Federal Office
for Information Security

A study commissioned by the Federal Office for Information Security (BSI)

Project 197

Secure Implementation of a Universal Crypto Library

Cryptographic Documentation

Version 1.2.2 / 2017-04-05
Botan 2.0.1-RSCS1

Summary
The objective of this project is the secure implementation of a universal crypto library which contains all common cryptographic primitives that are necessary for the wide use of cryptographic operations. These include symmetric and asymmetric encryption and signature methods, PRFs, hash functions and RNGs. Additionally, security standards such as X.509 and SSL/TLS have to be supported. The library will be provided to manufacturers of VS-NfD products which will help the Federal Office for Information Security (BSI) to evaluate these products.

This document describes the cryptographic implementations of Botan.

Authors
René Korthaus (RK)
Juraj Somorovsky (JSo)
Tobias Niemann (TN)

Secure Implementation of a Universal Crypto Library

Cryptographic Documentation

Botan 2.0.1-RSCS1

Changelog

Version	Authors	Comment	Date
1.0.0	JSo, RK, TN	Initial version	2016-11-19
1.1.0	JSo, RK, TN	Add introduction section Move all chapters one level up Replace top level chapter “Certificates” by “X.509 Path Validation” Add introductary text and subsections to RNG chapter Fix wrong and add missing paths to source files in RNG chapters Use full path to source files including src/ in all chapters SP800-90A refers to entry in bibliography Update SP800-90A bibliography entry to revision 1 Add example for keyed hash function to HMAC_DRBG Add note on platform availability of System_RNG Add description of HMAC_DRBG constructors Rework HMAC_DRBG section Rework X.509 path validation section Add chapter on entropy sources	2017-01-09
1.2.0	RK	Add PKCS11_RNG Add note on requirements for seeding a DRBG Remove System_RNG section heading Added a note on the security level of HMAC_DRBG Update DH group generation Update EC public key checks Update EC blinding	2017-03-02
1.2.1	RK	Correct RSA blinding operation Add description for HMAC_DRBG function security_level()	2017-03-09
1.2.2	RK	Correct description and add default values for function random_prime()	2017-04-05

Table of Contents

1 Introduction

This document describes the structure and implementation of cryptographic algorithms used in Botan. Our description follows [TR021021,TR021022] and analyzes the Botan implementation regarding BSI recommendations. If the implementation steps differ from the standard, conclusion about the standard compliance is provided.

The following chapters cover hash functions, symmetric encryption algorithms, message authentication codes, prime number generation, parameter generation for public key algorithms, key generation for public key algorithms, asymmetric encryption and key exchange schemes, signatures, random number generators, entropy sources, X.509 path validation and key derivation functions.

2 Hash Functions

The BSI documents [TR021021,TR021022] recommend usage of SHA-2 and SHA-3 algorithms in cryptographic software. In addition, SHA-1 is required for TLS versions up to 1.1.

SHA-1 and SHA-2 hash functions rely on the Merkle-Damgard construction. The general functionality of this construction is implemented in class `MDx_HashFunction` (located in `src/lib/hash/mdx_hash/mdx_hash.cpp`). This class handles splitting data into appropriate blocks and invocation of hash compression methods.

2.1 SHA-1

Botan provides two SHA-1 implementations: software and SSE2 (Streaming SIMD Extensions 2). Both versions implement SHA-1 as described in [NIST-SHS].

The software implementation is located in `src/lib/hash/sha1/sha160.cpp`. It uses the Botan structure `secure_vector` to implement data handling and compressions.

The SSE2 implementation is in `src/lib/hash/sha1_sse2/sha_sse2.cpp`. It is based on the code by Dean Gaudet[1].

2.2 SHA-2

SHA-2 hash functions are implemented in `src/lib/hash/sha2_64/sha2_64.cpp` and in `src/lib/hash/sha2_32/sha2_32.cpp`.

`sha2_32.cpp` implements the SHA-224 and SHA-256 hash functions. `sha2_64.cpp` implements SHA-384 and SHA-512. The description of these algorithms is provided in [NIST-SHS].

2.3 SHA-3

Botan implements the SHA-3 hash function in `src/lib/hash/sha3/sha3.cpp`. The implementation follows the standard [FIPS-202] and thus supports output lengths of 224, 256, 384 and 512 bits. SHA-3 does not rely on a Merkle-Damgard construction, as it uses a sponge construction to perform data compression.

1 http://arctic.org/~dean/crypto/sha1.html

3 Symmetric Encryption

In the following we describe symmetric encryption algorithms and schemes. We first present the implementation of AES in Botan. Afterwards, we give an overview of modes of operations and padding schemes used in symmetric encryption algorithms.

3.1 AES Block Cipher

Botan provides three AES implementations: software, assembly and hardware (AES-NI).

The software implementation (located in `src/lib/block/aes/aes.cpp`) is based on table lookups which are known to be vulnerable to cache and timing side channel attacks. The maintainers suggest to use assembly or hardware AES implementations in security products.

The assembly implementation is based on the Supplemental Streaming SIMD Extensions 3 (SSSE3) instruction set created by Intel. The implementation is based on the code of Mike Hamburg, which provides protection against cache and timing side channel attacks [AES-SSSE3]. The code is located in `src/lib/block/aes_ssse3/aes_ssse3.cpp`.

The AES-NI implementation provides an interface to the AES-NI instruction set. The code is in `src/lib/block/aes_ni/aes_ni.cpp`.

An application developer can enable and disable a specific implementation at compile time by using macros (BOTAN_HAS_AES, BOTAN_HAS_AES_NI, BOTAN_HAS_AES_SSSE3). All three AES implementations are enabled per default. Depending on the availability on the used processor, Botan first attempts to use AES-NI. If not available, Botan uses AES-SSSE3 or the software implementation. The code invoking a specific AES implementation is in `src/lib/block/aes/aes.cpp`.

3.2 AES-GCM

AES-GCM is an authenticated encryption scheme, which combines AES Counter mode with authentication over Galois fields. Botan implements AES-GCM according to the NIST specification [GCM].

The AES-GCM functionality is implemented in classes `GCM_Encryption` and `GCM_Decryption` (located in `src/lib/modes/aead/gcm/gcm.cpp`). These classes offer the following public methods, which are used by a developer when working with AES-GCM:

- `set_key(key)`: It initializes AES-GCM encryption / decryption with a symmetric key. The key length depends on the underlying AES block cipher size.
- `set_associated_data(ad)`: It performs a GHASH computation over this data.
- `start(nonce)`: It initializes the AES-GCM computation and the underlying Counter mode with the provided nonce. It encrypts the zeroth counter value, which is later used to compute the authentication tag.
- `process(buffer)`: It takes the buffer value, encrypts it in the counter mode and updates the GHASH.

- `finish(buffer)`: It finalizes the counter mode encryption and GHASH computation. It creates an authentication tag.

Remark 1: It is forbidden to re-use the same initialization vectors (nonces) with the same AES-GCM key. Otherwise, the attacker could break authenticity of the constructed ciphertext [GCM-FA,GCM-ND]. It is up to the application developer to choose the nonces properly.

Remark 2: AES-GCM specification prescribes the maximum length of the message to be encrypted to $(2^{32} - 1)$ blocks. Botan does not check the plaintext length explicitly. It is currently up to the application developer to choose correct data lengths.

Remark 3: Botan implements AES-GCM cipher suites in TLS. When encrypting TLS records, Botan sets the nonce value to zero and increments the nonce value with each new record. This effectively prevents nonce reuse attacks [GCM-ND].

Remark 4: We refer to [TR021021] for further security considerations on AES-GCM.

3.3 AES-CBC

AES-CBC [CBC] is implemented in classes `CBC_Encryption` and `CBC_Decryption` (located in `src/lib/modes/cbc/cbc.cpp`). The constructors of these classes offer usage of different padding schemes. When using AES-CBC, the AES cipher has to be provided as a parameter.

The following public methods are used by a developer when working with AES-CBC:

- `set_key(key)`: It initializes AES-CBC encryption / decryption with a symmetric key.
- `start(nonce)`: It initializes the AES-CBC computation with the provided nonce.
- `process(buffer)`: It takes the buffer value, encrypts / decrypts it in the CBC mode, and puts the result into the buffer.
- `finish(buffer)`: It finalizes the CBC encryption / decryption process, and puts the result into the buffer.

Remark 5: AES-CBC does not provide authentication. Generated ciphertexts must be protected by MACs or signatures.

Remark 6: The developer must always use fresh unpredictable initialization vectors.

Remark 7: We refer to [TR021021] for further security considerations on AES-CBC.

3.4 XTS

The XEX-based tweaked-codebook mode with ciphertext stealing is a block cipher mode of operation. [TR021021] does not cover the XTS mode. Nevertheless, it mentions XTS to have good efficiency and security properties for raw storage media encryption. Referring to [NIST SP-800-38E] it should be avoided in other scenarios such as transit data encryption. In addition, it is recommended that the length of the ciphertext, protected with the same key should not exceed the length of 2^{20} cipher blocks. Botan implements XTS in `src/lib/modes/xts/xts.cpp` according to [IEEE P1619]. The following functions are available:

- `XTS_Mode(cipher)`: Constructs a XTS_Mode object with the passed block cipher. Only

the block sizes 64 and 128 bit are supported.

- `key_schedule(key, key length)`: Splits the passed key in half and sets the cipher and the tweak key. If the key length is odd or the underlying cipher does not support a key with length $\frac{key}{2}$, the function throws an error.
- `start_msg(nonce, nonce length)`: Sets nonce as input of tweak computation and compute initial tweak as $E_{k_2}(nonce)$.
- `process(buffer, buffer length)`: Processes the data from the passed buffer. Note that the function is only able to processes full plaintext blocks.
- `finish(buffer)`: Finalizes the data processing,

3.5 Padding Schemes

Botan implements the following block cipher padding schemes (see `src/lib/modes/mode_pad/mode_pad.cpp`):

- PKCS#7 [RFC5652]: The last byte in the padded block defines the padding length `p`, the remaining padding bytes are set to `p` as well.
- ANSI X9.23: The last byte in the padded block defines the padding length, the remaining padding is filled with 0x00.
- ISO/IEC 7816-4: The first padding byte is set to 0x80, the remaining padding bytes are set to 0x00.
- ESP [RFC4304]: The first padding byte is set to 0x01, the remaining padding bytes each increase by one.
- Null: No padding.

Remark 8: By processing a decrypted message the padding is validated in constant time. If the padding is invalid, Botan sets the padding length to 0. This is a countermeasure against side channel attacks. However, in specific cases this countermeasure is not sufficient and padding oracle attacks can be mounted [Lucky13]. The application developer is thus responsible for a proper design of his application: the application has to validate message authenticity before it is decrypted.

Remark 9: The TLS implementation introduces a constant time CBC unpadding functionality to prevent the Lucky 13 attack [Lucky13]. This can be found in `src/lib/tls/tls_cbc/tls_cbc.cpp`.

4 Message Authentication Codes

The next section examines the Message Authentication Codes recommended in [TR021021]: CMAC, HMAC and GMAC. All three algorithms are implemented in Botan.

4.1 CMAC

The Cipher-based Message Authentication Code or One-Key MAC (OMAC1) algorithm uses a block cipher to compute an authentication code. The CMAC output length equals the block size of the used block cipher. Botan implements the algorithm according to [NIST-OMAC]. The class CMAC is in `src/lib/mac/cmac/cmac.cpp` and offers the following relevant functions:

- `CMAC(block cipher)`: Constructs a CMAC object with the passed block cipher. Only block ciphers with a block size of 64, 128, 256 or 512 bits are supported.
- `clear()`: Clears all sensitive data of the CMAC object.
- `key_schedule(key, key length)`: Derives the subkeys k_1 and k_2 from $E_k(0)$, where $E_k()$ is the encryption function of the specified block cipher.
- `poly_double(input)`: Computes the multiplication $input * x$ in $GF(2^n)$. This calculation is used to derive the subkeys.
- `add_data(data, data length)`: Computes the CMAC over the passed data.
- `final_result(buffer)`: Finalizes the MAC computation, pads the last message block with $10...0_2$ if necessary, XORs the respective subkey and writes the calculated MAC to the passed buffer.

4.2 HMAC

The Keyed-Hashing for Message Authentication algorithm uses an underlying cryptographic hash function to compute an authentication code. The HMAC output length equals the output length of the underlying hash function. Botan implements the HMAC algorithm [RFC2104] in `src/lib/mac/hmac/hmac.cpp`. The following functions of the class HMAC are provided:

- `HMAC(hash function)`: Constructs an HMAC object with the passed cryptographic hash function.
- `clear()`: Clears all sensitive data of the HMAC object.
- `key_schedule(key, key length)`: Creates ipad and opad with the length of hash output, compresses the passed key with the specified hash function if its length exceeds the hash function's block size, and computes $key \oplus ipad$, $key \oplus opad$.
- `add_data(data, data length)`: Adds data to inner hash input.
- `final_result(buffer)`: Computes inner hash and subsequently the outer hash. Writes output of the outer hash function to the passed buffer.

4.3 GMAC

GMAC combines counter mode of operation with a Galois field computation. Botan implements GMAC according to the NIST specification [GCM].

The GMAC functionality is implemented in the class `GMAC` (located in `src/lib/mac/gmac/gmac.cpp`). This class offers the following public methods and constructors, which are used by a developer when working with GMAC:

- `GMAC(BlockCipher* cipher)`: Constructs a GMAC object with the underlying block cipher.
- `set_key(key)`: It initializes GMAC computation with a symmetric key. The key length depends on the underlying block cipher size.
- `start_msg(nonce)`: It initializes the GMAC computation with the provided nonce. It encrypts a block of zero bytes, which is later used as an input into the GHASH computation.
- `add_data(buffer)`: It takes the buffer value and updates the GHASH.
- `final_result(mac)`: It finalizes the GHASH computation and creates an authentication tag. It fills the provided mac parameter array with the authentication tag data.

Remark 10: It is forbidden to re-use the same initialization vectors (nonces) with the same GMAC key. Otherwise, the attacker could break the authenticity of the constructed ciphertext [GCM-FA,GCM-ND]. It is up to the application developer to choose the nonces properly. Botan ensures that the developer sets the nonce before each new GMAC computation.

Remark 11: GMAC is generally used in AES-GCM. For different encryption mechanisms HMAC and CMAC should be used in favor of GMAC.

5 Prime Number Generation

Asymmetric encryption, key exchange or signature algorithms like RSA and Diffie-Hellman require long random primes. Botan offers two prime number generation functions. Directive [TR021021] recommends the usage of the probabilistic Miller-Rabin primality test, which is used in both functions. The functions are part of `src/lib/math/numbertheory/make_prm.cpp`.

1. `random_prime(RandomNumberGenerator& rng, size_t bits, const BigInt& coprime, size_t equiv, size_t modulo)`
2. `random_safe_prime(RandomNumberGenerator& rng, size_t bits)`

The function `random_prime()` operates as follows:

Input:

- `bits:` Bit length of the prime
- `rng:` Random number generator
- `coprime:` Number to which `p-1` shall be coprime to. If this parameter is not given, it defaults to 1.
- `equiv, modulo:` Number the prime shall be equivalent to (if not given, defaults to 1), using modulus `modulo` (if not given, defaults to 2).

Output:

- `p:` prime of `bits` length sampled from the random number generator passed with `rng` , where `p-1` is coprime to `coprime` and equivalent to `equiv` modulus `modulo`.

Steps:

1. Preliminary parameter requirement checks are conducted. The algorithm requires an even `modulo`, a positive `coprime` and an `equiv` smaller than `modulo`. Furthermore, the algorithm terminates if a length of 1 is passed, as no primes with that bit-length exist.
2. For lengths up to 4 bits only 2 different primes exist for each length. In this case, the algorithm samples one byte from the passed random number generator and returns one of 2 possible primes based on the parity of the sampled random byte.
   - 2 bit length: 2,3
   - 3 bit length: 5,7
   - 4 bit length: 11,13
3. The algorithm retrieves a random number candidate `p` of the passed length from the specified random number generator. Subsequently the 2 highest and the lowest bit are set. Therefore, each candidate is odd and the multiplication of 2 candidates with the same bit length results in a doubling of the bit length. This is especially helpful when generating an RSA key pair.
4. The function ensures that the candidate `p` is equivalent to `equiv` modulus `modulo`, by

optionally adding $(modulus - p \bmod modulus) + \text{equiv}$ to it. Note that `equiv` defaults to 1 and `modulo` to 2. In that case the condition is always met as the candidate is odd.

5. To eliminate non-prime candidates, two primality and one additional test are conducted consecutively. If the candidate fails one of the tests, it is incremented by `modulo` to preserve the equivalence modulus `modulo` and tested again. Note that the candidate is incremented by `modulo` a single time prior to the first test. Thus, the sampled candidate `p` itself is never checked. If a candidate fails 4095 times, go to step 3.

   - Divisibility by the first $\lfloor \frac{bits}{2} \rfloor$ of 6541 included precomputed primes q_i (without 2).

     Therefore, the algorithm checks if the equation $p \bmod q_i = 0$ applies for one of the primes q_i . If that is the case, the candidate is composite and thus not prime.

   - Checks if equation $gcd(p-1, coprime) = 1$ applies. This measure very likely aims at preventing cycle attacks on RSA (which are not practical when long RSA primes are used) by limiting the number of fixpoints $m^e = m \bmod N$, when passing the RSA exponent `e` as `coprime`. As `coprime` defaults to 1, this condition is always fulfilled, if `coprime` is not passed.

   - Conducts Miller-Rabin primality test with 65 test iterations (function `is_prime()` in `src/lib/math/numbertheory/numthry.cpp`).

The function `random_safe_prime()` generates a safe prime:

Input:

- `bits`: bit length of the prime
- `rng`: random number generator

Output:

- `p`: safe prime of form $p = 2 * q + 1$ (q is prime) with `bits` length sampled from the random number generator `rng`.

Steps:

1. Call random_prime() to sample a prime `q` of $bits - 1$ length.
2. Compute candidate `p` as $q * 2 + 1$
3. Check `p` with Miller-Rabin primality test with 65 test iterations. If the candidate fails the test, go to step 1.

Remark 12: The second highest bit is set in step 3 of the `random_prime()` algorithm. This reduces the complexity of guessing the generated prime by 50%. However, this is a standard procedure by generating prime numbers and does not influence the security if proper key sizes are used.

Conclusion: The prime number generation algorithm `random_prime()`, implemented in Botan, complies with the recommendations in [TR021021]. As the algorithm performs more than 50

iterations of the Miller-Rabin primality test, the generated primes are suitable for usage in long-term keys.

6 Parameter Generation for Public Key Algorithms

In the following we describe the parameter generation for DH and DSA. Generation of parameters for elliptic curve algorithms is not included and only usage of named curves or curves provided by a trusted authority is recommended.

6.1 Random Number Generation for Probabilistic Public Key Algorithms

The algorithms analyzed in this section and the following sections often require random number generation from a specific range. Typical random number generators only generate numbers from a range $r \in \{0, \ldots, 2^n - 1\}$, where `n` is the maximum bit size of a generated number. This is not suitable, for example, for the generation of random parameters for DSA signatures.

In order to generate integers from arbitrary range, Botan uses the `random_integer()` method, which is used by the implemented public key algorithms. It works as follows:

Input:

- `rng`: random number generator
- `min`: minimum integer value
- `max`: maximum integer value

Output:

- `r: min < r < max`

Steps:

1. Retrieve the bit length `n` of the `max` value
2. Use `rng` to generate $r \in \{0, \ldots, 2^n - 1\}$
3. `if (min < r < max): return r`
4. Go to step 2

Conclusion: The algorithm equals to the first technique described in Section B.4 in [TR021021]. The algorithm thus complies with the recommendations in [TR021021].

6.2 DH/DSA

These algorithms require both parties to agree on a finite multiplicative cyclic group $(\mathbf{Z}/p\mathbf{Z})^*$, a generator `g` of the group or a respective subgroup. These parameters are called DH parameters and are typically precomputed, as the generation process is very resource intensive. Botan implements a generic discrete logarithm group class in `src/lib/pubkey/dl_group/dl_group.cpp`. The class DL_Group offers the constructor `DL_Group(RandomNumberGenerator& rng,PrimeType type, size_t pbits, size_t qbits = 0)`, which can be used to generate DH parameters. Alternatively, Botan offers several precomputed IETF standardized discrete logarithm groups via the constructor `DL_Group::DL_Group(const`

`std::string& name)`. The DL_Group generating constructor operates as follows:

Input:

- `rng`: random number generator
- `pbits`: bit length of the prime p
- `qbits`: bit length of the prime q. q is the order of the subgroup.
- `type`: defines characteristics of the prime p and the generation process. Can be either Strong, Prime_Subgroup or DSA_Kosherizer

Output:

- DL_Group ($\mathbf{Z}/p\mathbf{Z}$)* and a generator of the subgroup of order q.

In any case, the algorithm initially checks if the passed value `pbits` is bigger than 1023. If that is not the case, the generation process is terminated. Thus, only groups with a prime longer than 1023 bits can be generated with Botan.

If the function is called with `type=Strong`, the algorithm performs the following steps:

Steps:

1. Sample a random safe prime `p` of length `pbits` from `rng` with the function `random_safe_prime()`.
2. Set generator `g` to 2. Since `p` is a safe prime, the order (`p`-1) of ($\mathbf{Z}/p\mathbf{Z}$)* has the prime factors $q=\frac{p-1}{2}$ and 2. Thus the selected generator has an order of `q` or $2*q=p-1$. Note that the passed parameter `qbits` is completely ignored, as the length of `q` is always `pbits`-1.
3. Verify that the selected generator `g` is a quadratic residue modulo `p` and subsequently has order `q`. That is the case if the following equation applies $\left(\frac{g}{p}\right)=g^{\frac{p-1}{2}} \bmod p=g^q \bmod p=1$ ($\left(\frac{g}{p}\right)$ is the Legendre symbol). If the equation does not apply, the next generator candidate is selected from the first 6541 precomputed primes q_i (without 2) and the algorithm continues at 3. This effectively prevents small subgroup attacks.

If the function is called with `type=Prime_Subgroup`, the algorithm operates differently and without the need for safe primes:

Steps:

1. If a `qbits` value of 0 is passed, the algorithm evaluates the bit length of `q` according to the estimated difficulty of the discrete logarithm problem as $2*\log_2\left(e^{1.92*\sqrt[3]{\ln(2^{pbits})*\ln(\ln(2^{pbits}))^2}}*k\right)$ with $k=1$ by `calling dl_exponent_size()` from `src/lib/pubkey/workfactor.cpp`. This computation differs from [RFC3766],

where $k = 0.02$ is used. Hence the algorithm overestimates the recommended bit length of `q` by $\log_2(50) \approx 6$ bits. Furthermore, it ensures that `q` is at least 128 bit long, even if a small `pbits` value is passed.

2. Sample the prime `q` of length `qbits` from the passed `rng` by calling random_prime().
3. Sample `X` with length `pbits` from `rng` and set the highest bit.
4. Calculate candidate `p` as $X - (X \bmod (2*q) - 1)$. Thus `q` is a factor of $p-1$.
5. Verify if candidate `p` has length `pbits`. Otherwise, go to 3.
6. Perform Miller-Rabin primality test with 65 iterations. If `p` fails the test, go to 3.
7. Compute generator `g` of the subgroup with order `q` with function `make_dsa_generator()`. The function receives `p` and the order of its subgroup `q`.
   1. Verify that $p-1 > q$ applies. If not, the algorithm terminates with respective error.
   2. Verify that `q` is a factor of $p-1$. If not, algorithm terminates with respective error.
   3. Iterate over the first 6541 precomputed primes q_i (without 2) and compute `g` as $q_i^{\frac{p-1}{q}} \bmod p$. If `g` is 1, it chooses the next prime q_i and repeats the process. Once all available primes q_i have been used and no suitable generator is found, the function terminates with an error.

The function call with `type=DSA_Kosherizer` generates the primes `p` and `q` using the SHA1/2 hash functions. The implementation follows the algorithm described in Section A.1.1.2 in [FIPS-186-4] and operates as follows:

Steps:

1. If a `qbits` value of 0 is passed, set it to 256 or 160, if $pbits \leq 1024$ applies.
2. Sample a random `seed` of bytes $\frac{qbits}{8}$ length from the passed random number generator.
3. Check if the sizes `qbits` and `pbits` are allowed by [FIPS-186-4]. Only the length combinations listed below are valid. If another combination is passed, the algorithm terminates.
   - If $qbits = 160$ applies, `pbits` must be 1024.
   - If $qbits = 224$ applies, `pbits` must be 2048.
   - If $qbits = 256$ applies, `pbits` must be 2048 or 3072.
2. Choose hash function `H()` as SHA-`qbits`.
3. Compute prime candidate `q` as H(`seed`) and set the highest two bits.
4. Perform Miller-Rabin primality test of `q` with 64 iterations. If `q` fails to pass the test, go to step 2.
5. Compute $V_j = H(seed + 1 + j)$ for all $j = 0$ to n and construct `X` as $V_0 || V_1 || \ldots || V_J$.

6. Set the highest bit of `X`.
7. Compute `p` as $X - (X \ mod (2 * q) - 1)$
8. Check if `p` has the desired bit length `pbits` and performs Miller-Rabin primality test with 64 iterations. If the check fails, go to 5. After $4 * pbits - 1$ failures, go to 2.
9. Compute generator `g` of the subgroup with order `q` with function `make_dsa_generator()`. The function receives `p` and the order of its subgroup `q`.
   - Verify that $p - 1 > q$ applies. If not, the algorithm terminates with respective error.
   - Verify that `q` is a factor of $p - 1$. If not, algorithm terminates with respective error.
   - Iterate over the first 6541 precomputed primes q_i (without 2) and computes `g` as $q_i^{\frac{p-1}{q}} \ mod \ p$. If `g` is 1, it chooses the next prime q_i and repeats the process. Once all available primes q_i have been used and no suitable generator is found, the function terminates with an error.

Remark 13: If the DL_Group is generated with `type=Prime_Subgroup` and `qbits` is not passed, the length of `q` is estimated based on the algorithm presented in [RFC3766]. [TR021021] states that `q` should be at least 250 bits long. This is only the case, if a pbits value larger than 2451 is passed. We advise to pass `qbits` directly to the function, to ensure this criteria.

Conclusion: Botan does still allow the generation of 1024 bit DH parameters. This lower bound should be increased to at least 2048 bit.

6.3 Elliptic Curve Algorithms

In order to compute a shared secret with ECDH, it is required that both participating parties agree on a domain, which consists of an elliptic curve, a base point of the curve, its group order and the cofactor. It is theoretically possible to generate a new elliptic curve suitable for ECDH. As this process is very costly and comes with many pitfalls, only precomputed standardized curves are used in Botan. Thus the feature of elliptic curve parameter generation is not implemented. 25 standardized curves are provided in src/`lib/pubkey/ec_group/named.cpp`. All curves recommended in [TR021021] are included. In addition, it is possible to import a single custom elliptic curve at compile time (house-curve). Imported custom Elliptic Curve domains can and should be manually validated with the provided `EC_Group::verify_group` function. The verification function operates as follows.

Input:

- `EC domain (curve parameters (first coefficient a, second coefficient b, prime p), base point G, ord(G) n, cofactor of the curve h)`
- `rng`: random number generator

Ouput:

- `true` if group `EC_Group` is valid. `false` otherwise

Steps:

1. Compute the discriminant as $D = 4 * a + 27 * b \ mod \ p$. If $D = 0$ applies, the curve is

singular and thus invalid. In this case false is returned.

2. Verify that `G` is on the curve. If not return false.
3. Assure that `G` has the correct order `n`. This is the case if $h*G \neq \infty P$ and $n*G = \infty P$ apply. If one of the equations does not apply, return false.
4. Execute Miller-Rabin primality test for `n` with 65 test iterations. If the test fails return false.

The `verify_group` function follows the main recommendations from [ReqEC], it misses a check whether $|E| = h*n$.

Botan implements the elliptic curve standard [ISO-15946-1]. The implementation internally differs between NIST reduction and Montgomery reduction curves and implements the reduction algorithms and curve operations in the respective classes `CurveGFp_Montgomery` and `CurveGFp_NIST`. For efficiency purposes Botan uses Jacobian projective coordinates for all elliptic curve points and point operations as described in [ISO-15946-1] with the line at infinity defined as [0, Y, 0]. The affine coordinates can be obtained by using the conversion functions `get_affine_x()` and `get_affine_y()`.

Function `get_affine_x()` operates as follows.

Input:

- elliptic curve: `CurveGFp_Montgomery` or `CurveGFp_NIST`
- Point in Jacobian projective coordinates: `[X,Y,Z]`

Ouput:

- affine `x` coordinate of the input point

Steps:

1. Verify that the input point is not on the line at infinity with the coordinates `[0,Y,0]`. As the point at infinity has no representative in affine coordinates, terminate with respective error, if a representative of the point at infinity is passed.
2. Compute affine `x` coordinate as $\frac{X}{Z^2}$

The conversion function `get_affine_y()` performs the following steps.

Input:

- elliptic curve: `CurveGFp_Montgomery` or `CurveGFp_NIST`
- Point in Jacobian projective coordinates: `[X,Y,Z]`

Ouput:

- affine `y` coordinate of the input point

Steps:

1. Verify that the input point is not on the line at infinity with the coordinates `[0,Y,0]`. As

the point at infinity has no representative in affine coordinates, terminate with respective error, if a representative of the point at infinity is passed.

2. Compute affine y coordinate as $\frac{Y}{Z^3}$

Conclusion: Botan defines all the elliptic curve parameters recommended in [TR021021].

7 Key Generation for Public Key Algorithms

7.1 DH

The implementation of the Diffie Hellmann key exchange in `src/lib/pubkey/dh/dh.cpp` provides the DH public key class `DH_PublicKey` and the DH private key class `DH_PrivateKey`. The public key consists of the DH parameters and a public value `y`. In addition to the public values, the DH private key includes the private parameter `x`. The algorithm requires both participating parties to generate a DH private key with the same input discrete logarithm group. A private and an associated public key is generated by calling the constructor `DH_PrivateKey(RandomNumberGenerator& rng,const DL_Group& grp,const BigInt& x_arg = 0)`, where `x_arg` of 0 has to be passed to generate a new private parameter. Otherwise `x_arg` is set as the secret value of the `DH_PrivateKey`. The key generation algorithm operates as follows:

Input:

- `rng`: random number generator
- `grp`: DL_Group ($\mathbf{Z}/p\mathbf{Z}$)* : `p`, generator `g` with order `q`
- `x_arg`: private DH parameter

Output:

- DH_PrivateKey: `x`, `y`, DL_Group ($\mathbf{Z}/p\mathbf{Z}$)* : `p`, generator `g` with order `q`

Steps:

1. If `x_arg` was provided, set `x` to `x_arg`. Otherwise:
   - Determine needed exponent length by calling `dl_exponent_size()`.
   - Sample random number of determined exponent length from random number generator `rng` as secret DH value `x`.
2. Compute `y` as $g^x \bmod p$ with `g` and `p` taken from the input group `grp`.

   Optionally the generated parameters and the DL_Group parameters can be verified with a call to `check_key(RandomNumberGenerator& rng, bool strong)`. The generated values and the used discrete logarithm group *fail* the check if one of the following conditions is met.
   - $y < 2$
   - $y \geq p$
   - $x < 2$
   - $x \geq p$
   - $g < 2$
   - $p < 3$

- $q < 0$
- $q \neq 0 \wedge p - 1 \bmod q \neq 0$
- $q \neq 0 \wedge g^q \bmod p \neq 1$
- Miller–Rabin primality test for `p` and `q` fails with 6 (65 if `strong` is true) test iterations.
- $y \neq g^x \bmod p$

Conclusion: The secret DH parameter is sampled as described in [TR021021].

To avoid potentially malicious system parameters from computational peers, the `check_key` function should be manually called before the key from the peer has been accepted.

7.2 DSA

The DSA implementation offers the DSA Key classes `DSA_PrivateKey` and `DSA_PublicKey` with respective constructors. A DSA public key consists of a discrete logarithmic group (DH parameter) and a public value `y`. The associated private key contains an additional private parameter `x`. A `DSA_PrivateKey` can be generated with the constructor `DSA_PrivateKey(RandomNumberGenerator& rng, const DL_Group& grp, const BigInt& x_arg)`. The implementation operates as follows:

Input:

- `rng`: random number generator
- `grp`: DL_Group $(\mathbf{Z}/p\mathbf{Z})^*$: `p`, generator `g` with order `q`
- `x_arg`: private DSA parameter

Output:

- DSA_PrivateKey: `x`, `y`, DL_Group $(\mathbf{Z}/p\mathbf{Z})^*$: `p`, generator `g` with order `q`

Steps:

1. If private value `x_arg` was provided, set `x` to `x_arg`. Otherwise, sample `x` as a random number $1 < x < q-1$ from `rng` using the algorithm described in Section 6.1.
2. Compute public value `y` as $g^x \bmod p$

   Optional verification of the generated key with `check_key(RandomNumberGenerator& rng, bool strong)`. See DH key generation check.

7.3 Elliptic Curve Algorithms

Botan provides the elliptic curve private key class `EC_PrivateKey`, the respective public key class `EC_PublicKey`, and the key generation algorithm in `src/lib/pubkey/ecc_key/ecc_key.cpp`. To generate a private key the constructor `EC_PrivateKey(RandomNumberGenerator& rng, const EC_Group& ec_group, const BigInt& x, bool with_modular_inverse)` is called. The constructor operates

as follows:

Input:

- `rng`: random number generator
- `ec_group`: domain(curve parameters(first coefficient a, second coefficient b, prime p), base point G, ord(G) n, cofactor of the curve h)

Output:

- EC_Privatekey: `d`, `Q`, domain(curve parameters(first coefficient a, second coefficient b, prime p), base point G, ord(G) n, cofactor of the curve h)

Steps:

1. Sample private value `d` as a random number $1 \leq d < n$ using the algorithm described in Section 6.1, where n is the order of the base point G on the curve taken from the domain parameters. It is also possible to pass `d` as `x` to the constructor. In this case `d` is not sampled.
2. Compute public point `Q` as point multiplication $d * G$, where `G` is the base point defined in the domain. Note that if the passed parameter `with_modular_inverse` is set to `true`, the public point `Q` is instead computed as $d^{-1} * G$. This is required for ECKDSA and ECGDSA key generation, but results in an invalid ECDH/ECDSA key.
3. Verify that the computed public point `Q` is on the curve (function `on_the_curve()`). As a consequence, the key generation algorithm resists fault attacks and computational errors.

Optionally `EC_PublicKeys` can be extensively checked with a call to `check_key`. The extensive check performs the following steps. Note that `on_the_curve()` is always automatically checked.

1. Verify the `ec_group` by calling `EC_Group::verify_group`. If the domain does not pass the verification, return false.
2. Assure that the public point `Q` is not the point at infinity.
3. Check that the public point `Q` is on the curve (function `on_the_curve()`). If the point does not satisfy the curve equation, return false.
4. If $h > 1$ applies perform the following additional steps. Else return true.
    1. Verify that the public point has the correct order `n`. This is the case if $h * Q \neq \infty P$ and $n * Q = \infty P$ apply. If one of the equations does not apply, return false.
    2. Return true.

Conclusion: The algorithm fulfills all requirements of [TR03111]. The public key validation follows the requirements described in [ReqEC].

7.4 RSA

The appropriate RSA key pair constructor `RSA_PrivateKey(RandomNumberGenerator& rng,size_t bits, size_t exp = 65537)` of class RSA_PrivateKey is called when

generating a new RSA key pair. `rng` is a random number generator, `bits` the desired bit length of the modulus `N` and `exp` the public exponent to be used.

The key generation process works as follows:

Input:

- `rng`: random number generator
- `bits`: bit length of RSA modulus `N`
- `e`: public exponent

Output:

- RSA_PrivateKey: `N`, `p`, `q`, `e`, `d`, d_1, d_2 , `c`

Steps:

1. The algorithm initially checks if the passed key length is at least 1024. If this is not the case, the function terminates with an error message. Thus, only keys with a desired length of at least 1024 bits can be generated.
2. Subsequently the passed exponent is validated, as it must be odd and larger than 3.
3. The algorithm samples 2 primes by successively calling `random_prime()` from `src/lib/math/numbertheory/make_prm.cpp`, passing the public exponent as `coprime`. The first prime `p` has a bit length of $\lceil \frac{bits}{2} \rceil$ and the second prime `q` is $\lfloor \frac{bits}{2} \rfloor$ long.
4. If the product of the resulting primes `N` has not the specified bit length, go to step 3.
5. The private exponent `d` is computed as $e^{-1} \bmod lcm(p-1, q-1)$. For this purpose, the extended Euclidean algorithm, implemented in src/`lib/math/numbertheory/numthry.cpp`, is used.
6. Additional values needed for CRT-RSA are computed as follows.
   - $d_1 = d \bmod (p-1)$
   - $d_2 = d \bmod (q-1)$
   - $c = q^{-1} \bmod p$

   The key values can be manually checked for consistency with the `check_key(RandomNumberGenerator& rng, bool strong)` function. The key pair fails the check if one of the following conditions is met:
   - $N < 35$
   - $N \bmod 2 = 0$

- $e < 2$
- $d < 2$
- $p < 3$
- $q < 3$
- $p * q \neq N$
- $d_1 \neq d \bmod (p-1)$
- $d_2 \neq d \bmod (q-1)$
- $c \neq q^{-1} \bmod p$
- Miller–Rabin primality test of `p` and `q` fails with 7 (65 if `strong` is true) test iterations
- Only if `strong` is true:
  - $e * d \bmod lcm(p-1, q-1) \neq 1$
  - The creation and verification of a test signature fails (`signature_consistency_check()`)

Remark 14: The side condition $0.1 < |\log_2(p) - \log_2(q)| < 30$ is not guaranteed by the algorithm.

For an odd `bits` value and $n = \lceil \frac{bits}{2} \rceil$ $2^{n-1} + 2^{n-2} + 1 \leq q < p \leq 2^{n+1} + 1$ applies. Thus the conditions upper limit of 30 is never violated

$\lim_{n \to \infty} |\log_2(2^{n+1}+1) - \log_2(2^{n-1}+2^{n-2}+1)| = \left|\log_2(\frac{8}{3})\right| \approx 1.415$ To satisfy the lower condition, `p` and `q` have to at least differ in the third of fourth most significant bit. As the first 2 bits are always set (`random_prime()`), the side condition only applies with a probability of approximately 57%[2]. Since the primes are sampled independently, it it however It is highly probable that they differ in subsequent bits.

Conclusion: The algorithm fulfills all main requirements listed in [TR021021]. The minimum possible bit length of the modulus N should be increased to the recommendation of 2000 bit.

7.5 XMSS with WOTS+

Botan implements the single tree version of the eXtended Merkle Signature Scheme (XMSS) using Winternitz One Time Signatures+ (WOTS+) in `src/lib/pubkey/xmss/*`. The implementation is based on the sixth IETF Internet Draft [XMSS]. The list of supported algorithms and their parameters is depicted in Table 1.

2 This approximation is the result of an analysis of 10000 1024 bit RSA keys generated with Botan.

XMSS algorithm	Parameters			
	n	w	len	h
XMSS_SHA2-256_W16_H10	32	16	67	10
XMSS_SHA2-256_W16_H16	32	16	67	16
XMSS_SHA2-256_W16_H20	32	16	67	20
XMSS_SHA2-512_W16_H10	64	16	131	10
XMSS_SHA2-512_W16_H16	64	16	131	16
XMSS_SHA2-512_W16_H20	64	16	131	20
XMSS_SHAKE128_W16_H10	32	16	67	10
XMSS_SHAKE128_W16_H16	32	16	67	16
XMSS_SHAKE128_W16_H20	32	16	67	20
XMSS_SHAKE256_W16_H10	64	16	131	10
XMSS_SHAKE256_W16_H16	64	16	131	16
XMSS_SHAKE256_W16_H20	64	16	131	20

Table 1: Supported XMSS Signature algorithms and their parameters (see Section 5.2 in [XMSS])

XMSS and WOTS+ rely on the hash function address scheme (`ADRS`). This scheme consists of 256 bits and stores OTS hash addresses and hash tree addresses, see Section 2.5 in [XMSS]. `ADRS` is implemented in `src/lib/pubkey/xmss/xmss_address.h` and offers the following methods:

- `set_key_mask_mode (Key_Mode=0 / Mask_Mode=1, Mask_LSB_Mode=1, Mask_MSB_Mode=2)`
- `set_chain_address (i)`
- `set_hash_address (i)`
- `set_ots_address (i)`
- `set_ltree_address (i)`
- `set_type (OTS Hash Address / L-Tree Address / Hash Tree Address)`
- `set_tree_height(i)`
- `set_tree_index(i)`

XMSS and WOTS+ use a specific ***base w*** number representation. For example, this representation turns a string `X=0x1234` into a byte array `{1,2,3,4}=base_w(X, 16, 4)`. We refer to Section 2.6, Algorithm 1 [XMSS] for more details.

7.5.1 WOTS+

WOTS+ uses a chaining function `chain(X,i,s,ADRS,seed)` to iteratively execute `s` PRF calls on a given input string `X`, the start index `i`, number of steps `s`, combined with `ADRS` and a `seed` value. See Algorithm 2 in [XMSS] for more details.

WOTS+ is implemented in `src/lib/pubkey/xmss/xmss_wots_privatekey.cpp` and `src/lib/pubkey/xmss/xmss_wots_publickey.cpp`.

The key generation process works as follows:

Input:

- `rng`: random number generator
- `oid`: XMSS WOTS+ parameters (`n, w, len, PRF`), see Table 1
- `public_seed`: public seed used for the pseudo random generation of public keys derived from the generated private key[3]

Output:

- XMSS_WOTS_PrivateKey: `sk`, `pk`, `private_seed`

Steps:

1. Use `rng` to generate `private_seed` of length `n`.
2. Use `private_seed` to generate `len` secret keys `sk[i]` of length `n`.
3. Generate `len` public key `pk[i]` of length `len` using `sk`, `public_seed`, `ADRS`, and the chaining function:
    1. Set i=0;
    2. While `(i < len)` do:

```
1. ADRS.set_chain_address(i);
2. pk[i] = chain(sk[i], 0, w-1, public_seed, ADRS);
```

Remark 15: Botan uses two PRNG seed values: `public_seed` and `private_seed`. The `private_seed` value is used to dynamically generate further arrays of secret keys: `sk[i]=PRF(private_seed,i)`. The advantage is that only the `private_seed` value must be stored instead of the full secret key (see Section 3.1.7 in [XMSS]).

7.5.2 XMSS

XMSS functionality is implemented in `src/lib/pubkey/xmss/xmss_privatekey.h` and `src/lib/pubkey/xmss/xmss_publickey.h`.

The algorithm for key generation relies on the method `treeHash` from Algorithm 9 in [XMSS]. The `treeHash` method takes as input secret key `sk`, start index `s`, target node height `t`, and address `ADRS`. The algorithm uses the input parameters and the secret key `sk` stored in the XMSS_PrivateKey object to return the root node of a given tree, whose height is `t`. The index s represents the index of the left most leaf of the WOTS+ public key. Botan implements the function as described in Algorithm 9.

Based on the `treeHash` function the key generation process follows Algorithm 10 in [XMSS] and it works as follows:

Input:

3 This variable is optional and can also be generated using rng.

- `rng`: random number generator
- `xmss_algo_id`: XMSS signature parameter identifier (`n, w, len, PRF`), see Table 1

Output:

- XMSS_PrivateKey: `SK`, `PK`

Steps:

1. Create a new XMSS public key for the chosen XMSS signature method.
2. Generate new `public_seed` and `SK_PRF` seed using `rng`. Each seed has length `n`. Public seed is used for the pseudo random generation of WOTS+ public keys. Private seed `SK_PRF` is used for the pseudo random generation of XMSS private keys and WOTS+ private seeds.
3. WOTS+ private and public keys are initialized with `public_seed` and `private_seed` (see the algorithm from previous section). The values of `private_seed` for each WOTS+ private key are generated using `SK_PRF`.
4. Initiate the index registry with `idx=0`. This value references the first unused leaf index.
5. Compute the `root` node value using the `treeHash` function:

   ```
   root = treeHash(0, h, ADRS);
   ```

6. ```
   SK = {idx, SK_PRF, root, public_seed}
   ```

   ```
   PK = {root, public_seed}
   ```

Remark 16: Note that Botan does not store the whole XMSS keys in memory. Similarly to WOTS+, only `public_seed, private_seed,` and `SK_PRF` are stored, and are used to construct keys on demand. See also Section 4.1.11 in [XMSS].
   ```

8 Asymmetric Encryption and Key Exchange Schemes

8.1 RSA

Botan implements RSA in `src/lib/pubkey/rsa/rsa.cpp`. RSA encryption and decryption are implemented in classes `RSA_Encryption_Operation` and `RSA_Decryption_Operation`.

The RSA decryption operation is implemented as follows:
Input:

- `rng`: random number generator
- `c`: ciphertext
- RSA_PrivateKey: `N`, `p`, `q`, `e`, `d`, d_1, d_2

Output:

- `m`: decrypted message

Steps:

0. Before the first decryption: The random number generator `rng` is used to initialize the blinding values m_e and its inverse m_d. Their length is set to `(m_modulus_bits - 1)`.
1. Blind ciphertext: $c' = c * m_e^e \, mod \, N$
2. Use Chinese Reminder Theorem (CRT) to decrypt `c'` and retrieve `m'`
3. Unblind message: $m = m' * m_d \, mod \, N$

Remark 17: After each blinding step, the blinding values change: $m_e = m_e * m_e \, mod \, N$, $m_d = m_d * m_d \, mod \, N$. This is performed at most 32 times (defined in `BOTAN_BLINDING_REINIT_INTERVAL`). Afterwards, `rng` is used to generate new blinding values.

RSA-PKCS#1 v1.5

The RSA-PKCS#1 v1.5 padding functionality is implemented in `src/lib/pk_pad/eme_pkcs1/eme_pkcs.cpp`, in the methods `pad` and `unpad`.

From the security perspective, the unpadding method is crucial since it has to resist the Bleichenbacher attack [Blei], and has to provide timing constant validation. This method is implemented as follows:

Input:

- `valid_mask`: message validity mask indicating whether the padding structure was valid
- `m`: padded message

- `in_len`: message length

Output:

- `k`: unpadded message
- `valid_mask`: message validity mask indicating whether the padding structure was valid

Steps:

1. `bad_input_m = 0`
2. `seen_zero_m = 0`
3. `bad_input_m |= (m[0] != 0x00)`
4. `bad_input_m |= (m[0] != 0x02)`
5. `for (i = 2; i < |m|; i++)`
   - `seen_zero_m |= (m[i] != 0x00)`
   - `bad_input_m |= (m[i] == 0x00 && i<9)`
6. `bad_input_m |= ~seen_zero`
7. `valid_mask = ~bad_input_m;`
8. Set `k` to the byte array behind the first `0x00`
9. `return k, valid_mask`

Remark 18: For TLS, Botan uses a different unpadding function `decrypt_or_random()`, which is located in `src/lib/pubkey/pubkey.cpp`.

RSA-OAEP

The RSA-OAEP functionality is implemented in `src/lib/pk_pad/eme_oaep/oaep.cpp`, in the functions `pad()` and `unpad()`.

From the security perspective, the unpadding method is crucial since it has to resist Manger's attack [Man], and has to provide timing constant validation. The decryption process cannot provide any information whether the first message byte was zero or not. This method is implemented as follows:

Input:

- `valid_mask`: message validity mask indicating whether the padding structure was valid
- `m`: padded message
- `in_len`: message length

Output:

- `k`: unpadded message
- `valid_mask`: message validity mask indicating whether the padding structure was valid

Steps:

The first byte is extracted as follows:

```
1. skip_first = (in[0]==0) & 0x01;
2. m' = array(m + skip_first, m + in_len);
```

The remaining steps operate on the message `m'`, and proceed according to the RSA-OAEP specification.

8.2 Diffie-Hellman (DH)

In the following section we describe the implementation of the Diffie-Hellman key exchange over cyclic groups (**Z**/*p***Z**)*. The respective classes and functions can be found in `src/lib/pubkey/dh/dh.cpp`.

Botan computes the shared Diffie-Hellman secret with the following algorithm, implemented in `raw_agree(const byte w[], size_t w_len)` which is part of the respective DH operation class `DH_KA_Operation`. The function receives the other parties public value y_b and computes the shared secret as follows:

Input:

- y_b : DH public value of the other party
- DH_PrivateKey: `x`, `y`, DL_Group (**Z**/*p***Z**)* : `p`, generator `g` with order `q`

Output:

- `s`: shared DH secret

Steps:

1. Sample a blinding nonce m_e and compute its inverse m_d . m_e has a length of $length(p)-1$.
2. Verify that y_b is valid. That is the case if $1<y_b<p-1$ applies. The algorithm terminates with an exception, if y_b is invalid.
3. Blind y_b as $y_b'=y_b*m_e \bmod p$.
4. Compute the blinded shared secret s' as $s'=y_b'^x \bmod p$.
5. Unblind the shared secret $s=s'*m_d \bmod p$

Optionally a specified KDF is applied to the shared secret.

Conclusion: The algorithm fulfills all DH criteria listed in [TR021021].

8.3 Elliptic Curve Diffie-Hellman (ECDH)

The elliptic curve variant of the Diffie-Hellman key exchange is implemented in `src/lib/pubkey/ecdh/ecdh.cpp`.

The shared secret is computed by calling `raw_agree(const byte w[], size_t w_len)`

from the respective ECDH operation class `ECDH_KA_Operation`. The algorithm receives the public point of the other party and computes the shared secret as follows:

Input:

- `rng: random number generator`
- Q_b : ECDH public point of the other party
- EC_Privatekey: `d`, `Q`, domain (curve parameters (first coefficient `a`, second coefficient `b`, prime `p`), base point `G`, `ord(G) n`, cofactor of the curve `h`)

Output:

- `S`: shared ECDH secret point

Steps:

1. Compute intermediate value $i=(h^{-1} \bmod n)*d$, where `h` is the cofactor taken from the agreed domain.
2. Verify that the received public point Q_b is on the elliptic curve. This check is part of the decode function `OS2ECP()`.
3. Sample a $\left\lceil \frac{length(n)}{2} \right\rceil$ bit long random blinding `mask` from `rng` and compute $i'=i+n*mask$. This blinding mechanisms can be disabled by unsetting `BOTAN_POINTGFP_USE_SCALAR_BLINDING` in `build.h` at compile time.
4. Compute the shared secret point `S` as $S=(h*Q_b)*i'=(h*Q_b)*(h^{-1} \bmod n)*d=Q_b*d$. This computation utilizes randomized Jacobian point coordinates with 80 bit blinding masks by default. The length of these blinding masks is controlled by `BOTAN_POINTGFP_RANDOMIZE_BLINDING_BITS` in `build.h` at compile time.
5. Verify that the computed shared secret point `S` is on the selected elliptic curve (`on_the_curve()`).
6. Return affine x coordinate of shared point `S` as shared secret. Before the transformation to affine coordinates is carried out, it is checked, if the shared point S is the point at infinity (`is_zero()`). If that is the case, a respective error is thrown.

Optionally a specified KDF is applied to the shared secret.

Conclusion: The implemented ECDH key agreement algorithm complies with the algorithm shown in chapter 4.3.1 of [TR03111] and thus fulfills the ECDH criteria listed in [TR021021], if a recommended curve was chosen. Furthermore, it is recommended to utilize the optional KDF to derive a symmetric key.

8.4 Hybrid Encryption Schemes

A hybrid encryption scheme is a combination of an asymmetric and a symmetric cryptosystem. In

detail the participating parties agree on a shared secret, which is then used to encrypt or decrypt data with a symmetric cipher. In addition, the authenticity of the data is secured with a MAC. The following schemes are both Integrated Encryption Schemes and standardized by the IEEE, ANSI and ISO.

8.4.1 DLIES

The Discrete Logarithm Integrated Encryption Scheme (DLIES) utilizes the Diffie-Hellman key exchange as the asymmetric component of the scheme. The symmetric cipher and MAC can be chosen. Botan implements the DLIES encryption scheme in `src/lib/pubkey/dlies/dlies.cpp`, providing the classes DLIES_Decryptor and DLIES_Encryptor. DLIES can be used in either stream or block cipher mode. Both modes are implemented according to [ISO-18033-2].

The DLIES_Encryptor constructor requires a `KDF`, `MAC` and `cipher` algorithm and a `DH private key`. The class offers the following functions:

- `set_other_key(key)`: Sets the other parties public Diffie-Hellman public key.
- `set_initialization_vector(IV)`: Sets the `IV` to use for the plaintext encryption.
- `enc(plaintext, plaintext length)`: Encryption function.
- Ensure that the other parties DH public key has been set correctly. If not terminate with respective error.
    1. Compute the Diffie-Hellman secret `s` using the provided DH private key and the other parties public value. Confer section 1.7.3.
    2. Pass `s` to the specified KDF and derive keybits for usage with the cipher and extra keybits for the `MAC`. If the KDF did not provide enough output bits, terminate with respective error.
    3. Encrypt the passed `plaintext` using the specified symmetric `cipher` and the derived encryption key.
    4. Compute the tag over the ciphertext using the specified `MAC` function and the derived MAC key.
    5. Return the concatenation of the own DH public key, the ciphertext and the computed tag.
- If no cipher is passed, the algorithm operates in stream mode. In this mode, the ciphertext is computed as $plaintext \oplus keybits$.

The DLIES_Decryptor requires similar parameters. DLIES_Decryptor offers the following functions:

- `set_initialization_vector(IV)`: Sets the `IV` to use for the ciphertext decryption.
- `do_decrypt(input, input length)`:
    1. Perform preliminary length checks of the input.
    2. Extract the other parties public Diffie-Hellman key from `input` and calculate the

shared DH secret `s`.

1. Derive the `cipher` and `MAC` key from the specified `KDF`. If the KDF did not provide enough output bits, terminate with respective error.
3. Extract the ciphertext from `input` and calculate the `MAC` function using the derived `MAC` key.
4. Validate that the calculated tag and the tag provided with input are equal. If they are not equal, return an empty plaintext vector.
5. Decrypt the cipthertext using the derived cipher key.

- If no cipher is passed, the algorithm operates in stream mode. In this mode, the ciphertext is computed as $plaintext \oplus keybits$.

8.4.2 ECIES

The Elliptic Curve Integrated Encryption Scheme (ECIES) resembles the DLIES algorithm. Instead of the Diffie-Hellman key exchange, the Diffie-Hellman key exchange over elliptic curves is used as the asymmetric component of the hybrid scheme. Botan implements the scheme according to [ISO-18033-2]. The implementation offers the operator classes ECIES_Encryptor and ECIES_Decryptor and the ECIES_System_Params class in `src/lib/pubkey/ecies/ecies.cpp`. [ISO-18033-2] requires the definition of ECIES specific system parameters, called ECIES flags. The available ECIES flags dictate certain computation rules:

- SINGLE_HASH_MODE: Prefix the input of the ECDH key exchange with the encoded public point.
- COFACTOR_MODE: Use cofactor multiplication during ECDH key exchange for decryption.
- OLD_COFACTOR_MODE: Use ECDH cofactor multiplication on both sides.
- CHECK_MODE: Test if the received point is on the curve.
- To support all ECIES flags defined in [ISO-18033-2], two distinct implementations of the ECDH key agreement are required. The agreement function with cofactor multiplication is part of Botans default ECDH implementation. The ECIES specific implementation without cofactor multiplication is implemented in class `ECIES_ECDH_KA_Operation` of `src/lib/pubkey/ecies/ecies.cpp`. The agreement without cofactor mode operates as follows:

Input:

- `rng: random number generator`
- Q_b : ECDH public point of the other party
- EC_Privatekey: `d`, `Q`, domain(curve parameters(first coefficient `a`, second coefficient `b`, prime `p`), base point `G`, ord(G) `n`, cofactor of the curve `h`)

Output:

- `S`: shared ECDH secret point

Steps:

1. Verify that the received public point Q_b is on the elliptic curve. This check is part of the decode function `OS2ECP()`.
2. Sample a $\left\lceil \frac{length(n)}{2} \right\rceil$ bit long random blinding `mask` from `rng` and compute $d' = d + n * mask$. This blinding mechanisms can be disabled by unsetting `BOTAN_POINTGFP_USE_SCALAR_BLINDING` in `build.h` at compile time.Compute the shared secret point `S` as $S = Q_b * d'$. This computation utilizes randomized Jacobian point coordinates with by default 80 bit long blinding masks. The length of these blinding masks is controlled by `BOTAN_POINTGFP_RANDOMIZE_BLINDING_BITS` in `build.h` at compile time.. Verify that the computed shared secret point `S` is on the selected elliptic curve (`on_the_curve()`).Return affine x coordinate of shared point `S` as shared secret. Before the transformation to affine coordinates is carried out, it is checked, if the shared point S is the point at infinity (is_zero()). If that is the case, a respective error is thrown.

The ECIES_Encryptor constructor requires a ECDH private key and ECIES system parameters, which consist of the ECIES flags, a EC domain, a KDF, a Cipher and a MAC algorithm. The class offers the following functions:

- `set_other_key(key)`: Sets the other parties public key.
- `set_initialization_vector(IV)`: Sets the `IV` to use for the plaintext encryption.
- `enc(plaintext, plaintext length)`:
  1. Ensure that the other parties ECDH public point has been set correctly and is not the point at infinity. If not terminate with respective error.
  2. Compute the ECDH secret `s` using the provided ECDH private key and the other parties public point. This operation honors the defined ECIES flags. Thus the implementation uses either the ECDH implementation described in section 1.7.3, if OLD_COFACTOR_MODE is set or else the custom implementation without cofactor mode `ECIES_ECDH_KA_Operation`, described above.
  3. Pass `s` to the specified KDF and derive keybits for the usage with the cipher and additional bits for the `MAC`. If the KDF did not provide enough output bits, terminate with respective error.
  4. Encrypt the passed `plaintext` using the specified symmetric `cipher` and the derived encryption key.
  5. Compute the tag over the ciphertext using the specified `MAC` function and the derived MAC key.

- Return the concatenation of the own encoded ECDH public point, the ciphertext and the computed tag.

The ECIES_Decryptor of the integrated scheme requires similar parameters. The class offers the following functions:

- `set_initialization_vector(IV)`: Sets the `IV` to use for the ciphertext decryption.
- `do_decrypt(input, input length)`:
  1. Peform preliminary length checks of the input.
  1. Extract the public point from input and compute the ECDH secret `s` using the provided ECDH private key and the other parties public point. This operation honors the defined ECIES flags. Thus the implementation uses either the ECDH implementation described in section 1.7.4, if OLD_COFACTOR_MODE or COFACTOR_MODE is set. Else the custom implementation without cofactor mode `ECIES_ECDH_KA_Operation`, described above, is used.
  1. Pass `s` to the specified KDF and derive key bits for the usage with the cipher and additional bits for the `MAC`. If the KDF did not provide enough output bits, terminate with respective error.
  1. Extract the ciphertext from `input` and calculate the `MAC` function using the derived `MAC` key.
  2. Validate that the calculated tag and the tag provided in input are equal. If they are not equal, return an uninitialized plaintext vector.
- Decrypt the ciphertext using the derived cipher key.

Signatures

8.5 RSA

The RSA signature algorithm is provided in the class `RSA_Signature_Operation` in `src/lib/pubkey/rsa/rsa.cpp`. The respective verification algorithm is implemented in the class `RSA_Verify_Operation`.

RSA Signature Schemes

Before a message can be signed it must be processed to achieve the bit length of the RSA modulus `N`. Therefore, a padding scheme is applied to the message `m`. Botan implements multiple padding schemes. For RSA signatures the probabilistic RSA-PSS scheme (EMSA4) implemented in `src/lib/pk_pad/emsa_pssr/pssr.cpp` is recommended [TR02102]. The RSA-PSS implementation follows the definition in [RFC3447]. The ISO 9796-2 DS2 and ISO 9796-2 DS3 padding schemes are implemented in `src/lib/pk_pad/iso9796/iso9796.cpp`. Both implementations follow the specification [ISO9796-2]. Alternatively, Botan provides the deterministic PKCS#1 v1.5 RSA signature scheme (EMSA3), which is obsolete and thus not recommended.

Signature Creation

To sign data, the function `secure_vector<byte> raw_sign(const byte msg[], size_t msg_len,RandomNumberGenerator&)` is called. The bytes to sign `m` and its length is passed to the function. The RSA signature algorithm operates as follows:

Input:

- `m`: raw bytes to sign
- RSA_PrivateKey: `N`, `p`, `q`, `e`, `d`, d_1, d_2 , `c`

Output:

- `x`: raw RSA_Signature

Steps:

1. Verify that $m < N$. If the message to sign exceeds the modulus of the RSA private key, the algorithm terminates with error.
2. Blind message m with random blinding nonce `k`.
3. Compute $x_1 = m^{d_1} \bmod p$ and $x_2 = m^{d_2} \bmod q$. This computations takes place simultaneously (additional thread). d_1 and d_2 are part of the RSA private key.
4. Compute $h = (x_1 - x_2) * q^{-1} \bmod p$, where q^{-1} is taken from the private key.
5. Compute signature $x = h * q + x_2$
6. Unblind `x` with k^{-1} .

7. Verify that $x^e \bmod N = m$ applies. Thus it is assured, that the algorithm returns a valid signature. As a consequence, the implementation resists fault attacks.

Signature Verification

To verify a RSA signature the signature value is passed to the function `secure_vector<byte> verify_mr(const byte msg[], size_t msg_len)`. This function proceeds as follows:

Input:

- `x`: raw RSA _Signature
- RSA_PublicKey: `N`, `e`

Output:

- `v`: value to compare signed data to

Steps:

1. Verify that the signature is smaller than the modulus of the public key. If $x < N$ does not apply, the algorithm throws a respective error.
2. Compute $v = x^e \bmod n$

8.6 DSA

The Digital Signature Algorithm (DSA) is implemented in `src/lib/pubkey/dsa/dsa.cpp`.

DSA Signature Schemes

For DSA signatures no padding is required. The only suitable signature scheme DL/ECSSA (EMSA1) uses a cryptographic hash function to compute a representative message with the length of q from the DSA public key. Botan implements EMSA1 in `src/lib/pk_pad/emsa1/emsa1.cpp`. If the computed hash is longer than the specified `output_bits` (length of `q`), the algorithm returns only the `output_bits` highest bits of the computed hash.

Signature Creation

The message representative created by the EMSA1 encoding algorithm is passed to `raw_sign(const byte msg[], size_t msg_len,RandomNumberGenerator& rng)` of class `DSA_Signature_Operation`. Botan computes the DSA signature with the following algorithm.

Input:

- `rng: random number generator`
- `m`: raw bytes to sign (EMSA1 encoded data)
- DSA_PrivateKey: `x`, `y`, DL_Group

Output:

- (r,s): DSA signature

Steps:

1. Perform conditional subtractions $m = m - q$, while $m \geq q$.
2. Generate parameter k as a random number $0 < k < q$ from the passed rng using the algorithm described in Section 6.1 or as HMAC_DRBG output [RFC6979]. If Botan is compiled with the module rfc6979 the HMAC_DRBG is used, otherwise k is sampled from the passed random number generator rng. HMAC_DRBG is deterministic and k thus depends on the HMAC_DRBG inputs m and x.
3. Compute $r = (g^k \bmod p) \bmod q$ and $k^{-1} \bmod q$ simultaneously (in separate threads).
4. Compute $s = k^{-1} * (x * r + m) \bmod q$
5. If $s = 0 \vee r = 0$ applies, the algorithm terminates with an error.

Remark 19: If Botan is built with the RFC6979 module, it implements deterministic DSA signatures, which are not covered by [TR021021]. In this case the implemented DSA signature algorithm is not [FIPS-186-4] conform. This cryptographic construct does not need a random number generator during signature computation. However, the RFC6979 module is prohibited in the BSI module policy.

Signature Verification

To verify a DSA signature the function `verify(const byte msg[], size_t msg_len, const byte sig[], size_t sig_len)` in class `DSA_Verification_Operation` is implemented. The function receives a signature, the respective EMSA1 processed message and the lengths of the parameters. The algorithm operates as follows:

Input:

- (r,s): DSA signature
- m: message bytes
- DSA_PublicKey: y, DL_Group

Output:

- true, if the signature for message m is valid. false otherwise

Steps:

1. Verify that the signature (r, s) has length $2 * qbits$ and $m < q$ applies. If that is not the case, the signature is invalid and false is returned.
2. Assure that $0 < r < q \wedge 0 < s < q$ applies. Otherwise the signature is invalid and false is returned.
3. Compute $w = s^{-1} \bmod q$
4. Compute $v_i = g^{w * i \bmod q} \bmod p$ and $v_r = y^{w * r \bmod q} \bmod p$ simultaneously (in separate

threads).

5. Compute $v = v_i * v_r \bmod p$
6. Return `true`, if $v \equiv r \bmod p$ applies and `false` otherwise.

8.7 ECDSA

The Digital Signature Algorithm over elliptic curves is implemented in `src/lib/pubkey/ecdsa/ecdsa.cpp`.

ECDSA Signature Schemes

Similarly to DSA, ECDSA uses the DL/ECSSA (EMSA1) signature scheme to compute a representative of the message to be signed.

Signature Creation

The signature generation algorithm works as follows:

Input:

- `rng: random number generator`
- `m`: raw bytes to sign (EMSA1 encoded data)
- EC_Privatekey: `d`, `Q`, domain (curve parameters (first coefficient `a`, second coefficient `b`, prime `p`), base point `G`, `ord(G)` `n`, cofactor of the curve `h`)

Ouput:

- (`r`,`s`): ECDSA signature

Steps:

1. Generate parameter `k` as a random number $0 < k < |E|$ using the algorithm described in Section 6.1 or as HMAC_DRBG output [RFC6979]. If Botan is compiled with the module RFC6979 the HMAC_DRBG is used, otherwise `k` is sampled from the passed random number generator `rng`. HMAC_DRBG is deterministic and k thus depends on the HMAC_DRBG inputs `m`, n and d.
2. Sample a $\left\lceil \frac{length(n)}{2} \right\rceil$ bit long random blinding `mask` from `rng` and compute $k' = k + n * mask$. This blinding mechanisms can be disabled by unsetting `BOTAN_POINTGFP_USE_SCALAR_BLINDING` in `build.h` at compile time. Compute the point multiplication $k_p = (x_1, y_1) = k' * G$, where G is the base point of the domain. This computation utilizes randomized Jacobian point coordinates with 80 bits blinding masks by default. The length of these blinding masks is controlled by `BOTAN_POINTGFP_RANDOMIZE_BLINDING_BITS` in `build.h` at compile time. Compute $r = x_1 \bmod n$ and $s = k^{-1} * (r * d + m) \bmod n$ If $s = 0 \vee r = 0$ applies, the algorithm terminates with an error.

Remark 20: If Botan is built with the RFC6979 module, it implements deterministic ECDSA signatures, which are not covered by [TR021021]. In this case the implemented ECDSA signature algorithm is not [FIPS-186-4] conform. However, the RFC6979 module is prohibited in the BSI module policy.

Signature Verification

The signature verification algorithm works as follows:

Input:

- `m`: message bytes
- EC_Publickey: `Q`, domain (curve parameters (first coefficient `a`, second coefficient `b`, prime `p`), base point `G`, `ord(G)` `n`, cofactor of the curve `h`)
- (`r`,`s`): ECDSA signature

Ouput:

- `true`, if the signature for message `m` is valid. `false` otherwise.

Steps:

1. Verify the passed signature has length $2*qbits$. If that is not the case `false` is returned.
2. Assure that $0<r<n \wedge 0<s<n$. Otherwise the signature is invalid.
3. Compute $w = s^{-1} \bmod n$
4. Compute $v_1 = m*w \bmod n$ and $v_2 = r*w \bmod n$
5. Compute the point $v = (x_1, y_1) = v_1*G + v_2*Q$ with Shamir's trick [DI08].
6. Return `true` if $v \equiv r \bmod n$ applies. `false` otherwise.

8.8 ECKCDSA

The Korean Certificate-based Digital Signature Algorithm over elliptic curves is implemented in `src/lib/pubkey/eckcdsa/eckcdsa.cpp`. The implementation follows [TR-03111].

ECKCDSA Signature Schemes

Similarly to other DSA variants, ECKCDSA uses the DL/ECSSA (EMSA1) signature scheme to compute a representative of the message to be signed.

Signature Creation

The signature generation algorithm works as follows:

Input:

- `m`: raw bytes to sign (EMSA1 encoded data)
- EC_Privatekey with inverse: `d`, `Q`, domain(curve parameters(first coefficient a, second

coefficient b, prime p), base point G, ord(G) n, cofactor of the curve h)

Ouput:

- (r,s): ECKCDSA signature

Steps:

1. Sample parameter k as a random number $0<k<n$ from `rng` using the algorithm described in Section 6.1 .
2. Sample a $\left\lceil \frac{length(n)}{2} \right\rceil$ bit long random blinding `mask` from `rng` and compute $k'=k+n*mask$. This blinding mechanisms can be disabled by unsetting `BOTAN_POINTGFP_USE_SCALAR_BLINDING` in `build.h` at compile time.
3. Compute point $W=(x_{1,}y_1)=k'*G$. This computation utilizes randomized Jacobian point coordinates with 80 bit blinding masks by default. The length of these blinding masks is controlled by `BOTAN_POINTGFP_RANDOMIZE_BLINDING_BITS` in `build.h` at compile time.
4. Compute $r=H(x_1)$, where H is the hash function used in the current instance of the EMSA1 signature scheme.
5. Compute $s=d*(k-r\oplus m) mod n$. If $s=0$ applies, the algorithm terminates with an error.
6. Return ECKCDSA signature (r,s).

Signature Verification

The signature verification algorithm works as follows:

Input:

- `m`: message bytes
- EC_Publickey: `Q`, domain(curve parameters(first coefficient a, second coefficient b, prime p), base point G, ord(G) n, cofactor of the curve h)
- (`r`,`s`): ECKCDSA signature

Ouput:

- `true`, if the signature for message `m` is valid. `false` otherwise

Steps:

1. Perform preliminary parameter checks and verifies that $0<s<n$ applies. Terminates otherwise.
2. Compute $e=r\oplus m mod n$.
3. Compute point $W=s*Q+e*G$ with Shamir's trick.

4. Return `true` if $r = H(x_1)$ applies, where H is the hash function used in the current instance of the EMSA1 signature scheme. Otherwise returns `false`.

8.9 ECGDSA

ECGDSA Signature Schemes

The German Digital Signature Algorithm over elliptic curves is implemented in `src/lib/pubkey/ecgdsa/ecgdsa.cpp`. The implementation follows [ISO-14888-3].

Signature Creation

The signature generation algorithm works as follows:

Input:

- `m`: raw bytes to sign (EMSA1 encoded data)
- EC_Privatekey with inverse: `d`, `Q`, domain(curve parameters(first coefficient a, second coefficient b, prime p), base point G, ord(G) n, cofactor of the curve h)

Ouput:

- (r,s): ECGDSA signature

Steps:

1. Sample parameter `k` as a random number $0 < k < n$ from `rng` using the algorithm described in Section 6.1 .
2. Sample a $\left\lceil \frac{length(n)}{2} \right\rceil$ bit long random blinding `mask` from `rng` and compute $k' = k + n * mask$. This blinding mechanisms can be disabled by unsetting `BOTAN_POINTGFP_USE_SCALAR_BLINDING` in `build.h` at compile time.
3. Compute point $W = (x_1, y_1) = k' * G$. This computation utilizes randomized Jacobian point coordinates with 80 bits blinding masks by default. The length of these blinding masks is controlled by `BOTAN_POINTGFP_RANDOMIZE_BLINDING_BITS` in `build.h` at compile time.
4. Set $r = x_1 \bmod n$
5. Compute $s = d * (k * r - m) \bmod n$.
6. If $s = 0 \vee r = 0$ applies, the algorithm terminates with an error.
7. Return ECGDSA signature (r,s).

Signature Verification

The signature verification algorithm works as follows:

Input:

- `m`: message bytes
- EC_Publickey: `Q`, domain(curve parameters(first coefficient a, second coefficient b, prime p), base point G, ord(G) n, cofactor of the curve h)
- (`r`,`s`): ECGDSA signature

Ouput:

- `true`, if the signature for message `m` is valid. `false` otherwise

Steps:

1. Perform preliminary parameter checks and verify that $0<r<n \wedge 0<s<n$ applies.
2. Compute $r^{-1} \bmod n$
3. Compute $v_1 = r^{-1} * m \bmod n$ and $v_2 = r^{-1} * s \bmod n$.
4. Compute point $W = v_1 * G + v_2 * Q$
5. Return `true` if $r \equiv x_1 \bmod q$ applies. Otherwise it returns `false`.

8.10 XMSS with WOTS+

8.10.1 WOTS+

Signature Creation

WOTS+ signing follows Algorithm 5 in [XMSS]. It is implemented in `src/lib/pubkey/xmss/xmss_wots_privatekey.cpp` and `src/lib/pubkey/xmss/xmss_wots_signature_operation.cpp`.

The signature generation process works as follows:

Input:

- `m`: message to be signed
- `oid`: XMSS WOTS+ parameters (`n, w, len, PRF`), which are chosen automatically based on the XMSS parameters from Table 1, see [XMSS]
- `ADRS`: Address
- `public_seed`: public seed
- `private_seed`: private seed

Output:

- `sig`: signature

Steps:

1. Convert the message `m` into base_w representation.

2. Compute a checksum over the converted message and convert this checksum into base_w representation. Append the checksum to the message `m`.
3. Generate the resulting signature bytes `sig` as follows:
    1. Set i=0;
    2. While `(i < len)` do:

        1. `ADRS.set_chain_address(i);`
        2. `sig[i] = chain(sk[i], 0, m[i], public_seed, ADRS);`

Remark 21: Note that the algorithm retrieves `sk[i]` in each round dynamically by using `private_seed` and `ADRS`.

Signature Validation

WOTS+ signature validation strictly follows Algorithm 6 in [XMSS]. It is implemented in `src/lib/pubkey/xmss/xmss_wots_publickey.cpp` and `src/lib/pubkey/xmss/xmss_wots_verification_operation.cpp`.

The signature validation process works as follows:

Input:

- `m`: message to be validated
- `oid`: XMSS WOTS+ parameters (`n, w, len, PRF`), which are chosen automatically based on the XMSS parameters from Table 1, see [XMSS]
- `sig`: Signature
- `ADRS`: Address
- `public_seed`: public seed

Output:

- `tmp_pk`: Temporary WOTS+ public key. This public key is afterwards compared with the provided public key.

Steps:

1. Convert the message `m` into base_w representation.
2. Compute a checksum over the converted message and convert this checksum into base_w representation. Append the checksum to the message `m`.
3. Generate the temporary public key `tmp_pk` as follows:
    1. Set i=0;
    2. While `(i<len)` do:
        1. `ADRS.set_chain_address(i);`

2. `tmp_pk[i] = chain(sig[i], m[i], w-1-m[i], public_seed, ADRS);`

8.10.2 XMSS

Signature Creation

XMSS signature generation functionality is implemented in `src/lib/pubkey/xmss/xmss_privatekey.h` and `src/lib/pubkey/xmss/xmss_signature_operation.h`

The algorithm for signature generation follows methods `treeSig` and `XMSS_sig` from Algorithms 11 and 12 in [XMSS]. The algorithm works as follows:

Input:

- `m`: message to be signed
- `SK`: XMSS secret key, `SK = {idx, SK_PRF, root, public_seed}`

Output:

- `Sig:` XMSS signature

Steps:

1. Initialize the signature operation and reserve a new leaf index `idx` of an *unused* WOTS+ signature. This index cannot be reused in further operations. Calculate a pseudorandom value r using the output of PRF on `SK_PRF || idx`.
2. Generate a hash over the message `m`, Merkle tree root, index `idx`, and output of the PRF function over the secret seed `SK_PRF`.
3. Build an authentication path `auth_path` by using the leaf index `idx`, and address `ADRS`.
4. Compute a WOTS+ signature `sig_ots` over the constructed hash value.
5. `Sig = {idx, r, auth_path, sig_ots}`

Signature Validation

XMSS signature validation functionality is implemented in `src/lib/pubkey/xmss/xmss_publickey.h` and `src/lib/pubkey/xmss/xmss_verification_operation.h`

The algorithm for signature verification follows methods `XMSS_rootFromSig` and `XMSS_verify` from Algorithms 13 and 14 in [XMSS]. The algorithm works as follows:

Input:

- `m`: message to be validated
- `Sig:` XMSS signature
- `PK`: XMSS public key, `PK = {root, public_seed}`

Output:

- `true`, if the signature for message `m` is valid. `false` otherwise

Steps:

1. Generate a hash over randomness `r`, Merkle tree root and index `idx` stored in the signature `Sig`, and message `m`.
2. Compute the root node `node` using the computed hash value, signature `Sig`, address `ADRS`, and public seed `public_seed` (the root node is computed using the `XMSS_rootFromSig` method from Algorithm 13 [XMSS]).
3. Return `(node == root)`

Remark 22: XMSS does not specify any format for the storage of private and public keys. Currently, Botan serializes keys as plain byte arrays.

Remark 23: Botan uses 8 bytes to encode the leaf index, instead of 4 bytes as described in Section 4.1.8 in [XMSS].

Remark 24 (Private key management): The XMSS private key is stateful. Therefore, information about used OTS needs to be persisted *before* using the result of an XMSS_Signature_Operation. Following this rule ensures that, in case the signature generation is interrupted due to an error (i.e. application crash, hardware failure), the private key update must have been completed before using the signature. If the private key update terminates erroneously and the leaf index is not updated, also the signature based on this leaf index will not have been published. However, if a signature is published before a failed private key update, the same leaf index might be used repeatedly on application restart. This reuse of OTS compromises the security of XMSS.

Remark 25 (Sharing private keys): Even in the absence of soft- or hardware errors, which may disturb the process of persisting the private key, reuse of OTS can occur if multiple instances of the same private key exist within: (a) a single process, (b) multiple processes, or (c) multiple physical devices. To mitigate the risk of OTS reuse in the case of (a), the XMSS index registry is implemented as part of XMSS in Botan. This ensures that different threads cannot update the same private key independently, creating an inconsistent state of the private key unused leaf index property. However, users of the Botan's XMSS capabilities are still required to ensure that reuse of OTS in cases of (b), (c) is eliminated by their implementation. This may be achieved by either not sharing XMSS private keys across multiple processes/devices or by implementing a central key store that synchronizes read and write access to XMSS private keys.

9 Random Number Generators

Random number generators are used for different purposes inside the library, as explained in the former chapters, and can also be used by application developers. All functions in Botan that need random numbers take a reference to a random number generator instance as a parameter.

Random number generators in Botan include deterministic generators, such as HMAC_DRBG and AutoSeeded_RNG, system-specific generators such as System_RNG and hardware random number generators such as RDRAND_RNG.

All random number generators in Botan implement the `RandomNumberGenerator` interface. This interface provides the following important member functions:

- `randomize(output, length)`: Extracts *length* random bytes from the random number generator and writes the output to *output*.
- `add_entropy(input, length)`: Incorporates *length* bytes of entropy from the input buffer *input* into the random number generator's entropy pool.
- `randomize_with_input(output, output_len, input, input_len)`: Incorporates *input_len* bytes of entropy from the input buffer *input* into the random number generator's entropy pool and then extracts *output_len* random bytes from the random number generator and writes the output to *output*.
- `randomize_with_ts_input(output, output_len)`: Incorporates a 64 bit system timestamp and a 64 bit processor timestamp into the random number generator's entropy pool and then extracts *output_len* random bytes from the random number generator and writes the output to *output*.
- `reseed(entropy_sources, poll_bits, poll_timeout)`: Polls the *entropy_sources* for up to *poll_bits* bits of entropy or until the *poll_timeout* expires, calls `add_entropy()` on this random generator and returns an estimate of the number of bits collected. The default value for *poll_bits* is `BOTAN_RNG_RESEED_POLL_BITS`, which defaults to 128. The default value for *poll_timeout* is `BOTAN_RNG_RESEED_DEFAULT_TIMEOUT`, which defaults to 50 milliseconds.
- `reseed_from_rng(rng, poll_bits)`: Polls the *rng* for *poll_bits* bits of entropy and calls `add_entropy()` on this random generator. The default value for *poll_bits* is `BOTAN_RNG_RESEED_POLL_BITS`, which defaults to 128.

9.1 Deterministic Generators

There are two classes of random bit generators. Non-deterministic random bit generators are based on a physical process that is unpredictable. In contrast, deterministic random bit generators compute bits deterministically using a specific algorithm. Deterministic generators **must** be seeded with a seed of sufficiently high entropy. For the requirements on seed generation, see [TR021021, sec. 9.5].

9.1.1 HMAC_DRBG

HMAC_DRBG is a deterministic random bit generator specified in [SP80090A]. The `HMAC_DRBG` class derives from the `Stateful_RNG` base class, which provides such functionality as automatic reseeding after a defined interval and after a process fork. The HMAC_DRBG is provided in `src/lib/rng/hmac_drbg/hmac_drbg.cpp`, the Stateful_RNG in `src/lib/rng/stateful_rng/stateful_rng.cpp`.

HMAC_DRBG Instantiation

HMAC_DRBG can be instantiated with different types of entropy sources. Therefore, HMAC_DRBG provides five constructors.

1. No entropy sources, just the keyed hash function: This instance will not be able to seed and reseed itself. Seeding must be done by explicitly calling the function `initialize_with()`. If a fork is detected during calls to `randomize()`, the instance will throw an exception.
2. *underlying_rng*: An object implementing the `RandomNumberGenerator` interface used for seeding and reseeding. Automatic reseeding from *underlying_rng* will take place after reseed interval many requests or after a fork was detected.
3. *entropy_sources*: A collection of objects implementing the `Entropy_Source` interface used for seeding and reseeding. Automatic reseeding from *entropy_sources* will take place after reseed interval many requests or after a fork was detected.
4. *underlying_rng*, *entropy_sources*: An object implementing the `RandomNumberGenerator` interface and a collection of objects implementing the `Entropy_Source` interface both used for seeding and reseeding. Automatic reseeding from *underlying_rng* and *entropy_sources* will take place after reseed interval many requests or after a fork was detected.
5. No entropy sources, just the hash function name: This instance will not be able to seed and reseed itself. Seeding must be done by explicitly calling the function `initialize_with()`. If a fork is detected during calls to `randomize()`, the instance will throw an exception.

The first constructor is implemented as follows.

Input:

1. *prf*: An approved keyed hash function, e.g., HMAC(SHA-512).

Output:

1. An HMAC_DRBG instance

Steps:

1. Set *max_number_of_bytes_per_request* = 64*1024
2. Set Stateful_RNG.*reseed_counter* = 0

3. Set Stateful_RNG.*last_pid* = 0
4. Set *V* = 0x01 01..01 Comment: prf.*outlen* bits
5. Set *Key* = 0x00 00..00 Comment: prf.*outlen* bits

The second constructor is implemented as follows:

Input:

1. *prf*: An approved keyed hash function, e.g., HMAC(SHA-512).
2. *underlying_rng*: A random number generator used for seeding and reseeding.
3. *reseed_interval*: The reseed interval.
4. *max_number_of_bytes_per_request*: The maximum number of bytes per request.

Output: An HMAC_DRBG instance

Steps:

1. If (*max_number_of_bytes_per_request* = 0) or (*max_number_of_bytes_per_request* >= 64*1024), then Return “Invalid Argument”
2. Set Stateful_RNG.*underlying_rng* = *underlying_rng*
3. Set Stateful_RNG.*reseed_counter* = 0
4. Set Stateful_RNG.*last_pid* = 0
5. Set *V* = 0x01 01..01 Comment: prf.*outlen* bits
6. Set *Key* = 0x00 00..00 Comment: prf.*outlen* bits

The third constructor is implemented as follows:

Input:

1. *prf*: An approved keyed hash function, e.g., HMAC(SHA-512).
2. *entropy_sources*: A collection of entropy sources used the source for seeding and reseeding.
3. *reseed_interval*: The reseed interval.
4. *max_number_of_bytes_per_request*: The maximum number of bytes per request.

Output: An HMAC_DRBG instance

Steps:

1. If (*max_number_of_bytes_per_request* = 0) or (*max_number_of_bytes_per_request* >= 64*1024), then Return “Invalid Argument”
2. Set Stateful_RNG.*entropy_sources* = *entropy_sources*
3. Set Stateful_RNG.*reseed_counter* = 0
4. Set Stateful_RNG.*last_pid* = 0

5. Set *V* = 0x01 01..01 Comment: prf.*outlen* bits
6. Set *Key* = 0x00 00..00 Comment: prf.*outlen* bits

The third constructor is implemented as follows:

Input:

1. *prf*: An approved keyed hash function, e.g., HMAC(SHA-512).
2. *underlying_rng*: A random number generator used for seeding and reseeding.
3. *entropy_sources*: A collection of entropy sources used the source for seeding and reseeding.
4. *reseed_interval*: The reseed interval.
5. *max_number_of_bytes_per_request*: The maximum number of bytes per request.

Output: An HMAC_DRBG instance

Steps:

1. If (*max_number_of_bytes_per_request* = 0) or (*max_number_of_bytes_per_request* >= 64*1024), then Return “Invalid Argument”
2. Set Stateful_RNG.*underlying_rng* = *underlying_rng*
3. Set Stateful_RNG.*entropy_sources* = *entropy_sources*
4. Set Stateful_RNG.*reseed_counter* = 0
5. Set Stateful_RNG.*last_pid* = 0
6. Set *V* = 0x01 01..01 Comment: prf.*outlen* bits
7. Set *Key* = 0x00 00..00 Comment: prf.*outlen* bits

The fifth constructor is implemented as follows.

Input:

1. *hash*: A hash function name, e.g., SHA-512.

Output:

1. An HMAC_DRBG instance

Steps:

1. Set *max_number_of_bytes_per_request* = 64*1024
2. Set Stateful_RNG.*reseed_counter* = 0
3. Set Stateful_RNG.*last_pid* = 0
4. Set *V* = 0x01 01..01 Comment: prf.*outlen* bits
5. Set *Key* = 0x00 00..00 Comment: prf.*outlen* bits

Function security_level():

`security_level()` is a pure virtual function that must be implemented by classes derived from Stateful_RNG. It returns the security level of the DRBG. For HMAC_DRBG, the security level of the DRBG depends on the security level of the hash function used in the PRF, given in [SP80057-P1, Table 3]. For SHA-1, a maximum of 128 bits is supported, for SHA-224 and SHA-512/224 a maximum of 192 bits is supported and for SHA-256, SHA-512/256, SHA-384, SHA-512 and SHA3-512 a maximum security level of 256 bits is supported.

Function initialize_with():

Stateful_RNG's `initialize_with()` can be used to manually seed an HMAC_DRBG, e.g., if it has no entropy sources given during construction. `initialize_with()` adds the given entropy to HMAC_DRBG's entropy pool and resets the reseed counter if at `security_level()` entropy bytes were passed.

Input:

- *input*: The string of bits obtained from the entropy source.
- *input_len*: Length of *input* in bytes.

Output: None

Steps:

1. Call `add_entropy(input, input_len)`
2. If (`8*input_len >= security_level()`) then do set `reseed_counter = 1`

HMAC_DRBG Reseeding

HMAC_DRBG can be reseeded using the *add_entropy()* function, which internally calls the function *update()* to update the entropy pool.

Input:

1. *input*: The string of bits obtained from the entropy source.
2. *input_len*: Length of *input* in bytes.

Output: None

Steps:

1. Call *update(input, input_len)*

Function update():

The `update()` function resets the internal state values *V* and MAC *Key* with new values according to [SP80090A, section 10.1.2.2].

Input:

1. *input*: The string of bits obtained from the entropy source.

2. *input_len*: Length of *input* in bytes.

Output: None

Steps:

1. *K* = **HMAC**(*K*, *V* || 0x00 || *input*)
2. *V* = **HMAC**(*K*, *V*)
3. If (*input_len* > 0) then:
    1. *K* = **HMAC**(*K*, *V* || 0x01 || *input*)
    2. *V* = **HMAC**(*K*, *V*)

HMAC_DRBG Randomize

Random bytes can be requested from HMAC_DRBG using the `randomize()`, `randomize_with_input()` and `randomize_with_ts_input()` functions. `randomize()` takes an output buffer and the number of requested bytes while `randomize_with_input()` additionally takes entropy input that is added to the entropy pool by calling the `update()` before extracting random bytes from the pool. `randomize_with_ts_input()` queries a system clock timestamp, a processor timestamp, the PID and the reseed counter and passes this as additional entropy to `randomize_with_input()`. `randomize()` does not implement any logic, but simply calls `randomize_with_input()` with an entropy input buffer of length zero.

The `randomize_with_input()` function extracts the requested number of random bytes from the internal state value *V* using the PRF given during construction according to [SP80090A, section 10.1.2.5].

In contrast to [SP80090A, section 10.1.2.5], the `randomize_with_input()` will not output an error if a reseed is required, but instead perform an automatic reseed from the entropy source given during construction. Additionally, `randomize_with_input()` will also not output an error if *requested_number_of_bytes* > *max_number_of_bytes_per_request*, but instead treat such calls as if multiple subsequent calls to `randomize_with_input()` were made.

`randomize_with_input()` also attempts to detect a fork of the process on Unix systems by comparing the process ID between calls. If the process ID changed, it will automatically perform a reseed. Seeding and reseeding is done in the Stateful_RNG's `reseed_check()` member function.

`randomize_with_input()` is implemented as follows.

Input:

1. *output*: Output buffer to hold the requested random bytes.
2. *output_len*: Number of requested random bytes.
3. *input*: A string of bits obtained from an entropy source to be mixed into the entropy pool before extraction.

4. *input_len*: Number of bytes in *input*.

Output:

1. *returned_bytes*: The pseudorandom bits to be returned to the consuming application.

Steps:

1. While (*output_len* > 0) do:
    1. Set *this_req* = **min**(*max_number_of_bytes_per_request*, *output_len*)
    2. Set *requested_number_of_bytes* = *output_len* - *this_req*
    3. Call Stateful_RNG's `reseed_check()`
    4. If **len**(*additional_input*) != 0, then (*Key*, *V*) = **HMAC_DRBG_Update**(*additional_input*, *Key*, *V*)
    5. While (*this_req* > 0) do:
        1. *to_copy* = **min**(*this_req*, **len**(*V*))
        2. *V* = **HMAC**(*Key*, *V*)
        3. *returned_bytes* = *returned_bytes* || **leftmost**(*V*, *to_copy*)
    6. **HMAC_DRBG_Update**(*additional_input*, *Key*, *V*)
2. Return (SUCCESS, *returned_bytes*, *Key*, *V*, *reseed_counter*)

`randomize_with_ts_input()` incorporates a system clock timestamp in nanoseconds precision, a 64 bit processor timestamp, using QueryPerformanceCounter's QuadPart value on Windows and an inline assembly to query the processor counter on other platforms, the 32 bit process ID (PID) and the 64 bit reseed counter into additional input of 24 bytes length. It is implemented as follows.

Input:

1. *output*: Output buffer to hold the requested random bytes.
2. *output_len*: Number of requested random bytes.

Output: None

Steps:

1. Add a 64 bit system clock timestamp to `additional_input`
2. Add a 64 bit processor timestamp to `additional_input`
3. Add the 32 bit process ID to `additional_input`
4. Add 64 bit reseed counter to `additional_input`
5. Call `randomize_with_input(output, output_len, additional_input, sizeof(additional_input))`

Function Stateful_RNG::reseed_check():

Stateful_RNG's `reseed_check()` initially seeds HMAC_DRBG and reseeds HMAC_DRBG if a fork occurred in the calling process or if the reseed interval is exceeded. If a seed or reseed is required, it requests `security_level()` bits from the entropy sources. `reseed_check()` is implemented as follows.

Input: None

Output: None

Steps:

1. Set *cur_pid* = **Get_Current_Process_ID()**
2. If (*reseed_counter* = 0) Or ((*last_pid* > 0) And (*cur_pid* != *last_pid*)) Or ((*reseed_interval* > 0) And (*reseed_counter* >= *reseed_interval*)) then do:
    1. Set *reseed_counter* = 0
    2. Set *last_pid* = *cur_pid*
    3. If the HMAC_DRBG was constructed with at least an underlying RNG as an entropy source, `security_level()` bits of entropy are requested from the underlying RNG and added to HMAC_DRBG's entropy pool by calling Stateful_RNG's `reseed_from_rng()`, which works as follows:
        1. Request `security_level()` bits of entropy from the underlying RNG by calling it's `randomize()` member function, which returns a buffer and an entropy estimation
        2. Mix the returned entropy bytes into HMAC_DRBG's entropy pool by calling it's `add_entropy()` member function (both steps via an indirection to the RandomNumberGenerator's `reseed_from_rng()` member function)
        3. If the returned entropy estimation is equal to or exceeds `security_level()` then:
            1. Set *reseed_counter* = 1
    4. If the HMAC_DRBG was constructed with at least a collection of entropy sources, `security_level()` bits of entropy are requested from the underlying RNG and added to HMAC_DRBG's entropy pool by calling Stateful_RNG's `reseed_from_rng()`, which works as follows:
        1. Request `security_level()` bits of entropy from the entropy sources by calling Entropy_Sources' `poll()` member function, which mixes entropy bytes into HMAC_DRBG's entropy pool by calling it's `add_entropy()` member function and returning the number of bits collected; `poll()` takes a timeout value in milliseconds after which polling of the entropy sources is stopped, the value used here is `BOTAN_RNG_RESEED_POLL_BITS`, which defaults to 50 milliseconds

2. If the returned number of bits collected is equal to or exceeds `security_level()` bits then:
    1. Set *reseed_counter* = 1
    2. Return the number of bits collected
5. If (*reseed_counter* = 0) then do:
    1. If ((*last_pid* > 0) And (*cur_pid* != *last_pid*)) then output “Fork detected, but unable to reseed” Else output “PRNG not seeded: HMAC_DRBG”

3. Else do:
    1. If (*reseed_counter* = 0) then output “RNG not seeded”
    2. *reseed_counter* = *reseed_counter* + 1

Conclusion: HMAC_DRBG conforms to [SP80090A], although it differs from the standard in two ways: It automatically reseeds if required instead of outputting an error in this case and it outputs random bytes even if the requested number of bytes is greater than the max_number_of_bytes_per_request parameter permits. In both cases though, the internal state is updated with fresh entropy if required and thus the security is ensured.

9.1.2 AutoSeeded_RNG

AutoSeeded_RNG is a random number generator that is automatically seeded. AutoSeeded_RNG internally uses the HMAC_DRBG. If no entropy source is explicitly given, AutoSeeded_RNG uses the System_RNG as the entropy source for HMAC_DRBG. If the System_RNG is not available, that means it is not part of the library build because it was explicitly disabled manually or because it is not available[4] for this platform, it uses a default[5] set of entropy sources. As the name implies, AutoSeeded_RNG is automatically seeded (and reseeded) from these sources. The AutoSeeded_RNG is provided in `src/lib/rng/auto_rng/auto_rng.cpp`.

9.2 System Generators

System_RNG provides access to an operating system provided random generator. On Windows, this is the CryptGenRandom API, on all Unix platforms this is /dev/urandom. The latter can be customized using the BOTAN_SYSTEM_RNG_DEVICE build.h variable, for example, it could be set to /dev/random instead. In the following, the Instantiate, Add_Entropy and Generate functions of both implementations are specified. The System_RNG is provided in `src/lib/rng/system_rng/system_rng.cpp`.

CryptGenRandom

Construction

Input:

4 Note that the System_RNG is available on most platforms, including Android, BSD, Cygwin, Darwin, Linux, MinGW and Windows.

5 "timestamp", "rdseed", "rdrand", "proc_info", "darwin_secrandom", "dev_random", "win32_cryptoapi", "proc_walk", "system_stats"

1. provider_type: A null-terminated string that contains the name of the CSP to be used (default: PROV_RSA_FULL).

Output:

1. provider_handle: Handle to the CSP.

Steps:

1. If (**CryptAcquireContext**(&*provider_handle*, 0, 0, *provider_type*, CRYPT_VERIFYCONTEXT) = 0) then output “System_RNG failed to acquire crypto provider”

Reseeding

Input:

1. *provider_handle*: Handle to the CSP.
2. *input*: Additional input received from the consuming application.
3. *len*: Size of *input* in bytes.

Steps:

1. **CryptGenRandom**(*provider_handle*, *len*, *input*)

Randomize

Input:

1. *provider_handle*: Handle to the CSP.
2. *buf*: The buffer receiving the pseudorandom bytes.
3. *len*: The number of pseudorandom bytes to be returned.

Output:

1. *buf*: The pseudorandom bits to be returned to the consuming application.

Steps:

1. **CryptGenRandom**(*provider_handle*, *len*, *buf*)

/dev/[u]random

Construction

Instantiate first attempts to open a file descriptor to the system RNG device in read-write mode, so additional entropy can be added using Add_Entropy later. In some, especially sandboxed, systems though, attempting to open in read-write mode will fail. In this case, the file descriptor will be opened in read-only mode as a fallback, allowing to get random bytes from the system RNG while preventing Add_Entropy from working.

Input:

1. *system_rng_device*: A null-terminated string that contains the name of the system RNG device to be used (default: /dev/urandom).

Output:

1. *fd*: File descriptor to the RNG device.

Steps:

1. *fd* = **open**(*system_rng_device*, O_RDWD | O_NOCTTY)
2. If (*fd* < 0) then do fd = **open**(*system_rng_device*, O_RDONLY | O_NOCTTY)
3. If (*fd* < 0) then output “System_RNG failed to open RNG device”

Reseeding

Input:

1. *fd*: File descriptor to the RNG device.
2. *input*: Additional input received from the consuming application.
3. *len*: Size of *input* in bytes.

Steps:

1. While (*len* > 0) do:
    1. *got* = **write**(*fd*, *additional_entropy*, *len*)
    2. If (*got* < 0) then do:
        1. If (errno = EINTR) do Continue
        2. If (errno = EPERM) do Return
        3. Output “System_RNG write failed error”
    3. *input += got*
    4. *len = len - got*

Randomize

Input:

1. *fd*: File descriptor to the RNG device.
1. *buf*: The buffer receiving the pseudorandom bytes.
4. *len*: The number of pseudorandom bytes to be returned.

Output:

1. *buffer*: The pseudorandom bits to be returned to the consuming application.
2. While (len > 0) do:
    1. *got* = **read**(*fd*, *buf*, lens)

2. If (*got* < 0) then do:
    1. If (errno = EINTR) do Continue
    2. Output “System_RNG read failed error”
3. If (*got* = 0) then output “System_RNG EOF on device”
4. *buf* += *got*
5. *len* = *len* – *got*

9.3 Hardware Generators

9.3.1 PKCS11_RNG

PKCS11_RNG is a random generator that uses the PKCS#11 interface to retrieve random bytes from a hardware security module (HSM) supporting the PKCS#11 standard, e.g., a smartcard. The PKCS11_RNG is provided in `src/lib/prov/pkcs11/pkcs11_randomgenerator.cpp`.

Construction

Input:

1. *session*: A PKCS#11 session object with the HSM.

Steps:

1. Store a reference to the session as *m_session* = *session*

Reseeding

Input:

1. *in*: Additional input received from the consuming application.
2. *length*: Length of *in* in bytes.

Steps:

1. **C_SeedRandom**(*m_session.get().module(), in, length*)

Randomize

Input:

1. *output*: The buffer receiving the pseudorandom bytes.
2. *length*: The number of pseudorandom bytes to be returned.

Output:

1. *output*: The pseudorandom bits to be returned to the consuming application.

Steps:

1. **C_GenerateRandom**(*m_session.get().handle(), in, length*)

9.3.2 RDRAND_RNG

RDRAND_RNG is a random generator that uses the rdrand instruction on modern Intel processors to retrieve random bytes. As there is no way to add entropy to the rdrand entropy pool, `add_entropy()` is a no operation. The RDRAND_RNG is provided in `src/lib/entropy/rdrand/rdrand.cpp`.

Construction

Steps:

1. If (**has_cpuid_bit**(CPUID_RDRAND_BIT) != true) then output “Current CPU does not support RDRAND instruction”

Reseeding

Not implemented.

Randomize

Input:

1. *out*: The buffer receiving the pseudorandom bytes.
2. *out_len*: The number of pseudorandom bytes to be returned.

Output:

1. *out*: The pseudorandom bits to be returned to the consuming application.

Steps:

1. While (*out_len* >= 4) do:
    1. *r* = **rdrand**()
    2. **store_le**(*r*, *buffer*)
    3. *out_len* = *out_len* - 4
2. If (*out_len* > 0) then do:
    1. *r* = **rdrand**()
    2. For (i = 0..*out_len*-1) do:
    3. *buffer*[i] = **get_byte**(*i*, *r*)

Helper Functions

rdrand(): Get 32 random bits from CPU using the rdrand instruction

Output: 32 bit unsigned integer

Steps:

1. *ok* = false

2. r = **rdrand_status**(ok)
3. While (*ok* != true) do:
    1. *r* = **rdrand_status**(*ok*)
4. Return *r*

rdrand_status(): Try to get 32 random bits from the CPU using the rdrand instruction

Output:

1. *retries*: How often the rdrand instruction should be retried in case of failure. Note that Intel guarantees that a call to the rdrand instruction succeeds after at most 10 tries. The default is 10, it can be customized using the BOTAN_ENTROPY_RDRAND_RETRIES variable.
2. *ok*: A boolean indicating whether the rdrand instruction succeeded or not.
3. *returned_bits*: An unsigned 32 bit integer returned from the rdrand instruction.

Steps:

1. *ok* = false
2. *returned_bits* = 0
3. For (i = 0..*retries*-1) do:
    1. cf = _rdrand32_step(&*returned_bits*)
    2. If (cf = 1) then do:
        1. ok = true
        2. Return *returned_bits*

Remark 26: On GNU GCC, instead an inline assembly for *rdrand %eax* is used in step 3.1.

10 Entropy Sources

An accompanying programming interface to the RandomNumberGenerator interface is the Entropy_Source interface. Objects of this class can be used to feed entropy in the pool of a `RandomNumberGenerator` object. There are several entropy sources available in Botan. All of these implement the abstract `Entropy_Source` interface. The interface consists only of a few the methods:

- std::unique_ptr<Entropy_Source> create(type): Returns a new entropy source of a particular type, or NULL if it's not available.
- `name()`: Returns the name of the entropy source.
- `poll(rng)`: Performs an entropy gathering poll and adds this entropy to the random number generator `rng`. Returns a conservative estimate in bits of the actual entropy added to the `rng` during this poll.

10.1 Available Sources

The following entropy sources are currently defined:

- `Win32_CAPI_EntropySource`: Only available on Windows and uses the system random number generator from the PROV_RSA_FULL Cryptographic Service Provider to gather entropy via `CryptGenRandom()`. The number of bytes that are polled can be adjusted by modifying the `BOTAN_SYSTEM_RNG_POLL_REQUEST` macro in the build.h file. Default is 64 bytes.
- `Darwin_SecRandom`: Only available on iOS and macOS and uses the system random number generator to gather entropy via `SecRandomCopyBytes()`. The number of bytes that are polled can be adjusted by modifying the `BOTAN_SYSTEM_RNG_POLL_REQUEST` macro in the build.h file. Default is 64 bytes.
- `Device_EntropySource`: This entropy source is available on the following operating systems: Aix, Android, Cygwin, Darwin, Dragonfly, Freebsd, Haiku, Hpux, Hurd, Irix, Linux, Netbsd, Openbsd, Qnx and Solaris. In default configuration, it tries to gather entropy from "`/dev/urandom`", "`/dev/random`" and "`/dev/srandom`". It blocks at most 20 milliseconds waiting for one of these devices to become ready. The timeout can be adjusted in build.h by changing the value of `BOTAN_SYSTEM_RNG_POLL_TIMEOUT_MS`. The number of bytes that are polled from each available device can be adjusted by modifying the `BOTAN_SYSTEM_RNG_POLL_REQUEST` macro in the build.h file. Default is 64 bytes.
- `ProcWalking_EntropySource`: This entropy source is available on the following operating systems: Aix, Android, Cygwin, Darwin, Dragonfly, Freebsd, Hpux, Hurd, Irix, Linux, Netbsd, Openbsd, Qnx and Solaris. The contents of /proc are provided as entropy inputs to the RNG. At maximum 2048 files are read per poll and per file a maximum of 4 KB are read. Each file counts as 4 bit of entropy regardless of the file size. The poll stops when more than 128 estimated entropy bits are collected.
- `Intel_Rdrand`: This entropy source is available on CPU's supporting the *RdRand* instruction. It polls entropy from the on-chip hardware random number generator. In default

configuration 128 bytes. Botan does not trust *RdRand* so `Intel_Rdrand::poll()` always returns 0.

- `Intel_Rdseed`: This entropy source is available on CPU's supporting the *RdSeed* instruction. It polls entropy from the on-chip hardware random number generator. Internally it polls RdSeed 32 times, whereas each poll generates 32 bit of entropy. Each poll is retried on failure at most 20 times because RdSeed is not guaranteed to generate a random number within a specific number of retries. In default configuration if each try was successful 128 bytes are gathered. Botan does not trust *RdSeed* so `Intel_Rdseed::poll()` always returns 0.
- `Win32_EntropySource`: This entropy source is available on Windows, Cygwin and MinGW.
  - First it gathers entropy from the following Win32 API functions. These inputs are not counted.
    - `GetTickCount()`
    - `GetMessagePos()`
    - `GetMessageTime()`
    - `GetInputState()`
    - `GetCurrentProcessId()`
    - `GetCurrentThreadId()`
    - `GetSystemInfo()`
    - `GlobalMemoryStatusEx()`
    - `GetCursorPos()`
    - `GetCaretPos()`
  - Afterwards it creates snapshots of modules, processes and threads. Each snapshot counts as 4 bits of entropy. If more than 256 bits are collected the poll is finished, otherwise snapshots of heap objects are fed into the entropy pool until the 256 bit limit is reached.

10.2 The Entropy_Sources class

This class manages the available sources. The static method

```
Entropy_Sources& Entropy_Sources::global_sources()
```

returns a reference to the default sources which are defined in build.h as follows:

```
#define BOTAN_ENTROPY_DEFAULT_SOURCES \
   { "rdseed", "rdrand", "darwin_secrandom", "dev_random", \
    "win32_cryptoapi", "proc_walk", "system_stats" }
```

These sources are used by the `AutoSeeded_RNG` if no system RNG is available.

RNGs can use an underlying RNG, entropy sources or both for reseeding. If they use the entropy sources they call the `poll()` method of the available and configured sources:

```
size_t Entropy_Sources::poll (RandomNumberGenerator& rng, size_t
poll_bits, std::chrono::milliseconds timeout)
```

All sources that are configured in the `Entropy_Sources` object are polled until `poll_bits` entropy bits are collected or the timeout is reached. So, the order in which the Entropy Sources were added to the `Entropy_Sources` is important here. The collected bits are returned after the poll is finished.

11 X.509 Path Validation

The following section describes how Botan performs X.509 certificate path validation, as defined in [RFC5280]. X.509 path validation is implemented in `src/lib/cert/x509/x509path.cpp`. Figure 1 shows how X.509 path validation is organized.

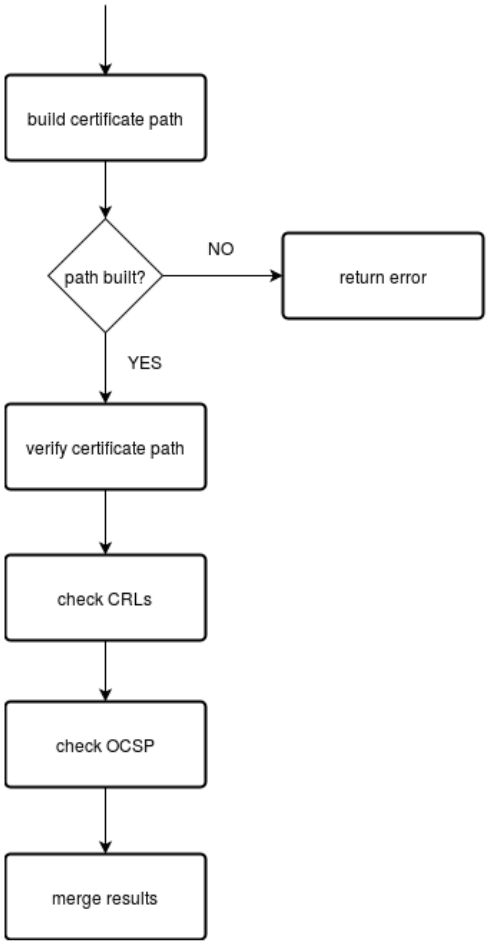

Figure 1: X.509 Path Validation in Botan

First, a certificate path is built from the end entity certificate to a trusted root certificate. If a path could not be build path validation returns with an error. If a path could be successfully built, the built certificate path is validated by checking signatures, validity periods, key usages and other extensions. From now on, any failures in the path validation are recorded and do not result in an early exit. Instead, Botan collects all error codes and merges them in the last step. Before, CRLs and OCSP responses are checked on the certificate path.

Botan provides four different variants of the function `x509_path_validate()` that differ in the first parameter, either a single end entity certificate or a certificate chain, and in the third parameter, either a single `Certificate_Store` or a list of `Certificate_Store` pointers. The variants taking a single element, either end entity certificate or `Certificate_Store` or both, just create a list of elements with the one element and then call the variant of `x509_path_validate()` that takes both a certificate chain to validate and a list of

`Certificate_Store` pointers. Thus, in the following, only the variant taking a certificate chain and a list of `Certificate_Store` pointers is described in detail.

All variants of `x509_path_validate()` take an object of class `Path_Validation_Restrictions`. This class can be used to enforce restrictions on path validation, namely the following:

- Require revocation information, e.g., from CRLs or OCSP
- Make OCSP requests for all CAs as well as the end entity certificate or only for the end entity certificate
- Only allow specific hash functions used in all certificates in the chain
- Require a minimum key strength for all keys in the certificate chain

All variants of `x509_path_validate()` take an object of class `Certificate_Store`. Such a `Certificate_Store` is used to store certificates of trusted parties, e.g., CAs, and CRLs used during path validation.

All variants of `x509_path_validate()` return an object of type `Path_Validation_Result`. `Path_Validation_Result` encapsulates the result of an X.509 path validation. It stores, amongst other information, the certificate verification status codes of each certificate in the chain to be validated. All the status codes can be queried by calling the `all_statuses()` member function, while the worst error code can be retrieved by calling the `result()` member function. To quickly check whether an X.509 path validation was successful the member function `successful_validation()` will return a boolean. All certificate validation status codes are defined in `src/lib/cert/cert_status.h`.

Input:

- `end_certs`: A list of certificates (certificate chain) of size `n` certificates to validate
- `restrictions`: Path validation restrictions
- `certstores`: List of certificate stores that contain trusted certificates
- `hostname`: The hostname of the peer (optional)
- `usage`: The usage type of the end entity certificate, one of [TLS Server, TLS Client, CA, OCSP Responder] (optional)
- `validation_time`: Reference time to use for validation (default: current system clock value)
- `ocsp_timeout`: Timeout for OCSP requests in milliseconds (0 means OCSP checks disabled; default: 0)
- `ocsp_responses`: Additional OCSP responses to consider

Output:

- An object of type `Path_Validation_Result`

Steps:

1. If `end_certs` is empty, throw an *InvalidArgument* exception // nothing to validate
2. Set `end_entity = end_certs[0]`
3. Set `end_entity_extra = end_certs[1] .. end_certs[n-1]`
4. Set `path_building_result = build_certificate_path(cert_path, trusted_roots, end_entity, end_entity_extra)`
5. If (`path_building_result` != Certificate_Status_Code::OK), then do Return Path_Validation_Result(`path_building_result`) // no chain to a trusted self-signed root
6. Set `status = check_chain(cert_path, ref_time, hostname, usage, restrictions.minimum_key_strength(), restrictions.trusted_hashes())` // check the certificate chain, but not rev. data
7. Set `crl_status = check_crl(cert_path, trusted_roots, ref_time)`
8. If (`oscp_responses.size()` > 0) then do:
   a) `ocsp_status = check_ocsp(cert_path, ocsp_resp, trusted_roots, ref_time)` // check additional OCSP responses
9. If (`ocsp_status.empty()` AND `ocsp_timeout` != 0) then do:
   a) `ocsp_status = check_ocsp_online(cert_path, trusted_roots, ref_time, ocsp_timeout, restrictions.ocsp_all_intermediates())` // check OCSP using online HTTP access
10. Call `merge_revocation_status(status, crl_status, ocsp_status, restrictions_require_revocation_information(), restrictions.ocsp_all_intermediates())` // merge all revocation information
11. Return Path_Validation_Result(status, cert_path)

Function build_certificate_path():

The `build_certificate_path()` function tries to build a certificate path from the end entity certificate given to a trusted root in one of the trusted certificate stores given. It returns the certificate path built and a certificate status code.

Input:

- `cert_path`: Holds the certificate path built (output parameter).
- `trusted_certstores`: List of certificate stores that contain trusted certificates.
- `end_entity`: The end entity certificate to be validated.
- `end_entity_extra`: An optional list of additional untrusted certificates for path building.

Output:

- The certificate path built
- A certificate status code: OK if path could be built, CANNOT_ESTABLISH_TRUST otherwise

Steps:

1. If the `end_entity` certificate is self-signed, Return CANNOT_ESTABLISH_TRUST
2. Append `end_entity` to `cert_path`
3. Append the SHA-256 fingerprint of `end_entity` to `certs_seen`
4. Load all certificates from `end_entity_extra` into a CertificateStore `ee_extras`
5. Iterate until we reach a root or cannot find the certificate issuer as follows:
   a) Set `trusted_issuer = false`
   b) Search for the issuer certificate `issuer` of the last certificate in `cert_path` and in all certificate stores in `trusted_certstores`; if found, set `trusted_issuer = true` and go to the next step
   c) If no issuer certificate was found in the previous step, search for the issuer certificate `issuer` of the last certificate in `cert_path` in `ee_extras`; if not found, Return CERT_ISSUER_NOT_FOUND
   d) If `certs_seen` contains the SHA-256 fingerprint of `issuer`, Return CERT_CHAIN_LOOP
   e) Append `issuer`'s SHA-256 fingerprint to `certs_seen`
   f) Append `issuer` to `cert_path`
   g) If `issuer` is self-signed then do: If(`trusted_issuer = true`), then do Return OK Else Return CANNOT_ESTABLISH_TRUST

Function check_chain():

The `check_chain()` function checks the certificate chain given for validity by walking up the certificate path and checking for validity period, signatures and extensions, e.g., key usage. It returns a list of sets of certificate status codes, each entry in the list contains the status codes for each certificate in the chain.

Input:

- `cert_path`: The certificate chain to check, of size `n`.
- `ref_time`: The time to perform validation against.
- `hostname`: The hostname to perform validation against.
- `usage`: End entity certificate usage to perform validation against.

- `min_signature_algo_strength`: Minimum strength of signatures in the certificate chain, given in symmetric key bits, e.g., 80 allows 1024 bit RSA and SHA-1, 110 allows 2048 bit RSA and SHA-2, using 128 requires ECC (P-256) or ~3000 bit RSA keys.
- `trusted_hashes`: a set of trusted hash functions in the certificate chain, an empty list means any known hash function is accepted.

Output:

- `cert_status`: A list of sets of Certificate_Status_Code.

Steps:

1. If `hostname` is given and `cert_path[0]` does not contain a match for `hostname` according to [RFC6125], Append Certificate_Status_Codes::CERT_NAME_NOMATCH to `cert_status[0]` // see function `matches_dns_name()` below
2. If `usage` is given and `cert_path[0]` does not contain key usage and extended key usage bits according to [RFC5280], sec. 4.2.1.12, Append INVALID_USAGE to `cert_status[0]`
3. For `i = 0...n-1` in `cert_path` do:
   a) Set `at_self_signed_root = (i == cert_path.size() - 1)` // last certificate in the chain?
   b) Set `subject = cert_path[i]`
   c) If (`at_self_signed_root = true` AND `cert_path[i]` is self-signed) then do Append CHAIN_LACKS_TRUST_ROOT to `cert_status[i]`
   d) If (issuer DN of `subject` NOT EQUAL TO subject DN of `issuer`) Append CHAIN_NAME_MISMATCH to `cert_status[i]`
   e) If `validation_time` < `cert_path[i]`'s *valid from* field, Append CERT_NOT_YET_VALID to `cert_status[i]`
   f) If `validation_time` > `cert_path[i]`'s *valid until* field, Append CERT_HAS_EXPIRED to `cert_status[i]`
   g) If `issuer` does not have the BasicConstraints.CA bit set AND `cert_path` contains more than one certificate, Append CA_CERT_NOT_FOR_CERT_ISSUER to `cert_status[i]`
   h) If (`issuer`'s BasicConstraints.pathLenConstraint < `i`) then do Append CERT_CHAIN_TOO_LONG to `cert_status[i]`
   i) If the `issuer` public key cannot be loaded from the `issuer` certificate, then do Append CERT_PUBKEY_INVALID to `cert_status[i]` and continue with step l)
   j) If the signature on `subject` can not be verified using `issuer`'s public key, Append SIGNATURE_ERROR to `cert_status[i]`
   k) If `issuer`'s public key strength < `min_signature_algo_strength` then do

Append SIGNATURE_METHOD_TOO_WEAK to `cert_status[i]`

l) If (`trusted_hashes` is not empty AND `at_self_signed_root = false` AND the hash function used in `subject` IS NOT IN `trusted_hashes`) then do Append UNTRUSTED_HASH to `cert_status[i]` // ignore untrusted hashes on self-signed root certs

m) Check all other certificate extensions in `subject`:

1. If the X.509 name constraints in `subject` do not match according to [RFC5280] section 4.2.1.10., Append NAME_CONSTRAINT_ERROR to `cert_status[i]`

4. Return `cert_status`

Function matches_dns_name():

The function `matches_dns_name()` of class X509_Certificate checks if a given hostname is present in the certificate's subject distinguished name, according to [RFC6125]. Note that it internally calls the host_wildcard_match() function, which's functionality is presented here as if it was contained in matches_dns_name() for readability.

Input:

- `name`: The hostname to check.

Output: true, if the hostname is present in the subject distinguished name, false otherwise

Steps:

1. If `name` is empty, Return false
2. Set `issued_names` = all entries in the *DNS* field of subject DN
3. If (`issued_names` is empty) then do set `issued_names` = all entries in the *CN* field of subject DN // fall back to CN only if no DNS names are set, RFC 6125 sec. 6.4.4
4. For `i=0..n` do:
   a) If (`issued_names[i] = name`) then do Return true
   b) If (`issued_names[i]` begins with "*.") then do:
      i. If `name` does not contain a dot "." character or contains only one dot character that is the last character, then do Return false
      ii. If `name` and `issues_names[i]` equal after the occurrence of the dot character, then do Return true
5. Return false

Function check_crl():

The `check_crl()` function checks certificate revocation lists (CRLs) for revocation data for the certificates in the given certificate chain (that was already validated by the `check_chain()` function). It returns a list of sets of certificate status codes, each entry in the list contains the status

codes for each certificate in the chain. Note than in the code, two functions `check_crl()` are present that call each other, whereas in the following description, both are combined into one function for readability.

Input:

- `cert_path`: The certificate chain to check, of size `n`.
- `certstores`: List of certificate stores that contain trusted certificates and CRLs.
- `ref_time`: The time to perform validation against.

Output:

- `status`: A list of sets of Certificate_Status_Code.

Steps:

1. If (`cert_path` is empty) throw Invalid_Argument exception
2. If (`certstores` is empty) throw Invalid_Argument exception
3. Try to find a CRL for each certificate in `cert_path` in each certificate store in `certstores` and Append them to `crls`
4. For `i=0..n-1` in `cert_path` do:
   a) If `crls[i]` does not contain a CRL then do continue with the next `i`
   b) Set `subject = cert_path[i]`
   c) Set `ca = cert_path[i+1]`
   d) If ca does not have the CRL_SIGN key usage bit set, Append CA_CERT_NOT_FOR_CRL_ISSUER to `cert_status[i]`
   e) If `ref_time` < `crls[i]` *valid from* field, Append CRL_NOT_YET_VALID to `cert_status[i]`
   f) If `ref_time` > crls[i] *valid until* field, Append CRL_HAS_EXPIRED to `cert_status[i]`
   g) If the signature on the `crls[i]` cannot be verified with `ca`'s public key, Append CRL_BAD_SIGNATURE to `cert_status[i]`
   h) Append VALID_CRL_CHECKED to `cert_status[i]`
   i) If crls[i] lists subject as REVOKED, Append CERT_IS_REVOKED to `cert_status[i]`
5. Remove all empty sets from `cert_status`
6. Return `cert_status`

Function check_ocsp():

The function `check_ocsp()` checks the given OCSP responses for a given certificate chain. It

returns a list of sets of certificate status codes, each entry in the list contains the status codes for each certificate in the chain.

Input:

- `cert_path`: The certificate chain to check, of size `n`.
- `ocsp_responses`: OCSP responses to check.
- `certstores`: List of certificate stores that contain trusted certificates and CRLs.
- `ref_time`: The time to perform validation against.

Output:

- `cert_status`: A list of sets of Certificate_Status_Code.

Steps:

1. If (`cert_path` is empty) throw Invalid_Argument exception
2. For `i=0..n-1` in `cert_path` do:
    1. Set `subject = cert_path[i]`
    2. Set `ca = cert_path[i+1]`
    3. If `ocsp_respones[i]` does not contain an OCSP response, then do continue with the next `i`
    4. Check the signature on the `ocsp_responses[i]` by calling `check_signature(trusted_certstores, cert_path)` on `oscp_responses[i]`
        1. If it returned OCSP_SIGNATURE_OK then do Append the OCSP status returned by calling `status_for(ca, subject, ref_time)` on `ocsp_responses[i]` to `cert_status[i]`
        2. Else Append the return value of `check_signature()` to `cert_status[i]`
3. Remove all empty sets from `cert_status`
4. Return `cert_status`

Function check_signature():

The function `check_signature()` of the `OCSP_Response` class verifies the signature on the OCSP response. The issuer's public key is looked up in the list of trusted certificates in a given list of certificate stores, in the already validation certificate chain given and last but not least in the certificate chain as part of the OCSP response. If an issuer could not be found, the function returns the OCSP_ISSUER_NOT_FOUND certificate status code. Internally, `check_signature()` calls another function `verify_signature()` that performs the actual signature verification using the issuer certificate found. To improve readability, `verify_signature()`'s functionality is specified below as if it was part of `check_signature()`, starting from step 6).

Input:

- `trusted_roots`: List of certificate stores that contain trusted certificates.
- `ee_cert_path`: The certificate chain validated, of size `n`.

Output:

- A certificate status code.

Steps:

1. Look for the OCSP response's issuer certificate in `trusted_roots` and if found, set `signing_cert` to it and continue with step 5)
2. Look for the OCSP response's issuer certificate in `ee_cert_path` and if found, set `signing_cert` to it and continue with step 5)
3. Look for the OCSP response's issuer certificate in the optional list of certificates sent with the OCSP response and if found, set `signing_cert` to it and continue with step 5)
4. If the issuer certificate could not be found eventually, Return OCSP_ISSUER_NOT_FOUND
5. If `signing_cert`'s key usage does not contain crlSign AND `signing_cert`'s extended key usage does not contain OCSPSigning then do Return OCSP_RESPONSE_MISSING_KEYUSAGE
6. If the signature algorithm sent in the OCSP response does not match the signature algorithm in signing_cert's certificate, then do Return OCSP_RESPONSE_INVALID
7. If the signature on the OCSP response cannot be verified using signing_cert's public key then do Return OCSP_SIGNATURE_ERROR, otherwise Return OCSP_SIGNATURE_OK

Function status_for():

The function `status_for()` of the `OCSP_Response` class searches for the OCSP response for a given issuer and subject certificate and returns an appropriate OCSP status code.

Input:

- `issuer`: The issuer certificate of the OCSP response.
- `subject`: The subject certificate.
- `ref_time`: The time to perform validation against.

Output:

- An OCSP status code, one of CERT_IS_REVOKED, OCSP_NOT_YET_VALID, OCSP_HAS_EXPIRED, OCSP_RESPONSE_GOOD, OCSP_BAD_STATUS or OCSP_CERT_NOT_LISTED.

Steps:

1. For each SingleResponse in the OCSP response do:

a) If the SingleResponse is not for `issuer` and `subject`, continue with step 2)

b) If the SingleResponse CertStatus is *revoked*, then do Return CERT_IS_REVOKED

c) If the SingleResponse's thisUpdate value > ref_time, then do Return OCSP_NOT_YET_VALID

d) If the SingleResponse contains the nextUpdate field and the ref_time > nextUpdate value, then do Return OCSP_HAS_EXPIRED

e) If the SingleResponse CertStatus is *good* ,then do Return OCSP_RESPONSE_GOOD

f) If the SingleResponse CertStatus is *unknown,* then do Return OCSP_BAD_STATUS

2. Return OCSP_CERT_NOT_LISTED

Function check_ocsp_online():

The function `check_ocsp_online()` checks the OCSP status for a given certificate chain by making HTTP requests. It returns a list of sets of certificate status codes, each entry in the list contains the status codes for each certificate in the chain. Internally, it fetches all OCSP responses and then passes them to `check_ocsp()`.

Input:

- `cert_path`: The certificate chain to check, of size `n`.
- `trusted_certstores`: List of certificate stores that contain trusted certificates.
- `ref_time`: The time to perform validation against.
- `timeout`: Timeout after which a HTTP request should time out.
- `ocsp_check_intermediate_CAs`: If true also performs OCSP on any intermediate certificates, if false only on the end entity certificate.

Output:

- `cert_status`: A list of sets of Certificate_Status_Code.

Steps:

1. If (`cert_path` is empty) throw Invalid_Argument exception
2. Set to_ocsp = 1
3. If (oscp_check_intermediate_CAs = true) then do set to_ocsp = cert_path.size()-1
4. If (cert_path.size() = 1) then do set to_ocsp = 0
5. For i=0..to_ocsp-1 do:

   a) Set `subject = cert_path[i]`

   b) Set `issuer = cert_path[i+1]`

   c) If subject's OCSP responder field is not empty then do:

   i. Make a OCSP request via HTTP for `issuer` and `subject` using `timeout` and Append the OCSP response to `ocsp_responses`

6. Return the value returned by calling `check_ocsp(cert_path, ocsp_responses, trusted_certstores, ref_time)`

Function merge_revocation_status():

The function `merge_revocation_status()` merges the results, that is, the certificate status codes from `check_chain()`, `check_crl()`, `check_ocsp()` and `check_ocsp_online()` into one list of sets of certificate status codes for the certificate chain validated.

Input:

- `chain_status`: Results of a call to `check_chain()` (also output parameter), of size `n`.
- `crl_status`: Results of the call to `check_crl()`.
- `ocsp_status`: Results of the call to `check_ocsp()`.
- `require_rev_on_end_entity`: Whether a valid CRL or OCSP is required for the end entity certificate.
- `require_rev_on_intermediates`: Whether a valid CRL or OCSP is required for all intermediate certificates.

Output: None

Steps:

1. If (`chain_status` is empty) throw Invalid_Argument exception
2. For `i=0..n-1` do:
   a) Set `had_crl = false`
   b) Set `had_ocsp = false`
   c) If `crl_status[i]` contains an empty set then do continue with step e)
   d) For `j=0..k-1` do:
      i. If (crl_status[i][j] = VALID_CRL_CHECKED) then do set had_crl = true
      ii. Append `crl_status[i][j]` to `chain_status[i]`
   e) If `ocsp_status[i]` contains an empty set then do continue with step g)
   f) For `j=0..k-1` do:
      i. If (`ocsp_status[i][j]` = OCSP_RESPONSE_GOOD) then do set `had_ocsp = true`
      ii. Append `ocsp_status[i][j]` to `chain_status[i]`
   g) If (`had_crl = false` AND `had_ocsp = false`) then do:

i. If (`require_rev_on_end_entity = true` AND `i = 0`) OR (`require_rev_on_intermediates = true` AND `i > 0`) then do Append NO_REVOCATION_DATA to `chain_status[i]`

12 Key Derivation Functions

12.1 KDF1 (ISO 18033)

KDF1 is a key derivation function defined in ISO 18033-2, section 6.2.2. KDF1 is implemented according to ISO-18033-2:2006. It can be instantiated with any hash function. The implementation can be found in `src/lib/kdf/kdf1_iso18033/kdf1_iso18033.cpp`.

12.2 NIST SP800-108

NIST SP800-108 defines three functions for key derivation using pseudorandom functions: KDF in Counter Mode (5.1), KDF in Feedback Mode (5.2) and KDF in Double-Pipeline Iteration Mode. All three implementations can be found in `src/lib/kdf/sp800_108/sp800_108.cpp`. They can all be instantiated with any message authentication code as a PRF.

The implementation of KDF in Counter Mode fixes the length of $[L]_2$ and $[i]_2$ (the value `r`) to 32 bits.

The implementation of KDF in Feedback Mode uses the optional counter `i` and fixes the length of $[L]_2$ and $[i]_2$ (the value `r`) to 32 bits.

The implementation of KDF in Double-Pipeline Iteration Mode uses the optional counter `i` and fixes the length of $[L]_2$ and $[i]_2$ (the value `r`) to 32 bits.

12.3 NIST SP800-56C

NIST SP800-56C defines a key derivation using extraction-then-expansion. The implementation can be found in `src/lib/kdf/sp800_56c/sp800_56c.cpp`. The implementation fixes the context value for the expansion step to the empty string.

13 Bibliography

[ABR01] M. Abdalla, M. Bellare and P. Rogaway: “DHIES: An encryption scheme based on the Diffie-Hellman Problem”. 2001. http://cseweb.ucsd.edu/~mihir/papers/dhies.html

[AES-SSSE3] Mike Hamburg. Accelerating AES with vector permute instructions. CHES 2009

[apachehttpd] Apache httpd. http://httpd.apache.org/

[ASK05] Onur Acıiçmez, Werner Schindler, Çetin K. Koç: „Improving Brumley and Boneh Timing Attack on Unprotected SSL Implementations“. ACM CCS 2005. http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=DCD55885759798FF35E87947FEEC7E58?doi=10.1.1.63.3510&rep=rep1&type=pdf

[BB2003] D. Boneh, D. Brumley: „Remote timing attacks are practical“, Proceedings of the 12th Usenix Security Symposium, 2003

[BBPV2012] B. B. Brumley, M. Barbosa, D. Page, and F. Vercauteren: Practical realisation and elimination of an ECC-related software bug attack. CTRSA, 2012

[BDFPS14] Bhargavan, Delignat-Lavaud, Fournet, Pironti, Strub: „Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS“, Security and Privacy 2014, http://prosecco.gforge.inria.fr/personal/karthik/pubs/triple-handshakes-and-cookie-cutters-sp14.pdf

[BDPLR] K. Bhargavan, A. Delignat-Lavaud, A. Pironti, A. Langley, M. Ray: „Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension“, http://tools.ietf.org/html/draft-bhargavan-tls-session-hash-00

[BEXT] Botan extra tests. https://github.com/randombit/botan/tree/master/src/extra_tests

[BEAST] Thai Duong, Juliano Rizzo: Here Come The ⊕ Ninjas. http://packetstormsecurity.com/files/download/105499/Beast-SSL.rar

[Blei] Daniel Bleichenbacher: „Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1“, Crypto,1998

[BnetzA] Bundesnetzagentur für Elektrizität, Gas, Telekommunikation, Post und Eisenbahnen: „Bekanntmachung zur elektronischen Signatur nach dem Signaturgesetz und der Signaturverordnung (Übersicht über geeignete Algorithmen)“, 13.1.2014

[Bouncy] The Legion of the Bouncy Castle Cryptographic Library. https://www.bouncycastle.org/java.html

[BPSSL2014] Ivan Ristić: „Bulletproof SSL and TLS“, Feisty Duck, 16. August 2014

[BR93] Mihir Bellare and Phillip Rogaway: „Entity authentication and key distribution“, in Douglas R. Stinson, editor, Advances in Cryptology – CRYPTO’93

[BREACH] Yoel Gluck, Neal Harris, Angelo Prado, BREACH. SSL, „Gone in 30 Seconds“, http://breachattack.com/

[BT2011] B. B. Brumley,N. Tuveri: „Remote Timing Attacks are Still Practical“, Cryptology ePrint Archive: Report 2011/232, Mai 2011

[CAB] CA/Browser Forum: „Baseline Requirements for the Issuance and Management of Publicly-Trusted Certificates“, v.1.2.3

[CBC] William F. Ehrsam, Carl H. W. Meyer, John L. Smith, Walter L. Tuchman, "Message verification and transmission error detection by block chaining", US Patent 4074066, 1976

[CK01] R. Canetti and H. Krawczyk: „Analysis of Key-Exchange Protocols and Their Use for Building Secure Channels“, Advances in Cryptology - EUROCRYPT ’01

[CODEMET] Steve Cornett, Bullseye Testing Technology: „Minimum Acceptable Code Coverage“, http://www.bullseye.com/minimum.html

[COMBA] P. G. Comba: „Exponentiation cryptosystems on the IBM PC“, IBM Systems Journal, Vol. 29, No. 4, December 1990

[CP2005] C. Percival: „CACHE MISSING FOR FUN AND PROFIT“, Proceedings of BSDCan 2005

[CPPBESTPR] Jason Turner, et al.: „C++ Best Practices: A Forkable Coding Standards Document“, https://github.com/lefticus/cppbestpractices/blob/master/02-Use_the_Tools_Available.md, Commit 3da497

[CRIME-BS13] Tal Be'ery, Amichai Shulman. „A perfect CRIME? Only TIME will tell“, Blackhat 2013, https://media.blackhat.com/eu-13/briefings/Beery/bh-eu-13-a-perfect-crime-beery-wp.pdf

[CRIME-koto] Krzysztof Kotowicz, „If it's CRIME, then I'm guilty“, http://blog.kotowicz.net/2012/09/if-its-crime-than-im-guilty.html

[CRIMEimp] Adam Langley: „CRIME“, https://www.imperialviolet.org/2012/09/21/crime.html

[DI08] Christophe Doche, Laurent Imbert. The Double-Base Number System in Elliptic Curve Cryptograhy. http://www.lirmm.fr/~imbert/talks/laurent_Asilomar_08.pdf

[Doxygen] Doxygen, http://www.stack.nl/~dimitri/doxygen/

[EarlyCCSadv] OpenSSL Security Advisory, „Early CCS“, https://www.openssl.org/news/secadv_20140605.txt, 5. Juni 2014

[EarlyCCSimp] Adam Langley, „Early ChangeCipherSpec Attack“, https://www.imperialviolet.org/2014/06/05/earlyccs.html

[EarlyCCSLep] Masashi Kikuchi: „How I discovered CCS Injection Vulnerability (CVE-2014-0224)“, http://ccsinjection.lepidum.co.jp/blog/2014-06-05/CCS-Injection-en/index.html

[ECBlind] Werner Schindler, Andreas Wiemers: “Efficient Side-Channel Attacks on Scalar Blinding on Elliptic Curves with Special Structure”, Workshop on Elliptic Curve Cryptography Standards, 2015

[ECGDSA] Erwin Hess, Marcus Schafheutle, Pascale Serf, “The Digital Signature Scheme ECGDSA”, Siemens AG, Corporate Technology, Dept. CT IC 3, Oct. 24, 2006, https://www.teletrust.de/fileadmin/files/oid/ecgdsa_final.pdf

[EtEKDF] Krawczyk, H., "Cryptographic Extraction and Key Derivation: The HKDF Scheme", Proceedings of CRYPTO 2010, 2010, http://eprint.iacr.org/2010/264

[FAUT] S. Schinzel, I. Schmitt: „FAU-Timer“, https://code.google.com/p/fau-timer/, 2013

[FIPS186-1]	U.S. DEPARTMENT OF COMMERCE/National Institute of Standards and Technology: „DIGITAL SIGNATURE STANDARD (DSS)“, 15. Dezember 1998
[FIPS-186-4]	Federal Information Processing Standards Publication 186-4. Digital Signature Standard (DSS). http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf
[FIPS-202]	Federal Information Processing Standards Publication 202. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf
[FLAGS]	https://www.openssl.org/docs/crypto/X509_VERIFY_PARAM_set_flags.html
[FLUSH]	Y. Yarom, K. Falkner: „FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack“, SEC'14 Proceedings of the 23rd USENIX conference on Security Symposium
[FORK1]	Andrew Ayer: „LibreSSL's PRNG is Unsafe on Linux“, https://www.agwa.name/blog/post/libressls_prng_is_unsafe_on_linux
[FORK2]	Diverse Autoren, https://news.ycombinator.com/item?id=8034045
[GCM]	U.S. DEPARTMENT OF COMMERCE/National Institute of Standards and Technology: „Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC“, November 2007
[GCM12]	David A. McGrew (Cisco Systems, Inc.), John Viega (Secure Software): “The Galois/Counter Mode of Operation (GCM)”, 2012, http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-spec.pdf
[GCM-FA]	A. Joux. Authentication failures in NIST version of GCM. http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/comments/800-38_Series-Drafts/GCM/Joux_comments.pdf
[GCM-ND]	Hanno Böck, Aaron Zauner, Sean Devlin, Juraj Somorovsky, and Philipp Jovanovic. Nonce-Disrespecting Adversaries: Practical Forgery Attacks on GCM in TLS. WOOT'16
[GCV]	„Gcov – Using the GNU Compiler Collection (GCC)“, https://gcc.gnu.org/onlinedocs/gcc/Gcov.html
[GitLab]	„GitLab – Git repository management, issue tracking, code review and more” - https://about.gitlab.com/
[GitSCM]	„Git – distributed version control system”, https://git-scm.com/
[GKS13]	Florian Giesen, Florian Kohlar, and Douglas Stebila: „On the security of TLS renegotiation“,. CCS '13, 2013
[HAC]	Alfred J. Menezes, Paul C. van Oorschot, Scott A. Vanstone: „Handbook of Applied Cryptography“, Fifth Printing, 2001, ISBN 0-8493-8523-7
[HeartbeatRFC]	R. Seggelmann, M. Tuexen, M. Williams: „Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension“, 2012
[Heartbleed]	„The Heartbleed Bug“, http://heartbleed.com/
[HeartbleedAdv]	OpenSSL Security Advisory:“Heartbleed,“ https://www.openssl.org/news/secadv_20140407.txt,7. April 2014

[HeartbleedCloud] Nick Sullivan: „Heartache and Heartbleed: The insider's perspective on the aftermath of Heartbleed“, 2014, http://events.ccc.de/congress/2014/Fahrplan/events/6212.html

[HOST] https://www.openssl.org/docs/crypto/X509_check_host.html

[IANAECC] Internet Assigned Numbers Authority: „Transport Layer Security (TLS) Parameters: EC Named Curve Registry“, http://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml#tls-parameters-8, vom 8.12.2014

[ISO9796] ISO/IEC 9796-2: Information technology — Security techniques — Digital signature schemes giving message recovery — Part 2: Integer factorization based mechanisms

[ISO148883] ISO/IEC 14888-3:2006: Information technology – Security techniques – Digital signatures with appendix – Part 3: Discrete logarithm based mechanisms (with technical corrigendum 2, published 2009-02-15)

[ISO180332] ISO/IEC 18033-2:2006: Information technology – Security techniques – Encryption algorithms – Part 2: Asymmetric ciphers (2006)

[JKSS12] Tibor Jager, Florian Kohlar, Sven Schäge, and Jörg Schwenk: „On the security of TLS-DHE in the standard model. In Reihaneh Safavi-Naini and Ran Canetti,
editors“, Advances in Cryptology – CRYPTO 2012

[KKO63] A. Karatsuba, YU. Ofman , „Multiplication of multidigit numbers on automata“, Soviet Physics – Doklady, 7 (1963), 595–596.

[KM07] B. Kopf and H. Mantel. Transformational Typing and Unification for Automatically Correcting Insecure Programs. 2007

[KO120015] TTAK.KO-12.0015/R2: Digital Signature Mechanism with Appendix – Part 3: Korean Certificate-based Digitial Signature Algorithm using Elliptic Curves (EC-KCDSA)

[KR02] Vlastimil Klíma, Tomáš Rosa: „Attack on Private Signature Keys of the OpenPGP format, PGP programs and other applications compatible with OpenPGP“, 2002, https://eprint.iacr.org/2002/076.pdf

[LEH02] Anja Lehmann : „On the Security of Hash Function Combiners„ , 2010, https://tuprints.ulb.tu-darmstadt.de/2094/1/thesis.lehmann.pdf

[LinuxRNG] Stephan Müller, Gerald Krummeck, Mario Romsy (atsec information security GmbH): “Dokumentation und Analyse des Linux-Pseudozufallszahlengenerators”, Version 3.8, 2013-12-02.

[LMSS2014] M. Lochter, J. Merkle, J.-M. Schmidt, T. Schütze: „Requirements for Standard Elliptic Curve“, Oktober 2014, http://eprint.iacr.org/2014/832

[Lucky13] N.J. AlFardan and K.G. Paterson: „Lucky Thirteen: Breaking the TLS and DTLS Record Protocols“, IEEE Security and Privacy, 2013, http://www.isg.rhul.ac.uk/tls/Lucky13.html

[Lucky13imp] Adam Langley: „Lucky Thirteen attack on TLS CBC“, 2013, https://www.imperialviolet.org/2013/02/04/luckythirteen.html

[MALSHA1] A. Albertini, J.-P. Aumasson, M. Eichlseder, F. Mendel, M. Schläffer: „Malicious Hashing: Eve's Variant of SHA-1“, Cryptology ePrint Archive: Report 2014/694, 03. September 2014, http://eprint.iacr.org/2014/694 FAQ: http://malicioussha1.github.io/#faq

[Man] James Manger. A Chosen Ciphertext Attack on RSA Optimal Asymmetric Encryption Padding (OAEP) as Standardized in PKCS#1 v2.0. Crypto 2001

[mitmAdv] OpenSSL Security Advisory [07-Jan-2009]: „Incorrect checks for malformed signatures“, https://www.openssl.org/news/secadv_20090107.txt

[MONA] Mona Timing Library. https://github.com/seecurity/mona-timing-report.git

[MS14] Daniel A. Mayer, Joel Sandin: „Time Trial - Racing Towards Practical Remote Timing Attacks“, Blackhat 2014

[MS15] Heiko Mantel and Artem Starostin. Transforming Out Timing Leaks, More or Less. ESORICS 2015

[MSWS] Christopher Meyer, Juraj Somorovsky, Eugen Weiss, and Jörg Schwenk, Sebastian Schinzel, Erik Tews: „Revisiting SSL/TLS Implementations: New Bleichenbacher Side Channels and Attacks“, USENIX Security ,2014

[NCsupport] Tom Leek: „OpenSSL ECC named curve support“, 2013, http://security.stackexchange.com/questions/43992/openssl-ecc-named-curve-support

[NETSEC] John Viega, Matt Messier, Pravir Chandra: „Network Security with OpenSSL“, Frist Edition, 2002, ISBN 0-596-00270-X

[nginx] Nginx, http://nginx.org/

[NIST-OMAC] National Institute of Standards and Technology. OMAC: One-Key CBC MAC — Addendum. http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/omac/omac-ad.pdf

[NIST-SHS] National Institute of Standards and Technology. Secure Hash Standard (SHS). 2015. http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf

[openssh] OpenSSH. http://www.openssh.com/

[OpenSSLA] W. Meyer zu Bergsten, R. Korthaus, J. Somorowsky, C. Mainka, Prof. Dr. J. Schwenk: „Quellcode-basierte Untersuchung von kryptographisch relevanten Aspekten der OpenSSL-Bibliothek“, Juni 2015

[OpenSSLDH] OpenSSL Dokumentation zu DH. https://www.openssl.org/docs/ssl/SSL_CTX_set_tmp_dh_callback.html

[P1OVECT1] PKCS#1 Test Vectors, Original: ftp://ftp.rsa.com/pkcs-1v2/p1ovect1.txt (offline), available at ftp://ftp.sunet.se/pub/security/pca/docs/PKCS/ftp.rsa.com/pkcs-1v2/p1ovect1.txt

[PCK1996] Paul C. Kocher: „Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems“, Advances in Cryptology - CRYPTO ’96

[PKCS1] B. Kaldinski, J. Staddon: „PKCS #1: RSA Cryptography Specifications“, Version 2.0, October 1998, available at https://www.ietf.org/rfc/rfc2437.txt

[PKCS7] B. Kaliski: “PKCS #7: Cryptographic Message Syntax”, Version 1.5, March 1998, available at https://tools.ietf.org/rfc/rfc2315.txt

[PKI] Carlisle Adams, Steve Lloyd: „Understanding PKI“, Second Edition, 2003, ISBN 0-672-32391-5

[POODLE] Bodo Möller, Thai Duong, Krzysztof Kotowicz: „This POODLE bites: Exploiting the SSLv3.0 Fallback“, 2014, https://www.openssl.org/~bodo/ssl-poodle.pdf

[POODLEadv] OpenSSL Security Advisory, „POODLE und SSLv3.0 Fallback“, https://www.openssl.org/news/secadv_20141015.txt, 15. Oktober 2014

[PyCrypto] PyCryptodome Cryptographic Library. https://www.pycryptodome.org

[RenegAdv] OpenSSL Security Advisory: „TLS Renegotiation Attack“, https://www.openssl.org/news/secadv_20091111.txt, 11. November 2009

[ReqEC] BSI. Minimum Requirements for Evaluating Side-Channel Attack Resistance of Elliptic Curve Implementations. Version 1.0.4, Juli 2011

[RFC2104] H. Krawczyk, M. Bellare, R. Canetti. RFC2104. “HMAC: Keyed-Hashing for Message Authentication”, 1997

[RFC2785] R. Zuccherato: Methods for Avoiding the "Small-Subgroup" Attacks on the Diffie-Hellman Key Agreement Method for S/MIME, https://www.ietf.org/rfc/rfc2785.txt

[RFC3061] Internet Engineering Task Force: RFC3061: “A URN Namespace of Object Identifiers”, https://www.ietf.org/rfc/rfc3061.txt, February 2001

[RFC3279] Internet Engineering Task Force: RFC3279: „Algorithms and Identifiers for the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile“, http://tools.ietf.org/html/rfc3279, April 2002

[RFC3280] Internet Engineering Task Force: RFC3280: „Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile“, http://tools.ietf.org/html/rfc3280, April 2001

[RFC3447] Internet Engineering Task Force: RFC3447: „Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1“, http://tools.ietf.org/html/rfc3447, February 2003

[RFC3713] Internet Engineering Task Force: RFC3713: “A Description of the Camellia Encryption Algorithm”, https://tools.ietf.org/html/rfc3713, April 2004

[RFC3766] H. Orman, P.Hoffman. RFC3766. “Determining Strengths For Public Keys Used For Exchanging Symmetric Keys”, 2004

[RFC3779] Internet Engineering Task Force: RFC3779: „X.509 Extensions for IP Addresses and AS Identifiers“, http://tools.ietf.org/html/rfc3779, June 2004

[RFC3962] Internet Engineering Task Force: RFC3962: „Advanced Encryption Standard (AES) Encryption for Kerberos 5“, https://www.ietf.org/rfc/rfc3962.txt, February 2005.

[RFC4304] S. Kent, BBN Technologies: RFC4304: „Extended Sequence Number (ESN) Addendum to IPsec Domain of Interpretation (DOI) for Internet Security Association and Key Management Protocol (ISAKMP)”

[RFC4419] Internet Engineering Task Force: RFC4419: „Diffie-Hellman Group Exchange for the Secure Shell (SSH) Transport Layer Protocol“, http://tools.ietf.org/html/rfc4419, March 2006

[RFC4492] Internet Engineering Task Force: RFC4492: “Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)”, http://tools.ietf.org/html/rfc4492, May 2006

[RFC5246] Internet Engineering Task Force: RFC5246: „The Transport Layer Security (TLS) Protocol Version 1.2“, https://tools.ietf.org/html/rfc5246, August 2008

[RFC5280] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, W. Polk. RFC5280. “Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile”, 2008

[RFC5580] Internet Engineering Task Force: RFC5580: „Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile“, http://tools.ietf.org/html/rfc5280, March 2008

[RFC5639] Internet Engineering Task Force: RFC5639: „Elliptic Curve Cryptography (ECC) Brainpool Standard Curves and Curve Generation“, https://tools.ietf.org/html/rfc5639, March 2010

[RFC5652] R. Housley. RFC-5652. Cryptographic Message Syntax. 2009

[RFC5746] Internet Engineering Task Force: RFC5746: „Transport Layer Security (TLS) Renegotiation Indication Extension“, http://tools.ietf.org/html/rfc5746, February 2010

[RFC6125] Internet Engineering Task Force: RFC6125: „Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS)“, http://tools.ietf.org/html/rfc6125, March 2011

[RFC6979] Internet Engineering Task Force: RF6979: “Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)“, August 2013

[RS2000] Jean-Francois Raymond, Anton Stiglic: „Security Issues in the Diffie-Hellman Key Agreement Protocol“, December 2000

[RSAblAdv] OpenSSL Security Advisory [17 March 2003], RSA Blinding, https://www.openssl.org/news/secadv_20030317.txt

[RSAblinding] RSA Blinding patch and a recent snapshot, OpenSSL mailing list, http://comments.gmane.org/gmane.comp.encryption.openssl.devel/4919

[RTOpenSSL] Issue Tracking System for OpenSSL, 2014, http://rt.openssl.org/Ticket/Display.html?id=3180&user=guest&pass=guest

[RTOpenSSL] Issue Tracking System for OpenSSL, 2014, http://rt.openssl.org/Ticket/Display.html?id=3180&user=guest&pass=guest

[RTSS09] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage: Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds, CCS '09

[SAFEECC] D.J. Bernstein: „SafeCurves: choosing safe curves for elliptic-curve cryptography“, http://safecurves.cr.yp.to/, January 2014

[sloccount] David A. Wheeler: „SLOCCount“, http://www.dwheeler.com/sloccount/

[SP80038A-A] National Institute of Standards and Technology: “Recommendation for Block Cipher Modes of Operation: Three Variants of Ciphertext Stealing for CBC Mode”, Addendum to NIST Special Publication 800-38A, October 2010.

[SP80090A]	NIST Special Publication 800-90A Revision 1: “Recommendation for Random Number Generation Using Deterministic Random Bit Generators”, Elaine Barker and John Kelsey, Computer Security Division, Information Technology Laboratory, June 2015
[Sqlite]	SQLite, https://sqlite.org/
[SSL3]	A. Frier, P. Karlton, P. Kocher: "The SSL 3.0 Protocol", Netscape Communications Corp., Nov 18, 1996.
[sslyze]	SSLyze, https://github.com/iSECPartners/sslyze
[testsslsh]	testssl.sh: Testing SSL/TLS, https://testssl.sh/
[TIOBE]	Paul Jansen, TIOBE Software: „The TIOBE Quality Indication – a pragmatic way of measuring code quality“, Version 3.5, 16. Oktober 2015
[TLSA]	TLS-Attacker. https://github.com/RUB-NDS/TLS-Attacker
[TR021021]	BSI Technische Richtline BSI TR-02102-1: „Kryptografische Verfahren: Empfehlungen und Schlüssellängen“, Version 2016-01, 15.2.2016
[TR021022]	BSI Technische Richtline BSI TR-02102-2: „Verwendung von Transport Layer Security (TLS)“, Version 2016-01
[TR03111]	BSI Technical Guideline TR-03111: “Elliptic Curve Cryptography”, Version 2.0
[TRUST]	https://www.openssl.org/docs/apps/x509.html#TRUST_SETTINGS
[VFY-CERT]	https://www.openssl.org/docs/crypto/X509_verify_cert.html
[VFY]	https://www.openssl.org/docs/ssl/SSL_CTX_set_verify.html
[WEAKDH]	David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, and Paul Zimmermann. Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice, CCS'15. https://weakdh.org/
[x509test]	https://github.com/yymax/x509test
[X923]	Americal Nation Standards Institute: “Financial Institution – Encryption of Wholesale Financial Messages”, ANSI Standard X.923, 1988.
[XMSS]	A. Huelsing, D. Butin, S. Gazdag, A. Mohaisen. XMSS: Extended Hash-Based Signatures draft-irtf-cfrg-xmss-hash-based-signatures-06. https://datatracker.ietf.org/doc/draft-irtf-cfrg-xmss-hash-based-signatures/06/, 2016
[ZDNET]	http://www.zdnet.com/68-percent-of-top-free-android-apps-vulnerable-to-cyberattack-researchers-claim-7000032875/
[ZJRR]	Y. Zhang, A. Juels, M.K. Reiter, T. Ristenpart: „Cross-Tenant Side-Channel Attacks in PaaS Clouds“, Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security
[ZJRR12]	Yinqian Zhang, Ari Juels, Mike Reiter, and Thomas Ristenpart: Cross-VM Side Channels and Their Use to Extract Private Keys, CCS '12