Forward-secure XMSS based on RFC 8391 – Documentation *

Andreas Hülsing Matthias Kannwischer Peter Schwabe

Mar 05, 2020

XMSS is a stateful hash-based signature scheme described in RFC 8391. The supplied archive contains an implementation of a forward-secure version of XMSS. This forward-secure version is based on the original XMSS reference implementation. The difference is limited to the way the one-time signature keys are computed. RFC 8391 only limits the used methods for this task to methods that achieve the same security level as the rest of the construction. The method used in the provided implementation achieves the necessary security level and is thereby compliant with RFC 8391.

In the following we briefly describe system requirements, the API of the delivered code, parameters for the use case of IOHK, and security considerations.

1 System requirements

The provided software was developed and tested on different Linux distributions, including Ubuntu 19.10 and Arch Linux using gcc 9.2.1. It comes with a Makefile which compiles some examples in the `test/` directory that demonstrate the usage of the API that is described in the following.

The only dependency is libcrypto (OpenSSL) for the SHA-2 hash functions (i.e., SHA-256 and SHA-512).

2 API

The API is defined through the following set of data types and functions:

2.0.1 `xmss_params`.

This type is used to capture all relevant parameters of one XMSS or XMSSMT parameter set. From a user's perspective what is most important is that it contains information about how much space needs to be allocated for keys and signed messages. The type is defined in the header file `params.h`.

2.1 Functions for (single-tree) XMSS

2.1.1 `xmss_str_to_oid`.

This function takes an XMSS parameter string and maps it to a so-called OID, i.e., a compact 32-bit integer ID that can be efficiently stored and passed around: The function is defined in the header file `params.h` as follows:

```
int xmss_str_to_oid(uint32_t *oid, const char *s);
```

The function takes the following arguments:

`uint32_t *oid`: pointer to the location the computed OID (of type `uint32_t`) is written to.

*This software was developed in collaboration with and supported by IOHK (`https://iohk.io`).

`const char *s`: XMSS parameter string, for example `"XMSS-SHA2_10_256"`. For a list of supported parameter strings see the implementation of `xmss_str_to_oid` in the file `params.c`.

The function returns `0` on success or `-1` if an invalid parameter string is provided.

2.1.2 xmss_parse_oid.

This function takes an OID as input and computes the corresponding parameters in a result of type `xmss_params`. The function is defined in the header file `params.h` as follows:

```
int xmss_parse_oid(xmss_params *params, const uint32_t oid);
```

The function takes the following arguments:

`xmss_params *params`: Pointer to the variable that the parameters are written to.

`const uint32_t oid`: A valid XMSS OID, which is typically first set by invoking the function `xmss_str_to_oid(&oid, xmss_str)`, where `xmss_str` is a valid XMSS parameter string (see above).

The function returns `0` on success or `-1` if an invalid OID is provided.

2.1.3 xmss_keypair.

This function takes as input an OID and computes an XMSS keypair. The function is defined in the header file `xmss.h` as follows:

```
int xmss_keypair(unsigned char *pk, unsigned char *sk, const uint32_t oid);
```

The function takes the following arguments:

`unsigned char *pk`: Pointer to the location that the public key is written to. The caller needs to ensure that at least `XMSS_OID_LEN + params.pk_bytes` bytes are allocated at the location that `pk` points to. The value `XMSS_OID_LEN` is defined in the header file `params.h`; the variable `params` of type `xmss_params` needs to be set by invoking `xmss_parse_oid(&params, oid)` before calling `xmss_keypair`.

`unsigned char *sk`: Pointer to the location that the secret key is written to. The caller needs to ensure that at least `XMSS_OID_LEN + params.sk_bytes` bytes are allocated at the location that `sk` points to. The value `XMSS_OID_LEN` is defined in the header file `params.h`; the variable `params` of type `xmss_params` needs to be set by invoking `xmss_parse_oid(&params, oid)` before calling `xmss_keypair`.

`const uint32_t oid`: A valid XMSS OID, which is typically first set by invoking the function `xmss_str_to_oid(&oid, xmss_str)`, where `xmss_str` is a valid XMSS parameter string (see above).

The function returns `0` on success or negative values for failure (for example, if no valid OID is provided).

2.1.4 xmss_sign.

This function receives as input a message and a secret key and computes a signed message and updates the secret key. The function is defined in the header file `xmss.h` as follows:

```
int xmss_sign(unsigned char *sk,
              unsigned char *sm, unsigned long long *smlen,
              const unsigned char *m, unsigned long long mlen);
```

The function takes the following arguments:

`unsigned char *sk`: Pointer to the location that the secret key is read from and written to. This secret key needs to be previously set by `xmss_keypair` and possibly updated by calls to `xmss_sign`.

`unsigned char *sm`: Pointer to the location that the signed message will be written to. The caller needs to ensure that at least `mlen + params.sig_bytes` bytes are allocated at the location that `sm` points to. The variable `params` of type `xmss_params` needs to be set by invoking `xmss_parse_oid(&params, oid)` before calling `xmss_keypair`.

`unsigned long long *smlen`: Pointer to the location that the length of the signed message (of type `unsigned long long`) will be written to.

`const unsigned char *m`: Pointer to the location, where the message is stored.

`unsigned long long mlen`: Length of the message `m`.

The function returns `0` on success or negative values for failure (for example, if no valid OID is provided). In particular, it returns `-2` if the secret key can no longer be used to compute signatures, because it has run out of one-time-signature keys.

2.1.5 `xmss_sign_open`.

This function receives as input a signed message and verifies the signature. If verification is successful the message is written to an output buffer. The function is defined in the header file `xmss.h` as follows:

```
int xmss_sign_open(unsigned char *m, unsigned long long *mlen,
                   const unsigned char *sm, unsigned long long smlen,
                   const unsigned char *pk);
```

The function takes the following arguments:

`unsigned char *m`: Pointer to the location that the message will be written to if verification is successful. The caller needs to ensure that at least `smlen` bytes are allocated at the location that `m` points to.

`unsigned long long mlen`: Pointer to the location that the length of the message (of type `unsigned long long`) will be written to.

`const unsigned char *sm`: Pointer to the location, where the signed message is stored.

`unsigned long long smlen`: Length of the signed message `sm`.

`const unsigned char *pk`: Pointer to the location where the public key is stored. This public key needs to be previously set by `xmss_keypair`.

The function returns `0` on success or `-1` on any failure including failed signature verification due to an invalid signature.

2.2 Functions for XMSSMT

2.2.1 `xmssmt_str_to_oid`.

This function takes an XMSSMT parameter string and maps it to a so-called OID, i.e., a compact 32-bit integer ID that can be efficiently stored and passed around: The function is defined in the header file `params.h` as follows:

```
int xmssmt_str_to_oid(uint32_t *oid, const char *s);
```

The function takes the following arguments:

`uint32_t *oid`: pointer to the location the computed OID (of type `uint32_t`) is written to.

`const char *s`: XMSSMT parameter string, for example `"XMSSMT-SHA2_22/2_192"`. For a list of supported parameter strings see the implementation of `xmssmt_str_to_oid` in the file `params.c`.

The function returns `0` on success or `-1` if an invalid parameter string is provided.

2.2.2 `xmssmt_parse_oid`.

This function takes an OID as input and computes the corresponding parameters in a result of type `xmss_params`. The function is defined in the header file `params.h` as follows:

```
int xmssmt_parse_oid(xmss_params *params, const uint32_t oid);
```

The function takes the following arguments:

`xmss_params *params`: Pointer to the variable that the parameters are written to.

`const uint32_t oid`: A valid XMSSMT OID, which is typically first set by invoking the function `xmssmt_str_to_oid(&oid, xmssmt_str)`, where `xmssmt_str` is a valid XMSSMT parameter string (see above).

The function returns `0` on success or `-1` if an invalid OID is provided.

2.2.3 `xmssmt_keypair`.

This function takes as input an OID and computes an XMSSMT keypair. The function is defined in the header file `xmss.h` as follows:

```
int xmssmt_keypair(unsigned char *pk, unsigned char *sk, const uint32_t oid);
```

The function takes the following arguments:

`unsigned char *pk`: Pointer to the location that the public key is written to. The caller needs to ensure that at least `XMSS_OID_LEN + params.pk_bytes` bytes are allocated at the location that `pk` points to. The value `XMSS_OID_LEN` is defined in the header file `params.h`; the variable `params` of type `xmss_params` needs to be set by invoking `xmssmt_parse_oid(&params, oid)` before calling `xmssmt_keypair`.

`unsigned char *sk`: Pointer to the location that the secret key is written to. The caller needs to ensure that at least `XMSS_OID_LEN + params.sk_bytes` bytes are allocated at the location that `sk` points to. The value `XMSS_OID_LEN` is defined in the header file `params.h`; the variable `params` of type `xmss_params` needs to be set by invoking `xmssmt_parse_oid(&params, oid)` before calling `xmssmt_keypair`.

`const uint32_t oid`: A valid XMSSMT OID, which is typically first set by invoking the function `xmssmt_str_to_oid(&oid, xmssmt_str)`, where `xmssmt_str` is a valid XMSSMT parameter string (see above).

The function returns `0` on success or negative values for failure (for example, if no valid OID is provided).

2.2.4 `xmssmt_sign`.

This function receives as input a message and a secret key and computes a signed message and updates the secret key. The function is defined in the header file `xmss.h` as follows:

```
int xmssmt_sign(unsigned char *sk,
                unsigned char *sm, unsigned long long *smlen,
                const unsigned char *m, unsigned long long mlen);
```

The function takes the following arguments:

`unsigned char *sk`: Pointer to the location that the secret key is read from and written to. This secret key needs to be previously set by `xmssmt_keypair` and possibly updated by calls to `xmssmt_sign`.

`unsigned char *sm`: Pointer to the location that the signed message will be written to. The caller needs to ensure that at least `mlen + params.sig_bytes` bytes are allocated at the location that `sm` points to. The variable `params` of type `xmss_params` needs to be set by invoking `xmssmt_parse_oid(&params, oid)` before calling `xmssmt_keypair`.

`unsigned long long *smlen`: Pointer to the location that the length of the signed message (of type `unsigned long long`) will be written to.

`const unsigned char *m`: Pointer to the location, where the message is stored.

`unsigned long long mlen`: Length of the message `m`.

The function returns `0` on success or negative values for failure (for example, if no valid OID is provided). In particular, it returns `-2` if the secret key can no longer be used to compute signatures, because it has run out of one-time-signature keys.

2.2.5 `xmssmt_sign_open`.

This function receives as input a signed message and verifies the signature. If verification is successful the message is written to an output buffer. The function is defined in the header file `xmss.h` as follows:

```
int xmssmt_sign_open(unsigned char *m, unsigned long long *mlen,
                     const unsigned char *sm, unsigned long long smlen,
                     const unsigned char *pk);
```

The function takes the following arguments:

`unsigned char *m`: Pointer to the location that the message will be written to if verification is successful. The caller needs to ensure that at least `smlen` bytes are allocated at the location that `m` points to.

`unsigned long long mlen`: Pointer to the location that the length of the message (of type `unsigned long long`) will be written to.

`const unsigned char *sm`: Pointer to the location, where the signed message is stored.

`unsigned long long smlen`: Length of the signed message `sm`.

`const unsigned char *pk`: Pointer to the location where the public key is stored. This public key needs to be previously set by `xmssmt_keypair`.

The function returns `0` on success or `-1` on any failure including failed signature verification due to an invalid signature.

3 Parameters for IOHK use case

As discussed, the specific use case of IOHK requires XMSSMT with 2 layers, each of height 11 (i.e., a total tree height of 22) and security parameter $n = 192$. This parameter set is supported through the parameter string `XMSSMT-SHA2_22/2_192`. As a consequence, any code invoking key-generation, signing, or verification, needs to start by setting parameters as follows:

```
xmss_params params;
uint32_t oid;

xmssmt_str_to_oid(&oid, "XMSSMT-SHA2_22/2_192");
xmssmt_parse_oid(&params, oid);
```

This sets the variables `params` (which can be used to retrieve all relevant information for allocating space for keys and signed messages) and `oid`, which can be passed to `xmss_keypair` or `xmssmt_keypair` to generate a key pair.

4 Security considerations

The delivered software uses the following forward-secure pseudo-random number generator to derive a forward-secure version of XMSS:

$$\mathrm{OUT}_i = \mathrm{hash}(\mathrm{ROOT}\|\mathrm{SEED}_i\|\mathrm{OTS\_ADDR}\|0)$$

$$\mathrm{SEED}_{i+1} = \mathrm{hash}(\mathrm{ROOT}\|\mathrm{SEED}_i\|\mathrm{OTS\_ADDR}\|1)$$

This is the forward secure PRG construction from the original XMSS paper [BDH11] with multi-target protection added. Using an n-bit hash function and n-bit seeds, this function achieves n-bit security.

It must be noted that all security considerations of RFC 8391 apply. Especially the warning about the stateful nature of the system:

```
In contrast to traditional signature schemes, the signature schemes
described in this document are stateful, meaning the secret key
changes over time.  If a secret key state is used twice, no
cryptographic security guarantees remain.  In consequence, it becomes
feasible to forge a signature on a new message.  This is a new
property that most developers will not be familiar with and requires
careful handling of secret keys.  Developers should not use the
schemes described here except in systems that prevent the reuse of
secret key states.

Note that the fact that the schemes described in this document are
stateful also implies that classical APIs for digital signatures
cannot be used without modification.  The API MUST be able to handle
a secret key state; in particular, this means that the API MUST allow
to return an updated secret key state.
```

Even more, to achieve forward-security it is not sufficient to guarantee that a secret key state is not reused. To achieve forward-security **it must be guaranteed that all copies of old secret key states are deleted as soon as a new secret key state is available**. Forward-security is only guaranteed up to the index of the oldest recoverable key state at the time of the key compromise. Consequently, special measures have to be taken to ensure that old key states do not remain on disk while, for example, only their file-system index gets deleted. How secure erasure of key states can be implemented is system dependent and beyond the scope of this documentation.

References

[BDH11] Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. XMSS - a practical forward secure signature scheme based on minimal security assumptions. In Bo-Yin Yang, editor, *Post-Quantum Cryptography*, volume 7071 of *LNCS*, pages 117–129. Springer Berlin Heidelberg, 2011.