Stateful Hash Objects: API and Constructions

Trevor Perrin (trevp@trevp.net)

Revision 1, 2018-02-26, unofficial/unstable

Contents

1. Introduction

This document defines the **Stateful Hash Object (SHO)** API. This is an API for working with hash functions that provides several useful features:

- "Customization labels" for domain-separation.
- Arbitrary-length (XOF-style) output.
- Immunity to "length-extension" of hash inputs (unlike functions such as SHA-256).
- "Stateful hash objects" which can "absorb" inputs incrementally, so can be used in streaming cases or protocols where inputs are absorbed at different stages.
- A "ratcheting" function for a stateful hash object which makes the object's internal state minimum-sized and a one-way function of all preceding inputs, for forward-secrecy and reducing storage size.
- An optional encryption capability which can be provided by a stateful hash object to efficiently encrypt some data while absorbing the ciphertext.

This document also describes constructions which extend existing hash functions to support the SHO API.

2. Overview

The SHO API is based on the notion of a **Stateful Hash Object** (or **SHO object**). A SHO object is **Initialized** with some **customization label**. The customization label provides domain separation so that if the same values are input to differently-customized SHO objects, independent outputs will be produced.

The SHO object is then used to **Absorb** inputs. Eventually an output is **Squeezed** from the SHO object.

SHO objects typically contain a small, fixed-size buffer plus some internal chaining variable. Absorbing input appends to the buffer, and calls some cryptographic function when the buffer is full. The cryptographic function mixes the buffered data into the chaining variable and then resets the buffer to empty.

To provide the caller more control over this buffer, a **Ratchet** function can be called to force the SHO object to cryptographically process any buffered data so that its chaining variable becomes a one-way function of all preceding inputs, and the SHO object's state is reduced to its minimum size (i.e. with no buffered data).

If the caller wishes to squeeze multiple outputs from a sequence of inputs at different points in time, the caller can **Clone** the SHO object and squeeze output from the clones.

While SHO objects provide several features targeted at stateful hashing (where a single SHO object is used to absorb inputs over a period of time), SHO features like customization labels, immunity to length-extension, and arbitrary output lengths are useful even if exposed via a single-shot (non-stateful) API.

3. SHO API

3.1. SHO object creation

A SHO API is capable of creating new SHO objects based on either a **customization label** (which may be an empty byte sequence) or by cloning an existing SHO object:

- **`Initialize(customization_label: bytes)`**: Creates and returns a new SHO object based on the `customization_label` byte sequence, which must be from 0-65535 bytes in length. Initialization with a non-empty `customization_label` is recommended (but not required) to leave the new SHO object in a minimum-sized state (e.g. by calling `Ratchet` internally), to aid in storing precalculated SHO objects.
- **`Clone()`**: This function is a method which is called "on" some pre-existing SHO object. The pre-existing object's state is copied into a new SHO object, which is returned.

3.2. SHO input and output

In addition to its `Clone` method, a SHO object supports `Absorb`, `Squeeze`, and `Ratchet` methods:

- **`Absorb(input: bytes)`**: This method passes input into the SHO object. When output is eventually squeezed from the SHO object, that output will be a hash of the concatenation of all absorbed inputs. Note that inputs are considered to be concatenated without separators, i.e. `Absorb("abc")` is the same as `Absorb("ab")` followed by `Absorb("c")`.
- **`Squeeze(output_length: uint64)`**: This method returns a byte sequence of length `output_length` which is a hash of the customization label and all absorbed input. The requested output is considered to be a prefix of some infinite-length output, so requesting a longer `output_length` will give the same initial bytes as requesting a shorter `output_length`. After

this function is called on a SHO object the object is consumed and can't be used any further.

- `Ratchet()`: This method causes the SHO object to update its state to be a one-way cryptographic function of all preceding inputs, and to be minimum-sized (i.e. any buffered data which has not been hashed yet will be hashed and then cleared). Calling `Ratchet` will affect the eventual results from `Squeeze` in one of the following three ways (the exact effect will be determined by the current state of the SHO object).
  - The `Ratchet` call doesn't modify the object at all (if it is already in a minimum-sized state)
  - The `Ratchet` call is equivalent to an `Absorb` call, e.g. an `Absorb` call that absorbs some padding bytes to fill out the next hash block.
  - The `Ratchet` call is equivalent to an `Absorb` call that absorbs a special symbol that cannot be passed into `Absorb` directly but which is hashed into the eventual output.

3.3. SHO function

A SHO API should also provide a more traditional, non-stateful hash function. For convenience, we simply call this the **SHO function** (as distinct from a SHO object), or call this function by the name of the SHO algorithm (e.g. the SHO/SHA256 function).

- **`SHO(customization_label: bytes, input: bytes, output_length: uint64)`**: This is shorthand for initializing a new SHO object with the customization label, absorbing the `input`, and then squeezing `output_length` bytes of output. To simplify the API, the `customization_label` field defaults to the empty string, and the `output_length` field defaults to the collision-resistant output length for the underlying hash function (e.g. 32 bytes for SHA-256, SHAKE128, or BLAKE2s; 64 bytes for SHA-512, SHAKE256, or BLAKE2b).

4. SHO with encryption (SHOE) API

An extended form of SHO object is a **Stateful Hash Object with Encryption**, or **SHOE** object. A SHOE object is a SHO object which additionally supports `Encrypt` and `Decrypt` functions.

These functions effectively derive an encryption key from the SHOE object's state, use it to perform authenticated-encryption on some message, and then absorb the ciphertext into the SHOE object. This functionality is useful in protocols such as Noise where handshake messages are being encrypted and hashed simultaneously. Providing these functions via a special `SHOE` object allows low-level optimizations, such as combining the authentication and hashing calculations, or using a sponge/duplex authenticated-encryption mode.

- **`Encrypt(plaintext: bytes)`**: This method returns a ciphertext byte sequence of length equal to the plaintext length plus 16. The ciphertext is an authenticated encryption of the plaintext. This function should provide equivalent functionality to first cloning the SHOE object, then absorbing some special symbol into the clone which cannot be directly passed to `Absorb`, then squeezing a key from the clone, then using the key to perform authenticated-encryption on the plaintext, then absorbing the ciphertext into the original SHOE object.
- **`Decrypt(ciphertext: bytes)`**: This method returns a plaintext byte sequence of length equal to the ciphertext length minus 16, or a decryption error. The plaintext is an authenticated decryption of a ciphertext that was encrypted using a SHOE object in the same state.

5. SHO constructions

5.1. Generic SHO constructions

To build SHO variants of existing hash function we place these functions in one of four categories, based on whether or not they are an extensible output function (XOF), and whether or not their underlying cryptographic function is a sponge.

For example, we classify functions in the SHAKE, SHA2, and BLAKE2 families as follows:

	XOF	Not XOF
Sponge	SHAKE128, SHAKE256	(SHA-3)
Not Sponge	(BLAKE2X)	SHA256, SHA512 BLAKE2s, BLAKE2b

SHA-3 and BLAKE2X are shown only as examples of the classification scheme, but for the other functions we will define SHO variants, named as:

- SHO/SHAKE128 and SHO/SHAKE256
- SHO/SHA256 and SHO/SHA512
- SHO/BLAKE2s and SHO/BLAKE2b

These SHOs are defined using the following generic construction, which can be applied to other hash functions based on the above categorization.

The generic construction is described with Python-like pseudocode, assuming a SHO object where the following functions are defined:

- The `update()`, and `finalize()` functions call a stateful API for the underlying hash function. The `update()` function appends to the input byte sequence, and the `finalize()` function produces the final hash or XOF output. We assume the `update()` function is appending inputs to a buffer of length `BLOCKLEN` (for a sponge, `BLOCKLEN` is the size of the "rate"), and whenever the buffer is full the inputs are immediately mixed into an internal chaining variable and the buffer is cleared. We assume the `finalize()` function produces `HASHLEN` bytes of output (for an XOF, `HASHLEN` is the recommended output length to provide collision-resistance).
- The `buffered_data_len()` function returns the number of bytes the hash function has buffered since last invoking the underlying compression function (or for a sponge, how many bytes have been written into the sponge's rate since last invoking the underlying permutation). This value will be from 0 to `BLOCKLEN-1`.
- The `zeroize_rate()` function can only be called if the underlying hash is a sponge, in which case it erases (sets to zero) the contents of the sponge's "rate". This requires low-level access to the sponge which is not typically provided by a SHAKE API.
- The `is_xof()` and `is_sponge()` functions return True or False depending on the category of the existing function.
- The `zeros(n)` function returns a byte sequence of length `n` filled with zeros.
- The `new_byte_sequence()` function returns an empty byte sequence, and the `new_hash()` function returns a new hash object for the underlying hash function.
- The `uint16()` and `uint64()` functions encode an unsigned integer into a big-endian byte sequences.

Generic SHO construction pseudocode:

```
def Init(self, customization_label):
    if not self.is_xof():
        self.update(zeros(BLOCKLEN))
    self.update(uint16(len(customization_label)))
    if len(customization_label) != 0:
        self.update(customization_label)
        self.Ratchet()

def Absorb(self, input):
    self.update(input)

def Ratchet(self):
    if self.buffered_data_len() != 0:
        self.update(zeros(BLOCKLEN - self.buffered_data_len()))
    if self.is_sponge():
        self.zeroize_rate()

def Squeeze(self, output_length):
    if not self.is_xof():
        inner_hash = self.finalize()
        output = new_byte_sequence()
        for count in range(math.floor(output_length-1 / HASHLEN)+1):
            h = new_hash()
            h.update(inner_hash)
            h.update(uint64(count))
            output.append(h.finalize())
        return output[ : output_length]  # truncate to output_length
    else:
        return self.finalize(output_length)
```

5.2. Generic construction examples

The simplest form of the above constructions would result from calling the single-shot `SHO` function with default (empty) `customization_label` and default `output_length`. This would result in the following output, for each of the listed SHO variants, where `||` indicates concatenation of byte sequences.

SHO/SHA256

- SHA256(SHA256(zeros(66) || input) || zeros(8))

SHO/SHA512

- SHA512(SHA512(zeros(130) || input) || zeros(8))

SHO/BLAKE2s

- BLAKE2s(BLAKE2s(zeros(66) || input) || zeros(8))

SHO/BLAKE2b

- BLAKE2b(BLAKE2b(zeros(130) || input) || zeros(8))

SHO/SHAKE128

- SHAKE128(zeros(2) || input, 32)

SHO/SHAKE256

- SHAKE256(zeros(2) || input, 64)

With non-empty `customization_label` the last two bytes of the initial `zeros()` field would be replaced by a `uint16(len(customization_label))` field, followed by the `customization_label`, then followed by zero-padding to fill out the remainder of the hash block. For the SHAKE variants, `zeroize_rate()` would have to be called after absorbing a non-empty `customization_label`, so the non-empty `customization_label` case could no longer be expressed as a simple call to SHAKE128 or SHAKE256.

With different output lengths, the `zeros(8)` field in the non-XOF functions would be replaced with a 64 bit counter which increments 0,1,... until sufficient output is produced.

5.3. HKDF construction

HKDF [1], [2] (with some underlying hash function) can be used with the SHO API as follows:

- HKDF's `salt` parameter is used for the `customization_label`.
- HKDF's `ikm` parameter is used for the absorbed input.
- HKDF's `info` parameter is set to a zero-length byte sequence.
- HKDF's `output_length` is set to the `output_length` from `Squeeze`.

The `Absorb` and `Ratchet` functions are the same as if the underlying hash function was used with the generic SHO construction.

As an example, calling HKDF-SHA256 as a single-shot `SHO` function with default `customization_label` and `output_length` would result in the following output:

- HKDF-SHA256(salt=customization_label, ikm=input, info="", output_length=32)

6. SHOE constructions

6.1. STROBE

STROBE [3] can be used to implement the SHOE API using the following Python-like pseudocode, where the STROBE operations (AD, meta-AD, RATCHET, send_MAC/recv_MAC, send_ENC/recv_ENV) are methods on the SHOE object.

SHOE/STROBEv1.0.2 pseudocode:

```
def Init(self, customization_label):
    self.meta-AD(customization_label)

def Absorb(self, input):
    self.AD(input, more=true)

def Ratchet(self):
    self.RATCHET()

def Squeeze(self, output_length):
    return self.PRF(output_length)

def Encrypt(self, plaintext):
    ciphertext = self.send_ENC(plaintext)
    ciphertext.append(self.send_MAC(16))
    return ciphertext

def Decrypt(self, ciphertext):
    plaintext = self.recv_ENV(ciphertext[:-16])
    tag = self.recv_MAC(16)
    if not consttime_equal(tag, ciphertext[-16:]):
        raise Error("decryption failure)
    return plaintext
```

7. Security considerations

The constructions here are all new and should not be used until more analysis has been done.

8. Rationales

The generic SHO construction uses nested hashing with a prepended zero block for the innner hash. This construction was analyzed in [4] (where it was somewhat confusingly called the "HMAC construction", though differing from the more widely known HMAC function).

9. IPR

This document is hereby placed in the public domain.

10. Acknowledgements

This proposal resulted from extensive discussion with Gilles van Assche about stateful hashing, and was also inspired by Mike Hamburg's STROBE, and discussions with Mike.

Samuel Neves proposed the nested-hashing construction from [4]. Peter Schwabe proposed the importance of domain-separation and explicit customization labels.

Discussions with Henry de Valence regarding his Merlin proposal, and with David Wong regarding his Disco proposal, were also helpful, as was feedback from Paul Rösler.

11. References

[1] H. Krawczyk, "'Cryptographic extraction and key derivation: The hkdf scheme'." Cryptology ePrint Archive, Report 2010/264, 2010. http://eprint.iacr.org/2010/264

[2] H. Krawczyk and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)." Internet Engineering Task Force; RFC 5869 (Informational); IETF, May-2010. http://www.ietf.org/rfc/rfc5869.txt

[3] Mike Hamburg, "The STROBE protocol framework." Cryptology ePrint

Archive, Report 2017/003, 2017. http://eprint.iacr.org/2017/003

[4] J.-S. Coron, Y. Dodis, C. Malinaud, and P. Puniya, “Merkle-damgård revisited: How to construct a hash function,” in Proceedings of the 25th annual international conference on advances in cryptology, 2005, pp. 430–448. https://cs.nyu.edu/~dodis/ps/merkle.pdf