Permutation-based APIs

A framework for future-proof cryptographic APIs.

Frank Denis @jedisct1 — PBC2023
Fastly Inc.

Where do APIs come from?

- Standards
- Specifications
- Reference implementations
- Very useful for implementers, in order to quickly understand what the components are and how they relate to each other

But most implementations will invent their own, idiomatic APIs

After a paper is released, we may quickly see multiple **standalone** implementations

What’s the goal of dedicated APIs?

- To expose all the functionalities **of a primitive**.
  In a way that feels elegant **in the context of that primitive**.
- However, these APIs won’t be used much, because **they don’t fit well within the rest of the ecosystem**. If not rewritten, implementations are going to be merged into other projects, with their own APIs.

What people actually use

- General-purpose cryptographic libraries
- Cryptographic services from the Operating System
- APIs from cloud vendors
- Standard libraries from programming languages
- Company internal frameworks

Anything that doesn’t follow existing conventions can be confusing

MulMod(a,b,m)

Are a and b assumed to be reduced (mod m)?

How a cryptographic library is built

- Start with the low-hanging fruits and the **most common primitives**
- Design an **API** that's a perfect fit for what is implemented.
- Goal: to be good. To look good. **Clean**. With nice **abstractions**.

The good API

- Functions are grouped by categories
- Strong typing to enforce separation
- In a given category, everything is consistent
- Feels clean and satisfactory
- But time passes…

H(ctx,p,...)

Parameters set doesn’t match the current API for hash functions

What can we do?

- Add new functions for that special case?
  Ugly.
- Revamp the entire API to use one namespace per function?
  May be too late, and API surface would balloon.
- Only expose the lowest common denominator?
  Sad.
- Expose the lowest common denominator + additional functions?
  Redundant and confusing.

H(ctx,k,len,...)

Where does that function fit?

Permutation-based cryptography?

Doesn’t fit anywhere in APIs of general-purpose libraries

Adding permutation-base cryptography

- Shall we make breaking changes to the current APIs?
- Introduce a new category and duplicate everything from other categories?
- What could we do if we could start over?

WebAssembly

A virtual machine for C, C++, Zig, Go, Kotlin, Ruby, Rust, C#, …

System calls

- Allow applications to interact with the kernel
- Critical API
- Has to be **small**
- Has to be **secure**
- Has to be **stable**
- Every system call must be carefully designed, with a **long-term** view.

WebAssembly hostcalls

- Allow applications to communicate with the WebAssembly runtime
- Small, well-defined, trusted APIs, that have to commit to long-term stability
- WASI: domain-specific sets of standard APIs
- **WASI-Crypto**

These APIs are not meant to be directly used by applications

Symmetric cryptography API for WebAssembly

Types
(Handles)

- symmetric_key
  ```
  key_handle = symmetric_key_import("SHA-256", bytes)
  ```
- symmetric_state
- symmetric_tag

Algorithms are strings

- `key_handle = symmetric_key_import(`**`"SHA-256"`**`, bytes)`

Creating a state

- `state = symmetric_state_open("HMAC/SHA-256", key_handle, options)`
- `options` is a `{string, integer|string|memory}` map
- Allows new primitives to be added, and their custom features to be supported **without breaking changes**.

Option name	Description	Type
`context`	Context/domain for hash functions and XOFs	Byte string
`salt`	Salt for hash functions	Byte string
`nonce`	Nonce or IV for ciphers	Byte string
`memory_limit`	Memory limit in bytes for memory-hard KDFs	Unsigned integer
`ops_limit`	Computational cost for CPU-hard KDFs	Unsigned integer
`parallelism`	Number of threads to use	Unsigned integer
`buffer`	Scratch buffer for memory-hard KDFs	Memory

Consistency between keys and algorithms is always enforced

The complete symmetric crypto API

`clone()`	`absorb()`	`squeeze()`	`squeeze_tag()`
`squeeze_key()`	`max_tag_len()`	`encrypt()`	`encrypt_detached()`
`decrypt()`	`decrypt_detached()`	`ratchet()`	`tag_len()`
`tag_pull()`	`tag_verify()`	`close()`	`reset()`

Symmetric operations are performed by composing the following functions:

- `symmetric_state_absorb()`: absorb data into the state.
  - **Hash functions:** adds data to be hashed.
  - **MAC functions:** adds data to be authenticated.
  - **Tuplehash-like constructions:** adds a new tuple to the state.
  - **Key derivation functions:** adds to the IKM or to the subkey information.
  - **AEAD constructions:** adds additional data to be authenticated.
  - **Stateful hash objects, permutation-based constructions:** absorbs.

- `symmetric_state_squeeze()`: squeeze bytes from the state.
    - **Hash functions:** this tries to output an `out_len` bytes digest from the absorbed data. The hash function output will be truncated if necessary. If the requested size is too large, the `invalid_len` error code is returned.
    - **Key derivation functions:** : outputs an arbitrary-long derived key.
    - **RNGs, DRBGs, stream ciphers::** outputs arbitrary-long data.
    - **Stateful hash objects, permutation-based constructions:** squeeze.

Other kinds of algorithms MUST return `invalid_operation` instead.

For password-stretching functions, the function MAY return `in_progress`.

In that case, the guest SHOULD retry with the same parameters until the function completes.

Hash functions, XOF

- `{ absorb(), squeeze() }`

```
let mut out = [0u8; 64];
let state_handle = symmetric_state_open("SHA-256", None)?;
symmetric_state_absorb(state_handle, b"data")?;
symmetric_state_absorb(state_handle, b"more_data")?;
symmetric_state_squeeze(state_handle, &mut out)?;
```

MAC

- `{ absorb(), squeeze_tag() }`
- Tag object can be copied or verified

```
let mut raw_tag = [0u8; 64];
let key_handle = symmetric_key_import("HMAC/SHA-512", b"key")?;
let state_handle = symmetric_state_open("HMAC/SHA-512", Some(key_handle), None)?;
symmetric_state_absorb(state_handle, b"data")?;
symmetric_state_absorb(state_handle, b"more_data")?;
let computed_tag_handle = symmetric_state_squeeze_tag(state_handle)?;
symmetric_tag_pull(computed_tag_handle, &mut raw_tag)?;
```

```
let state_handle = symmetric_state_open("HMAC/SHA-512", Some(key_handle), None)?;
symmetric_state_absorb(state_handle, b"data")?;
symmetric_state_absorb(state_handle, b"more_data")?;
let computed_tag_handle = symmetric_state_squeeze_tag(state_handle)?;
symmetric_tag_verify(computed_tag_handle, expected_raw_tag)?;
```

HKDF

- Extract: `{ absorb(), squeeze_key() }`
- Expand: `{ absorb(), squeeze() }`

```
let mut prk = vec![0u8; 64];
let key_handle = symmetric_key_import("HKDF-EXTRACT/SHA-512", b"key")?;
let state_handle = symmetric_state_open("HKDF-EXTRACT/SHA-512", Some(key_handle), None)?;
symmetric_state_absorb(state_handle, b"salt")?;
let prk_handle = symmetric_state_squeeze_key(state_handle, "HKDF-EXPAND/SHA-512")?;
```

```
let mut subkey = vec![0u8; 32];
let state_handle = symmetric_state_open("HKDF-EXPAND/SHA-512", Some(prk_handle), None)?;
symmetric_state_absorb(state_handle, b"info")?;
symmetric_state_squeeze(state_handle, &mut subkey)?;
```

Password hashing

- Hash string: `{ absorb(), squeeze_tag() }`
  Returned tag is a string that can be used to verify the input.
- KDF: `{ absorb(), squeeze() }`

```
let mut memory = vec![0u8; 1_000_000_000];
let options_handle = symmetric_options_open()?;
symmetric_options_set_guest_buffer(options_handle, "memory", &mut memory)?;
symmetric_options_set_u64(options_handle, "opslimit", 5)?;
symmetric_options_set_u64(options_handle, "parallelism", 8)?;

let state_handle = symmetric_state_open("ARGON2-ID-13", None, Some(options))?;
symmtric_state_absorb(state_handle, b"password")?;

let pw_str_handle = symmetric_state_squeeze_tag(state_handle)?;
let mut pw_str = vec![0u8; symmetric_tag_len(pw_str_handle)?];
symmetric_tag_pull(pw_str_handle, &mut pw_str)?;
```

AEADs

- AEADs **must** support the following operations:
  - `absorb()`
  - `max_tag_len()`
  - `encrypt()`, `encrypt_detached()`, `decrypt()`, `decrypt_detached()`
- if padding is required, it is included in the tag
- Where’s the nonce?

Nonce is optional

Automatically generated if safe

```
let key_handle = symmetric_key_generate("AES-256-GCM-SIV", None)?;
let message = b"test";
let mut nonce = [0u8; 24];

let state_handle = symmetric_state_open("AES-256-GCM-SIV", Some(key_handle), None)?;

let nonce = symmetric_state_options_get(state_handle, "nonce")?;

let mut ciphertext = vec![0u8; message.len() + symmetric_state_max_tag_len(state_handle)?];
symmetric_state_absorb(state_handle, "additional data")?;
symmetric_state_encrypt(state_handle, &mut ciphertext, message)?;
```

Sessions are supported out of the box

Required, recommended and optional algorithms

- Implementations are **encouraged** to support Xoodyak and Kyber
- Official test suite will include test for these.

Current status

- The API for permutation-based cryptography **is** the API, not an additional API
- API is small and comprehensive
- Yet extensible without breaking changes
- Traditional APIs can easily be built on top of it

- Developers understand it
- Makes permutation-based cryptography more widely available

Thanks!

https://github.com/WebAssembly/wasi-crypto