Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Implementing a Puncturable Key Wrapping Library

Semester Project

Younis Khalil

January 12, 2023

Advisors: Prof. Dr. Kenny Paterson, Dr. Felix Günther, Matilda Backendal

Applied Cryptography Group
Institute of Information Security
Department of Computer Science, ETH Zürich

Abstract

Puncturable Key Wrapping (PKW) was recently outlined in [1]. It extends the symmetric primitive of key wrapping [2] with an additional operation, called *puncture*, which, when used on a wrapped key, makes that key irrecoverable. In this work, a functional implementation is realized. The goal is to determine the feasibility of parameters allowing for real-world usage and to gather insights from practical issues. To that end, an abstract application programming interface (API) is defined and instantiated with two concrete realizations in C++. The first is a naïve approach, and amounts to an indexed collection of keys. The second is an approach put forward by [1] and based on puncturable pseudo-random functions (PPRFs) [3] and utilizes an AEAD scheme [4]. Considerations are made regarding exportability of secrets and secure operation of the library.

The library is made available under the MIT license.

Contents

Chapter 1

Introduction

Key-wrapping [2] describes a symmetric cryptographic primitive which provides protection of keys under some secret master key, for example during their transport and storage. The protection is defined in terms of confidentiality and integrity. As put forward by [1], key-wrapping can be extended to provide fine-grained forward security for the wrapped keys. They call this scheme puncturable key-wrapping (PKW). In addition to the usual wrapping and unwrapping operations, it provides the possibility to *puncture* the secret key on wrapped keys, rendering them irrecoverable (i.e., they cannot be unwrapped) with the updated secret key produced by this operation. This is of interest in a number of settings, and the authors put forward two examples: TLS ticketing and protected file storage (PFS) in the cloud. To provide some intuition, consider the cloud storage setting: a data encryption key (DEK) is be used to encrypt a file, the DEK is wrapped and stored in the cloud alongside the encrypted file. To read the file, it is downloaded alongside the corresponding wrapped DEK, the DEK is unwrapped and used to decrypt the file. To *delete* a file, the (wrapped) DEK is punctured. Because the DEK now is irrecoverable, the file it encrypts cannot be decrypted anymore; whether the file is physically deleted is irrelevant. The forward security of the wrapped DEK provides secure deletion for the file.

The goal of this work is to implement the puncturable key-wrapping scheme: to design an appropriate API and to realize the scheme. We provide two instantiations: one, called naïve PKW and another which combines a puncturable pseudo-random function (PPRF) [5, 6, 7] and an AEAD [4] scheme.

A number of obstacles were encountered in the process and they are outlined in this report.

The first obstacle to overcome was the choice of a programming language. There were different aspects to be weighed against each other, including the reuse of this work in future projects, the extent to which results could be

produced (depending on the author's familiarity with the programming language, more, or less, time would be spent on understanding the language, rather than implementing and testing the scheme). In an exploratory phase, the design of the API was started in Python. It became clear quickly that there is a necessity of being able to manipulate memory directly in order to reliably scrub secrets from memory, as the forward security properties if the PKW scheme depend on the reliable deletion of old secret keys. This led to a choice being made for C++, as it is a widely used language and provides the required access to memory. Once the choice was made, subtle problems like *dead store elimination* [8] (a compiler optimization, which removes code during compilation) posed a challenge. The realization of the functionalities of the scheme in a functioning library led to considerations of key management and attack vectors. The secret keys have to be stored and loaded between different sessions. During this time they have to be stored securely; to enable this, an encryption functionality was included alongside serialization. In the first prototype, implemented in Python, serializing an object was simple. In C++, a library could be used or it could be done manually – both were more time-consuming than anticipated, but manual serialization was selected as it guarantees full control of the process. Finally, some consideration had to be given to licensing. If software is to be made available to the world, which is the goal for a library, a license has to be specified, declaring which use of the software is permitted.

This report starts by providing some preliminaries, introduces the PKW scheme and shows how it was implemented, also presenting some benchmarks for the implementation. It then explains choices made about the programming environment and general considerations about library design, describes some of the intricacies of memory sanitization in C++ and considers copyright questions and software licenses.

Further work is needed to analyze the efficiency as well as evaluate the function of the library in applications, such as protected file storage.

Chapter 2

Preliminaries

We present some notation used throughout the report and provide the syntax of AEAD.

2.1 Notation

Throughout the report, there are some passages with formal definitions. In those, we use the symbol $||$ to denote concatenation, $u \leftarrow v$ to denote assignment of value v to variable u, $x \leftarrow\!\!\$\ \mathcal{X}$ to denote that x is sampled from the set $\mathcal{X}$ uniformly at random, $\{0,1\}^n$ and $\{0,1\}^*$ to denote bitstrings of length n or of any length (including the empty string), respectively. The special symbol $\perp$ (pronounced *bot*) is used to denote rejection.

2.2 AEAD

We restate the definition of AEAD [4] as it is presented in [1].

Definition 2.1 (AEAD scheme)**.** An authenticated encryption with associated data scheme, $\mathsf{AEAD} = (\mathsf{Enc}, \mathsf{Dec})$, is a pair of algorithms with four associated sets; the secret-key space $\mathcal{SK}$, the nonce space $\mathcal{N}$, the associated data space $\mathcal{AD}$ and the message space $\mathcal{M}$. Further associated with AEAD is a ciphertext-length function $cl : \mathbb{N} \to \mathbb{N}$. The algorithms of AEAD operate as follows.

- Via $C \leftarrow \mathsf{Enc}(sk, N, ad, M)$, the deterministic encryption algorithm Enc on input the secret key $sk \in \mathcal{SK}$, a nonce $N \in \mathcal{N}$, associated data $ad \in \mathcal{AD}$ and a message $M \in \mathcal{M}$ produces a ciphertext $C \in \{0,1\}^{cl(|M|)}$.
- Via $M/\perp \leftarrow \mathsf{Dec}(sk, N, ad, C)$, the deterministic decryption algorithm Dec on input the secret key $sk \in \mathcal{SK}$, a nonce $N \in \mathcal{N}$, associated data $ad \in \mathcal{AD}$ and a ciphertext $C \in \{0,1\}^*$ produces a message $M \in \mathcal{M}$ or, to indicate failure, the special symbol $\perp$.

Correctness of a nonce-based AEAD scheme stipulates that $\mathsf{Dec}(sk, N, ad, \mathsf{Enc}(sk, N, ad, M)) = M$ for all $sk \in \mathcal{SK}$, $N \in \mathcal{N}$, $ad \in \mathcal{AD}$ and $M \in \mathcal{M}$.

Chapter 3

Puncturable Key Wrapping

Puncturable key wrapping (PKW) is a primitive providing fine-grained forward security properties in symmetric key hierarchies. It is of interest in different applications, e.g. protected file storage (PFS) [1]. The goal is to have key wrapping with an additional puncturing operation, which allows modifying the main secret key in such a way that makes a previously wrapped key irrecoverable. In doing so, the confidentiality of punctured (wrapped) keys is maintained after the compromise of the main secret key.

3.1 Defining the API

We repeat the formal definition, as stated in [1] and derive corresponding C++ function signatures:

Definition 3.1 (PKW scheme)**.** A puncturable key-wrapping scheme consists of four algorithms $\mathsf{PKW} = (\mathsf{Keygen}, \mathsf{Wrap}, \mathsf{Unwrap}, \mathsf{Punc})$ with four associated sets: the secret-key space $\mathcal{SK}$, the tag space $\mathcal{T}$, the header space $\mathcal{H}$ and the wrap-key space $\mathcal{K}$

- Via $sk \leftarrow\!\!\$\ \mathsf{KeyGen}()$, the probabilistic key generation algorithm KeyGen, taking no input, outputs a secret key $sk \in \mathcal{SK}$.
- Via $C/\perp \leftarrow \mathsf{Wrap}(sk, T, H, K)$, the deterministic wrapping algorithm Wrap on input a secret key $sk \in \mathcal{SK}$, a tag $T \in \mathcal{T}$, a header $H \in \mathcal{H}$ and a key $K \in \mathcal{K}$ outputs a ciphertext $C \in \{0,1\}^{cl(|K|)}$ or, to indicate failure, $\perp$.
- Via $K/\perp \leftarrow \mathsf{Unwrap}(sk, T, H, C)$, the deterministic unwrapping algorithm Unwrap on input a secret key $sk \in \mathcal{SK}$, a tag $T \in \mathcal{T}$, a header $H \in \mathcal{H}$ and a ciphertext $C \in \{0,1\}^*$ returns a key $K \in \mathcal{K}$ or, to indicate failure, $\perp$.

- Via $sk' \leftarrow \mathsf{Punc}(sk, T)$, the deterministic puncturing algorithm Punc on input of a secret key $sk \in SK$ and a tag $T \in \mathcal{T}$ returns a potentially updated secret key $sk' \in \mathcal{SK}$.

Translated into C++ function definitions, the function signatures of Wrap, Unwrap and Punc look as follows:

```
template<class T, class C>
class AbstractPKW {
    public:

        virtual C
        wrap(T tag, std::vector<unsigned char> &header,
    std::vector<unsigned char> &key) = 0;


        virtual std::vector<unsigned char>
        unwrap(T tag, std::vector<unsigned char> &header, C &c)
    = 0;

        virtual void punc(T tag) = 0;
}
```

`T` is the type of the tag, `C` is the type of the ciphertext, usually a variable-length byte array (`std::vector<unsigned char>`). The functions are defined as pure (the assignment of the value 0) `virtual` functions, virtual meaning they must be overridden in subclasses and pure designating that AbstractPKW is an abstract class, so no objects of this type can be created. Subclasses must provide an implementation for these functions, otherwise, they are considered abstract too.

Additionally, there is a necessity to be able to store the secret key: sometimes a machine running the library may need to be updated, or the key has to be moved to a different machine. Depending on the instantiation of the scheme, keys may come in quite different forms. The most general to which to serialize a key was considered a byte string. The API functions for importing and exporting were therefore defined as:

```
virtual SecureByteBuffer serializeKey() = 0;

virtual SecureByteBuffer serializeAndEncryptKey(const
    std::string &password) = 0;
```

For an explanation of the usage of `SecureByteBuffer`, see Chapter 5.

For deserialization the function signatures are

```
virtual std::shared_ptr<AbstractPKW<T, C>>
    fromSerialized(SecureByteBuffer &serialized) = 0;

```

```
3 std::shared_ptr<AbstractPKW<T, C>>
    fromSerializedAndEncrypted(SecureByteBuffer
    &serializedAndEncrypted, const std::string &password);
```

As the reader will have noticed, there is the option to encrypt the exported key (with a password-derived encryption key). This offers a layer of protection to programmers when unloading a key. Still, care should be taken in handling it. Especially after the deserialization (i.e., loading) of a key, it is vital that the exported key is deleted. It is vital because the forward security properties of the PKW scheme can only be maintained if the key is updated in a timely fashion; if an old key is still available after punctures were made, it can be loaded again, nullifying the punctures.

The return type for deserialization was elected to be an `AbstractPKW<T,C>`, to remain general. Because it is an abstract class, it cannot be instantiated, so the return type is a (smart) pointer to an `AbstractPKW<T,C>`.

We present two implementations of the PKW API.

3.2 NaïvePKW

A simple, or maybe naïve, instantiation of the PKW scheme is to use a list of keys used for wrapping, with one key per tag. If a tag is punctured, the key indexed by that tag is erased. While it is conceptually simple to understand, space efficiency is an issue, as all the keys must be stored. Still, it may be a valid scheme for applications with a small tag space. It also proved a useful way of ensuring that the API is sensible as it was easy to implement and perform basic tests on. In the C++ implementation, the list of keys is organized as a map:

```
1 #define MAC_LEN 12
2 #define NONCE_LEN 16
3 #define KEY_LEN 16
4
5 using Key = std::map<long, std::array<unsigned char, KEY_LEN> *>;
6
7 class NaivePKW : public AbstractPKW<long, std::vector<unsigned
    char>> {
8 ...
9 };
```

Listing 3.1: naive_pkw.h

3.3 PKW construction from a PPRF and AEAD scheme

As described by [1] in Chapter 4.2, a PKW can be instantiated with a PPRF and an Authenticated Encryption with Associated Data (AEAD) [4] scheme. For completeness, we repeat it in full in Figure 3.1.

PKW[PPRF, AEAD] :

PKW.KeyGen():

1 Return PPRF.KeyGen()

PKW.Wrap(sk_p, T, H, K):

1 $sk_a \leftarrow$ PPRF.Eval(sk_p, T)
2 $C \leftarrow$ AEAD.Enc(sk_a, N_0, H, K)
3 Return C

PKW.Unwrap(sk_p, T, H, C):

1 $sk_a \leftarrow$ PPRF.Eval(sk_p, T)
2 $K \leftarrow$ AEAD.Dec(sk_a, N_0, H, C)
3 Return K

PKW.Punc(sk_p, T):

1 $sk'_p \leftarrow$ PPRF.Punc(sk_p, T)
2 Return sk'_p

N_0 is a fixed nonce, for simplicity it is set to 0 in the implementation.

Figure 3.1: PKW from composition of PPRF and AEAD

The PPRF used in this project is the one described in Chapter 3.3.1. For the AEAD scheme, Crypto++'s [9] implementation of AES in GCM mode was chosen.

To wrap and unwrap, the PPRF is used to obtain the key-encryption key (sk_a), which is then used by the AEAD scheme for encryption or decryption, respectively. Puncturing is delegated to the PPRF and is performed as described in Chapter 3.3.1.

Listing 3.3 is a C++ code excerpt, showing the use of the PPRF and the AEAD scheme.

```
1  using ciphertext = std::vector<unsigned char>;
2
3  ciphertext PPRF_AEAD_PKW::wrap(Tag tag, vector<unsigned char>
       &header, vector<unsigned char> &key) {
4      SecureByteBuffer wrapping_key = pprf.eval(tag);
5      try {
6          CryptoPP::GCM<CryptoPP::AES>::Encryption e;
7          vector<unsigned char> iv(16, 0);
8          e.SetKeyWithIV(wrapping_key.data(), wrapping_key.size(),
       iv.data(), iv.size());
9          ciphertext cipher;
10         CryptoPP::AuthenticatedEncryptionFilter ef(
11             e,
12             new CryptoPP::VectorSink(cipher),
13             false,
14             TAG_SIZE /* MAC_AT_END */);
15         ef.ChannelPut(CryptoPP::AAD_CHANNEL, header.data(),
       header.size());
16         ef.ChannelMessageEnd(CryptoPP::AAD_CHANNEL);
17
18         // Confidential data comes after authenticated data.
19         ef.ChannelPut(CryptoPP::DEFAULT_CHANNEL, key.data(),
       key.size());
```

```
20          ef.ChannelMessageEnd(CryptoPP::DEFAULT_CHANNEL);
21          return cipher;
22      } catch (CryptoPP::Exception &e) {
23          throw WrappingException();
24      }
25  }
```

Listing 3.2: composition.]Implementation of wrap in the [PPRF, AEAD] composition.

3.3.1 Puncturable Pseudo-Random Functions (PPRF)

The PKW scheme can also be instantiated with a PPRF and an AEAD scheme (see Chapter 3.3). To that end, a PPRF based on the PRF from PRG construction defined by Goldreich, Goldwasser & Micali (GGM) [3] was implemented.

Again, we provide the definition of PPRFs from [1]:

Definition 3.2 (PPRF)**.** A puncturable pseudo-random function, $\mathsf{PPRF} = (\mathsf{KeyGen}, \mathsf{Eval}, \mathsf{Punc})$, is a triple of algorithms with three associated sets; the secret-key space $\mathcal{SK}$, the domain $\mathcal{X}$ and the range $\mathcal{Y}$.

- Via $sk \leftarrow\!\!\$\ \mathsf{KeyGen}()$, the probabilistic key generation algorithm KeyGen, taking no input, outputs the secret key $sk \in \mathcal{SK}$.
- Via $y/\perp \leftarrow \mathsf{Eval}(sk, x)$, the function evaluation algorithm Eval, on input the secret key sk and an element $x \in \mathcal{X}$ outputs $y \in \mathcal{Y}$ or, to indicate failure, $\perp$.
- Via $sk' \leftarrow \mathsf{Punc}(sk, x)$, the deterministic puncturing algorithm Punc, on input the secret key sk and an element $x \in \mathcal{X}$ outputs an updated secret key $sk' \in \mathcal{SK}$.

For *correctness* we require that for all $sk \in \mathcal{SK}$ and all $x, y \in \mathcal{X}$:

- $\Pr[\mathsf{Eval}(sk_0, x) \neq \perp | sk_0 \leftarrow\!\!\$\ \mathsf{KeyGen}()] = 1$.
- If $sk' \leftarrow \mathsf{Punc}(sk, x) and y \neq x, then \mathsf{Eval}(sk, y) = \mathsf{Eval}(sk', y)$.
- If $sk' \leftarrow Punc(sk, x)$, then $Eval(sk', x) = \perp$.

The GGM construction lends itself to being made puncturable [7]. In the original version, a pseudo-random generator (PRG) with input size s and output size $2s$ is used to derive the output. The construction defines a binary tree structure through which an element is evaluated by its bit-string representation. Depending on the bit at index i, the left or right half of the PRG's output is used for the next evaluation. At the root is a secret key of length s.

Defining the PRG as $G : \{0,1\}^s \rightarrow \{0,1\}^{2s}, G(x) = G_0(x)||G_1(x)$, where G_0 and G_1 denote the left and right half of G, respectively, and $||$ denotes

concatenation, then the evaluation of $x = 10010 \in \mathcal{X} = \{0,1\}^5$ can be written as $F(x) = F(10010) = G_0(G_1(G_0(G_0(G_1(sk)))))$.

To implement the PPRF in code, for the PRG a pseudo-random function for which the output length can be chosen was selected. The chosen function is `HKDF.Expand`, which takes as input a key (randomness), a context information string, and the output length [10]. The context information string is used for the right and left derivations, with the info string either being set to `'r'` or `'l'`, respectively. The construction of our PRG with output length $2s$ is then: $G(x) = \mathsf{HKDF.Expand}(x, 'l', s) || \mathsf{HKDF.Expand}(x, 'r', s)$.

Puncturing

For correctness of the scheme (see the definition), when the secret key is punctured on input x, the key has to be adapted in such a way that x can no longer be evaluated, but all other previously evaluable elements still can. This demands that the root of the key must be deleted after the first puncture and it makes sense to think of the secret key as a collection of nodes defining their respective subtrees in the tree, initially containing only the root. To achieve reachability for all nodes except the one which was punctured, during the evaluation of x, first, the node which defines the subtree needed to evaluate x has to be found. Then the co-path is captured during the evaluation of x in the subtree; this collection of nodes then replaces the node which previously defined the subtree in the new secret key sk'.

Looking at the example in Figure 3.2, using the blue nodes, all leaf nodes can still be reached, except for the one red node that was punctured.

Implementation

The crucial part being the key, a data structure containing a list of nodes and information about the key was defined. the information consists of the length of tags, the length of keys and the number of punctures performed on the key.

```
class PPRFKey {
        int keyLen;
        int tagLen;
        int puncs;
        std::vector<SecretRoot> nodes;
};

class SecretRoot {
        std::string prefix;
        SecureByteBuffer value;
};
```

The nodes contain a prefix and a value. The prefix is the bitstring which indicates the path to the node in the tree, with 0 meaning left and 1 meaning

right. The value is the partial evaluation of the PPRF on the prefix. In Figure 3.2, the middle blue node has prefix 10 and contains the value $G_0(G_1(sk))$.

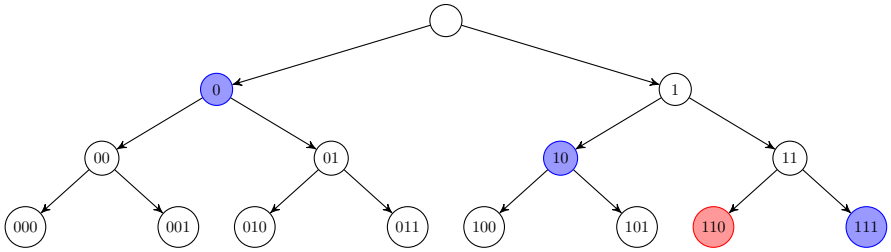

Figure 3.2: Example of a puncturing operation on the element 110, the co-path indicated in blue.

The nodes are ordered lexicographically based on the path that was taken to reach them (the prefix, in Figure 3.2 it is shown inside the nodes), and this ordering is maintained during all operations which change the key.

Initially, a key contains a single node, which in turn contains an empty prefix and the secret keying material (a byte string).

Eval As the key can grow substantially (at least initially), a quick evaluation function was necessary. To allow for quick evaluation, a binary search was implemented to find the node for which the prefix is indeed a prefix to the element which is being evaluated. A binary search is possible because the ordering invariant is maintained. There is always at most one ”matching-prefix” node since no node is contained in the subtree defined by another node (which would give the nodes a common prefix). After finding the correct node, the remainder of the bitstring representation of the element is used to derive the key.

```
1  long GGM_PPRF::findMatchingPrefix(Tag tag) {
2      std::string tagString = tag.to_string().substr(MAX_TAG_LEN -
       key.tagLen, MAX_TAG_LEN);
3      long min = 0;
4      long max = key.nodes.size();
5      while (max > min) {
6          long index = (max + min) / 2;
7          std::string pref = key.nodes[index].getPrefix();
8          if (pref == tagString.substr(0, pref.length())) {
9              return index;
10         } else if (pref < tagString) {
11             min = index + 1;
12         } else {
13             max = index;
14         }
15     }
16     return -1;
```

```
17 }
```

Listing 3.3: Find the index of the node with a matching prefix for a tag.

Looking at Figure 3.2 again, let's assume the key consists of the three blue nodes. When evaluating the element 010, first the node which shares the prefix with the tag to be evaluated among the set of nodes in the key has to be found. This is the root of the subtree in which the leaf node for the tag resides. If no such node exists, the tag must have been punctured and evaluation is not possible. In this example, the node is not punctured and is the left-most node in Figure 3.2. To evaluate, the value stored in the node $v = G_0(sk)$ is used to obtain the result by evaluating the tag in the subtree: $F(010) = G_0(G_1(v))$.

Punc As in Eval, a binary search is performed to find the node n with a matching prefix to the tag. If no such node exists, as in Eval, the key has been punctured on tag and there is nothing to be done. If the node is found, then during the evaluation of the remainder of the tag (the tag without the prefix), the co-path in the subtree is captured. Finally, n is replaced by the co-path, while maintaining the invariant that the nodes in the key are ordered lexicographically.

```
1  static const std::vector<unsigned char> RIGHT({'r'});
2  static const std::vector<unsigned char> LEFT({'l'});
3  using Tag = std::bitset<MAX_TAG_LEN>;
4
5  SecureByteBuffer evalAndGetCoPath(Tag tag, const SecretRoot
       &node, std::vector<SecretRoot> &coPath) const {
6      const int keyLenByte = key.keyLen / 8;
7      CryptoPP::HKDF<CryptoPP::SHA256> hkdf;
8
9      std::deque<SecretRoot> left;
10     std::deque<SecretRoot> right;
11
12     SecureByteBuffer curr(node.getValue());
13     SecureByteBuffer derived_right(keyLenByte);
14     SecureByteBuffer derived_left(keyLenByte);
15     std::string pref = node.getPrefix();
16     for (size_t i = node.getPrefix().size(); i < key.tagLen;
       i++) {
17         Tag mask;
18         mask.set(key.tagLen - i - 1, true);
19         hkdf.DeriveKey(derived_right.data(),
       derived_right.size(), curr.data(), curr.size(), nullptr, 0,
       RIGHT.data(), RIGHT.size());
20         hkdf.DeriveKey(derived_left.data(), derived_left.size(),
       curr.data(), curr.size(), nullptr, 0, LEFT.data(),
       LEFT.size());
21         if ((mask & tag).count() > 0) {
22             left.emplace_back(pref + "0", derived_left);
23             curr = derived_right;
```

```
24              pref += "1";
25          } else {
26              right.emplace_front(pref + "1", derived_right);
27              curr = derived_left;
28              pref += "0";
29          }
30      }
31      coPath.insert(coPath.end(), left.begin(), left.end());
32      coPath.insert(coPath.end(), right.begin(), right.end());
33      return curr;
34  }
```

Listing 3.4: Excerpt showing the function evalAndGetCoPath, which evaluates a tag by starting at `node` and gathers the nodes in the co-path in the list `coPath`.

Listing 3.4 shows how two deques, `left` and `right`, are used to gather the nodes of the co-path, depending on the bits of the tag (lines 21-29). The two deques are used in order to obtain the co-path in lexicographical ordering, so as to avoid an additional sorting operation at the end.

In Figure 3.2, initially the key contains the node at the root. After puncturing the key on tag 110, it contains the three blue nodes ordered as they appear in the figure, from left to right ('0' $<$ '10' $<$ '111').

3.3.2 PPRF Benchmarks

Some basic benchmarking was performed in order to judge the efficiency of the library. It was carried out on a machine with a 2.3 GHz 8-Core Intel Core i9 processor and access to 16 GB of memory.

A PPRF with tag lengths of 64, 32, and 16 bits and a key length of 128 bits was instantiated and repeatedly punctured on random tags (generated beforehand). After ten punctures, the time was recorded and the key was serialized to measure the growth in size.

Serialization size

The PKW API requires a serialization function for the key. In the [PPRF, AEAD] construction, the secret PKW key is really the secret key of the PPRF scheme. Serialization is therefore delegated to the PPRF implementation. This section shows how the key changes in size, relative to the number of punctures.

Figure 3.3 seems to show linear growth, but the line is actually convex. This is to be expected, since, with an increasing number of punctures, the matching prefix of a tag will be found "lower" in the tree, leading to a shorter co-path to be inserted into the key. At some point there will be mostly leaf nodes in the key, leading to decreasing key size with increasing punctures, as becomes visible with a small tag space, see Figure 3.4.

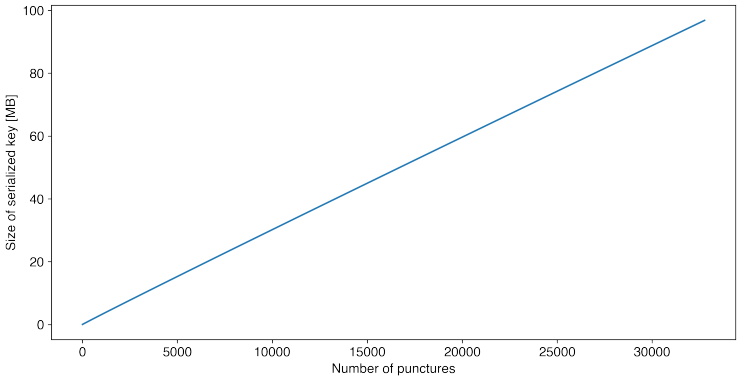

Figure 3.3: Size of the serialized (PPRF) key with tag size 64 bits

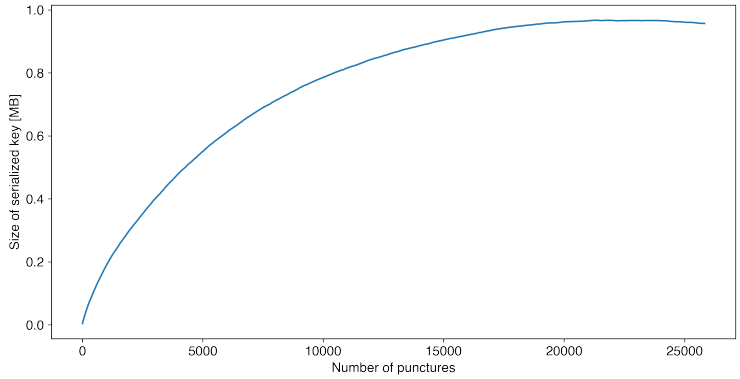

Figure 3.4: Size of the serialized (PPRF) key with tag size 16 bits

Comparing the size of the serialized key after a fixed number (2^{15}) of punctures for different tag lengths (Table 3.1), it becomes clear that larger tags lead to disproportionately larger key sizes after a fixed number of punctures. This may be because the larger the tag space, the more linear (smaller curvature) the curve up to 2^{15} punctures will appear. Or in other words, for smaller tag spaces, the curve already flattens out at this number of punctures (which can also be observed very clearly by comparing Figures 3.3 and 3.4). So, for shorter tag spaces the flattening-out effect is already visible at 2^{15} punctures, while with larger tag spaces the effect is not as present yet. As an illustration, we provide Figure 3.5, which compares the linear interpolation of the growth based on the first 20 punctures (orange) with the actual growth (blue) for a tag length of 16 bits.

tag length	size [MB]
16	0.90
32	11.78
64	44.97
128	155.29

Table 3.1: Serialization size of (PPRF) key after 15000 punctures

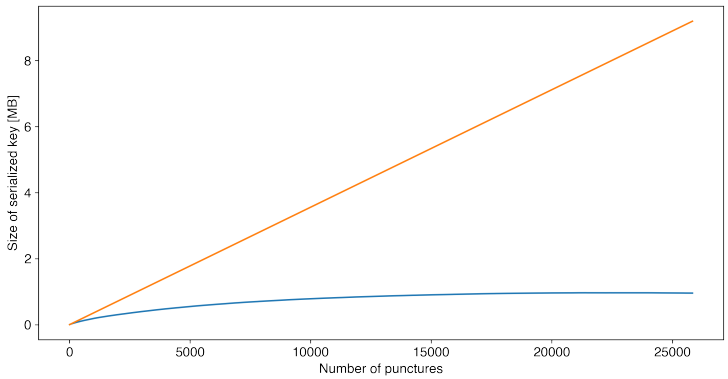

Figure 3.5: Serialization size for tags of 16 bits (blue) and linear interpolation of first 20 punctures (orange)

Time usage of puncturing

Figure 3.6 seems to be showing a linear relationship between the number of performed punctures and the time to perform additional punctures. This apparently linear growth in time usage was perplexing at first, however, when considering that the insertion time of nodes into the list is most probably the leading contributor, the observed behavior becomes more obvious. When the tag space is large, e.g. 128 bits, the key will contain many nodes after just a few punctures. For each puncture, during insertion of the co-path, the right part of the list has to be shifted, as the C++ vector keeps its elements in contiguous memory to allow access to elements in constant time. This ”big shift” introduces a time overhead in direct correlation to the key size, dwarfing any other time consumption during puncturing and as the size of the key is growing almost linearly, so is the time consumption. This seems inefficient, and a different data structure may be more suited to store the nodes.

Some very basic micro-benchmarking showed that most time ($> 90\%$) was spent inserting nodes (the co-path) and deleting the old node. This concretizes that there may be data structures more suited for the key. Further investigation into this issue is warranted. It also indicates that the time spent

in evaluation is not as excessive, but, this too requires a closer analysis.

As can be seen in Figure 3.7, the time for puncturing eventually stays in a certain range. This is because the size of the key has reached its maximum value, so the time of insertion of new nodes is no longer the leading contributor, but rather finding the node with a matching prefix for the tag.

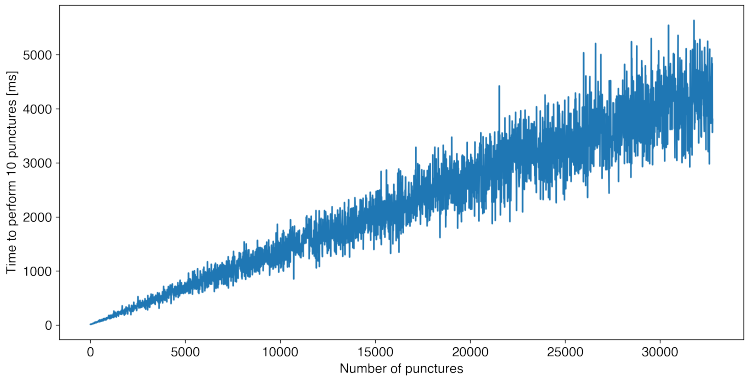

Figure 3.6: Duration of Punc with tag size of 64 bits)

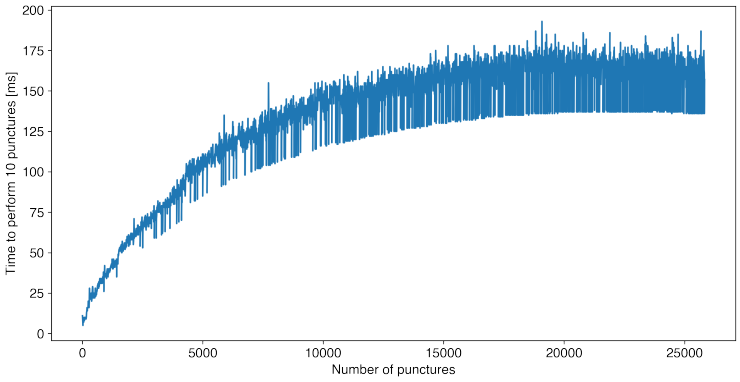

Figure 3.7: Duration of 10 punctures in PPRF with tag size of 16 bits

Comparison to NaïvePKW

No benchmarking was done for NaïvePKW. However, we provide a table listing the theoretical approximate serialization sizes (only the keying material is counted) for key sizes of 128 bits.

After 2^{15} punctures, the size would be about 4 MB smaller than the theo-

tag length	size [MB]
16	8
32	524288
64	$2.25 * 10^{15}$

Table 3.2: Theoretical serialization size with 0 punctures

retical values in Table 3.2 (the key size decreases with punctures). As can be seen, the relationship of the key size with respect to the tag length is exponential since all the keys have to be stored. Since the time for evaluation and puncturing are constant in this construction (access to elements in a `std::vector`), there is a trade-off to be considered for short tag sizes. If larger tags are required, NaïvePKW disqualifies itself because of its excessive memory needs.

Chapter 4

API and programming environment

4.1 Choice of programming language

When choosing a programming language for a project, one should be aware of the trade-off between ease of writing, readability, and maintainability (among many others). Some languages, such as Python, are easy to write, leading to a quick production of results, but offer no type safety. Other languages, among others Java and C++, require more boilerplate code but are easier to read and maintain because the required types provide documentation by themselves. Especially in a cryptographic library it is sensible to choose a language which preempts some problems by design, because it forces the developer to write more stable code (in the case of C++ this mostly refers to strong types for variables). Moreover, there are requirements that will disqualify some languages completely. As an example, in Java one cannot manipulate the memory underlying the variables directly.

4.1.1 Python vs. C++

The first objective was defining the abstract API – since the author was familiar with Python, and it is a widely used language, it seemed a sensible choice to start. However, it soon became apparent that the language did not offer as much control over memory as was required to guarantee the secure operation of the library (see Chapters 5, 6). Because of the requirement that memory be manageable, some previous experience with C and the popularity of C++, the project was continued in C++. Even then there were issues to overcome (cf. secure memory erasure, Chapter 5).

4.1.2 C++ environment

Build system CMake

Complex systems using many different files in C++ lead to complex build procedures where the files have to be linked during compilation. This can be done ”by hand” by specifying Makefiles, or a number of tools can be used to automate some of the work. In this project, the popular open-source tool CMake [11] was used. It uses a custom language for build specification, which presented a challenge in terms of time spent exploring its functionalities. However, it proved a capable tool once it was integrated.

Crypto++

Crypto++ [9] is an open-source, in part community-driven cryptographic library, from which standard cryptographic components needed for the implementation, among which AEAD and HKDF, were taken.

4.2 API considerations

In the design of a cryptographic library, many aspects are worth some deeper consideration. The authors of *Developers are Not the Enemy!: The Need for Usable Security APIs* [12] outline ten principles to which to adhere, in order to create secure and usable cryptographic APIs. One principle, for example, asks for cryptographic functionality to be integrated into (non-security) standard APIs, hiding the cryptography from developers. Another puts forward that the API should be easy to use, even without reading any documentation. While these and some others make sense for widely used cryptographic functionalities, where developers might not be familiar with cryptography, PKW seems like a scheme where developers should understand the underlying cryptography to use it. Depending on the construction, efficiency diminishes with continued use (with increasing punctures, operations take a longer time to complete, see chapter 3.3.2) and the library takes no steps to mitigate this. Furthermore, care must be taken in key handling to ensure the forward security properties are not broken.

Other principles demand general good practice, for example, that the API be understandable without reliance on documentation. To that end, care was taken in naming functions and files in a way that makes construction and serialization as straightforward as possible, as well as providing clear and concise function documentation. Additionally, we provide a demo file, demonstrating the usage (see Appendix A.2).

Chapter 5

Handling of secrets

Throughout the report, the class `SecureByteBuffer` is used to store keying material. This chapter provides an explanation for its conception and adoption.

5.1 Secure memory erasure

C++ offers developers strong control over memory, which is the key reason the language was chosen for this project. The compiler, however, introduces many optimizations, among which is the elimination of dead stores [8]. This means, that if a programmer explicitly sets a portion of memory to 0 by using `memset`, and does not read from the location at a later point, the compiler will remove the *dead store* in an effort to render the program more efficient. For security-critical applications this is an issue, since erasing secrets is not usually optional, but rather a requirement. There are a few workarounds to "trick" the compiler, and the authors of [8] have compiled them into a function called `secure_memzero`. Since C++11 there is the function `memset_s`, however, the authors of [8] point out that there is "no standard-compliant implementation", so even if the function was used the behavior would not be clearly defined for all compilers. In the current draft for C++23 there is the new function `memset_explicit` [13] with the same goal of providing a safe scrubbing method. As both `memset_s` and `memset_explicit` appear to be unusable at this time, `secure_memzero` is used instead.

5.1.1 SecureByteBuffer

The class `SecureByteBuffer` was conceived as a wrapper for a `std::vector <unsigned char>` (a variable-length list of bytes), which, during deconstruction, calls `secure_memzero`, thereby setting the occupied memory of the vector to zero before it is de-allocated. This way, the functionalities provided by a `std::vector` can be utilized, but it is still ensured that data be erased

once objects are no longer needed. Composition was chosen instead of specialization because it is generally not recommended to inherit from standard library (STL) classes.

```
class SecureByteBuffer {
    public:
        virtual ~SecureByteBuffer() {
            secure_memzero(vec.data(), vec.size());
        }

    private:
        std::vector<unsigned char> vec;
};
```

Deconstruction occurs any time an object has reached the end of its lifetime, for instance, if it is defined inside the scope of a function and the function returns. By using the `SecureByteBuffer` to store keys, their erasure can be guaranteed when an object is no longer used.

Chapter 6

Serialization

The structure of PKW keys is highly dependent on the used scheme, hence it was important to define a method to export and import a key. The most straightforward approach was to serialize the key objects into bytes, which is the method used in this project.

6.1 Boost

The Boost library [14] offers a serialization component. However, its use necessitates the addition of a function to each class which is to be serialized:

```
1 class ExampleClass {
2     private:
3         Key key;
4         friend class boost::serialization::access;
5
6         template<class Archive>
7         void serialize(Archive &arch, const unsigned int
    version);
8 };
```

There are a few problems here. Firstly, it requires the class `access` to be defined as a friend class, to allow access to the private field `key`. Secondly, during serialization, one cannot be sure to which places in memory the sensitive key was copied and, thirdly, there is no way to ensure its removal after serialization has been completed.

Because of this opacity in the library's functioning, keys are serialized *manually*. To ensure secrets do not linger in memory, the SecureByteBuffer (Chapter 5.1.1) class is used.

6.2 Manual serialization

6.2.1 Integers and endianness

Integers are stored in fixed-length memory, with their length depending on their type and the compiler. Depending on the endianness of the underlying machine, the bytes representing the integer are stored either from right to left or left to right (when considering the most significant byte).

For serialization this poses an issue if the serialized object has to be deserializable on other machines. For that reason, the decision was made to store integers in network order. The file `<arpa/inet.h>`, provided by UNIX-like systems (Windows provides similar functionalities), contains macros which can reorder the bytes from machine (**h**ost) **to n**etwork order (`htonll` for a 64-bit integer) and from network to machine order (`ntohll`). Listing 6.1 shows the handling of integers.

```
1  void writeInteger(std::vector<unsigned char> &buffer, uint64_t
       t1) {
2      uint64_t t2 = htonll(t1);
3      for (int i = 0; i < sizeof(uint64_t); ++i) {
4          unsigned char byte = (t2 >> (8 * i)) & 0xFF;
5          buffer.push_back(byte);
6      }
7  }
8
9  int getInt(SecureByteBuffer b, size_t offset) {
10     uint64_t ret = getUInt64(b, offset);
11     return ntohll(ret);
12 }
13 size_t getUInt64(SecureByteBuffer &b, size_t offset) {
14     if (b.size() < offset + sizeof(uint64_t)) {
15         throw PPRFDeserializationError();
16     }
17     uint64_t ret = 0;
18     for (int i = 0; i < sizeof(uint64_t); i++) {
19         uint64_t byte = b.data()[offset + i];
20         ret = ret | (byte << (i * 8));
21     }
22     return ret;
23 }
```

Listing 6.1: Serialization and deserialization of integers.

6.2.2 Byte arrays

Byte arrays are stored by first encoding their size (the number of elements) and then the content of the arrays. If the length of the array is fixed, the length encoding is omitted.

Chapter 7

Licensing

In Switzerland, software is copyright protected from the moment it is written down (see Art. 29 in [15]). To enable reuse of the library created in this project, a license has to be specified. The MIT license[1] was chosen, because it is very permissive and allows complete code reuse. The only requirement is that the original copyright text be included in derivative works. Since ETH may not be mentioned in student works [16], only the author is mentioned as copyright holder.

Listing 7.1: MIT license text.

```
Copyright <YEAR> <COPYRIGHT HOLDER>

Permission is hereby granted, free of charge, to any person
obtaining a copy of this software and associated documentation
files (the "Software"), to deal in the Software without
restriction, including without limitation the rights to use,
copy, modify, merge, publish, distribute, sublicense, and/or
sell copies of the Software, and to permit persons to whom the
Software is furnished to do so, subject to the following
conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
OTHER DEALINGS IN THE SOFTWARE.
```

[1] The name *MIT licence* is not an official name, but is commonly used to refer to this license.

Chapter 8

Conclusion

We outlined our process of the implementation of a PKW library, and provide two realizations of the scheme, a naïve PKW and one constructed from a PPRF and an AEAD scheme. In this process, we encountered a number of questions which had previously not surfaced, e.g. the serialization of keys and memory sanitization. Also, questions about software distribution were considered and a choice of a license was made.

We showed that, essentially, PKW can be implemented with realistic key and tag lengths. There remain a number of open questions, and more benchmarks could help in formulating more. The basic benchmarks which were performed indicate that the data structure in the PPRF + AEAD construction which stores the PPRF secret key merits further exploration. This would hopefully lead to a significant speedup of the puncturing operation. Different PPRF constructions may also be interesting to compare to the GGM construction used here. Possible applications, such as protected file storage (Chapter 6 in [1]), may also reveal more requirements, necessitating changes or additions to, and further solidifying the API.

Appendix A

Appendix

A.1 Additional benchmarks

Shown below are figures of size and timing benchmarks for a PPRF key with a tag size of 32 bits and a key size of 128 bits. Figure A.1 shows similar behavior to Figure 3.6 (tag size 64 bits) in Chapter 3.3.2.

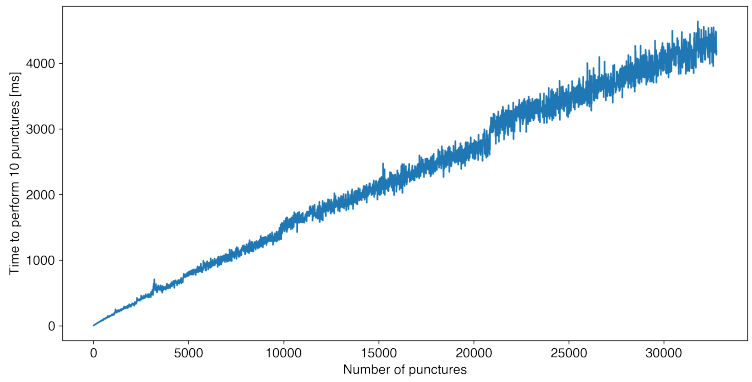

Figure A.1: Duration of 10 punctures in PPRF with tag size 32

The size of the key shows quasi-linear growth (but it is sub-linear, see Chapter 3.3.2) and the graph is very similar to the one for a tag size of 64 bits (Figure 3.3 in Chapter 3.3.2).

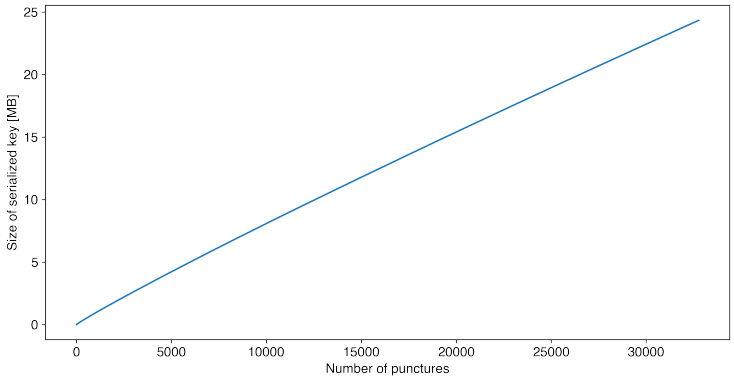

Figure A.2: Serialization size of PPRF with tag size 32

A.2 Demo of PKW API usage

We provide a demonstration file, showing the instantiation, serialization, deserialization and operations of the PKW scheme.

```
#include <iostream>
#include <pkw/PPRF_AEAD_PKW.h>
#include <pkw/exceptions.h>
#include <pkw/pkw.h>
void print(const std::vector<unsigned char> &v) {
    for (auto val: v)
        std::cout << val;
}

int main() {
    // Construct a new PKW abject with fresh randomness and
    tag-lenth 256, key-length 196
    PPRF_AEAD_PKW pkw(256, 196);

    // Define a header and a key to be wrapped
    std::vector<unsigned char> header({0, 2, 'a', 'b'});
    std::vector<unsigned char> key({
            's',
            'e',
            'n',
            's',
            1,
            't',
            1,
            'v',
            'e',
    });

    int tag = 12;
    // Wrap the key using the tag and the header
```

```
30      std::vector<unsigned char> wrapped_key = pkw.wrap(tag,
        header, key);
31
32      std::cout << "The ciphertext produced after wrapping key = ";
33      print(key);
34      std::cout << " with header = ";
35      print(header);
36      std::cout << " and tag = " << tag << ":" << std::endl;
37      print(wrapped_key);
38      std::cout << std::endl;
39
40      // Unwrapping with a different tag or header will lead to an
        error
41      try {
42          pkw.unwrap(11, header, wrapped_key);
43      } catch (PuncturableKeyWrappingException &e) {
44          std::cout << "Unwrapping with a wrong tag fails." <<
        std::endl;
45      }
46      try {
47          std::vector<unsigned char> other_header({0, 2, 'a', 'b',
        'c'});
48          pkw.unwrap(tag, other_header, wrapped_key);
49      } catch (UnwrappingException &e) {
50          std::cout << "Unwrapping with a wrong header fails." <<
        std::endl;
51      }
52
53      // Unwrap with the correct parameters:
54      std::vector<unsigned char> unwrapped_key = pkw.unwrap(tag,
        header, wrapped_key);
55      std::cout << "Unwrapping reveals the original key: ";
56      print(unwrapped_key);
57      std::cout << std::endl;
58
59      // Puncturing a tag
60      pkw.punc(tag);
61      std::cout << "Punctured on tag = " << tag << std::endl;
62
63      // Now wrapping and unwrapping using the tag will fail
64      try {
65          pkw.unwrap(tag, header, wrapped_key);
66      } catch (PuncturableKeyWrappingException &e) {
67          std::cout << "Unwrapping with a punctured tag fails." <<
        std::endl;
68      }
69      try {
70          pkw.wrap(tag, header, key);
71      } catch (PuncturableKeyWrappingException &e) {
72          std::cout << "Wrapping with a punctured tag fails." <<
        std::endl;
73      }
74
75      // To offload the key for storage it can be serialized
```

```
76      SecureByteBuffer serialized_pkw = pkw.serializeKey();
77
78      // It can also be protected by a password derived key
79      SecureByteBuffer serialized_encrypted_pkw =
        pkw.serializeAndEncryptKey("securepassword");
80
81      // Deserialization is handled by a factory, which constructs
        a shared pointer to the object
82      auto deserialized =
        PPRF_AEAD_PKW_Factory().fromSerialized(serialized_pkw);
83      auto deserialized_with_password =
        PPRF_AEAD_PKW_Factory().fromSerializedAndEncrypted(
84          serialized_encrypted_pkw,
85          "securepassword");
86
87      //  Wrapping and unwrapping using the tag will still fail
88      try {
89          deserialized->unwrap(tag, header, wrapped_key);
90      } catch (PuncturableKeyWrappingException &e) {
91          std::cout << "Unwrapping with a punctured tag fails
        after export and import." << std::endl;
92      }
93      try {
94          deserialized_with_password->wrap(tag, header, key);
95      } catch (PuncturableKeyWrappingException &e) {
96          std::cout << "Wrapping with a punctured tag fails after
        export and import." << std::endl;
97      }
98  }
```

Listing A.1: demo.cpp

Bibliography

[1] Matilda Backendal, Felix Günther, and Kenneth G. Paterson. Puncturable Key Wrapping and Its Applications. *Cryptology ePrint Archive*, 2022.

[2] Phillip Rogaway and Thomas Shrimpton. A provable-security treatment of the key-wrap problem. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 4004 LNCS:373–390, 2006.

[3] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal of the ACM (JACM)*, 33(4):792–807, Aug 1986.

[4] Phillip Rogaway. Authenticated-encryption with associated-data. *Proceedings of the ACM Conference on Computer and Communications Security*, pages 98–107, 2002.

[5] Elette Boyle, Shafi Goldwasser, and Ioana Ivan. Functional signatures and pseudorandom functions. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8383 LNCS:501–519, 2014.

[6] Dan Boneh and Brent Waters. Constrained pseudorandom functions and their applications. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8270 LNCS(PART 2):280–300, 2013.

[7] Aggelos Kiayias, Stavros Papadopoulos, Nikos Triandopoulos, and Thomas Zacharias. Delegatable pseudorandom functions and applications. *Proceedings of the ACM Conference on Computer and Communications Security*, pages 669–683, 2013.

[8] Zhaomo Yang, Brian Johannesmeyer, Anders Trier Olesen, Sorin Lerner, and Kirill Levchenko. Dead Store Elimination (Still) Considered Harm-

ful. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1025–1040, Vancouver, BC, Aug 2017. USENIX Association.

[9] Crypto++ Library 8.7 — Free C++ Class Library of Cryptographic Schemes. URL https://cryptopp.com/. Last accessed: 2022-11-10.

[10] Hugo Krawczyk. Cryptographic Extraction and Key Derivation: The HKDF Scheme. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2010.

[11] CMake. URL https://cmake.org/. Last accessed: 2022-11-17.

[12] Matthew Green and Matthew Smith. Developers are Not the Enemy!: The Need for Usable Security APIs. *IEEE Security and Privacy*, 14(5): 40–46, Sep 2016.

[13] memset, memset_explicit, memset_s - cppreference.com. URL https://en.cppreference.com/w/c/string/byte/memset. Last accessed: 2022-11-21.

[14] Boost C++ Libraries. URL https://www.boost.org/. Last accessed: 2022-11-05.

[15] Bundesgesetz über das Urheberrecht und verwandte Schutzrechte. 1992. URL https://www.fedlex.admin.ch/eli/cc/1993/1798{_}1798{_}1798/de.

[16] Copyright and Software — ETH Zurich, . URL https://ethz.ch/en/industry/researchers/licensing-software/copyright-ownership.html. Last accessed: 2022-11-10.

[17] MLS Secret Tree, 2022. URL https://www.ietf.org/id/draft-ietf-mls-protocol-16.html{#}name-secret-tree. Last accessed: 2022-07-21.

[18] RFC 2104: HMAC: Keyed-Hashing for Message Authentication. URL https://www.rfc-editor.org/rfc/rfc2104. Last accessed: 2022-08-27.

[19] SR 414.135.1 - Verordnung der ETH Zürich vom 22. Mai 2012 über Lerneinheiten und Leistungskontrollen an der ETH Zürich (Leistungskontrollenverordnung ETH Zürich), . URL https://www.fedlex.admin.ch/eli/cc/2012/446/de{#}a23. Last accessed: 2022-11-10.

[20] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. Binsec/Rel: Symbolic Binary Analyzer for Security with Applications to Constant-Time and Secret-Erasure. *ACM Transactions on Privacy and Security*, Sep 2022.

[21] Bjarne Stroustrup. A history of C++. *History of programming languages—II*, pages 699–769, Jan 1996.

[22] Matthew D. Green and Ian Miers. Forward secure asynchronous messaging from puncturable encryption. *Proceedings - IEEE Symposium on Security and Privacy*, 2015-July:305–320, Jul 2015.

[23] M J Dworkin. Recommendation for block cipher modes of operation :. 2007.

[24] Joel Reardon, David Basin, and Srdjan Capkun. SoK: Secure data deletion. *Proceedings - IEEE Symposium on Security and Privacy*, pages 301–315, 2013.

[25] Nimrod Aviram, Kai Gellert, and Tibor Jager. Session resumption protocols and efficient forward security for TLS 1.3 0-RTT. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11477 LNCS:117–150, 2019.

[26] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7533 LNCS:159–176, 2012.