Implementation ChaCha20-Poly1305 in Rust with a focus on Security

Le Duy Nguyen
Faculty of Information Technology
Brno University of Technology
Brno, Czech Republic
xnguye27@stud.fit.vut.cz

***Abstract*— In the realm of modern cryptography, the ChaCha20-Poly1305 AEAD construct has emerged as a robust and efficient authenticated encryption algorithm, providing confidentiality, authentication and integrity guarantees for sensitive data. In this paper, we present an implementation of ChaCha20-Poly1305 in Rust, focusing on security aspects. We delve into the security design and analysis of our Rust library, highlighting the steps taken to ensure a secure and efficient ChaCha20-Poly1305 implementation.**

I. Introduction

The increasing reliance on secure data transmission and storage has underscored the critical importance of implementing robust cryptographic solutions. Among the various algorithms available, ChaCha20-Poly1305 has gained recognition for its high performance and strong security properties, making it a preferred choice for securing data in various applications, including messaging platforms, network protocols, and file encryption. Additionally, ChaCha20-Poly1305 is among the authentication ciphers recommended by the Czech National Cyber and Information Security Agency (NÚKIB).

While Rust's reputation for security is widely recognized, the decision to implement ChaCha20-Poly1305 in Rust was primarily influenced by the language's zero-cost abstractions and its interoperability with C, enabling seamless integration with various programming languages.

II. The Algorithms

The subsections below describe the algorithms used and the AEAD construction.

For the pseudocode conventions:

- $a + b$ represents modular addition with a modulus of 2^{32}
- $a * b$ represents modular multiplication with a modulus of 2^{256}
- $a\%b$ represents the modulo operation
- $ao + b$ represents bitwise Exclusive OR (XOR)
- ab represents bitwise AND
- $a \ll b$ represents bit shifting to the left by b times
- $a \mid b$ represents vector concatenation

A. ChaCha20

ChaCha20, a symmetric stream cipher, was introduced by Daniel J. Bernstein in 2008[1] as an improved version of his Salsa20 cipher[2]. Unlike block ciphers, which operate on fixed-size blocks of data, ChaCha20 generates a keystream, allowing for the encryption and decryption of data on a byte-by-byte basis, which is critical in real-time communications.

The core of ChaCha20 is built around a 20-round function that shuffles the input data. This function relies on a 256-bit key, a 64-bit nonce, and a 64-bit counter, offering a substantial 256-bit security level.

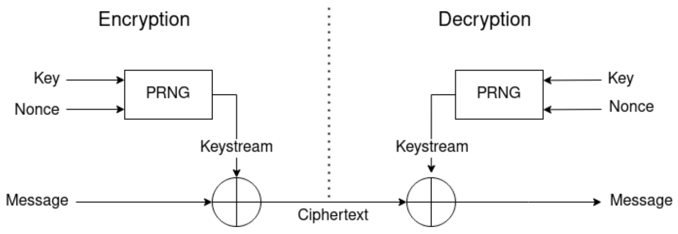

Fig 1: ChaCha20 Scheme

The efficiency of the ChaCha20 round function is notable, as it achieves full diffusion of any given input data by the 4th round, with this process persisting throughout all 20 rounds.

However, considering recommendations stemming from extensive cryptanalysis and the guidance of organizations like IETF, it is advisable to augment security by using a 92-bit nonce instead of the conventional 64-bit nonce while reducing the counter size to 32 bits.

Pseudocode

```
QuarterRound($a, b, c, d$):
1   $a \leftarrow a + b;$   $d \leftarrow d \oplus b;$   $d \leftarrow d \ll 16;$
2   $c \leftarrow c + d;$   $b \leftarrow b \oplus c;$   $b \leftarrow b \ll 12;$
3   $a \leftarrow a + b;$   $d \leftarrow d \oplus a;$   $d \leftarrow d \ll 12;$
4   $c \leftarrow c + d;$   $b \leftarrow b \oplus c;$   $b \leftarrow b \ll 12;$
```

```
GENERATE KEYSTREAM(key, nonce, counter):
state ← constants | key | counter | nonce
initial_state ← state
for $i$ ← 1 upto 10:
    // Column rounds
    QuarterRound(state[0], state[4], state[8], state[12])
    QuarterRound(state[1], state[5], state[9], state[13])
    QuarterRound(state[2], state[6], state[10], state[14])
    QuarterRound(state[3], state[7], state[11], state[15])
    // Diagonal rounds
    QuarterRound(state[0], state[5], state[10], state[15])
    QuarterRound(state[1], state[6], state[11], state[12])
    QuarterRound(state[2], state[7], state[8], state[13])
    QuarterRound(state[3], state[4], state[9], state[14])
    end
state ← initial_state
return serialize(state)
```

B. *Poly1305*

Poly1305, introduced by Daniel J. Bernstein in 2005[3], is a fast and secure message authentication code (MAC) designed for authenticating data and ensuring its integrity. Notably, Poly1305 is constructed based on the theory of universal hashing[4], which allows for efficient computation and provides strong security guarantees.

Poly1305 operates on a 256-bit one-time key and a message input of arbitrary length, generating a 128-bit authenticator as its output.

Pseudocode

```
POLY1305(key, msg):
$r$ ← le_bytes_to_num(key[0..15])
$r$ ← $r \wedge$ 0x0ffffffc0ffffffc0ffffffc0fffffff
$s$ ← le_bytes_to_num(key[16..31])
$a$ ← 0
for $i$ ← 1 upto ceil$\left(\frac{\text{msg length in bytes}}{16}\right)$
    $n$ ← le_bytes_to_num(msg[$((i-1)*16)..(i*16)$] | [0x01])
    $a \leftarrow a + n$
    $a \leftarrow (r * a)\%((1 \ll 130) - 5)$
    end
$a \leftarrow a + s$
return num_to_16_le_bytes($a$)
```

C. *AEAD Construction*

The ChaCha20-Poly1305 AEAD construct represents a powerful combination of the ChaCha20 stream cipher and the Poly1305 authenticator, providing a robust and efficient solution for achieving confidentiality, authentication and data integrity in secure communication protocols. Its usage in IETF protocols is standardized in RFC 8439[5].

The inputs are similar to ChaCha20, but with the addition of extra data (AD). The length of this additional data can range from 0 to $2^{\{64\}} - 1$ random bytes. The output is a ciphertext accompanied by an authentication tag.

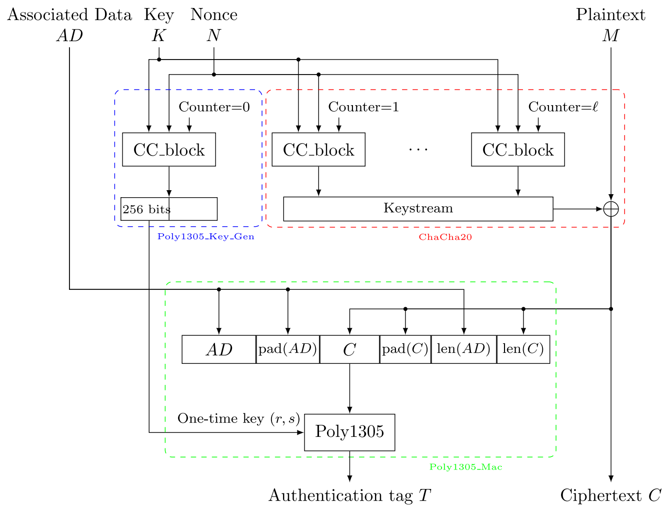

Fig 2: ChaCha20-Poly1305 AEAD construct. Taken from [6]

However, a limitation to be aware of is that this construction can only encrypt data up to 256 GiB. While this capacity is substantial for communication over networks, it may be restrictive for encrypting large volumes of data, such as encrypting an entire hard drive.

Pseudocode

```
POLY1305 KEY GEN(key, nonce):
counter ← 0
keystream ← chacha20_generate_keystream(key, nonce, counter)
return keystream[0..31]
```

```
CHACHA20 AEAD ENCRYPT(ad, key, nonce, msg):
otk ← Poly1305_key_gen(key, nonce)
ciphertext ← chacha20_encrypt(key, 1, nonce, plaintext)
mac_data ← ad | zero_padding_16(ad)
mac_data ← mac_data | ciphertext | zero_padding_16(ciphertext)
mac_data ← mac_data | num_to_8_le_bytes(ad.length)
mac_data ← mac_data | num_to_8_le_bytes(ciphertext.length)
tag ← poly1305_mac(mac_data, otk)
return (ciphertext, tag)
```

```
ChaCha20 AEAD decrypt($ad$, $key$, $nonce$, $msg$, $received\_tag$):
otk ← poly1305_key_gen(key, nonce)
mac_data ← ad | pad16(aad)
mac_data ← mac_data | ciphertext | pad16(ciphertext)
mac_data ← mac_data | num_to_8_le_bytes(ad.length)
mac_data ← mac_data | num_to_8_le_bytes(ciphertext.length)
tag ← poly1305_mac(mac_data, otk)
is_authenticated ← bitwise_compare(tag, received_tag);
ciphertext ← chacha20_encrypt(key, 1, nonce, plaintext)
return (ciphertext, is_authenticated)
```

III. Security Design

One of the paramount concerns in cryptographic algorithm implementations is guarding against side-channel attacks[7], with timing attacks being particularly worrisome due to their potential for remote exploitation. To mitigate the risk of timing attacks and enhance overall security, our ChaCha20-Poly1305 implementation adopts a security design that focuses on the following key principles:

A. Time Complexity Analysis

- Every operation involving secret keys or internal states must have a consistent time complexity of $O(1)$. This design principle is critical for thwarting timing attacks, which exploit variations in the execution time of cryptographic operations to deduce information about the secret key.
- Our approach involves analyzing every line of code that interacts with secret keys and internal states at the assembly level. The goal is to ensure that no variable-timed instructions are present in the code. Variable-timed instructions, such as divisions, modulo operations, and jumps (e.g., if statements, loops and function calls), introduce execution time variations that can be exploited by adversaries. We take meticulous care to eliminate such instructions for the target architecture.

B. Secure Memory Management

- Ensure that sensitive information about secret keys and internal stats is promptly and securely cleared from memory when it is no longer required. This process helps prevent potential information leakage through memory remnants or unauthorized access to sensitive data.
- This is especially challenging when dealing with compiler optimizations, since instructions for clearing memory often seen as “dead code”, because we modify the data but not consume it later, which is then being optimized away. This can introduce unintended vulnerabilities. More about it is in related literature[8]

C. Minimal Dependencies

- Minimizing dependencies on external libraries and components is another security-enhancing aspect of our design. Reducing reliance on external code mitigates potential vulnerabilities that could be introduced through external dependencies.
- In our implementation only use 1 dependency that is `zeroize`[9] for clean up memory securely as discussed above.

D. Testing

- A secure implementation is only as meaningful as its correctness.

IV. Realization in Rust

Rust’s inherent features have proven invaluable in realizing the security designs discussed above:

- **Modular arithmetic:** Rust provides a seamless way to perform modular arithmetic, or carryless arithmetic, using built-in functions `a.wrapped_operation(b)`. Simply replace “operation” with the appropriate arithmetic operation, such as `a.wrapped_add(b)`.
- **Inlining Sensitive Functions:** The process of inlining sensitive functions is simplified by adding `#[inline(always)]` at the top of the function declaration. This allows for code organization into functions for better readability and maintainability without concerns about function calls in assembly.
- **Unrestricted Loop Utilization:** Rust’s ability to accurately predict loop iterations allows us to utilize loops without restrictions. The compiler optimizes the loops by unrolling them, treating them as redundant when the number of iterations is predictable.
- **Drop Trait Implementation:** The `Drop` trait, Rust’s destructor function, is implemented for each data structure to ensure the secure clearing of sensitive data from memory. This is achieved with the assistance of the zeroize library[9].

For the Poly1305 component, the core of Poly1305-donna written by Andrew Moon[10] is ported due to its exceptional effectiveness in modular multiplication operations and is recommended by various papers including RFC 8439[5].

The implementation of other components, such as ChaCha20 and the AEAD construction, closely follows their specifications. These specifications were designed to be easily understood by computers, ensuring a straightforward and accurate translation into our implementation.

V. Usage

First, let's assume that we have these functions available for use, this in practice will be :

- `cryptographic_rng()` - generate random data securely
- `send()` - securely sending data
- `receive()` - securely receive data

To integrate this ChaCha20Poly1305 library into your Rust project, add it to the dependencies list in your project's `Cargo.toml` file:

```
[dependencies]
chacha20poly1305 = { git = "https://github.com/
tmokenc/VUT-FIT-SCO" }
```

A. Generate Documentation:

To explore the library's documentation, use the following command:

```
cargo doc --open --package chacha20poly1305
```

B. Module Import:

The library comprises three modules: `chacha20`, `poly1305`, and `chacha20poly1305`. You can either use a specific module or include them all with a wildcard import:

```
use chacha20poly1305::*;
```

C. ChaCha20

ChaCha20, being a stream cipher, operates on a keystream that is XORed with the message or ciphertext. This makes the encryption and decryption processes identical, and is unified as "`perform`" function in our implementation.

```
let (key, nonce) = cryptographic_rng();
let data = b"Your message or cipher text";
let mut cipher = ChaCha20::new(key, nonce);
match cipher.perform(&data) {
  Ok(result) => // Succesfully encrypted/decrypted
  Err(_) => // The data is too long
}
```

D. Poly1305

Poly1305 is a message authentication code used to calculate tags for messages. Here's how you can calculate and verify tags:

- Calculate tag of a message

```
let key = cryptographic_rng();
let message = b"Your message";
let mut mac = Poly1305::new(key);
mac.update(message);
let tag = mac.finalize(message);
send(key, message, tag);
```

- Verify tag

```
let (key, message, tag) = receive();
let mut mac = Poly1305::new(key);
mac.update(message);
if !mac.verify(message, tag) {
  // Tags not match
}
```

E. AEAD Construction

Encryption and decryption use the same interface but involve different functions. It's crucial to ensure that these two operations are not mixed up.

- Encrypting

```
let (key, nonce, aad) = cryptographic_rng();
let message = b"Your message";
let mut cipher = ChaCha20Poly1305::new(key,
nonce, aad)?;
match cipher.encrypt(message) {
  Ok(ciphertext) => // Success,
  Err(why) => // Something went wrong,
}
let tag = cipher.finalize();
```

- Decrypting

```
let (key, nonce, aad, cipher_text, tag) = receive();
let mut cipher = ChaCha20Poly1305::new(aad, key,
nonce)?;
match cipher.decrypt(cipher_text) {
  Ok(message) => // Success
  Err(why) => // Something went wrong
}
match cipher.verify(cipher_text) {
  Ok(()) => // Succesfully
  Err(why) => // The message was tamped
}
```

There are also "oneshot" functions that perform both encryption/decryption and tag generation/verification simultaneously. These include:

- `ChaCha20Poly1305::encrypt_oneshort`
- `ChaCha20Poly1305::decrypt_oneshort`

F. *In place operation*

Every encryption/decryption function has an in-place version denoted by adding the postfix in_place to the function name. These functions transform the data directly, taking a mutable reference `&mut [u8]` instead of an immutable reference `&[u8]`. Supported in-place functions include:

- `ChaCha20::perform_in_place`
- `ChaCha20Poly1305::encrypt_in_place`
- `ChaCha20Poly1305::decrypt_in_place`
- `ChaCha20Poly1305::encrypt_oneshot_in_place`
- `ChaCha20Poly1305::decrypt_oneshot_in_place`

```
let (key, nonce) = cryptographic_rng();
let mut data = b"Your message or cipher text";
let mut cipher = ChaCha20::new(key, nonce);
match cipher.perform_in_place(&mut data) {
  Ok(_) => // `data` is now the result
  Err(_) => // The data is too long
}
```

G. *Embedded environment*

The library is also compatible with embedded environments. To use it in such environments, disable the default features in the `Cargo.toml` dependencies list:

```
[dependencies]
chacha20poly1305 = { git = "https://github.com/
tmokenc/VUT-FIT-SCO", default-features = false }
```

Note that this disables the "`alloc`" feature, allowing only in-place encryption/decryption due to the lack of functionalities for allocating messages. Refer to the Security Analysis section for more details on the security implications of this configuration.

VI. Recommendations

The `Nonce` plays a pivotal role, and it is crucial to generate a unique `Nonce` for each use by a Cryptographic Random Number Generator[11]. This practice safeguards against various attacks, including well-known Relay Attacks[12].

While the `Key` does not necessitate unique generation each time like the `Nonce`, it is advisable to generate it randomly alongside the `Nonce`. This enhances the overall security posture of the ChaCha20-Poly1305.

In Poly1305, you can technically call "`finalize()`" to get the tag then compare with other tag manually. but this will allow attack to reduce the number of variants by timing the comparision speed. Always use "`verify(tag)`" instead as it compares tags in $O(1)$.

It is crucial to emphasize that while the AEAD construct of ChaCha20-Poly1305 provides robust confidentiality, integrity, and authentication, it does not offer non-repudiation—a objective in cryptography. To augment the security model, it is recommended to complement ChaCha20-Poly1305 with a digital signature algorithm such as DSA[13].

VII. Security Analysis

In this section, we conduct an analysis of the security aspects of our ChaCha20Poly1305 implementation.

A. *Timing Attack Resilience*

The security designs implemented in our library effectively neutralize timing attacks. By ensuring that the time complexity varies solely based on the length of the data and not the data itself or the internal states of the cipher, we eliminate vulnerabilities associated with variations in execution time. Through analysis at the assembly level, we confirm the absence of variable-timed instructions, such as divisions, modulo operations, and conditional jumps. This meticulous approach safeguards against potential timing side-channel attacks.

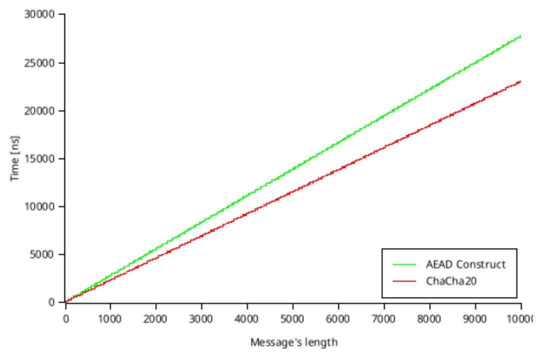

Fig 3: Execution time over message's length

It is important to note that while the Rust compiler can compile this implementation for a wide range of architectures[14], the timing attack resilience has been verified only for the x86 CPU family. The implementation has not been tested against other architectures.

B. *Power Analysis Vulnerabilities*

Despite the resilience to timing attacks, our implementation is susceptible to power analysis, as highlighted by the cryptanalysis work of Bernhard Jungk and Shivam Bhasin[15]. Power analysis exploits fluctuations in power consumption to deduce information about cryptographic operations. It's essential to acknowledge that power analysis requires physical access to the target system. Consequently, users take this potential threat into consideration when deploying this ChaCha20Poly1305 implementation.

C. *Test Coverage*

To validate the robustness of our implementation, we leverage extensive test cases outlined in RFC 8439[5]. These tests cover a myriad of scenarios, including many edge cases, ensuring a thorough examination of the correctness and security of our ChaCha20Poly1305 construction.

VIII. Future Work

Several possibilities for future development exist, including:

A. *Mitigating Power Analysis*

Our top priority is to address potential vulnerabilities to power analysis, a technique that can be countered through the implementation of masking techniques.

B. *Key and Nonce Generation*

The ChaCha20 diffusion algorithm, known for its exceptional effectiveness, can be modified to securely generate cryptographic keys and nonces.

C. *Hardware Acceleration*

Hardware acceleration, particularly through SIMD (Single Instruction, Multiple Data)[16] optimizations, holds great promise for enhancing the computational efficiency of the ChaCha20Poly1305 implementation.

D. *C Interface for Cross-Language Compatibility*

Expanding the usability of the current ChaCha20Poly1305 implementation involves creating a C interface to enable seamless integration with other programming languages. This step will enhance interoperability and facilitate the use of our cryptographic solution in diverse software ecosystems.

E. *Expand Architectural Support*

Extending compatibility to various architectures ensures the versatility and applicability of the ChaCha20Poly1305 implementation across a diverse range of computing platforms.

F. *Continuous Security Monitoring and Enhancement*

To fortify cryptographic implementations, continuous security monitoring is essential. Future efforts will focus on a vigilant approach to security, incorporating the latest cryptographic research advancements and promptly addressing emerging threats or vulnerabilities.

IX. Conclusion

This implementation offers a streamlined and user-friendly interface for ChaCha20, Poly1305, and their AEAD construction. Its simplicity makes it suitable for diverse scenarios, particularly for users seeking a library that works effortlessly. As of the time of writing this paper, it is recommended for use exclusively on remote platforms to mitigate potential threats posed by power analysis.

References

[1] D. J. Bernstein, "Chacha, a variant of salsa20." https://cr.yp.to/chacha/chacha-20080120.pdf (accessed: Nov. 9, 2023).

[2] D. J. Bernstein, "The salsa20 family of stream ciphers." https://cr.yp.to/snuffle/salsafamily-20071225.pdf (accessed: Nov. 1, 2023).

[3] D. J. Bernstein, "The poly1305-aes message-authentication code." https://cr.yp.to/papers.html#poly1305 (accessed: Nov. 9, 2023).

[4] Wikipedia, "Universal hashing." http://en.wikipedia.org/w/index.php?title=Universal\%20hashing&oldid=1182493673 (accessed: Nov. 9, 2023).

[5] Y. Nir, and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols," RFC Editor, 2018. [Online]. Available: https://www.rfc-editor.org/info/rfc8439 (RFC 8439)

[6] Wikipedia, "Chacha20-poly1305." http://en.wikipedia.org/w/index.php?title=ChaCha20-Poly1305&oldid=1182313000 (accessed: Nov. 9, 2023).

[7] Wikipedia, "Side-channel attack." http://en.wikipedia.org/w/index.php?title=Side-channel\%20attack&oldid=1182539399 (accessed: Nov. 9, 2023).

[8] C. Percival, "How to zero a buffer." https://www.daemonology.net/blog/2014-09-04-how-to-zero-a-buffer.html (accessed: Nov. 9, 2023).

[9] T. Arcieri, and rustcrypto, "Zeroize." https://crates.io/crates/zeroize (accessed: Nov. 9, 2023).

[10] A. Moon, "Poly1305-donna." https://github.com/floodyberry/poly1305-donna (accessed: Nov. 9, 2023).

[11] Wikipedia, "Cryptographically secure pseudorandom number generator." https://en.wikipedia.org/wiki/Cryptographically_secure_pseudorandom_number_generator (accessed: Nov. 14, 2023).

[12] Wikipedia, "Relay attack." https://en.wikipedia.org/wiki/Relay_attack (accessed: Nov. 14, 2023).

[13] Wikipedia, "Digital signature algorithm." https://en.wikipedia.org/wiki/Digital_Signature_Algorithm (accessed: Nov. 14, 2023).

[14] R. L. Team, “Rustc platform support.” https://doc.rust-lang.org/nightly/rustc/platform-support.html (accessed: Nov. 9, 2023).

[15] B. Jungk, and S. Bhasin, “Don’t fall into a trap: physical side-channel analysis of chacha20-poly1305.” https://past.date-conference.com/proceedings-archive/2017/pdf/7031.pdf (accessed: Nov. 9, 2023).

[16] Wikipedia, “Single instruction, multiple data.” https://en.wikipedia.org/wiki/Single_instruction,_multiple_data (accessed: Nov. 14, 2023).