**Cryptography with rust**

[[](https://medium.com/@technocrat21?source=post_page-----31957242299f--------------------------------)](https://medium.com/@technocrat21?source=post_page-----31957242299f--------------------------------)

[[CoderHack.com](https://medium.com/coderhack-com?source=post_page-----31957242299f--------------------------------)](https://medium.com/coderhack-com?source=post_page-----31957242299f--------------------------------)

[Amay B](https://medium.com/@technocrat21?source=post_page-----31957242299f--------------------------------)

·

Follow

Published in

[CoderHack.com](https://medium.com/coderhack-com?source=post_page-----31957242299f--------------------------------)

·

7 min read

·

Sep 19

14

Cryptography is the practice of secure communication in the presence of adversaries. It allows us to encrypt sensitive data, verify the authenticity of messages, and generate random numbers for security purposes. In this section, we’ll explore some core cryptographic primitives in Rust using various crates.



Photo by [Bernard Hermant](https://unsplash.com/@bernardhermant?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral" \t "_blank)

**Symmetric Encryption**

Symmetric encryption algorithms use the same key for both encryption and decryption. We’ll look at two popular symmetric ciphers — AES and ChaCha20.

**AES Encryption**

AES or Advanced Encryption Standard is a widely used cipher for encrypting and decrypting data. We can implement AES in Rust using the crypto crate.

To encrypt data with AES, we first need to generate a random AES key. We can do so like this:

use crypto::aes;   
  
let key = aes::Key::generate();

Then we can encrypt data using encrypt and decrypt using decrypt:

let cipher = aes::cipher(key, aes::Mode::Encrypt, aes::Padding::PKCS7);  
let mut buf = [0; 16];  
cipher.encrypt(&mut buf, b"my secret message");  
  
let decipher = aes::cipher(key, aes::Mode::Decrypt, aes::Padding::PKCS7);   
let mut decbuf = [0; 16];  
decipher.decrypt(&mut decbuf, &buf);  
assert\_eq!(b"my secret message", &decbuf);

We encrypted 16 bytes of data in buf and then decrypted it back to the original message in decbuf.

**ChaCha20 Encryption**

[ChaCha20](https://en.wikipedia.org/wiki/Salsa20#ChaCha_variant) is a stream cipher that is faster than AES while providing similar security. We can implement ChaCha20 in Rust using the chacha20poly1305 crate.

To use ChaCha20, we need to generate a random nonce and key:

let nonce = chacha20poly1305::gen\_nonce();   
let key = chacha20poly1305::gen\_key();

Then we can encrypt and decrypt data like this:

let cipher = chacha20poly1305::ChaCha20Poly1305::new(key, nonce);  
let mut encrypted = vec![0; 16];  
cipher.encrypt(&b"secret message"[..], &mut encrypted);  
  
let decipher = chacha20poly1305::ChaCha20Poly1305::new(key, nonce);   
let mut decrypted = vec![0; 16];  
decipher.decrypt(&encrypted, &mut decrypted);   
assert\_eq!(b"secret message", &decrypted);

We encrypted 16 bytes of data in encrypted and decrypted it back to the original secret message.

**Hashing algorithms**

Hashing algorithms take input data of arbitrary size and generate a fixed-length “digest” or “hash value” as output. They are useful for checksums, message authentication, and more.

**SHA-2 family**

The SHA-2 family of hash functions (SHA-224, SHA-256, SHA-384, SHA-512) are popular and trusted. We can use the [sha2](https://docs.rs/sha2/0.8.2/sha2/) crate to generate SHA-2 hash digests in Rust:

use sha2::{Sha224, Sha256, Sha384, Sha512};  
  
// SHA-224 hash  
let mut hasher = Sha224::new();  
hasher.update(b"Hello");  
let hash = hasher.finalize();  
  
// SHA-256 hash   
let mut hasher = Sha256::new();  
hasher.update(b"World");  
let hash = hasher.finalize();  
  
// SHA-384 hash  
let mut hasher = Sha384::new();  
hasher.update(b"This is a message");   
let hash = hasher.finalize();   
  
// SHA-512 hash  
let mut hasher = Sha512::new();  
hasher.update(b"To generate hashes");  
let hash = hasher.finalize();

**BLAKE2**

BLAKE2 is a cryptographic hash function faster than MD5, SHA-1, SHA-2, and SHA-3, yet is just as secure. We can use the [blake2](https://docs.rs/blake2/0.8.1/blake2/) crate to generate BLAKE2 hash digests:

use blake2::{Blake2b, Digest};  
  
// BLAKE2b hash  
let mut hasher = Blake2b::new();  
hasher.update(b"Hello");  
let hash = hasher.finalize();

The BLAKE2b variant generates digests of 256 bits (32 bytes) in size.

**Argon2**

[Argon2](https://en.wikipedia.org/wiki/Argon2) is a key derivation function that was selected as the winner of the Password Hashing Competition in 2015. It is designed for hashing passwords and is resistant to GPU cracking attacks. We can use the [argon2](https://docs.rs/argon2/0.2.5/argon2/) crate to generate Argon2 password hashes:

use argon2::{self, Config};  
  
// Hash a password   
let config = Config::default();  
let hash = argon2::hash\_encoded(  
 "mypassword",   
 "somesalt",   
 &config  
).unwrap();  
  
// Verify a password   
let valid = argon2::verify\_encoded(  
 "mypassword",   
 hash.as\_str(),   
 "somesalt",  
).unwrap();

The argon2 crate supports the Argon2i (for password hashing), Argon2d (for password hashing and password hardening), and Argon2id (a hybrid of Argon2i and Argon2d) variants.

**Message Authentication Codes**

Message Authentication Codes or MACs are a way to verify the integrity and authenticity of messages. They use a secret key and a hash function to generate a MAC for the message. The recipient who has the same secret key can then verify the MAC to check that the message hasn’t been tampered with or is not from an unauthorized sender.

**HMAC**

The [hmac](https://crates.io/crates/hmac" \t "_blank) crate provides an implementation of the HMAC algorithm in Rust. HMAC uses a hash function (like SHA-1, SHA-256, etc.) and a secret key to generate the MAC.

use hmac::{Hmac, Mac, NewMac};  
  
// Generate a random secret key  
let key = rand::random::<[u8; 32]>();  
  
// Create a new HMAC-SHA256 instance   
let mut hmac = Hmac::<sha2::Sha256>::new\_varkey(&key).unwrap();  
  
// Update with message data and calculate final MAC  
hmac.update(b"my message");  
let mac = hmac.finalize().into\_bytes();

The recipient can then verify the MAC to authenticate the message:

let mut hmac = Hmac::<sha2::Sha256>::new\_varkey(&key).unwrap();  
hmac.update(b"my message");  
  
assert!(hmac.verify(&mac).unwrap()); // Success! Message is authentic  
  
let mut invalid\_hmac = Hmac::<sha2::Sha256>::new\_varkey(&key).unwrap();  
invalid\_hmac.update(b"tampered message");  
  
assert!(!invalid\_hmac.verify(&mac).unwrap()); // Failure! Message is invalid

**Poly1305**

The [chacha20poly1305](https://crates.io/crates/chacha20poly1305) crate provides an implementation of the Poly1305 MAC algorithm. Poly1305 uses a 128-bit (16 byte) secret key and a random 64-bit nonce to generate and verify MACs.

use chacha20poly1305::{ChaCha20Poly1305, Key, Nonce};  
  
// Generate a random secret key   
let key = Key::random();   
  
// Generate a random nonce   
let nonce = Nonce::random();   
  
// Create an AEAD instance with the key and nonce  
let aead = ChaCha20Poly1305::new(key, nonce);  
  
// Calculate the MAC (tag)  
let tag = aead.tag(b"my message");  
  
// To verify, re-create the AEAD instance and check the tag   
let aead = ChaCha20Poly1305::new(key, nonce);   
assert!(aead.verify\_tag(b"my message", tag));

Poly1305 produces a 128-bit (16 byte) MAC and provides strong security and performance benefits over HMAC.

**IV. Digital Signatures**

Digital signatures are a cryptographic mechanism used to verify the authenticity and integrity of digital messages or documents. They are the digital equivalent of handwritten signatures and provide non-repudiation, meaning the sender cannot deny sending the message.

**RSA Signatures**

The rsa crate can be used to generate and verify RSA digital signatures. Here's an example:

use rsa::{PublicKey, PrivateKey};  
  
// Generate 2048-bit RSA keys  
let (public\_key, private\_key) = rsa::generate\_keys(2048);  
  
// Sign a message with the private key  
let signature = private\_key.sign(b"message");   
  
// Verify the signature with the public key  
assert!(public\_key.verify(&b"message"[..], &signature[..]));

We first generate RSA public and private keys using rsa::generate\_keys(). We then sign a message with the private key using private\_key.sign() which returns a signature. Finally, we verify the signature with the public key using public\_key.verify(). If the signature is valid, it returns true.

**ECDSA Signatures**

The ecdsa and elliptic-curve crates can be used to generate and verify ECDSA digital signatures. Here's an example:

use ecdsa::{Signature, PublicKey, SecretKey};  
use elliptic\_curve::sec1::{FromEncodedPoint, ToEncodedPoint};  
  
// Generate ECDSA keys   
let secret\_key = SecretKey::new(&elliptic\_curve::secp256k1::curve(), &mut rand::thread\_rng());  
let public\_key = PublicKey::from\_encoded\_point(&secret\_key.public\_key());  
  
// Sign a message with the secret key   
let signature = secret\_key.sign(&b"message");  
  
// Verify the signature with the public key  
assert!(public\_key.verify(&b"message", &signature));

We first generate an ECDSA secret key and derive the public key from it. We then sign a message using secret\_key.sign() and verify it with public\_key.verify(). If the signature is valid, it returns true.

This covers the basics of generating and verifying digital signatures using RSA and ECDSA in Rust. Let me know if you would like me to explain anything in more detail!

**Random number generation**

Randomness is crucial for many cryptographic operations like generating encryption keys, nonces, salts, etc. We need cryptographically secure random number generators that generate unpredictable random numbers suitable for these use cases.

The rand crate provides multiple random number generators in Rust, including cryptographically secure ones.

To generate cryptographically random numbers, we can use the thread\_rng() function to get a handle to the OS's random number generator:

use rand::Rng;  
  
let mut rng = rand::thread\_rng();  
let random\_number = rng.gen::<u32>(); // Generate a random u32  
let random\_bytes = rng.gen::<[u8; 16]>(); // Generate 16 random bytes

The thread\_rng() uses operating system cryptographic random number generators like /dev/urandom on Linux and CryptGenRandom() on Windows.

For reproducibility, we can use a deterministic random number generator like StdRng:

use rand::SeedableRng;  
  
let mut rng = rand::rngs::StdRng::seed\_from\_u64(0);   
let random\_number1 = rng.gen::<u32>();  
  
let mut rng = rand::rngs::StdRng::seed\_from\_u64(0);   
let random\_number2 = rng.gen::<u32>();  
  
assert\_eq!(random\_number1, random\_number2); // Same random numbers!

The StdRng generates pseudorandom numbers from a seed in a reproducible fashion.

Using the rand crate, we can generate random numbers suitable for cryptographic as well as general-purpose use cases in Rust applications. The crate provides a variety of useful random number generation features for Rust developers.

**Libraries and Crates**

There are several crates available in Rust for cryptography. Some of the major ones are:

* openssl - Bindings to OpenSSL, the popular C library. Provides a wide range of algorithms and formats.
* ring - A safe, fast, small crypto library in Rust. It provides primitives for symmetric encryption (AES, ChaCha20), MACs (HMAC, Poly1305), digital signatures (Ed25519), and hash functions (SHA-2, Blake2).
* sodiumoxide - Bindings to the Sodium crypto library. It also provides primitives for symmetric/asymmetric encryption, hash functions, MACs, etc.
* libsecp256k1 - Efficient implementation of the secp256k1 ECDSA curve. Useful for Bitcoin and other cryptocurrencies.
* rsa - Pure Rust implementation of RSA encryption. Provides primitives for generating RSA keys and encrypting/decrypting data.
* chacha20poly1305 - ChaCha20-Poly1305 AEAD symmetric cipher.
* blake2 - BLAKE2 hash functions.
* sha-1 - SHA-1 hash function.
* sha2 - SHA-2 hash function family (SHA-224, SHA-256, SHA-384, SHA-512).
* hmac - HMAC implementation for SHA-1, SHA-2 and BLAKE2.
* pbkdf2 - PBKDF2 password-based key derivation function.
* argon2 - The Argon2 password hashing function.

To use these crates, simply add them as dependencies in your Cargo.toml file. For example, to use the ring crate, you'd add:

[dependencies]  
ring = "0.16.6"

Then you can use it in your code like this:

use ring::{rand, signature};  
  
fn main() {  
 // Generate an Ed25519 key pair   
 let (pk, sk) = signature::Ed25519KeyPair::generate\_pkcs8(&rand::SystemRandom::new());  
  
 // Sign a message   
 let msg = b"Hello World!";  
 let sig = signature::Ed25519::sign(&msg, &sk);   
}

The Rust crypto ecosystem provides a lot of choice and solid primitives for cryptography in Rust applications. Let me know if you have any other questions!

I hope this article has been helpful to you! If you found it helpful please support me with 1) click some claps and 2) share the story to your network. Let me know if you have any questions on the content covered.

Feel free to contact us at coderhack.com(at)xiv.in