MASTER IN COMPUTER SCIENCE

Integrating BLS signatures in PROTECT

Master Thesis

Patrick Hodel

Faculty of Science
at the University of Bern

June 2020

Prof. Dr. Christian Cachin

Cryptology and Data Security Group
Insitute of Computer Science
University of Bern, Switzerland

UNIVERSITÄT BERN

UNIVERSITÉ DE NEUCHÂTEL

UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Abstract

The Boneh-Lynn-Shacham signature scheme is a signature scheme based on elliptic curves. It produces short signatures and supports threshold signatures [1]. A threshold signature scheme is a signature system where a signature can be generated by t out of n participants. The signature can be verified by anyone knowing the common public key [2]. PROTECT (a *P*latform for *RO*bust *T*hr*E*shold *C*ryp*T*ography) provides a platform for threshold-secure cryptography. The system can be used to implement systems and services and services that tolerate multiple simultaneous faults and security breaches without loss of privacy, availability or correctness. The system further self-heals from faults and self-recovers from breaches [3]. Protect is written in Java. The Java Pairing-Based Cryptography Library provides a port of the Pairing-Based Cryptography Library [4]. It allows to compute the mathematical operations used in pairing-based cryptosystems directly in Java. Using JPBC we implement the BLS signature scheme in Protect. We built a demo client to run requests against PROTECT and to test signature aggregation for a threshold signature scheme.

Ut In Omnibus Glorificetur Deus.

List of Figures

Listings

Nomenclature

BLS Boneh, Lynn and Shacham

GMP The GNU Multiple Precision Arithmetic Library

HTTP Hyper Text Transport Protocol

JPBC Java Pairing-Based Cryptography

LGPL Lesser General Public License

PBC Pairing-Based Cryptography

PROTECT A Platform for Robust Threshold Cryptography

RSA Rivest-Shamir-Adleman

Contents

1 Introduction

Cryptography (from Greek kryptós "hidden" and gráphein "writing") is the art of creating secure codes. Cryptography is most practically relevant to the problem of *secure communication*, which focuses on two goals: *secrecy* and *integrity* of communicated data [5]. Ensuring secrecy means that no user eavesdropping the traffic between two parties can infer any information about the communicated message; particularly not its content. Maintaining integrity means that any alteration of the message during transport is detectable. Digital signatures (see section 2.1) play a vital role in ensuring message integrity. An example for a signature scheme is the Boneh-Lynn-Shacham (BLS) signature scheme. It is based on elliptic curves and uses pairing and is a pairing-based cryptographic system. It outperforms other signature schemes in terms of security. BLS produces shorter signatures while providing comparable security [1]. The scheme is further described in section 2.2.

Let's imagine a group of 5 people is sending out messages. Let's further assume at least 3 people have to consent for a message to be sent on behalf of the group. Such a system can be realized with a threshold signature scheme. In this particular case it would be a 3 out of 5 threshold scheme. Section 2.3 gives more background on Threshold Cryptography. The BLS signature scheme support threshold signatures [1].

To implement such a system we make use of PROTECT. It provides a platform for threshold-secure cryptography. The system can be used to implement systems and services and services that tolerate multiple simultaneous faults and security breaches without loss of privacy, availability or correctness. The system further self-heals from faults and self-recovers from breaches [3]. Protect is written in Java. The Java Pairing-Based Cryptography Library provides a port of the Pairing-Based Cryptography Library [4]. It allows to compute the mathematical operations used in pairing-based cryptosystems directly in Java. Using JPBC we implement the BLS signature scheme in Protect.

This thesis is further structured in three chapters. Chapter 2 introduces digital signatures, BLS signatures, threshold cryptography and the Protect system. Chapter 3 highlights the design of PROTECT and particularly shows how to interact with the system to perform various operations. Chapter 4 explains the implementation. Finally this thesis ends in chapter 5 with a conclusion.

2 Background

This chapter presents the theoretical background and introduces related work.

2.1 Digital Signatures

One of the fundamental components of cryptography are digital signature schemes. Such schemes are used to prove the authenticity of messages and data [6]. A sender A who wishes to send data to a receiver B computes a signature on the data and sends the data and the signature to B. On receiving the data and the signatures, B validates the signature. If the signature proves to be valid, B has strong reason to believe in the authenticity of the sender and in the integrity of the message. That is, the message was most certainly sent by A and was not altered during the transit. A common way to compute and validate signatures employs public-key cryptography. Figure 2.1 gives a visual example of the signing process. The next paragraph explains the fundamental principle of this system.

In public-key cryptography each participant of the system possesses two keys, a private and a public key [8]. The private key is to be kept secret at all times, whereas the public key is known by all other participants. The private key allows the generation of signatures. With the corresponding public key the signature can be verified to belong to the owner of the key. Without the private key it is virtually impossible to create a corresponding signature.

To conclude, a digital signature scheme using public-key cryptography consists of a digital signature generation algorithm that, given a private key and a message, generates a data string called signature and a digital signature verification algorithm that, given a public key corresponding to the private key and a signature, verifies the signature [8].

2.2 BLS Signatures

BLS is a signature scheme introduced in 2004 that produces particularly short and secure signatures [1]. BLS features some interesting properties. In this section we will discuss these properties.

One of these properties is the possibility to aggregate signatures. Given public keys $p_1, p_2, \ldots, p_n$ and corresponding signatures $s_1, s_2, \ldots, s_n$ we can compute an aggregate public key P and an aggregate signature S which is validated with P. Signatures can be added incrementally [9]. That means we can create an aggregate signature s_{12} with s_1 and s_2. Afterwards s_3 can be added to create the aggregate signature s_{123}. Many applications such as a Public Key Infrastructure (PKI) or the Secure BGP protocol benefit from the method by compressing many signatures into one short signature of equal length [10].

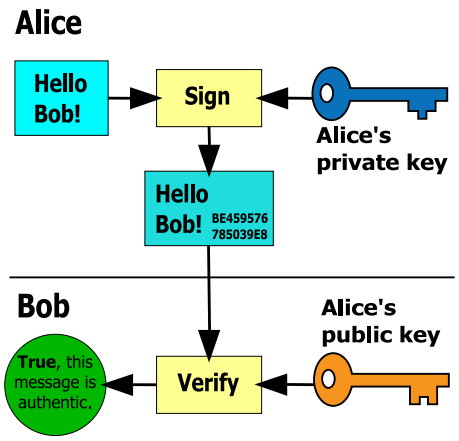

Figure 2.1: Example of the public key signature scheme [7]. Alice sings the message "Hello Bob!" with her private key, sends the message and its signature to Bob. Bob uses Alice's public key to verify that the message is indeed from Alice (authenticity) and was not altered during the transmission (integrity).

BLS produces signatures that are unique and deterministic [11]. For any given pair of public key and message, there can only be one valid signatures. In other schemes such as ECDSA the use of randomness results in many possible valid signatures. Somewhat related, all operations performed with BLS are deterministic.

For many applications it is essential to minimize the computational overhead. BLS allows batch verification that makes signature verification more efficient [12].

2.3 Threshold Cryptography

The scenario so far always required one participant to create a signature. In a different scenario we have a pool of participants. To create a signature at least k, where k is called threshold, participants are required to participate in the signature generation. It must not be possible for $i < k$ participants to generate a signature. This problem statement is subject of threshold cryptography research. One of the first solutions was presented more than 30 years ago [2]. The solution involved an adaption of the ElGamal [13] public key cryptosystem.

An important algorithm in threshold cryptography is Shamir's Secret Sharing. In this algorithm a secret is split into parts, such that each participant receives its own unique share [14]. The number of parts required to recover the original secret is given by the threshold. The threshold k as well as the total number of participants n can be chosen arbitrarily. Knowledge of any $i < k$ parts will leave the secret completely undetermined. Shamir's secret sharing builds on the idea that k points are required to define a polynomial of degree $k - 1$. For example, at least two points are required to define a line, at least three points are required to define a parabola etc. Figure 2.2 illustrates this. Let the secret S be an element of a finite field $F$of size P where P is a prime number and $S < P$. In a threshold scheme (k, m) with $0 < k \leq n < P$ we choose at random $k - 1$ positive integers $a_1, \ldots, a_{k-1}$ with $a_i < P$, and let $a_0 = S$. Define the polynomial $f(x) = \sum_{i=0}^{k-1} a_i x^i$ and select n points lying on the polynomial. Any k-subset of points is sufficient to compute f using interpolation. Having computed the polynomial, we also retrieve the secret which is the constant term a_0.

2.4 PROTECT

PROTECT provides a platform for threshold-secure cryptography [3]. It is an open source project hosted on GitHub under MIT license [15]. It is intended to run on multiple nodes simultaneously and provides functionality for secret maintenance. A user can initiate a distributed key generation, where each

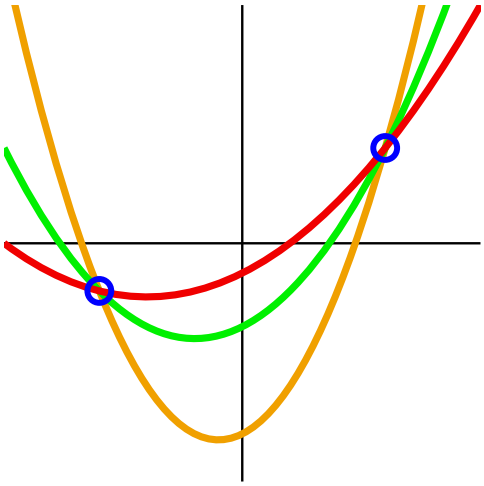

Figure 2.2: The knowledge of two points is not sufficient to define a parabola. We can find unlimited parabolas going through these two points. However a line is well defined by these two points and similarity a parabola would be well defined given a third point.

participant receives a share while the underlying secret value is never known to anyone. PROTECT can periodically generate new shares for an existing secret. All existing old shares become void to ensure which circumvents that a leaked share remains useful for an extended period. Further PROTECT can recover a share, that is, rebuild a lost or destroyed share without having to rebuild the secret or expose another share. The platform also provides various user actions related to share management. A user can store, read, delete, recover, disable and enable a specific share. All operations are subject to a fine-grained access control system permitting specific operations to specific users and specific role. As such all requests have to be authenticated. PROTECT currently supports signature generation, blinded signature generation and decryption based on RSA. Further it supports cryptographic functions on elliptic curves such as pseudorandom functions [16], oblivious pseudorandom functions [16], ECIES Encryption [17] and Elliptic Curve Diffie Hellman Key Agreement [18].

PROTECT communicates over HTTP which is also how the client interacts with PROTECT. For some functionalities a browser interface exists. Configuration of the system is done offline via various config files. All requests require previously generated certificate files to authenticate the user.

2.5 RSA Signatures with PROTECT

It is already possible to generate a signature using PROTECT. The algorithm that is implemented in PROTECT is explained in [19]. To do this, we obtain a signature share of at least k nodes of a total of l participants.

2.5.1 Initialization

Initially two large primes p and q, where $p = 2p' + 1$ and $q = 2q' + 1$ with p', q' themselves prime. We obtain $n = pq$ called RSA modulus and $m = p'q'$. Furthermore, a prime e, called public exponent with $e > l$ is chosen. Afterward $d \in \mathbb{Z}$ such that $de \equiv 1 \mod m$ is computed. A polynomial $f(X) = \sum_{i=0}^{k-1} a_i X^i \in \mathbb{Z}[X]$ with $a_0 = d$ and $a_1, \ldots, a_{k-1}$ randomly chosen from $\{0, \ldots, m-1\}$. Next, $s_i = f(i) \mod m$ for $1 \leq i \leq l$ is computed. s_i represents the secret key share for participant i.

For use in PROTECT the shares are usually computed offline and stored in each client individually through an HTTP request. Listing 1 shows how generated shares are added to PROTECT.

```
$ curl --cacert config/ca/ca-cert-server-1.pem
   --cert config/client/certs/cert-signing_user
   --key config/client/keys/private-signing_user "https://127.0.0.1:8081/store?
   secretName=rsa-secret&e=65537&n=103473605&v=891745958475&
   v_1=67838606285772972709589 63&
   v_2=49365490607479026700863538&
   v_3=70399170169480003218&
   v_4=396178385757170867 59&
   v_5=5710598023&
   share=326633247433522184"
```

Listing 1: Example HTTP request using curl to store RSA variables and secret shares to the secret with name "rsa-secret" on the client at 127.0.0.1:8081. The parameters "cacert", "cert" and "key" are for authentication purposes.

2.5.2 Obtaining a signature share

Generating a signature on a message using PROTECT requires obtaining a signature share of each node. Let H be a hash function mapping messages to elements of $\mathbb{Z}_n^*$ and $x = H(M)$ given a message M. A valid signature on M is $y \in \mathbb{Z}_n^*$ such that $y^e = x$. The signature share of participant i is $x_i = x^{2\Delta s_i} \in \mathbb{Q}_n$, where $\mathbb{Q}_n$ is the subgroup of squares in $\mathbb{Z}_n^*$. Further a proof of correctness is computed. For this, a security parameter L_1 and a hash function H' mapping to a L_1-bit integer is chosen. Each participant chooses a random number $r \in \{0, \ldots, 2^{L(n)+2L_1} - 1\}$, where $L(n)$ is the bit-length of n. The proof of correctness is $(s_i c + r, c)$ with $c = H'\left(v, x^{4\Delta}, v_i, x_i^2, v^r, x^{4\Delta r}\right)$.

In PROTECT a signature share is computed using the `/share` endpoint as illustrated in listing 2.

```
$ curl --cacert config/ca/ca-cert-server-1.pem
   --cert config/client/certs/cert-signing_user
   --key config/client/keys/private-signing_user
   "https://localhost:8081/sign?secretName=rsa-secret&message=896826402883" | jq .
```

Listing 2: Example HTTP request using curl to generate a signature share from client at localhost:8081. The sign request uses the secret "rsa-secret" and the message to sign is 896826402883.

The server computes the share, returns the verification keys and the share proof as shown in listing 3.

```
{
   "share_proof": [
      "3915452739578001",
      "20964253121137168757054695 3"
   ],
   "e": "65537",
   "v": "9766788291648112",
   "verification_keys": [
      "9252797476485432255805",
      "2662839095604955890458",
      "2588402847926746427981",
      "1095107926223706019872",
      "1145851297136005570977"
   ],
   "responder": 1,
   "epoch": 0,
   "share": "104089335183339073298944",
   "compute_time_us": 1596,
   "n": "124177666122443695422631 7049"
}
```


Listing 3: Example result containing the share proof, verification keys and the share of a sign request.

3 Design

3.1 BLS Signatures with PROTECT

The procedure to obtain BLS signatures is similar as it was with RSA: First it is required to store the secret before we can obtain signature shares from each participant. Combining the signatures shares to a complete signatures is done offline.

3.1.1 Initialization

The generation of the public key shares has been outlined in section 2.3. The `store` endpoint was extended to accept the following parameters:

public-key-x The x value of the public key

public-key-y The y value of the public key

share The participant's share of the private key

type A newly introduced variable containing either the value `bls-a` or `rsa` to indicate whether to use RSA or BLS for the given secret

Thus an example request is given in listing

```
curl --cacert ../../bin/config/ca/ca-cert-server-1.pem
   --cert ../../bin/config/client/certs/cert-signing_user
   --key ../../bin/config/client/keys/private-signing_user
    "https://127.0.0.1:8081/store?secretName=bls-secret
   &type=bls-a&share=28775751703195007910403010708335189384295681820 2
   &public-key-x=516381700716113131119310145346446369727684849460896829701851
   &public-key-y=383799672673297958114733040329484430297870898465036498916854"
```

Listing 4: Example store request for a bls secret.

3.1.2 Obtaining a signature share

Obtaining a signature share is analogous to RSA, as indicated in listing 5

```
curl --cacert ../../bin/config/ca/ca-cert-server-1.pem
   --cert ../../bin/config/client/certs/cert-signing_user
   --key ../../bin/config/client/keys/private-signing_user
   "https://localhost:8081/sign?secretName=bls-secret&type=bls-a&message
    =896826402883" | jq .
```

Listing 5: Example sign request for a bls secret.

3.2 BLS parameters

BLS signatures requires specific curves for the Weil pairing to be available [1]. Elliptic curves are plane curves over a finite field consisting of points satisfying the equation

$$y^2 = x^3 + ax + b.$$

Additionally the point at infinity is included. a, b are chosen from a fixed finite field [20]. JPBC provides different curves. We are using Type A curve defined as

$$y^2 = x^3 + x$$

over the field $\mathbb{F}_q$ for some prime $q \equiv 3 \mod 4$.

4 Implementation

This chapter discusses the implementation of BLS signatures in PROTECT, as well as the used library for the cryptographic operations.

4.1 JPBC Library

The JPBC Library [4] is a library introduced by Angelo De Caro and Vincenzo Iovino in 2011 providing a framework to perform mathematical operations of pairing-based cryptography. It is a Java port of the PBC Library [21] originally developed in C. The PBC library allows fast computation of pairings. Both libraries are freely available under the GNU LGPL version 3 license [22]. For some functionalities the JPBC library acts as a wrapper to the underlying PBC library. For example to compute pairings with JPBC internally the underlying PBC which in turn uses GMP library. The wrapper to call the GMP library directly from Java is JNA GMP [23]. The GMP library is written in C and allows for fast mathematical computations. Thus this library has to be available on the host system. In our case we have used the Ubuntu operating system where this library is available by default [24].

Figure 4.1 shows a UML diagram of relevant JPBC classes. The pairing interface serves as the main entry point. It provides access to the G_1 and Z_r which are both represented by the `Field` class. A point on a curve is represented as a `CurveElement` or an `ImmutableCurveElement`. Most classes have an immutable variant that cannot be altered. Any operation on an immutable object returns a new object. Alternatively the immutable element can also be duplicated. To generate a pairing we rely on the `PairingFactory` which uses predefined parameters describing a particular curve.

4.1.1 Obtaining keys

Listing 6 shows how a private public key pair is obtained using JPBC classes. In the first line the pairing is obtained. The variable `parametersPath` is a string containing the path to a text file containing the curve definition. Next we obtain a private key by drawing a random element from the field $\mathbb{Z}_r$ which is subsequently wrapped in a `PrivateKey` object. To generate the public key we first obtain a random generator point on the curve (line 4). We then multiply this element with the private key to get the public key. Finally a `PublicKey` element is constructed using the multiplied element and the generator point.

```
Pairing pairing = PairingFactory.getPairing(parametersPath);
Element privateKeyElement = pairing.getZr().newRandomElement();
PrivateKey privateKey = PrivateKey((ZrElement) privateKeyElement);

Element g = pairing.getG2().newRandomElement().getImmutable();
```

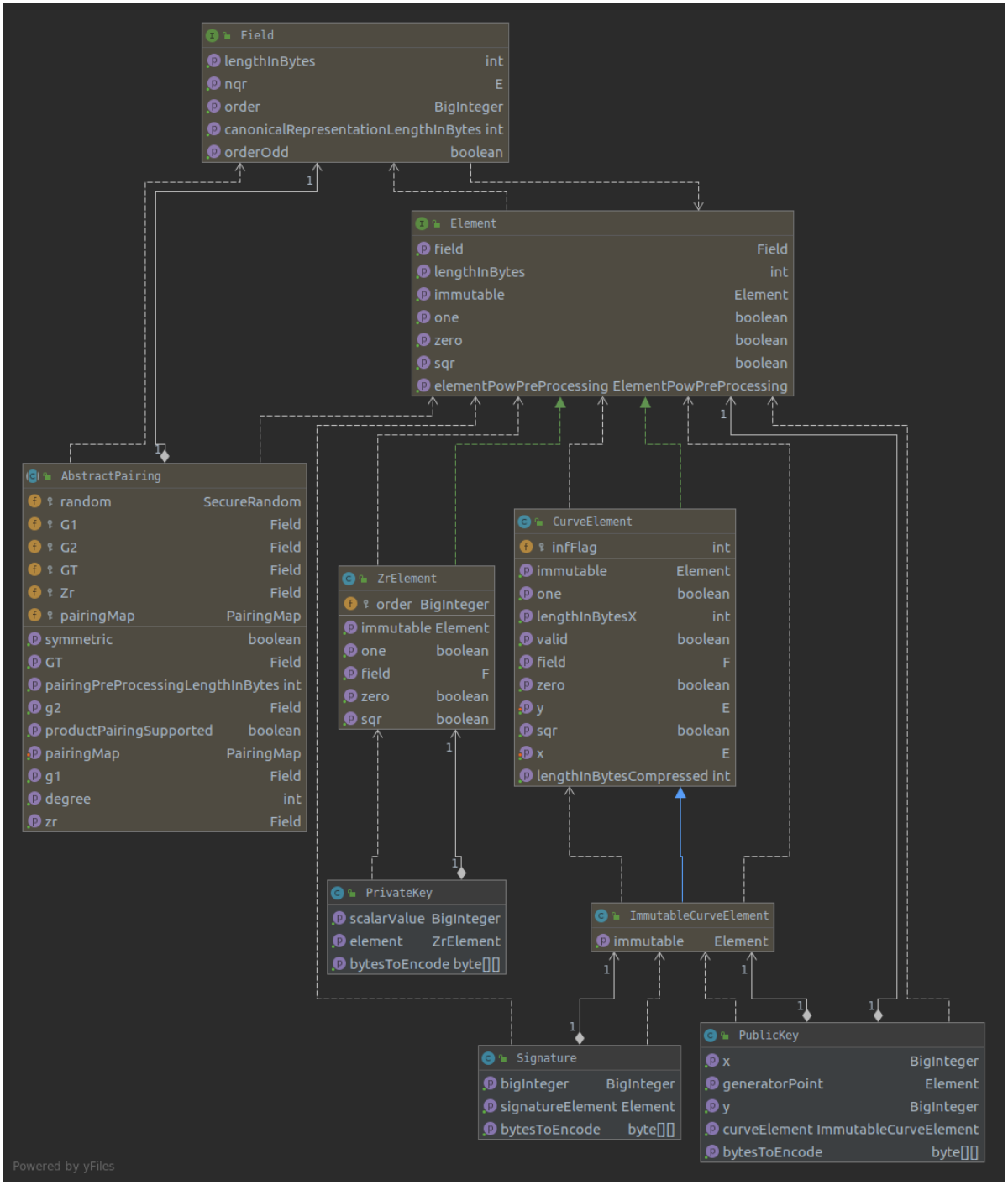

Figure 4.1: UML diagram of selected JPBC classes

```
ImmutableCurveElement publicKeyElement = (ImmutableCurveElement) g.mulZn(privateKey.
    getElement());
PublicKey publicKey = new PublicKey(publicKeyElement, g);
```

Listing 6: Generation of a private public key pair.

4.1.2 Verifying a signature

Listing 7 shows how to verify a signature. First the JPBC signature element is obtained. This is a point on the curve. The subsequent line creates a point on the curve given the bytes of a message. Next a pairing of the signature element and the generator point of the public key as well as a pairing of the message element and the public key element is computed. If those two pairings are equal the next line returns true. Otherwise the signature is incorrect and the function returns false.

```
Element signatureElement = signature.getSignatureElement();
Element messageElement = pairing.getG1().newElement().setFromHash(messageBytes, 0,
    messageBytes.length);
Element e1 = pairing.pairing(signatureElement, publicKey.getGeneratorPoint());
Element e2 = pairing.pairing(messageElement, publicKey.getCurveElement());
return e1.equals(e2);
```

Listing 7: Validating a signature.

4.2 PROTECT

In our work we extend the two HTTP endpoints `/store` and `/sign` to also work with BLS secrets. Previously these were only implemented to work with RSA secrets.

Before being able to store a secret, the secret's role have to be configured in `clients.config` that configures all secrets. These configurations associate the permissions of each user with the secret name. Listing 8 shows the declaration of a secret.

```
[my-secret]
   administrator       = generate,delete,disable,enable,info
   security_officer    = disable,info
   storage_user        = store,read,delete,info
```

Listing 8: Declaration of the secret `my-secret` in the file `clients.config`.

4.2.1 Communication

As in our example client PROTECT communicates through over HTTP using curl, the payload of the request has to be URL encoded. This poses a significant challenge since the client requires access to the JPBC elements. To overcome this challenge we encoded the cryptographic objects in base64. Figure 4.2 shows relevant classes to encode the cryptographic elements. We wrapped the JPBC elements in a element based on its semantics (`PublicKey`, `PrivateKey` and `Signature`). All elements that need to be transferred implement the `CurlEncodable` interface. Its only method, `getBytesToEncode` returns a two-dimensional array of bytes that are needed to recreate the element. Each array of bytes represents one field of the underlying class. To generate the array of bytes we can resort to the `getBytes` method available on all JPBC `Elements`. The second step is to generate a `string` from the `bytes` array. The generic class `UrlBase64ObjectEncoder` works on all objects implementing `CurlEncodable`. Its method `encodeObject` returns a `string` that is safe for use with `curl`. Internally we rely on `UrlBase64Encoder` of the Bouncycastle cryptographic library [25]. On the receiving side the class generic class `UrlBase64DecoderObjectFactory` decodes the received `string` and returns the decoded object. Internally it contains an object implementing `BytesToObjectFactory` which knows how to convert the `bytes` array to the underlying class.

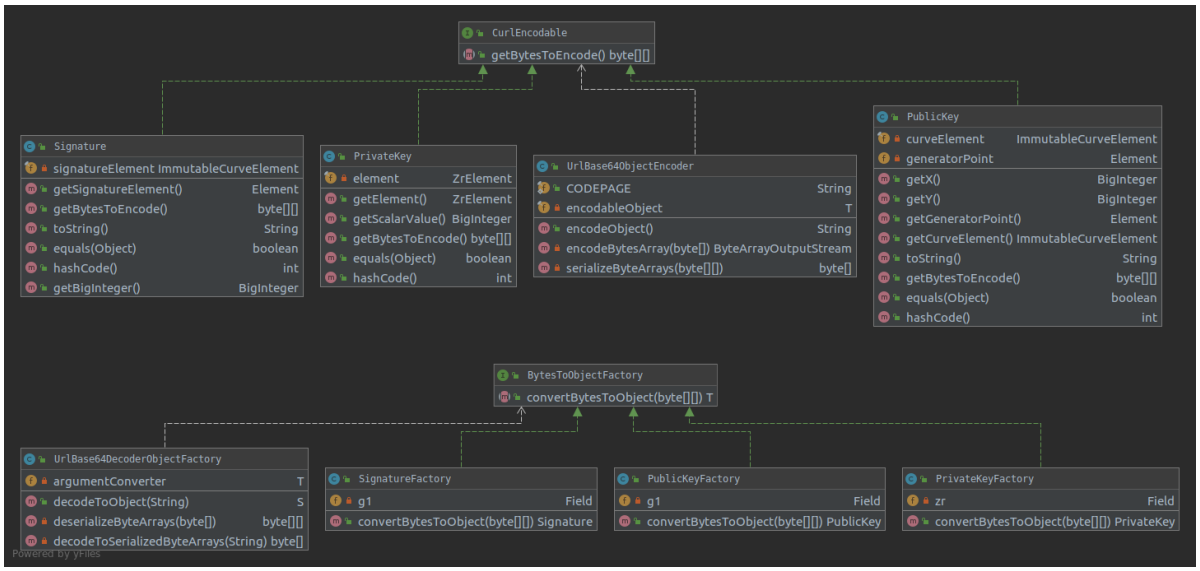

Figure 4.2: UML diagram of base64 encoding mechanism.

4.2.2 Application flow

This section discusses how a request to an endpoint is processed. The `HttpRequestProcesser` is the initial entry point, `RequestParameters` is an utility class to process the query string and the `StoreHandler` and `SignHandler` are responsible to generate a response. An overview of the main classes responsible for answering a store or sign request is provided in figure 4.3.

4.2.2.1 HttpRequestProcesser

Once a secret is defined, a secret can be stored by invoking a GET request to the `/store` endpoint with the name of the secret as the value of the `name` parameter. PROTECT uses the `HttpsServer` class provided by Java to process these requests [6]. This class is used in the `HttpRequestProcesser` class as shown in listing 9. It defines the endpoints and assigns a handler to each endpoint. Whenever an endpoint is accessed the respective handle method is called. In the case of the `StoreHandler` and the `SignHandler` the method invoked is called `authenticatedClientHandle`.

```
public class HttpRequestProcessor {

   /** omitted */

   private final HttpsServer server;

   /** omitted */

   public void addHandlers(/** omitted */) {

      // Returns basic information about this server: (quorum information, other
    servers)
      this.server.createContext("/", new RootHandler(serverIndex, serverConfig,
    shareholders));

      /** omitted */

      // Handlers for reading or storing shares
      this.server.createContext("/read", new ReadHandler(clientKeys,
    accessEnforcement, serverConfig, shareholders));
      this.server.createContext("/store", new StoreHandler(clientKeys,
    accessEnforcement, shareholders));
```

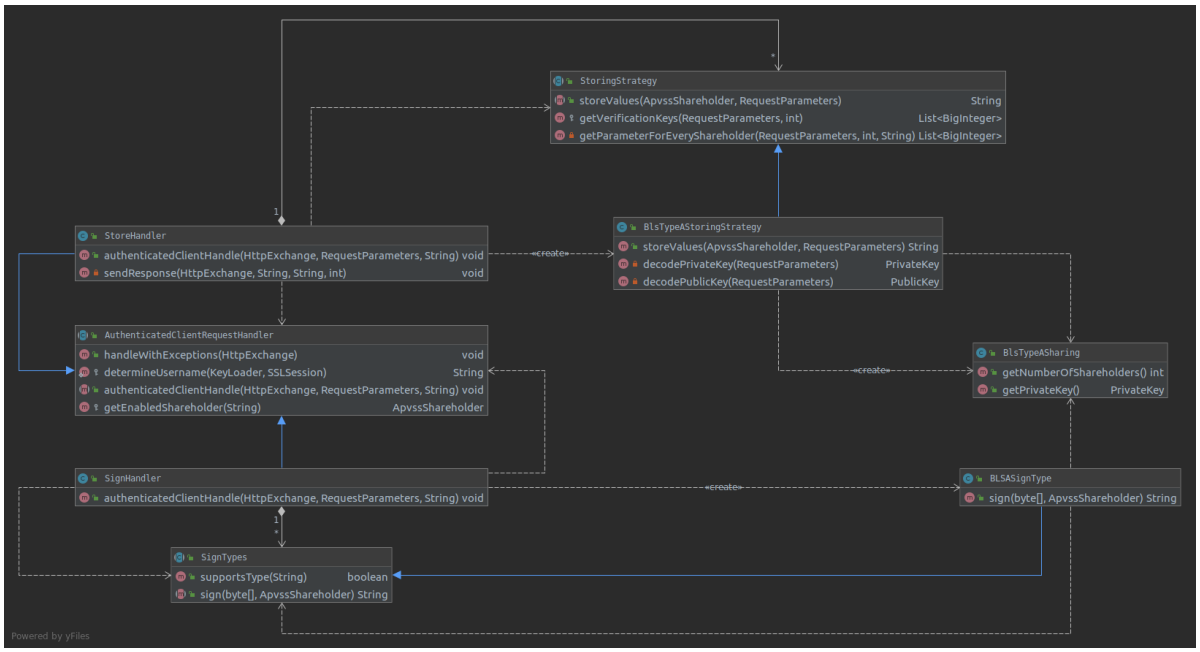

Figure 4.3: Classes responsible for generating a response.

```
    /** omitted */

    this.server.createContext("/sign", new SignHandler(clientKeys,
 accessEnforcement, shareholders));

    // Define server to server requests
    this.server.createContext("/partial", new PartialHandler(serverKeys,
 shareholders));
}
```

Listing 9: Method `addHandler` of the `HttpRequestProcessor` class.

4.2.2.2 Request Parameters

The constants for the parameter names as well as logic to parse the query parameters are located in the class `RequestParameters` (see listing 10). We have introduced this class with methods to obtain parameter values as `String` or `BigInteger`, to query whether a parameter exist and to automatically throw a `BadRequestExcepetion` in case a required parameter is missing. This makes for less code duplication, better code maintenance and easier reading of the handler classes. Additionally it becomes easier to implement new handlers. The class makes heavy use of Java's Optional Class [26]. This is built-in in many Java and allows to indicate whether a value is present or absent without passing null values.

The class is initialized by calling the constructor with a `HttpExchange` object. In the constructor the query string is parsed and stored in the parameters map. The map contains a list of parameter values ordered by parameter names as key. The main method is `getParameter` accepting the parameter name as `String` and returning an `Optional<String>`. If the value was present in the request, it can be found in the parameters map and is returned as the value of the optional. Otherwise an empty optional is returned. Most other methods rely on this method. The method `getRequiredParameter` is similar, but returns the value as `string` or throws a `BadRequestException` if the value was not present. Convenience methods to check whether a parameter is present without actually returning the value or to return the parameter value as a `BigInteger` are also implemented. The `String` constants contain the

names of the parameters. Using these constants instead of using the parameter names directly decreases the risk of misspelling type names.

```
/** omitted */

public class RequestParameters {

   // Query Parameters
   public static final String SECRET_NAME_FIELD = "secretName";
   public static final String MESSAGE_FIELD = "message";

   /** omitted */

   public static class CRYPTO_TYPES {
      public static final String RSA = "rsa";
      public static final String BLSA = "bls-a";
   }

   private final Map<String, List<String>> parameters;

   public RequestParameters(HttpExchange exchange) throws
    UnsupportedEncodingException {
      // Extract parameters from request
      final String queryString = exchange.getRequestURI().getQuery();
      parameters = parseQueryString(queryString);
   }

   /** omitted */

   public String getRequiredParameter(final String parameterName) throws
    BadRequestException {
      Optional<String> parameterValue = getParameter(parameterName);
      if (parameterValue.isPresent()) {
          return parameterValue.get();
      }
      throw new BadRequestException();
   }

   public BigInteger getRequiredParameterAsBigInteger(final String parameterName)
    throws BadRequestException {
      return new BigInteger(getRequiredParameter(parameterName));
   }

   /** omitted*/

   public Optional<String> getParameter(final String parameterName) {
      final List<String> parameterValues = this.parameters.get(parameterName);
      if ((parameterValues == null) || (parameterValues.get(0) == null)) {
         return Optional.empty();
      } else {
         return Optional.of(parameterValues.get(0));
      }
   }

   /** omitted */

   public boolean getParameterExists(String parameterName) {
      return getParameter(parameterName).isPresent();
   }

}
```

Listing 10: The `RequestParameter` class.

4.2.2.3 StoreHandler

The logic of storing the secrets sit in the classes `RSAStoringStrategy` and `BlsTypeAStoringStrategy`. These classes are initialized in the constructor of the `StoreHandler` and stored in a Map. This scheme can be easily extended to other secret types.

```
/** omitted */
public class StoreHandler extends AuthenticatedClientRequestHandler {
   public static final Permissions REQUEST_PERMISSION = Permissions.STORE;
   // Fields
   private final AccessEnforcement accessEnforcement;
   private Map<String, StoringStrategy> storingStrategies;
   public StoreHandler(final KeyLoader clientKeys, final AccessEnforcement
    accessEnforcement,
                  final ConcurrentMap<String, ApvssShareholder> shareholders) {
      super(clientKeys, shareholders);
      this.accessEnforcement = accessEnforcement;
      this.storingStrategies = new HashMap<>();
      this.storingStrategies.put(RequestParameters.CRYPTO_TYPES.RSA, new
    RSAStoringStrategy());
      this.storingStrategies.put(RequestParameters.CRYPTO_TYPES.BLSA, new
    BlsTypeAStoringStrategy());
   }
   /** omitted */
}
```

Listing 11: The `StoreHandler`'s constructor.

In both cases we have added a new type parameter within the `authenticatedClientHandle` method as can be seen in listing 12. The first parameter `exchange` contains information about the current request and is used to trigger the sending of the response. The third parameter `username` indicates the user of the current request and is provided by the authentication system of PROTECT. As part of our work we introduced a `RequestParameters` object which is passed as the second parameter and will be further explained later on. Whereas previously the system was intended to only work with RSA, the request now needs to contain a type parameter indicating whether a BLS secret or a RSA secret is given. The field `storingStrategies` contains a dictionary of `StoringStrategies` which will be selected based on the type parameter. If the type is unknown, `storingStrategy` will be null and a `NotImplementedException` will be thrown. Otherwise the method storeValues on the interface `StoringStrategy` will be called with the `requestParameters` and the `shareholder` containing the users private key. This method will process the `requestParameters` and store the private key share.

```
/** omitted */

public class StoreHandler extends AuthenticatedClientRequestHandler {

   /** omitted */

   @Override
   public void authenticatedClientHandle(final HttpExchange exchange, final
    RequestParameters requestParameters, final String username)
         throws IOException, UnauthorizedException, NotFoundException,
    BadRequestException,
         ResourceUnavailableException, ConflictException, InternalServerException {
      // Extract secret name from request
      final String secretName = requestParameters.getRequiredParameter(
    RequestParameters.SECRET_NAME_FIELD);

      /** omitted */

      String type = requestParameters.getRequiredParameter(RequestParameters.
    CRYPTO_TYPE);
```

```
      StoringStrategy storingStrategy = storingStrategies.get(type);
      if (storingStrategy == null) {
         throw new NotImplementedException();
      }
      String storeResponse = storingStrategy.storeValues(shareholder,
    requestParameters);
      /** omitted */
   }

   /** omitted */

}
```

Listing 12: The `StoreHandler`'s `authenticatedClientHandle` method.

4.2.2.4 SignHandler

The implementation of the `SignHandler` is achieved in a similar way as in the `StoreHandler`. The actual signature is computed in the method `produceBlsASignatureResponse` of the class `ThresholdSignatures`. Listing 13 shows how a signature is generated. The process largely depends on JPBC classes. The `BLS01Signer` is initialized and instructed to use SHA256 to generate the message hashes. The private key share of the authenticated user is obtained from the `blsTypeASharing`. A pairing is constructed using predefined curve properties (see also section 4.1). The key share is converted to a JPBC element using the Z_r field. Further the signer is initialized using the private key and curve parameters. The signature generation follows standard cryptography practice [27].

```
   public static byte[] produceBlsASgnatureResponse(byte[] message, BlsTypeASharing
    blsTypeASharing) {
      BLS01Signer signer = new BLS01Signer(new SHA256Digest());
      BigInteger keyShare = blsTypeASharing.getKeyShare();
      PairingParameters pairingParameters = PairingFactory.getPairingParameters("
    config/curves/a.properties");
      BigInteger order = PairingFactory.getPairing(pairingParameters).getZr().
    getOrder();
      ZrField zrField = new ZrField(order);
      ZrElement<ZrField> keyshareElement = new ZrElement<>(zrField, keyShare);
      BLS01Parameters bls01Parameters = new BLS01Parameters(pairingParameters,
    keyshareElement);
      CipherParameters privateKeyShare = new BLS01PrivateKeyParameters(
    bls01Parameters, keyshareElement.getImmutable());
      signer.init(true, privateKeyShare);
      signer.update(message, 0, message.length);
      try {
         return signer.generateSignature();
      } catch (CryptoException e) {
         throw new RuntimeException("Signature generation failed.");
      }
   }
```

Listing 13: The method `produceSignatureResponse` of the class `ThresholdSignatures` constructor.

5 Conclusion

In this thesis we have implemented the BLS signature scheme in PROTECT. We have extended the existing endpoint working with RSA key pairs to also work with BLS keys. With our demo client we have tested a threshold scheme and successfully generated signature shares and aggregated theses signatures. The signature generation with JPBC is reasonably fast: The generation of the key pair takes 13 milliseconds, the generation of a signature 37 milliseconds in average when creating 1000 key pairs and 1000 signatures for different messages and key pairs.

Further work may include the implementation of the encryption endpoint.

Bibliography

[1] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the weil pairing," *Journal of Cryptology*, vol. 17, no. 4, jul 2004.

[2] Y. Desmedt and Y. Frankel, "Threshold cryptosystems," in *Advances in Cryptology — CRYPTO' 89 Proceedings*, G. Brassard, Ed., 1990, pp. 307–315.

[3] jasonkresch, "protect - A Platform for Robus Threshold Cryptography," Mar 2019, [Online; accessed 10. Feb. 2020]. [Online]. Available: https://github.com/jasonkresch/protect

[4] A. De Caro and V. Iovino, "jpbc: Java pairing based cryptography," in *Proceedings of the 16th IEEE Symposium on Computers and Communications, ISCC 2011*. Kerkyra, Corfu, Greece, June 28 - July 1: IEEE, 2011, pp. 850–855. [Online]. Available: http://gas.dia.unisa.it/projects/jpbc/

[5] J. Katz, *Cryptography*, 2004, cited By 0.

[6] "The GNU MP Bignum Library," Jan 2020, [Online; accessed 23. Feb. 2020]. [Online]. Available: https://gmplib.org

[7] "Digital signing. Public-key cryptography / asymmetric cryptography." May 2019. [Online]. Available: https://commons.wikimedia.org/wiki/File:Private_key_signing.svg

[8] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography, "11 Digital Signatures"*. Taylor & Francis Inc, 1996.

[9] D. Boneh, C. Gentry, B. Lynn, and H. Shacham, "Aggregate and Verifiably Encrypted Signatures from Bilinear Maps," pp. 416–432, May 2003.

[10] ——, "A survey of two signature aggregation techniques," *CryptoBytes*, vol. 6, 09 2003.

[11] A. Block, "Secret Sharing and Threshold Signatures with BLS," *Medium*, Jul 2018. [Online]. Available: https://blog.dash.org/secret-sharing-and-threshold-signatures-with-bls-954d1587b5f

[12] J. Camenisch, S. Hohenberger, and M. Ø. Pedersen, "Batch verification of short signatures," *Lect. Notes Comput. Sci.*, vol. 4515 LNCS, pp. 246–263, 2007.

[13] T. ElGamal, "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms," *Lect. Notes Comput. Sci.*, vol. 196 LNCS, pp. 10–18, 1985.

[14] "Shamir's Secret Sharing - Wikipedia," Feb 2020, [Online; accessed 4. Feb. 2020]. [Online]. Available: https://en.wikipedia.org/wiki/Shamir%27s_Secret_Sharing

[15] "The MIT License | Open Source Initiative," Feb 2020, [Online; accessed 12. Feb. 2020]. [Online]. Available: https://opensource.org/licenses/MIT

[16] Contributors to Wikimedia projects, "Pseudorandom function family - Wikipedia," Nov 2019, [Online; accessed 5. Jun. 2020]. [Online]. Available: https://en.wikipedia.org/wiki/Pseudorandom_function_family

[17] ——, "Elliptic-curve cryptography - Wikipedia," Jun 2020, [Online; accessed 5. Jun. 2020]. [Online]. Available: https://en.wikipedia.org/wiki/Elliptic-curve_cryptography

[18] ——, "Diffie–Hellman key exchange - Wikipedia," Jun 2020, [Online; accessed 5. Jun. 2020]. [Online]. Available: https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange

[19] V. Shoup, "Practical threshold signatures," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 1807, pp. 207–220, 2000.

[20] Contributors to Wikimedia projects, "Elliptic-curve cryptography - Wikipedia," Jun 2020, [Online; accessed 18. Jun. 2020]. [Online]. Available: https://en.wikipedia.org/wiki/Elliptic-curve_cryptography

[21] B. Lynn, "Pbc library," in *The Pairing-Based Cryptography Library*. Kerkyra, Corfu, Greece, June 28 - July 1: IEEE, pp. 850–855. [Online]. Available: https://crypto.stanford.edu/pbc/

[22] "Gnu lesser general public license gnu.org," Feb 2020, [Online; accessed 23. Feb. 2020]. [Online]. Available: http://www.gnu.org/licenses/lgpl-3.0.html

[23] square, "jna-gmp," Nov 2019, [Online; accessed 23. Feb. 2020]. [Online]. Available: https://github.com/square/jna-gmp

[24] "The leading operating system for PCs, IoT devices, servers and the cloud | Ubuntu," Feb 2020, [Online; accessed 23. Feb. 2020]. [Online]. Available: https://ubuntu.com

[25] "bouncycastle.org," Sep 2019, [Online; accessed 8. Jun. 2020]. [Online]. Available: https://bouncycastle.org/documentation.html

[26] "Optional (Java Platform SE 8)," Jan 2020, [Online; accessed 24. Feb. 2020]. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html

[27] "Java Cryptography Architecture (JCA) Reference Guide," Jun 2020, [Online; accessed 8. Jun. 2020]. [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html