/blog/pairing-cryptography-in-rust/ (visited on Mar. 11, 2021).

[18] *cargo-binutils*. URL: https://crates.io/crates/cargo-binutils.

[19] B. Girgenti, P. Perazzo, C. Vallati, F. Righetti, G. Dini, and G. Anastasi. "On the Feasibility of Attribute-Based Encryption on Constrained IoT Devices for Smart Systems." In: *2019 IEEE International Conference on Smart Computing (SMARTCOMP)* (2019), pp. 225–232.

[20] V. Goyal, O. Pandey, A. Sahai, and B. Waters. "Attribute-based encryption for fine-grained access control of encrypted data." en. In: *Proceedings of the 13th ACM conference on Computer and communications security - CCS '06*. Alexandria, Virginia, USA: ACM Press, 2006, pp. 89–98. ISBN: 978-1-59593-518-2. DOI: 10.1145/1180405.1180418. URL: http://dl.acm.org/citation.cfm?doid=1180405.1180418 (visited on Nov. 28, 2020).

[21] M. Green and J. A. Akinyele. *The Functional Encryption Library (libfenc)*. URL: https://code.google.com/archive/p/libfenc/.

[22] J. Herranz. "Attacking Pairing-Free Attribute-Based Encryption Schemes." In: *IEEE Access* 8 (2020), pp. 222226–222232. DOI: 10.1109/ACCESS.2020.3044143.

[23] A. Joux. "A One Round Protocol for Tripartite Diffie–Hellman." In: *Algorithmic Number Theory*. Ed. by W. Bosma. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 385–393. ISBN: 978-3-540-44994-2.

[24] J. Katz and Y. Lindell. *Introduction to modern cryptography*. second edition. Chapman Hall, CRC cryptography and network security. Boca Raton ; London ; New York: CRC Press, 2015. ISBN: 978-1-4665-7026-9 1-4665-7026-1.

[25] M. S. Kiraz and O. Uzunkol. "Still Wrong Use of Pairings in Cryptography." en. In: *arXiv:1603.02826 [cs]* (Nov. 2016). arXiv: 1603.02826. URL: http://arxiv.org/abs/1603.02826 (visited on Dec. 2, 2020).

[26] A. Lewko, A. Sahai, and B. Waters. *Revocation Systems with Very Small Private Keys*. Tech. rep. 2008/309. Published: Cryptology ePrint Archive, Report 2008/309. 2008. URL: https://eprint.iacr.org/2008/309.

[27] B. Lynn. "ON THE IMPLEMENTATION OF PAIRING-BASED CRYPTOSYSTEMS." en. Dissertation. Stanford, California: Stanford University, 2007. URL: https://crypto.stanford.edu/pbc/thesis.pdf.

[28] B. Lynn. *The Pairing-Based Cryptography Library*. URL: https://crypto.stanford.edu/pbc/.

[29] M. L. Manna, P. Perazzo, and G. Dini. "SEA-BREW: A scalable Attribute-Based Encryption revocable scheme for low-bitrate IoT wireless networks." In: *Journal of Information Security and Applications* 58 (2021), p. 102692. ISSN: 2214-2126. DOI: https://doi.org/10.1016/j.jisa.2020.102692. URL: http://www.sciencedirect.com/science/article/pii/S2214212620308413.

[30] N. Nethercote. *Massif: a heap profiler*. URL: https://valgrind.org/docs/manual/ms-manual.html.

[31] *nRF52840 Product specification*. URL: https://infocenter.nordicsemi.com/pdf/nRF52840_PS_v1.1.pdf (visited on Mar. 12, 2021).

[32] L. B. Oliveira, D. F. Aranha, C. P. Gouvêa, M. Scott, D. F. Câmara, J. López, and R. Dahab. "TinyPBC: Pairings for authenticated identity-based non-interactive key distribution in sensor networks." In: *Special Issue of Computer Communications on Information and Future Communication Security* 34.3 (Mar. 2011), pp. 485–493. ISSN: 0140-3664. DOI: 10.1016/j.comcom.2010.05.013. URL: https://www.sciencedirect.com/science/article/pii/S0140366410002483.

[33] R. Ostrovsky, A. Sahai, and B. Waters. "Attribute-based encryption with non-monotonic access structures." en. In: *Proceedings of the 14th ACM conference on Computer and communications security - CCS '07*. Alexandria, Virginia, USA: ACM Press, 2007, p. 195. ISBN: 978-1-59593-703-2. DOI: 10.1145/1315245.1315270. URL: http://portal.acm.org/citation.cfm?doid=1315245.1315270 (visited on Feb. 8, 2021).

[34] *RustCrypto - Cryptographic algorithms written in pure Rust*. URL: https://github.com/RustCrypto.

[35] A. Sahai and B. Waters. "Fuzzy Identity-Based Encryption." In: *Advances in Cryptology – EUROCRYPT 2005*. Ed. by R. Cramer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 457–473. ISBN: 978-3-540-32055-5.

[36] A. H. Sánchez and F. Rodríguez-Henríquez. "NEON Implementation of an Attribute-Based Encryption Scheme." In: *Applied Cryptography and Network Security*. Berlin, Heidelberg: Springer, 2013, pp. 322–338.

[37] M. Scott. "On the Deployment of curve based cryptography for the Internet of Things." In: (2020). Published: Cryptology ePrint Archive, Report 2020/514. URL: https://eprint.iacr.org/2020/514.pdf (visited on Jan. 25, 2021).

[38] M. Scott, K. McKusker, A. Budroni, and S. Andreoli. *The MIRACL Core Cryptographic Library*. URL: https://github.com/miracl/core.

[39] A. Shamir. "How to share a secret." en. In: *Communications of the ACM* 22.11 (Nov. 1979), pp. 612–613. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/359168.359176. URL: https://dl.acm.org/doi/10.1145/359168.359176 (visited on Jan. 7, 2021).

[40] K. Sowjanya, M. Dasgupta, S. Ray, and M. S. Obaidat. "An Efficient Elliptic Curve Cryptography-Based Without Pairing KPABE for Internet of Things." en. In: *IEEE Systems Journal* 14.2 (June 2020), pp. 2154–2163. ISSN: 1932-8184, 1937-9234, 2373-7816. DOI: 10.1109/JSYST.2019.2944240. URL: https://ieeexplore.ieee.org/document/8869901/ (visited on Jan. 12, 2021).

[41] S.-Y. Tan, K.-W. Yeow, and S. O. Hwang. "Enhancement of a Lightweight Attribute-Based Encryption Scheme for the Internet of Things." In: *IEEE Internet of Things Journal* 6.4 (Aug. 2019), pp. 6384–6395. ISSN: 2327-4662, 2372-2541. DOI: 10.1109/JIOT.2019.2900631. URL: https://ieeexplore.ieee.org/document/8651482/ (visited on Dec. 3, 2020).

[42] *The Rust Rand Book*. URL: https://rust-random.github.io/book/intro.html (visited on Mar. 12, 2021).

[43] X. Wang, J. Zhang, E. M. Schooler, and M. Ion. "Performance evaluation of Attribute-Based Encryption: Toward data privacy in the IoT." In: *2014 IEEE International Conference on Communications (ICC)*. 2014, pp. 725–730. DOI: 10.1109/ICC.2014.6883405.

[44] L. C. Washington. *Elliptic curves: number theory and cryptography*. en. 2nd ed. Discrete mathematics and its applications. OCLC: ocn192045762. Boca Raton, FL: Chapman & Hall/CRC, 2008. ISBN: 978-1-4200-7146-7.

[45] B. Waters. "Ciphertext-Policy Attribute-Based Encryption: An Expressive, Efficient, and Provably Secure Realization." In: *Public Key Cryptography – PKC 2011*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 53–70. ISBN: 978-3-642-19379-8.

[46] X. Yao, Z. Chen, and Y. Tian. "A lightweight attribute-based encryption scheme for the Internet of Things." en. In: *Future Generation Computer Systems* 49 (Aug. 2015), pp. 104–112. ISSN: 0167-739X. DOI: 10.1016/j.future.2014.10.010. URL: http://www.sciencedirect.com/science/article/pii/S0167739X14002039 (visited on Dec. 3, 2020).

[47] S. Yonezawa, S. Chikara, T. Kobayashi, and T. Saito. *Pairing-Friendly Curves*. Mar. 2019. URL: https://tools.ietf.org/id/draft-yonezawa-pairing-friendly-curves-01.html (visited on Mar. 12, 2021).

[48] S. Zickau, D. Thatmann, A. Butyrtschik, I. Denisow, and A. Küpper. "Applied Attribute-based Encryption Schemes." en. In: Paris, 2016, p. 8. URL: http://opendl.ifip-tc6.org/db/conf/icin/icin2016/1570228068.pdf (visited on Feb. 9, 2021).