Master thesis
Computing Science

Radboud University

Tink Outside the Deck

Author:
Dor Mariel Alter
s1027021
dor.alter@ru.nl

First supervisor/assessor:
Prof. dr. Joan Daemen
j.daemen@cs.ru.nl

Second supervisor:
dr. Silvia Mella
silvia.mella@ru.nl

Second assessor:
dr. Bart Mennink
b.mennink@cs.ru.nl

September 17, 2024

Abstract

To ease the integration of cryptographic schemes into different applications, Google introduced a new cryptographic library called Google Tink, which includes cryptographic schemes for different tasks. For the task of authenticated encryption, Google introduced Google Tink streaming Authenticated Encryption with Associated Data (AEAD). In this thesis, we examine the history of Google Tink AEAD's security and the construction that inspired Google Tink developers. To prove the security of Google Tink AEAD, a new security notion, nOAE2, had to be introduced from the existing nOAE notion. We believe a better approach is to think about the security proof while designing the scheme. In this thesis, we introduce a new authenticated encryption mode called Deck-Tink, for which we prove a security bound using the jammin cipher as a model. We go further and provide an instantiation for the mode called Xoofff-Tink, which uses Xoofff as its deck function. Xoofff-Tink provides additional features that other deck-based authenticated encryption modes, e.g. Deck-Plain, do not possess, such as out-of-order and lost messages. We implement our Xoofff-Tink scheme in C and Rust to increase its usability and integration potential. We go on to provide an analysis of performance benchmarks between Google Tink streaming AEAD and Xoofff-Tink for the C version and a comparison to various authenticated encryption schemes for the Rust variant. We also discuss our collaboration with iHub, where we successfully integrated Xoofff-Tink into their PostGuard project, demonstrating its practicality and ease of adoption in real-world applications.

Contents

Chapter 1

Introduction

In the contemporary field of digital security, cryptography stands as the cornerstone of protecting data confidentiality, integrity, and authenticity. As cyber threats evolve in complexity and sophistication, the development of advanced cryptographic solutions becomes essential. Among these solutions, Authenticated Encryption (AE) has emerged as a possibility for integrating encryption with authentication to ensure both privacy and data integrity in a single streamlined process. Traditional encryption techniques are effective in concealing content, but they often lack mechanisms to verify the integrity and authenticity of data. While authentication methods could ensure data integrity, they do not protect content privacy. AE unifies these capabilities, offering a holistic security solution that encrypts data and simultaneously ensures its integrity.

Today, the landscape of cryptography and AE schemes is marked by various methodologies designed to meet different security needs and operational environments. Modern AE schemes, such as Galois/Counter Mode (GCM) [MV05] and ChaCha20-Poly1305 [NL18], are widely implemented in protocols like Transport Layer Security (TLS) and provide robust, efficient, and scalable security measures.

To ease the integration of cryptographic schemes into different applications, Google introduced a new cryptographic library called Google Tink, which includes cryptographic schemes for different tasks. "It was born out of our extensive experience working with Google's product teams, fixing implementation weaknesses" [Goo23b]. The core idea is to have one library with different functions, which has a variety of cryptographic schemes that are easy to use, secure, and compatible with existing cryptography libraries [Goo23a]. For authenticated encryption, Google provides Google Tink streaming Authenticated Encryption with Associated Data's (AEAD) scheme.

In this thesis, we examine the history of Google Tink AEAD's security and the construction that inspired Google Tink developers. We discuss the history of notions that led to the notion being used for Google Tink AEAD security-bound proof. We start with the first definition of the security notion OAE, then the improved version OAE2, the looser version nOAE, and finally, the version Google Tink AEAD is compared to, i.e., nOAE2. The security bound of Google Tink was proven in [HS20], where the authors created their ideal world inspired by nOAE [HRRV15] and shaped it into a version that fits the scheme's design. We believe we ought to do better and think about the proof when designing a scheme. As we see it, a scheme's security proof should come naturally from the scheme itself, and one should consider the proof while designing the scheme, not design and then prove it. We believe an ideal model should be practical to ensure it can be compared to real-world designs without the need to create a new model, as was done between the nOAE and nOAE2.

To that end, we develop a new authenticated encryption mode that we call Deck-Tink, which has proof of security by design and uses the jammin cipher [BDH+22] as a model for its security proof. We go further and provide an instantiation for the mode called Xoofff-Tink, which uses XOOFFF as its deck function [Alt24]. Xoofff-Tink provides additional features that other deck-based AE schemes do not possess, such as out-of-order and lost messages. Google Tink streaming AEAD inspired us to write this thesis as it showcases the approach of design and then prove, therefore, it is important to mention it.

We implement our Xoofff-Tink scheme in two languages, C and Rust, to increase its usability and integration potential. We also benchmark the performance of Google Tink stream AEAD and compare it to Xoofff-Tink. As Google Tink does not have a C implementation, we will compare our C implementation of Xoofff-Tink to the C++ implementation of Google Tink. We do not expect to see a significant improvement in performance as Google Tink uses AES, which is implemented in hardware for most higher-level devices. Nevertheless, it is essential to report the speed performance of the two schemes, as speed is an important aspect in today's cryptographic world and plays a role when choosing one scheme over the other.

Another important aspect of an implementation is the ease of use. To that end, we collaborated with iHub on a project to showcase our implementation and its practicality. iHub is a Radboud interdisciplinary research hub on digitalization and society. They are currently working on different projects with a variety of goals. We collaborated on their PostGuard project [BBJ+23], which aims to prove an easy-to-use email platform that allows encryption and decryption of emails.

Our primary goal in this collaboration is to provide a real-life example of how easy it is to use our new implementation and measure the performance of our scheme compared to the scheme used in PostGuard, AES-GCM. We decided to go further and benchmark our Rust implementation against more popular authenticated encryption schemes. As our scheme is relatively new, it is essential to be able to place it among the more well-known authenticated encryption schemes, and by benchmarking the performances, we can achieve that. We picked Ascon, AES-GCM, AES-GCM-SIV, and Chacha20Poly1305 for that end. We used the implementation from the Rust Crypto AEAD library [dev23], which is becoming the standard library for cryptography in Rust these days.

The thesis is structured in the following way: in Chapter 2, we explain the working of stream encryption, Mac function, and deck function. In Chapter 3, we introduce the notions of AE and the nOAE2 notions used by Google Tink. In Chapter 4, we introduce the jammin cipher, which is used in the security analysis of Deck-Tink. In Chapter 5, we explain Google Tink AEAD and the scheme that inspired its design. In Chapter 6, we discuss a mode of AE scheme using a deck function and the deck function we used for our instance later in the thesis. In Chapter 7, we provide the specification for our mode Deck-Tink, and in 8, we analyze its security. In Chapter 9, we discuss an implementation of an instance of Deck-Tink called Xoofff-Tink and in Chapter 10, we provide the result of benchmarking it against various schemes. Finally, in Chapter 11, we discuss the collaboration results with iHub, and in Chapter 12, we conclude the thesis.

Chapter 2

Stream Encryption and MAC functions introduction

2.1 Stream Encryption

Encryption has existed for over 4,000 years in many different variants [PNT$^+$15]. One variant believed to have been used by Julius Caesar was called Caesar cipher. The idea is simple: one picks a letter x and then encrypts by replacing each letter in the message with the letter x positions down the alphabet. For instance, if a equals 1, the sentence "bob" becomes "cpc". This idea was improved by the Vigenere cipher, used by Giovan Battista Bellaso in the 16th century [AO16]. The same encoding is used for the letter, but instead of a number, a, a keyword is used, and we shift each letter in the sentence based on the corresponding letter in the keyword. We first repeat the word as many times as needed to get the length of the sentence and then shift the corresponding letter. For example, if the word is *key* and we encrypt hello, we will have h shifted k (11) times to the right, e is shifted by e (5) times, and so on. If the keyword is used only once and has the size of the sentence, then it is the encryption technique we call today one-time-pad.

Although this cipher is too primitive for our days, as it would require the key material to be as long as the text, its idea of using one-time-pads is still common today. Today, instead of using letters, we map the message, which we call plaintext, into bits and use that for encryption/decryption. Simply put, a one-time-pad takes a key value, called keystream, and performs XOR with the plaintext to get the ciphertext, denoted by C. The security of this technique depends on the randomness of the key. If the key is not random, one might be able to easily guess it and decrypt the ciphertext back to the original message. If we know that the key is a value between zero and ten, we will need at most ten guesses to find the key rather than two to the power of the length of the key, assuming the input/key is long enough. However, if the plaintext is an entirely random value, then the attacker that obtains the

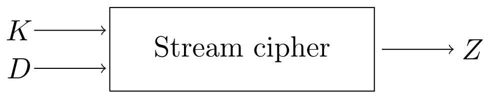


Figure 2.2: Illustration of a modern stream cipher.

ciphertext cannot decrypt it unless provided with the keystream.

To reduce the amount of time the keystream needs to be shared, one usually uses a key K, which is communicated once and future keystreams are then generated based on K. The algorithm that generates keystreams from the key, K, is called stream cipher. A stream cipher is a cryptographic primitive that takes a key in defined domain space $\{0,1\}^l$ and returns an output in some other specified domain. Given a key K of short fixed length l, SC returns an arbitrary long keystream Z, $SC : \{0,1\}^l \to \{0,1\}^*, K \mapsto Z$.

For example, consider the case where Alice and Bob agreed on a key K of length 128 bits. Alice wishes to encrypt a 2 MB file of text, simply using K, is not possible as it is too short. She, therefore, uses the following stream cipher:

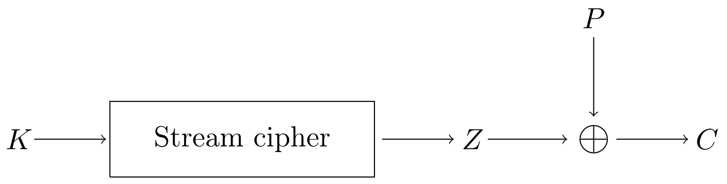


Figure 2.1: Illustration of how a ciphertext is generated using stream encryption.

Alice then sends C to Bob. Bob performs the same operations, with the only difference being that instead of P, he uses C and will receive the original message.

Modern stream ciphers also include a diversifier, D, in the input to generate multiple keystreams per key. If we use the same key, we will get the same keystream every time, making it easier to guess. To avoid using different keys for each message, the diversifier was introduced. For each plaintext, a different diversifier is used, thus resulting in different keystreams that can be XORed with the original message. An example can be seen in Figure 2.2.

2.2 MAC function

While a stream cipher provides us with a keystream to encrypt a message, it does not authenticate it. For authentication, the receiver should be able to determine who wrote the message. To that end, one can append an authentication token or a signature. In symmetric cryptography, the token/signature

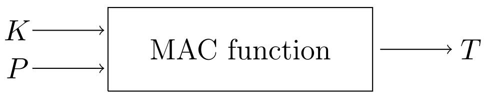


Figure 2.3: Illustration of a MAC function.

is called either a message authentication code (MAC) or a tag. In this thesis, we use the name tag for the authentication token, denoted by T. A MAC function is a cryptographic primitive that takes an arbitrarily long plaintext, P, and a key, K, and returns a fixed-size tag, T. An illustration can be seen in Figure 2.3.
Consider the case where Alice wishes to send a message to Bob, and Bob wants to ensure that the message indeed arrived from Alice. Alice and Bob already share a secret key, which we will call K. Alice will make a call to the MAC function with secret key K and plaintext P, obtaining $\mathrm{MAC}_\mathrm{K}(P) = T$ different notation exists for the MAC function such as $\mathrm{MAC}(K, P)$. Throughout this thesis, we will stick to the $\mathrm{MAC}_\mathrm{K}(P)$ notation as we do not often change the key. Therefore, we consider the plaintext to be the only inputted variable. Alice will then send to Bob (P, T). Bob can verify if the plaintext comes from Alice by computing $\mathrm{MAC}_\mathrm{K}(P) = T'$ and verifying it with the received tag T. In a good MAC function, if the plaintext is modified, the computation of T' will differ from T, and assuming that K is secret, using a MAC function with random K is unlikely to result in a computation that will verify correctly (considering that the tag is sufficiently long).

Some MAC functions require an additional nonce for their security. Similar to the idea we saw with the diversifier for stream ciphers, for some MAC functions, we want to use a nonce to ensure the safe reuse of their key. Consider the case of a MAC function that performs XOR between the plaintext and the key. If an attacker uses this method to authenticate the stream of all zeros, they will get the key. Some MAC functions can prevent such cases and ensure their security even if the key is reused by adding a nonce to the computation. This nonce should be checked by the sender to guarantee they will not use the same nonce twice. Considering this approach, we modify the definition of the MAC function to take not only a key and a plaintext but also a nonce, such that $\mathrm{MAC}_K(N, P) = T$.

2.3 Deck-function

Doubly-Extendable Cryptographic Keyed function, known as deck function, is a keyed function that takes a sequence of strings and returns a pseudorandom string of arbitrary length [BDH+22]. Formally defined, we can see a deck function as a function F that takes as input a secret key $K \in \mathcal{K}$, where $\mathcal{K}$ represents the set of all possible keys and a sequence of an arbitrary number of strings $X^{(0)}; \ldots; X^{(m-1)} \in (\mathbb{Z}_2^*)^+$, where $(\mathbb{Z}_2^*)^+$ is the set of all

bit strings containing at least one string, and produces a string of bits of arbitrary length and takes from it the range starting from a specified offset $q \in \mathbb{N}$ and for a specified length $n \in \mathbb{N}$ [BDH+22]. We denote this as $Z = 0^n + F_K(X^{(0)}; \ldots ; X^{(m-1)}) \ll q$, where 0^n means that the length of Z will be n and $\ll q$ specifies the range of the offset.
Deck functions can be used for different purposes. We focus on the deck function as a stream cipher, MAC function, and in modes for authenticated encryption. We can use the deck function as a stream cipher in the same manner as we have seen above in 3.3. We have a diversifier D, and we encrypt by XORing the result of F on D with the message M, $C = M + F_K(D)$ and decrypt in the same way $M = C + F_K(D)$. We can use the deck function as MAC by applying F on the message instead of on a diversifier, $T = 0^t + F_K(M)$. We verify the tag by computing $T' = 0^t + F_K(M)$ and comparing it to T.

If we combine those two into one ciphertext, say $C = M + F_K(D)||0^t + F_K(M)$, then we have an instantiation for a mode of authenticated encryption. By changing the order of operations we can create a different modes. For example, if ciphertext is computed as $C = M + F_K(A)$, where A is the associated data containing a nonce, and the tag is computed as $T = 0^t + F_K(A; C)$. Those are mere examples of how to use a deck function for AE, in different modes, more about this will be discuss in section 3.2. Where we will see that the first example is an instance of the Encrypt and MAC (E&M) mode and the second one is an instance of the Encrypt then MAC (EtM) mode. More possibilities are available, and depending on the task at hand, one might prefer one over another. The latest approach is usually chosen because when decrypting, we can first authenticate and verify the tag and only then decrypt, ensuring that if the tag is invalid, we do not reveal anything about the message or the key.

Chapter 3

Authenticated Encryption and the nOAE2 Notions

Before we discuss Google Tink Authenticated Encryption with Associated Data scheme, we explain the security claim it is trying to achieve and the ideal-world schemes it is compared to. The security notion is called nonce-based online authenticated encryption 2, nOAE2. We start with explaining distinguishing models in section 3.1, which is used in most security analyses including nOAE2. We then define authenticated encryption and its modes in sections 3.2 and 3.3. Finally, in sections 3.4, we provide the evolution of the nOAE2 notion.

3.1 Distinguishing Models

Before discussing any security notions, we need to investigate how they can be used to prove a security bound for a mode. We model a scheme's security by creating a game, also called an experiment, in which an adversary, Eve, is provided with access to two different schemes. The first is the ideal world, and the second is the scheme for which we wish to express a security bound, called the real-world scheme. Eve does not know which scheme she is communicating with. At the beginning of the game, the scheme is randomly picked with a probability of 50%. Eve then sends her messages to an interface that forwards them to either the ideal world or the real world, depending on the scheme picked. Eve's goal is to determine if she is taking to the real-world or ideal-world scheme. She can send several queries to the interface, and eventually, she must provide her guess. Without any knowledge, Eve has a success probability of $\frac{1}{2}$ as there are two options, and if she randomly picks one, the probability of it being correct is $\frac{1}{2}$. So her goal is to be able to guess to which scheme she is talking with probability which is greater than $\frac{1}{2}$. We call the advantage, denoted by δ between the real world and the ideal world schemes, twice the difference between Eve's probability of guess and $\frac{1}{2}$.

To make this idea more concrete, we provide the following example where we instantiate the two schemes: the real world with the stream cipher we have seen in 2.1 and the ideal one with the following model we call random oracle. A random oracle is an ideal cryptographic primitive, denoted by $\mathcal{RO}$, that generates a random response to each query and the same response for queries with the same value. It can generate arbitrary long output, which we denote by Z. Given a message M and a length l, it generates a l bits long output string Z. $\mathcal{RO}$ keeps an archive map AR of the messages it received, M, and the outputs it generated, Z. Initially, this archive is empty. The procedure of $\mathcal{RO}$ is as follows:

1. If M is not in archive AR, randomly generate l bit long string Z and store (M, Z) in AR.
2. If M is in AR, take the generated Z. If Z length is l, return Z. If Z is larger than l, return the first l bits of it. If it is smaller than l, randomly generate the missing bits, append them at the end, update Z in AR, and return Z.

Random oracle distinguishability is modeled by following the game in Figure 3.1 where adversary Eve is denoted as $\mathcal{A}$. Eve communicates with an interface that either forwards the communication to SC_K or $\mathcal{RO}$. We generate a random bit $b \xleftarrow{\$} \{0,1\}$ and select SC_K if $b = 1$ and $\mathcal{RO}$ if $b = 0$. As described above, Eve can send as many queries to the interface as she wishes, but at the end, she needs to provide a guess bit $b' \in \{0,1\}$. Eve is successful if $b' = b$, denoted by the event *success*. We now can formally express Eve's advantage as the probabilistic difference between the real world and the ideal model: $\Delta_{\mathcal{A}}(SC_K; \mathcal{RO}) = \mathbf{Pr}(\mathcal{A}^{SC_K} = 1) - \mathbf{Pr}(\mathcal{A}^{\mathcal{RO}} = 1)$. In words, this distance is the difference between the probability that Eve returns $b' = 1$ in the real world minus the probability that she returns $b' = 1$ in the ideal world scheme.

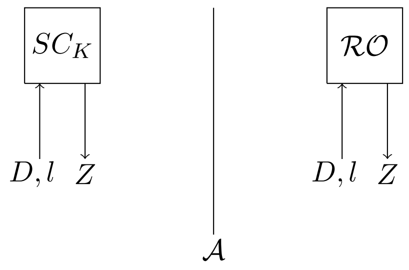


Figure 3.1: Illustration of the distinguishing game between stream cipher SC and random oracle $\mathcal{RO}$.

Random oracle distinguishability is an example of how we can model the security of schemes like a stream cipher, where the random oracle is the ideal world scheme. For different schemes, we may need to create other ideal worlds, such that both the real and ideal worlds share the same interface. This is because the random oracle might be too simplistic, or the schemes may possess additional properties we wish to distinguish. In this thesis, we mainly focus on AE, and therefore, we are looking at ideal schemes that can be used to distinguish and prove the security bounds of such AE schemes and we give the first example of such scheme in section 3.4 and another in Chapter 4.

3.2 Authenticated Encryption

Following the discussion on stream encryption and MAC functions, we next look at authenticated encryption. Authenticated encryption is achieved when one not only encrypts the message but also provides means to authenticate its content. One way to achieve such a scheme is by combining the stream encryption and MAC function. We can send a tag T alongside the encrypted message that allows the receiver to verify the message's origin. Using a Tag is one way of performing AE, but in some cases, the ciphertext itself has the means to allow the verification step.

Often, AE schemes include an additional parameter called associated data, which adds some additional non-confidential information about the message and is sent in plaintext to be used for the authentication part. Some AE schemes also use a nonce for their security. This is not mandatory, but all the schemes in this thesis used it, so we include it in our definition. Given a key K, a nonce N, an associated data A, and a plaintext P, the sender applies the wrap operation of the AE scheme to generate ciphertext C, as can be seen in Figure 3.2.

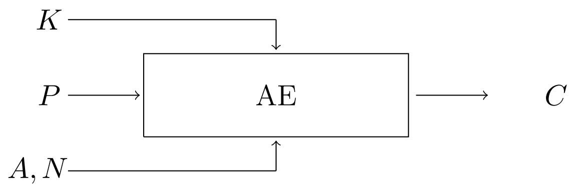


Figure 3.2: Illustration of wrap operation of a nonce-based authenticated encryption scheme.

The receiver will then unwrap the ciphertext to retrieve the plaintext if the ciphertext is valid or an error otherwise, as can be seen in Figure 3.3.

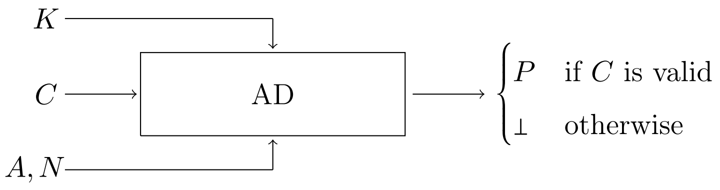


Figure 3.3: Illustration of unwrap operation for a nonce-based authenticated encryption scheme.

3.3 Authenticated Encryption Modes

There are different modes for building an authenticated encryption scheme based on underlying primitives. For simplicity, we use the two schemes we mentioned above, stream encryption, denoted by SC, and MAC function, denoted by MAC, and use the concatenation operation, $||$, to encode the associated data and the message. Three common approaches for creating authenticated encryption, which are usually called modes of use in authenticated encryption, are [BN00]:

- Encrypt-and-MAC (E&M) - on input of a nonce N, associated data A, and plaintext P, compute:

$$C = P + \mathrm{SC}_K(N)$$
$$T = \mathrm{MAC}_L(A||M)$$

- Mac-then-Encrypt MtE - on input of a nonce N, associated data A, and plaintext P, compute:

$$T = \mathrm{MAC}_L(A||M)$$
$$C = T||P + \mathrm{SC}_K(N)$$

- Encrypt-then-Mac EtM - on input of a nonce N, associated data A, plaintext P, compute:

$$C = P + \mathrm{SC}_K(N)$$
$$T = \mathrm{MAC}_L(A||C)$$

Above, we provide the specification for the wrap operation, the unwrap is left as an exercise for the reader.

Another common notion for authenticated encryption is being online. For an AE scheme to be online, we require that it allows the un/wrap operation to be implemented with a constant memory and a single one-direction pass over the plaintext (ciphertext), writing out the result during that pass [Boz24]. For wrap, this implies that one can go through the plaintext without saving it and generate a ciphertext in a single pass with constant memory. For unwrap, it means that if one splits a ciphertext into smaller segments and sends each, the receiver can unwrap each segment without waiting for the whole ciphertext to arrive. Consider the following example: we would like to wrap, send, and unwrap a rather big message (e.g., a movie of a few gigabytes). If our scheme is online, we can slice the movie into smaller chunks of data (each a few seconds/minutes, e.g. a few kilobytes), wrap each, and send them one by one to the receiver, who unwraps them and displays the movie to the user. Such a property allows us to have a small memory buffer on the receiver rather than wait and store the cryptogram of the whole message in memory before being able to unwrap it. This also derives another concept called session. Session has multiple definitions depending on the context and the scheme used. We distinguish between processing and communication sessions to avoid confusing the two definitions.

- When talking about authentication sessions, we refer to authenticating a message in the context of previously sent ones within the sequence. In such cases, the cryptogram C_3 authenticates the sequence $M_1; M_2; M_3$ (where $M_i = P_i; A_i$).
- When using communication sessions, we refer to the sequences of messages from when a party initialized their communication until they finalized it. If they later decide to continue their communication, a new communication session should be started.

3.4 Nonce-Base Online Authenticated Encryption notions

In this section, we discussed the different online authenticated encryption (OAE) notions and the most important one, nOAE2, which was used by Google Tink AEAD. We start by providing the syntax and notation used in this section in section 3.4.1 and then talk about the OAE notions in section 3.4.2. Notice that in the context of OAE and Google Tink we use the term encryption to describe wrap and decryption to describe unwrap as those are the terms used in their specifications.

3.4.1 Syntax and Notations

We start by defining a message M and a segment of a message, M_i, such that M_i is a string, and M is a sequence of segments $M = M_0 M_1, \ldots, M_l$, where l is the length of the message divided by the segment length. An empty message is an empty sequence rather than an empty string. We denote the numbers of segments in the message X as $|X|$, and the ith segment of message X as $X[i]$.

Following the notation from [HRRV15] we denote an AE scheme in the OAE notions using $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ where the key space $\mathcal{K}$ is a nonempty set with an associated distribution, and the encryption $\mathcal{E}$ and decryption $\mathcal{D}$ both consist out of three calls init, *next* and *last*. Associated to Π are the nonce space, $\mathcal{N} \subseteq \{0,1\}^*$, and the state space $\mathcal{S}$. The state is used to track the different stages of the instance. The first call of a Π scheme is the *init* call, which returns a state. From that moment onwards, we move from one state to another until *last* is called, which terminates the state. We have additionally the associated data (A) space $\mathcal{A} = \{0,1\}^*$, plaintext space $\mathcal{M} = \{0,1\}^*$ and ciphertext space $\mathcal{C} = \{0,1\}^*$. We can define the function signatures of the encryption $\mathcal{E}$ and decryption $\mathcal{D}$ as follows:

$$\begin{array}{ll}
\mathcal{E}.\text{init: } \mathcal{K} \times \mathcal{N} \to \mathcal{S} & \mathcal{D}.\text{init: } \mathcal{K} \times \mathcal{N} \to \mathcal{S} \\
\mathcal{E}.\text{next: } \mathcal{S} \times \mathcal{A} \times \mathcal{M} \to \mathcal{C} \times \mathcal{S} & \mathcal{D}.\text{next: } \mathcal{S} \times \mathcal{A} \times \mathcal{C} \to (\mathcal{M} \times \mathcal{S}) \cup \{\perp\} \\
\mathcal{E}.\text{last: } \mathcal{S} \times \mathcal{A} \times \mathcal{M} \to \mathcal{C} & \mathcal{D}.\text{last: } \mathcal{S} \times \mathcal{A} \times \mathcal{C} \to \mathcal{M} \cup \{\perp\}
\end{array}$$

Where an algorithm produces or takes a state $s \in \mathcal{S}$ from its state space, it is understood that a fixed encoding of S is employed. We assume that a single nonce is provided for the entire sequence of segments and that each plaintext is provided with its A.

For a given AE scheme $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$, encryption and decryption algorithms are defined as follows: $\boldsymbol{\mathcal{E}}, \boldsymbol{\mathcal{D}} : \mathcal{K} \times \mathcal{N} \times \{\{0,1\}^*\}^* \times \{\{0,1\}^*\}^* \to \times \{\{0,1\}^*\}^*$ and we require that the following holds: if $K \in \mathcal{K}$, $N \in \mathcal{N}$, $A \in \{\{0,1\}^*\}^*$, $M \in \{\{0,1\}^*\}^*$ and $C = \boldsymbol{\mathcal{E}}(K, N, A, M)$ then $M = \boldsymbol{\mathcal{D}}(K, N, A, C)$. Suppose we encrypt a plaintext with a nonce, a key, and an associated data. In that case, the decryption of the ciphertext with the same key, nonce, and associated data should return the original plaintext, as expected from an AE scheme. Notice that if a ciphertext is invalid, the function returns an error, $\perp$.

Finally, we define the ciphertext expansion, denoted by τ. We consider the case where $\tau > 0$ as if $\tau = 0$, the scheme only considers encryption without authentication. We call the value we get with the length of this expansion the redundancy in a cipher. We define segment/ciphertext expansion of Π as a number $\tau \geq 0$ such that if $K \in \mathcal{K}$, $N \in \mathcal{N}$, $A \in \{\{0,1\}^*\}^*$, $M \in \{\{0,1\}^*\}^*$, $m = |M| = |A|$ and $C = \mathcal{E}(K, N, A, M)$ then $|C[i]| = |M[i]| + \tau$ for all $i \in [1, \ldots, m]$ or in words the length of ciphertext is the length of the plaintext plus the amount of redundancy.

3.4.2 nOAE2

This section explains the evolution of the nOAE2 [HS20] from its first definition in [FFL12]. The first notion is the online authenticated encryption (OAE), introduced in [FFL12]. OAE was criticized in [HRRV15], where a new corrected definition was provided due to some error in the original definition.

However, even with the corrected definition, OAE suffers from some issues, according to the authors of [HRRV15], and they decided to provide their definition, which they call OAE2. There are three variants for the OAE2 notion: OAE2a, OAE2b, and OAE2c. The purpose of creating three different notions was to help clarify what OAE2 means. The OAE2 mentions the use of nonce, which are values that should be used only once. If a notion requires a value to be used only once, we say it is nonce respective, as it upholds the requirement of nonce not being repeated. However, if a value that should be a nonce is used multiple times, we call that nonce misuse, as the nonce is not used as expected. The OAE2 notion allows nonce misuses. The authors of [HRRV15] created a new notion called nonce-base online authenticated encryption (nOAE), where they define the security under the presence of a nonce, or in other words, an OAE2 variable, which is nonce respecting.

The nOAE claim assumes that the user will perform decryption in an in-order fashion, meaning we first decrypt segment m_i before decrypting segment m_{i+1}. However, some schemes might allow decrypting ciphertext segments in arbitrary order, we call this property random access decryption. In other words, segment m_2 might arrive before m_1, and the receiver would like to decrypt it before waiting for m_1, under the random access decryption, this should be possible.

nOAE also only considers security in the single-target scenario where an attacker only targets a single user to attack. However, often an attacker would not target a single user but rather a set of users to increase the probability of success. When discussing security in scenarios where an attacker targets multiple users simultaneously, we consider the multiple-target security notion.

To include those two aspects into the notion, the authors of [HS20] enhanced the nOAE notion to include multi-target security and allow random access decryption, resulting in the new nOAE2 notion.

For an AE scheme $\mathit{\Pi} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$, the nOAE2 advantage of an adversary $\mathcal{A}$ is defined as:

$$Adv_{\mathit{\Pi}}^{noae2}(\mathcal{A}) = 2\Pr[\mathbf{G}_{\Pi}^{\mathrm{noae2}}(\mathcal{A})] - 1,$$

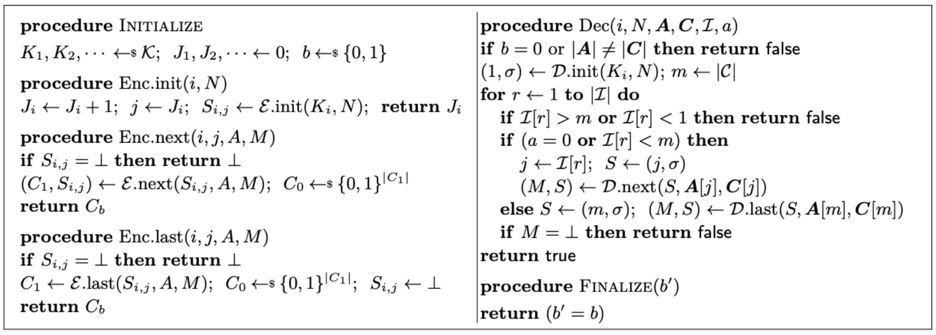
where game $\mathbf{G}_{\Pi}^{\mathrm{noae2}}$ is defined in Figure 3.4. Adversary $\mathcal{A}$ is restricted to be nonce respecting and is not allowed to create trivial forgeries:

- Calling $j \leftarrow \mathrm{Enc.init}(i, N)$, and $C_k \leftarrow \mathrm{Enc.next}(i, j, A_k, M_k)$ for $k = 1, \ldots, m$ and then querying $\mathrm{Dec}(i, N, \mathbf{A}, \mathbf{C}, \mathcal{I}, 0)$ such that $\mathbf{A}[j] = A_j$ and $\mathbf{C}[j] = C_j$ for every $j \in \mathcal{I}$.

- Calling $j \leftarrow \mathrm{Enc.init}(i, N)$, and $C_k \leftarrow \mathrm{Enc.next}(i, j, A_k, M_k)$ for $k = 1, \ldots, m$ and then querying $\mathrm{Dec}(i, N, \mathbf{A}, \mathbf{C}, \mathcal{I}, 1)$ such that $|\mathbf{C}| \in \mathcal{I}$ and $\mathbf{A}[j] = A_j$ and $\mathbf{C}[j] = C_j$ for every $j \in \mathcal{I}$.

- Calling $j \leftarrow \mathrm{Enc.init}(i, N)$, and $C_k \leftarrow \mathrm{Enc.next}(i, j, A_k, M_k)$ for $k =$

$1, \dots, m - 1$ and $C_m \leftarrow$ Enc.last(i, j, A_m, M_m) and then querying Dec$(i, N, \mathbf{A}, \mathbf{C}, \mathcal{I}, 1)$ such that $\mathbf{A}[j] = A_j$ and $\mathbf{C}[j] = C_j$ for every $j \in \mathcal{I}$.



```
procedure INITIALIZE
$K_1, K_2, \cdots \leftarrow_\$ \mathcal{K}$;  $J_1, J_2, \cdots \leftarrow 0$;  $b \leftarrow_\$ \{0,1\}$
procedure Enc.init$(i, N)$
$J_i \leftarrow J_i + 1$;  $j \leftarrow J_i$;  $S_{i,j} \leftarrow \mathcal{E}$.init$(K_i, N)$;  return $J_i$
procedure Enc.next$(i, j, A, M)$
if $S_{i,j} = \perp$ then return $\perp$
$(C_1, S_{i,j}) \leftarrow \mathcal{E}$.next$(S_{i,j}, A, M)$;  $C_0 \leftarrow_\$ \{0,1\}^{|C_1|}$
return $C_b$
procedure Enc.last$(i, j, A, M)$
if $S_{i,j} = \perp$ then return $\perp$
$C_1 \leftarrow \mathcal{E}$.last$(S_{i,j}, A, M)$;  $C_0 \leftarrow_\$ \{0,1\}^{|C_1|}$;  $S_{i,j} \leftarrow \perp$
return $C_b$

procedure Dec$(i, N, \boldsymbol{A}, \boldsymbol{C}, \mathcal{I}, a)$
if $b = 0$ or $|\boldsymbol{A}| \neq |\boldsymbol{C}|$ then return false
$(1, \sigma) \leftarrow \mathcal{D}$.init$(K_i, N)$; $m \leftarrow |\mathcal{C}|$
for $r \leftarrow 1$ to $|\mathcal{I}|$ do
  if $\mathcal{I}[r] > m$ or $\mathcal{I}[r] < 1$ then return false
  if $(a = 0$ or $\mathcal{I}[r] < m)$ then
    $j \leftarrow \mathcal{I}[r]$;  $S \leftarrow (j, \sigma)$
    $(M, S) \leftarrow \mathcal{D}$.next$(S, \boldsymbol{A}[j], \boldsymbol{C}[j])$
  else $S \leftarrow (m, \sigma)$;  $(M, S) \leftarrow \mathcal{D}$.last$(S, \boldsymbol{A}[m], \boldsymbol{C}[m])$
  if $M = \perp$ then return false
return true
procedure FINALIZE$(b')$
return $(b' = b)$
```

Figure 3.4: nOAE2 definition of the distinguishing game taken from [HS20] for an AE scheme $\mathit{\Pi} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ with expansion τ, nonce space $\mathcal{N}$, and associated data space $\mathcal{A}$.

The game represents the interface that an adversary is communicating with, as mentioned in 3.1. The game has six procedures, as shown in Figure 3.4. The first is the initialize procedure, which sets up the scene for the interface. As nOAE2 considers the settings in a multiple-target scenario, each user should have scheme-related attributes, which in this case are a key and a counter. So, in the initialize procedure, the interface assigns a key K_i and a counter J_i for each user, randomly selects whether the interface would forward the communication to the real or ideal world, and stores it in attribute b.

The interface allows an adversary to start multiple communication sessions by calling the Enc.init and Enc.last procedures. Each time Enc.init is called, a communication session is started, given a nonce N and user identifier $i \in \mathbb{N}$. More specifically, the Enc.init procedure increases the counter by one, generates the state S, which will be used for encryption, from the key and the nonce, and returns the new value of the counter.

There are two types of calls for encryption, either next or last. In both, the message is encrypted in the same matter, but in the case of the Enc.last, the communication session is terminated, and the state is set to error. If a call to either method is done after Enc.last, the procedure will return an error instead of a cryptogram. Both Enc.next and Enc.last generate two ciphertexts C_0 and C_1. C_0 is generated in the real world, using the current state to encrypt message M and associated data A. C_1 is created randomly and considered the output in the ideal world. Based on b, which we set in the *initialize* procedure, we either return C_0 or C_1. Finally, in the case of last, we set the current state to $\perp$ to indicate that the state has been terminated.

Although we only have one function in decryption, the three calls, init, next, and last, are integrated into that function. It first performs the $\mathcal{D}$.init call, as we would expect in a Dec.init call. As the behavior of next and last is very similar, combining them into one function is possible. This is done using an additional variable a, which is 0 or 1, depending on whether it is the last call. The function takes i, the user's index, N, the nonce, A, the additional information, or as we call it, associate data, C, the cipher text, $\mathcal{I}$ a list of indices of segments in C, and an a to specify if it is the last message. The main difference here is the set $\mathcal{I}$. This is a new variable compared to what we have seen so far. It specifies which messages should be decrypted. As the notion allows decryption in out-of-order or missing messages, the user should specify which messages they intend to decrypt. For example, we might wish to decrypt the set of messages m_i where $i \in \{0..5\} \cup \{8..10\}$. So we provide set $\mathcal{I} = \{0..5\} \cup \{8..10\}$ in the decryption call such that the function will know which index to use and decrypt. In the ideal world, we always return false for decryption, meaning the ciphertext is invalid. In the case of the real world, for each value in list $\mathcal{I}$, we extract the index of the message and retrieve the state used to encrypt it. Then we use the $\mathcal{D}$.next or $\mathcal{D}$.last depending on a to decrypt the ciphertext and return true if the ciphertext is valid.

The final procedure is *finalize*, where the adversary provides their guess of whether they are talking to the real or ideal world and gets back true if they are correct and false otherwise.

Chapter 4

Jammin Cipher

We have seen in section 3.4.2, the nOAE2 notion and their definition of a game to distinguish the real and ideal world. This is one example of a security notion for an AE scheme, but not the only one. This section explains the jammin cipher ideal world and compares it to the nOAE2 notion. The jammin cipher originates from the idea that the OAE2 definition is not operational, as it does not have a functioning unwrap. According to [BDH+22], jammin cipher has several interesting features and compares favorably to OAE2:

- It can serve as a security reference for both nonce-enforcing and nonce-misuse-resistant schemes. For OAE2, different ideal-world schemes are required such as nOAE or dOAE.
- It produces cryptograms whose distribution is intuitive and is as random as allowed while leaving the possibility for decryption. In contrast, the definition of Ideal2A/B uses a rather complex building block IdealOAE(τ), called uniformly sampled τ-expanding injective functions.
- It has ciphertext expansion as a parameter, which is required when dealing with schemes with variable ciphertext expansion due to the use of block encryption.
- It supports unwrap and wrap calls in any order, including bi-directional communication. Instead, an instance of Ideal2B can only encipher messages or decipher cryptograms but not both.

Following those arguments, we discuss the design of the jammin cipher in section 4.1 and in section 4.2 a more in depth explanation of the inner workings.

4.1 Jammin Cipher Design

The jammin cipher is specified in an object-oriented manner, making it easier to compare to written code or implement code out of the specification of a scheme. The world is split into object instances that the communicating parties can use. For two parties to communicate, they need to have a shared value, ID, with which their instances will be initialized. This can be seen as a key in the real world. Consider the case where Alice wishes to communicate with Bob, then they will both share a $ID_{\text{Alice and Bob}}$ while if Edward and Emma wish to communicate with one another, they will share $ID_{\text{Edward and Emma}}$. The instances support two functions: wrap and unwrap. A wrap call computes a cryptogram C out of a plaintext P, associated data A, which can both be arbitrarily long and the *history*. The unwrap function computes the plaintext from cryptogram C, associated data A, and the history. The jammin cipher is parameterized with the function WrapExpand. This function takes the length of the plaintext and returns the length of the cryptogram. Finally, the jammin cipher has a clone function that allows the user to make a copy of the state of the current instance and restart it freely.

Algorithm 1 The jammin cipher $\mathcal{J}^{\text{WrapExpand}(p)}$

```
1:  **Parameter:** WrapExpand, a $t$-expanding function
2:  **Global variables:** codebook initially set to $\bot$ for all, taboo initially set
    to *empty*

3:  **Instance constructor:** $init$(ID)
4:  **return** new instance inst with attribute inst.history = ID

5:  **Instance cloner:** inst.$clone()$
6:  **return** new instance inst$'$ with the history attribute copied from inst

7:  **Interface:** inst.$wrap(A, P)$ returns $C$
8:  context $\leftarrow$ inst.history; $A$
9:  **if** codebook(context; $P$) = $\bot$ **then**
10:     $\mathcal{C} = \mathbb{Z}_2^{\text{WrapExpand}(|P|)} \setminus (\text{codebook(context}; *) \cup \text{taboo(context)})$
11:     **if** $\mathcal{C} = \varnothing$ **then return** $\bot$
12:     codebook(context; $P$) $\xleftarrow{\$} \mathcal{C}$
13: inst.history $\leftarrow$ inst.history; $A$; $P$
14: **return** codebook(context; $P$)

15: **Interface:** inst.$unwrap(A, C)$ returns $P$ or $\bot$
16: context $\leftarrow$ inst.history; $A$
17: **if** $\exists! P$ : codebook(context; $P$) = $C$ **then**
18:     inst.history $\leftarrow$ inst.history; $A$; $P$
19:     **return** $P$
20: **else**
21:     taboo(context) $\leftarrow C$
22:     **return** $\bot$
```

The jammin cipher is stateful, where the sequence of messages exchanged so far are stored in a local attribute called *history.* It additionally stores a mapping of all associated data and plaintext with their matching cryptogram in a global archive called codebook. The codebook is initially empty, and for each call to wrap, the mapping between the context (consisting of the *history* and associated data) and plaintext to the cryptogram is added to the archive. At the beginning of each communication session, the history is initialized with the identifier ID, and for each wrap and unwrap call afterward, the associated data, A, and the plaintext, P, are appended. Each instance keeps a local version of the *history* and uses it to wrap and unwrap messages for the duration of the communication session. Thus, when creating a cryptogram using the *history*, not only the current message is authenticated, but the whole communicated session is. In other words, the session of the jammin cipher is an authenticated session.

4.2 Inner workings

The jammin cipher keeps a mapping of all the wrap queries and their cryptogram or error codes, $\text{codebook}(\text{history}; A; P) \to C/\bot$. The *history* and the associated data A form the context for the encryption of a plaintext P, where different contexts will result in different encryption for the same plaintexts. Apart from this mapping, the jammin cipher keeps a mapping of invalid cryptograms called taboo. When one tries to unwrap an invalid cryptogram, the taboo will store this cryptogram, $\text{taboo}(\text{context}) \to C$. Initially, both mappings are empty, and they get populated with calls to wrap and unwrap, denoted as $\text{codebook}(\text{context}; P) \overset{\$}{\leftarrow} C$ and $\text{taboo}(\text{context}) \leftarrow C$. The \$ in the expression before denotes the assignment of a random element chosen uniformly from C, and the expression $\text{codebook}(\text{context}, *)$ denotes the set of the values of $\text{codebook}(\text{context}, P)$ over all P.

Consider the case where Alice wishes to send a cryptogram to Bob. She first initializes her instance with her preshared identifier $ID_{\text{Alice and Bob}}$. Then, she calls the wrap function for a message P and associated data A. The wrap function creates a context variable consisting of *history*, being $ID_{\text{Alice and Bob}}$, and A. Then, it checks that the combination of the context and message is fresh, meaning that no such combination exists in the codebook. If so, it will generate a cryptogram in $\mathbb{Z}_2^{\text{WrapExpand}(|P|)}$ that is not already present in the codebook or in the taboo, which at this stage are both empty. Recall that WrapExpand returns a length based on the size of the plaintext and the expected size of the tag, and $\mathbb{Z}_2$ means bits. If the value returned in line 10 is the empty set, meaning that all the possibilities for a cryptogram have been exhausted, the function returns an error. Consider that a cryptogram has the size of $|P| + t$, where t is 32, then there are at least 4 billion (2^{32}) options for each message size before changing the key. The other option is if the taboo is full, resulting from some other party trying to brute force our key, so changing the key is also wise. Then, the function adds the new cryptogram to the codebook for the given combination of context and plaintext, adds the new associated data and plaintext to the *history*, and returns the cryptogram. Then Alice will send Bob A, C. Bob will make a call to unwrap with those values. The unwrap function will get the context from *history* and check that the codebook has a plaintext such that the context and plaintext return the received cryptogram, if so, it will update the *history* and return the plaintext. Otherwise, it will add the context and cryptogram to the taboo list and return an error.

The Jammin cipher has a few useful properties that can be used to prove the security of a real-world scheme. For simplicity, we do not provide their proofs but state them, for the full proof, refer to [BDH+22].

Proposition 1. *From the codebook, one always recovers at most one plaintext value (taken from Proposition 1 in [BDH+22]):*

$$\forall (context, C) |\{P : codebook(context; P) = C\}| \leq 1$$

Proposition 2. *If WrapExpand is t-expanding with* $t \geq 2$*, wrap is successful unless there were at least* 2^t *unsuccessful unwrap queries with the same context (taken from Proposition 2 in [BDH+22]).*

Chapter 5

STREAM and Google Tink

In this chapter, we examine the design of Google Tink Authenticated Encryption with Associated Data [Goo23a] and the construction that inspired Google developers' choices, STREAM [HRRV15]. We discuss why Google Tink AEAD's designers decided to deviate from the original STREAM construction, and we talk about the ideal scheme used to prove the security claim of Google Tink AEAD in [HS20].

5.1 STREAM Construction

The STREAM Construction takes the idea of nOAE and provides a more practical idea of how to implement it. We define the STREAM Construction as follows taken from [HRRV15]: fix an encoding function $< \cdot >$ that maps a tuple of strings $(N, i, d) \in \mathcal{N}' \times \mathcal{I} \times \{,\}$ to a string $< N, i, d >$. Here $\mathcal{I} = \mathbb{N}$ or else $\mathcal{I} = \{1, 2, \dots, \mathbf{max}\}$ for some $\mathbf{max} \in \mathbb{N}$ (the maximum number of segments in any message). Let $\mathit{\Pi} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be an nAE scheme with A space $\mathcal{A}$, and nonce space $\mathcal{N}$ that includes all possible values of $< N, i, d >$. Such an nOAE scheme is said to be compatible with the encoding function. We now describe the STREAM construction, defined and illustrated in Figure 5.1, to turn $\mathit{\Pi}$ and $< \cdot >$ into an nOAE-secure segmented-AE scheme $\mathbf{STREAM}[\mathit{\Pi}, < \cdot >]$ whose A space is $\mathcal{A}$ and whose nonce space is $\mathcal{N}'$.
As shown in Figure 5.1, there are three calls: init, next, and last. The next and the last can take any size of messages l, and together with the length of redundancy τ, we get the ciphertext length $l + \tau$. Each call uses a nonce, a counter, and an is-the-last-bit value, which is 0 for the next calls and 1 for the last call. We can also observe that different A sizes are allowed. STREAM is proven to be nOAE, for the full proof, refer to [HRRV15].

```
proc $\mathcal{E}$.init$(K, N)$                                   $\mathcal{E}$ algorithms
return $(K, N, 1)$

proc $\mathcal{E}$.next$(S, A, M)$
$(K, N, i) \leftarrow S$;  $S \leftarrow (K, N, i+1)$
$C \leftarrow \mathbf{E}_K(\langle N, i, 0\rangle, A, M)$
return $(C, S)$

proc $\mathcal{E}$.last$(S, A, M)$
$(K, N, i) \leftarrow S$;  return $\mathbf{E}_K(\langle N, i, 1\rangle, A, M)$
```

```
proc $\mathcal{D}$.init$(K, N)$                                   $\mathcal{D}$ algorithms
return $(K, N, 1)$

proc $\mathcal{D}$.next$(S, A, C)$
$(K, N, i) \leftarrow S$;  $M \leftarrow \mathbf{D}_K(\langle N, i, 0\rangle, A, C)$
if $M = \perp$ then return $(\perp, \perp)$
$S \leftarrow (K, N, i+1)$;  return $(M, S)$

proc $\mathcal{D}$.last$(S, A, M)$
$(K, N, i) \leftarrow S$;  return $\mathbf{D}_K(\langle N, i, 1\rangle, A, C)$
```

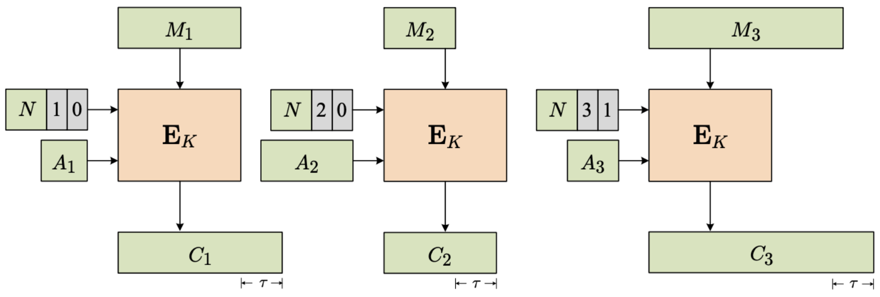


Figure 5.1: The STREAM construction taken from [HRRV15].

5.2 Google Tink Streaming AEAD Encryption

Google Tink is a library that implements different cryptography schemes for various tasks. “It was born out of our extensive experience working with Google’s product teams, fixing implementation weaknesses” [Goo23b]. The core idea is to have one library with different functions, which has a variety of cryptographic schemes that are easy to use, secure, and compatible with existing cryptography libraries [Goo23a].

A couple of the functions in the library implement the task of authenticated encryption. Google provides both Google Tink AEAD and Google Tink Streaming AEAD set of functions. The main difference is that Google Tink AEAD is a primitive where the user can pick what sub-scheme to use. Google Tink Streaming AEAD is more specialized for streaming, which they defined as the case where one needs to use an authenticated encryption scheme for a large file and send it over the Internet. An example can be sending a movie or parts of a movie between a server and a client, where the segments sent are rather large files. The two schemes are similar, with the main difference being the additional functionality of breaking large files into segments in Google Tink Streaming AEAD, and Google Tink AEAD allows more options for the scheme used for the key derivations. This thesis uses both Google Tink AEAD and Google Tink Streaming AEAD. For the security proof, we use the Google Tink Streaming AEAD version as used in [HS20]. For benchmarking, we use the Google Tink AEAD as we believe it is a fairer comparison to our Xoofff-Tink scheme as Xoofff-Tink is also more of a primitive that requires some additional implementing around it. With this being said, the primitives used for the benchmark are the same as the one mentioned in Google Tink Streaming AEAD, so when talking about Google Tink AEAD one can have Figure 5.2 in mind.

Google Tink attempted to use the previously discussed STREAM construction, where the E_k is instantiated with AES-GCM (GCM construction using AES) scheme, one of the most popular AE schemes [MV04]. While doing so, they encountered the following issue: a triple (N, i, a) needs to be encoded as a 12-byte GCM nonce, Tink uses four bytes to encode i and one byte to encode a. That means that the nonces of STREAM[GCM] will be only 7-byte long, and thus, the only viable option is to implement them as counters, as there is not enough entropy for a random nonce. However, there are situations when random nonces are desirable [HS20]:

- They are booted frequently in routers, meaning their counters will be reset often, resulting in many nonce repetition.
- Synchronizing counters among busy distributed servers might be impractical.

For this reason, Google developers decided to deviate from STREAM, such that Tink's streaming-encryption does not take nonces from the user. Instead, it picks a 7-byte random nonce prefix P, a 16-byte random salt S^*, and together with the messages header H it forms $R \leftarrow S^* \| H$ and derives a subkey $L \leftarrow KD(K, R)$, where KD is a key-derivation function that will be instantiated via HMAC-SHA256. It then runs STREAM[GCM] with key L and "nonce" P. Such short nonces will repeat, but under different subkeys, and thus will cause no harm to security [HS20]. An illustration of Google Tink Streaming encryption is given in Figure 5.2.
By making those decisions, Tink's streaming encryption suffers from the following issues according to the authors of [HS20]:

- As it does not follow the typical syntax of an online AE scheme, it is unclear what kind of security Tink's streaming-encryption provides.
- Relying on true randomness rather than the uniqueness of nonces is a step backward in robustness towards randomness failure, which is quite common [DGP07, HDWH12, LHA+12].

In order to prove Tink's streaming-encryption security to be nOAE2, the author of [HS20] distinguish it from their ideal world scheme they call SE1. They showed how one can view Tink's streaming-encryption as an online AE scheme SE1[KD, Π] where a nonce is a pair (R, P) and Π is a segmented AE scheme as defined in 3.4.1, and Tink chooses to pick nonces at random [HS20]. Following this convention, the paper confirms that SE1 is indeed secure, provided that Π is a good conventional AE scheme and KD is a good PRF. Also, the paper discusses that SE1 is not robust against randomness

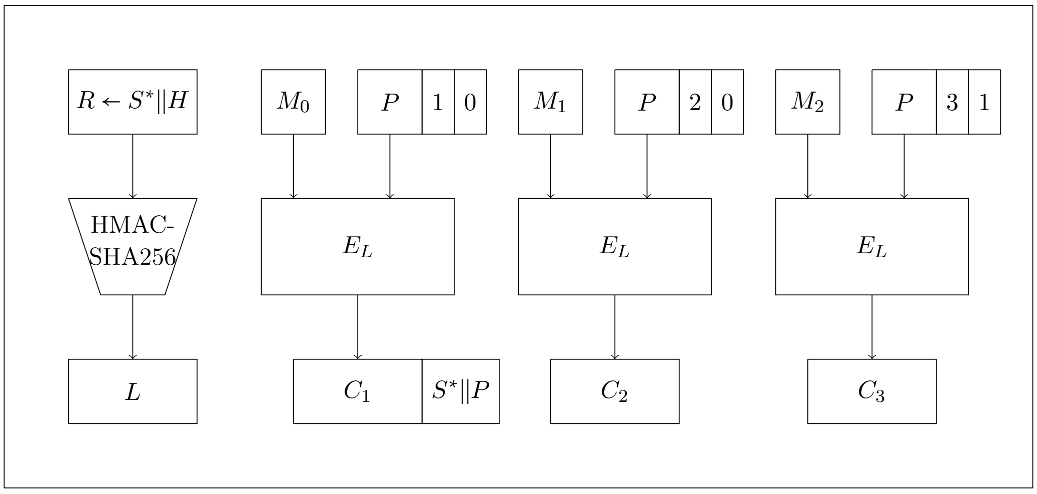
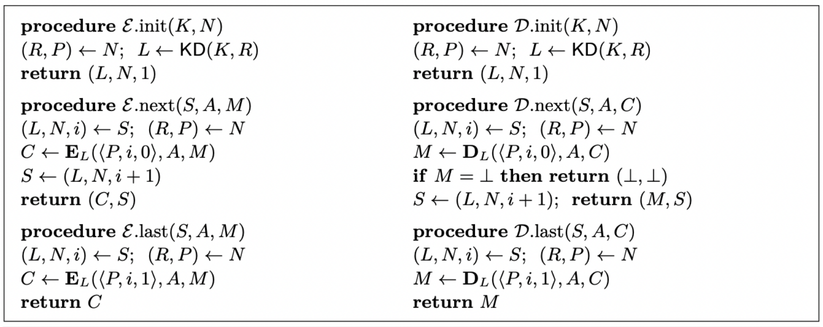


Figure 5.2: Tink procedure where P is a 7 byte nonce prefix, H is the header of the messages, S^* is a 16 bytes random salt, M_i is the i message in the sequence, and E is the encryption procedure of GCM using AES.

failure. In particular, its security would degrade if R is random, but P is a constant string. The procedure of SE1 can be seen in Figure 5.3.
As we can see, SE1 has associated data for each message, but Tink's streaming-encryption does not process such a possibility. The authors of [HS20] overcame this issue by taking the specific case of SE1 where A_i is always empty. This way, we can compare Tink's streaming encryption to the generalized canonical scheme SE1 that supports segmented A. Furthermore, we can view the pair (R, P) as a nonce, with $R = S^*||H$. This way, Tink's streaming-encryption is robust to randomness failure, guaranteeing that the same nonce will not repeat.

To summarize, Tink deviates from the syntax of AE in several ways:

- There is no associated data for each message. Instead, one provides a potentially empty header H at the beginning of the scheme. This header has the same role as the associated data but is only given at the beginning of the scheme.
- Nonces cannot be picked, but they result from applying HMAC-SHA256 to a randomly picked 16-byte salt S^* and a 7-byte nonce prefix P. Out of which, the scheme generates a subkey L from $R \leftarrow S^*||H$, and runs STREAM[GCM] (without segmented A) under the key L and nonce P.



```
procedure $\mathcal{E}$.init$(K, N)$
    $(R, P) \leftarrow N$;  $L \leftarrow \mathsf{KD}(K, R)$
    return $(L, N, 1)$

procedure $\mathcal{E}$.next$(S, A, M)$
    $(L, N, i) \leftarrow S$;  $(R, P) \leftarrow N$
    $C \leftarrow \mathbf{E}_L(\langle P, i, 0\rangle, A, M)$
    $S \leftarrow (L, N, i+1)$
    return $(C, S)$

procedure $\mathcal{E}$.last$(S, A, M)$
    $(L, N, i) \leftarrow S$;  $(R, P) \leftarrow N$
    $C \leftarrow \mathbf{E}_L(\langle P, i, 1\rangle, A, M)$
    return $C$
```

```
procedure $\mathcal{D}$.init$(K, N)$
    $(R, P) \leftarrow N$;  $L \leftarrow \mathsf{KD}(K, R)$
    return $(L, N, 1)$

procedure $\mathcal{D}$.next$(S, A, C)$
    $(L, N, i) \leftarrow S$;  $(R, P) \leftarrow N$
    $M \leftarrow \mathbf{D}_L(\langle P, i, 0\rangle, A, C)$
    if $M = \perp$ then return $(\perp, \perp)$
    $S \leftarrow (L, N, i+1)$;  return $(M, S)$

procedure $\mathcal{D}$.last$(S, A, C)$
    $(L, N, i) \leftarrow S$;  $(R, P) \leftarrow N$
    $M \leftarrow \mathbf{D}_L(\langle P, i, 1\rangle, A, C)$
    return $M$
```

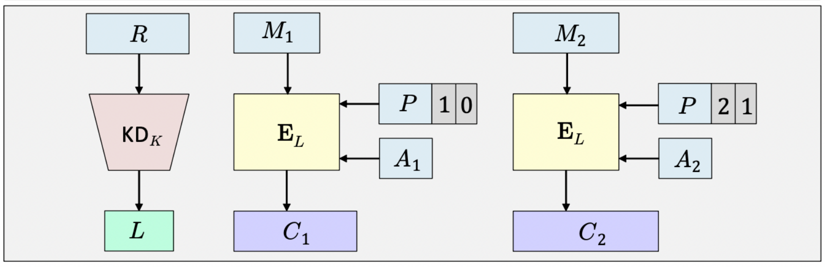


Fig. 8: **The canonical online AE scheme SE1**$[\mathsf{KD}, \Pi] = (\mathcal{K}, \mathcal{E}, \mathcal{D})$**, where** $\Pi = (\mathbf{K}, \mathbf{E}, \mathbf{D})$ **is a conventional AE scheme of the same key-generation algorithm, and** $\mathsf{KD} : \mathcal{K} \times \{0,1\}^* \rightarrow \mathcal{K}$ **is a key-derivation function.**

Figure 5.3: Definition of SE1 taken from [HS20].

Chapter 6

Deck-Plain and Xoofff Instantiation

In Chapter 4, we introduced the notion of the jammin cipher and in Chapter 2 the deck function. In this chapter, we provide a real-world mode that uses the deck function called Deck-Plain, and then we provide an instantiation for a deck function called XOOFFF [BDH+22].

6.1 Deck-Plain

Deck-Plain is a deck function mode for nonce-based authentication session supporting AE. It is a real-world design of an AE scheme introduced in [BDH+22] together with the jammin cipher discussed above. The main difference is in the initialization. While in the ideal world, we use an identifier ID, here, we use a key that can be derived from a shared secret or a master key. On the sending end, the scheme wraps a plaintext and associated data into a cryptogram consisting of a ciphertext of the original plaintext length and the length of the redundancy t. At the receiver's end, it unwraps the cryptogram to the original plaintext if the cryptogram is valid or an error otherwise. Both associated data and plaintext are optional. This allows us to have acknowledgment messages where the receiver wishes to inform the sender that they received the messages but not to send any messages in return, think about cases like TCP protocol where such a feature is required. If a key is used for multiple authentication sessions, the first message's associated data must be a nonce. One can use a counter for that goal and include it in the associated data. Deck-Plain was optimized for the case of an initial message with associated data and subsequent messages with only plaintext. It then requires only a single deck function call per fragment of the plaintext. The specification for the Deck-Plain mode can be seen in Algorithm 2, as taken from [BDH+22].

Algorithm 2 Definition of Deck-PLAIN(F, t, ℓ)

```
**Parameters:** deck function $F$, tag length $t \in \mathbb{N}$ and alignment unit length $\ell \in \mathbb{N}$
Let offset = $\ell \lceil \frac{t}{\ell} \rceil$: the smallest multiple of $\ell$ not smaller than $t$

**Instance constructor:** $init(\vec{K}, i)$ taking key array $\vec{K}$, key index $i$
$(\mathsf{inst}.K, \mathsf{inst}.\text{history}) \leftarrow (\vec{K}[i], \varnothing)$
**return** Deck-PLAIN instance
**Note:** in the sequel, $K$, history denote the attributes of inst

**Instance cloner:** inst.$clone()$
**return** new instance inst′ with all attributes ($K$, history) copied from inst

**Interface:** inst.$wrap(A, P)$ returns $C$
**if** $|P| = 0$ **then**
  history ← history; $A||00$
**else if** $|A| > 0$ or history = $\varnothing$ **then**
  context ← history; $A||10$
  $Z \leftarrow P + F_k\mathsf{context}$
  history ← context; $Z||1$
**else**
  context ← history
  $Z \leftarrow P + F_K(\mathsf{context}) \ll \text{offset}$
  history ← context; $Z||1$
$T \leftarrow 0^t + F_k\text{history}$
**return** $C = Z||T$

**Interface:** inst.$unwrap(A, C)$ returns $P$ or $\bot$
**if** $|C| < t$ **then return** $\bot$
Parse $C$ in $Z$ and $T$
**if** $|Z| = 0$ **then**
  history′ ← history; $A||00$
**else if** $|A| > 0$ or history = $\varnothing$ **then**
  history′ ← history; $A||10; Z||1$
**else**
  history′ ← history; $Z||1$
$T' \leftarrow 0^t + F_k\text{history}'$
**if** $T' \neq T$ **then return** $\bot$
**if** $|A| > 0$ or history = $\varnothing$ **then**
  context ← history; $A||10$
  $P \leftarrow Z + F_k\mathsf{context}$
**else**
  context ← history
  $P \leftarrow Z + F_K(\mathsf{context}) \ll \text{offset}$
history ← history′
**return** $P$
```

Inner workings

Similarly to the jammin cipher, Deck-Plain keeps track of messages in a variable called history. The history is then absorbed in the deck function state during a authentication session, if the message is missing the state will be out of sync and the unwrap will fail. In a wrap call, Deck-Plain encrypts a message by XORing it with a keystream generated from the deck function call on the history and associated data. The tag is generated by calling the deck function on the history, associated data, and ciphertext. This way, the scheme follows the encrypt-then-MAC approach. Allowing on the receiver end to verify the tag and only then start the decryption, ensuring we do not leak any partial decryption of the ciphertext if the tag is invalid. In the case of unwrapping, tag' is computed in the same manner as in wrapping and compared to the received tag. If valid, the ciphertext will be decrypted to the plaintext message. Otherwise, the unwrap will return an error message. In the case of a plaintext-only message (where we do not have associated data), the scheme reserves the first t bits of the deck function output for the tag and the remaining one as the keystream for the encryption. More correctly, it takes the smallest multiple of ℓ such that $\mathit{offset} = \ell\lceil \frac{t}{\ell} \rceil$ not shorter than t and shifts the result of the deck function to the right $offset$ times. For acknowledgement messages, Deck-Plain skips the en(de)cryption step and for plaintext-only messages, it skips the absorbing of the associated data unless the message is blank.

6.2 Xoofff - Farfalle over Xoodoo

So far, we have seen the abstract interface of the deck function where we call a function, F, and receive an output string that we use as a keystream or tag. This section presents an implementation for a deck function called Xoofff [DHAK18]. Before we can understand Xoofff, we need to understand its building blocks. More specifically, we need to understand Xoodoo, the primitive used in Xoofff and the Farfalle construction, which on top of the Xoodoo creates the Xoofff.

6.2.1 Xoodoo

Xoodoo is a family of permutations parameterized by the number of rounds n_r and denoted Xoodoo$[n_r]$. We give the specification as written in [DHAK18]. There are 12 round constants, which allow at most 12 rounds, but depending on the scheme used, a choice between 6 and 12 rounds is usually made. The design approach is similar in nature to Keccak-p, where we have a state and apply a round to it iteratively. The state consists of 3 equally sized horizontal planes, each with 4 parallel 32-bit lanes. Similarly, the state can be seen as a set of 128 columns of 3 bits, arranged in a 4×32

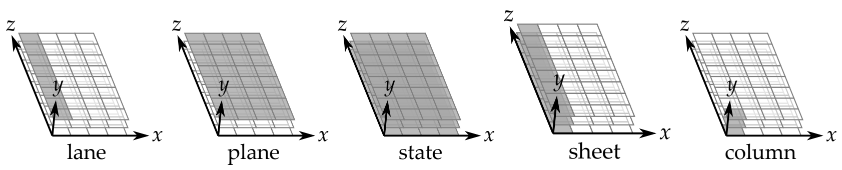


Figure 6.1: Toy version of the XOODOO state, with lanes reduced to 8 bits, and different parts of the state highlighted, taken from [DHAK18].

A_y	Plane y of state A
$A_y \lll (t, v)$	Cyclic shift of A_y moving bit in (x, z) to position $(x + t, z + v)$
$\overline{A_y}$	Bitwise complement of plane A_y
$A_y + A_{y'}$	Bitwise sum (XOR) of planes A_y and $A_{y'}$
$A_y \cdot A_{y'}$	Bitwise product (AND) of planes A_y and $A_{y'}$

Table 6.1: Notational conventions, taken form [DHAK18]

array. The planes are indexed by y, with plane $y = 0$ at the bottom and plane $y = 2$ at the top, an illustration can be seen in Figure 6.1. Within a lane, we index bits with z. The lanes within a plane are indexed by x, so the position of a lane in the state is determined by the two coordinates (x, y). The bits of the state are indexed by (x, y, z) and the columns by (x, z). Sheets are arrays of 3 lanes on top of each other, and they are indexed by x. The permutation consists of the iteration of a round function R_i that has 5 steps: a mixing layer θ, a plane shifting ρ_{west}, the addition of round constants ι, a non-linear layer χ and another plane shifting ρ_{east}. We specify XOODOO in Algorithm 3, completely in terms of operations on planes and use thereby the notational conventions we specify in Table 6.1. We illustrate the step mappings in a series of figures: the χ operation in Figure 6.2, the θ operation in Figure 6.3, the ρ_{east} and ρ_{west} operations in Figure 6.4. The round constants C_i are planes with a single non-zero lane at $x = 0$, denoted as c_i. We specify the value of this lane for indices -11 to 0 in Table 6.2. Finally, in many applications, the state must be specified as a 384-bit string s with the bits indexed by i. The mapping from the three-dimensional indexing (x, y, z) and i is given by $i = z + 32(x + 4y)$.

6.2.2 Farfalle

Farfalle is parameterized by four permutations denoted as p_b, p_c, p_d and p_e and two rolling functions denoted as $roll_c$ and $roll_e$. Rolling functions process data in a way that allows parts of its internal state or output to change incrementally as the input is updated. One can split Farfalle into three parts: the key mask derivation, the compression layer, and the expansion layer. The

i	c_i	i	c_i	i	c_i	i	c_i
−11	`0x00000058`	−8	`0x000000D0`	−5	`0x00000060`	−2	`0x000000F0`
−10	`0x00000038`	−7	`0x00000120`	−4	`0x0000002C`	−1	`0x000001A0`
−9	`0x000003C0`	−6	`0x00000014`	−3	`0x00000380`	0	`0x00000012`

Table 6.2: The round constants c_i with $-11 \leq i \leq 0$, in hexadecimal notation (the least significant bit is at $z = 0$), taken form [DHAK18].

Algorithm 3 Definition of Xoodoo[n_r] with n_r the number of rounds

```
**Parameters:** Number of rounds $n_r$
**for** Round index $i$ from $1 - n_r$ to 0 **do**
  $A = R_i(A)$

Here $R_i$ is specified by the following sequence of steps:
        $\theta :$
              $P \leftarrow A_0 + A_1 + A_2$
              $E \leftarrow P \lll (1,5) + P \lll (1,14)$
              $A_y \leftarrow A_y + E$ for $y \in \{0,1,2\}$
        $\rho_{\text{west}} :$
              $A_1 \leftarrow A_1 \lll (1,0)$
              $A_2 \leftarrow A_2 \lll (0,11)$
        $\iota :$
              $A_0 \leftarrow A_0 + C_i$
        $\chi :$
              $B_0 \leftarrow \overline{A_1} \cdot A_2$
              $B_1 \leftarrow \overline{A_2} \cdot A_0$
              $B_2 \leftarrow \overline{A_0} \cdot A_1$
              $A_y \leftarrow A_y + B_y$ for $y \in \{0,1,2\}$
        $\rho_{\text{east}} :$
              $A_1 \leftarrow A_1 \lll (0,1)$
              $A_2 \leftarrow A_2 \lll (2,8)$
```

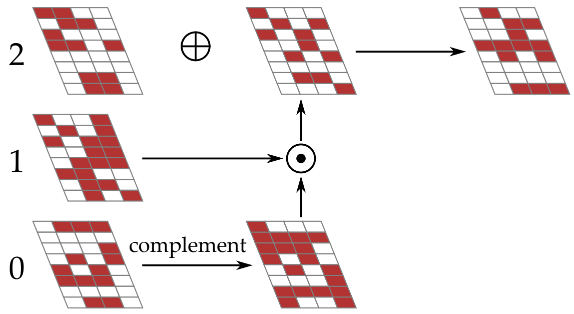


Figure 6.2: Effect of χ on one plane, taken from [DHAK18].

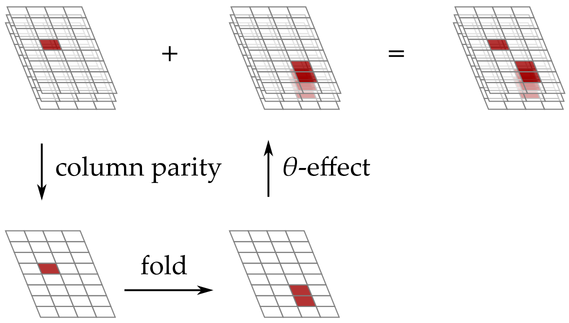


Figure 6.3: Effect of θ on a single-bit state, taken from [DHAK18].

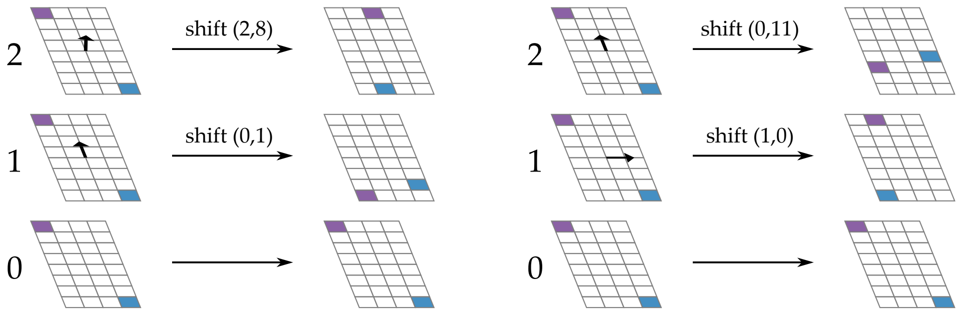


Figure 6.4: Illustration of ρ_{east} (left) and ρ_{west} (right), taken from [DHAK18].

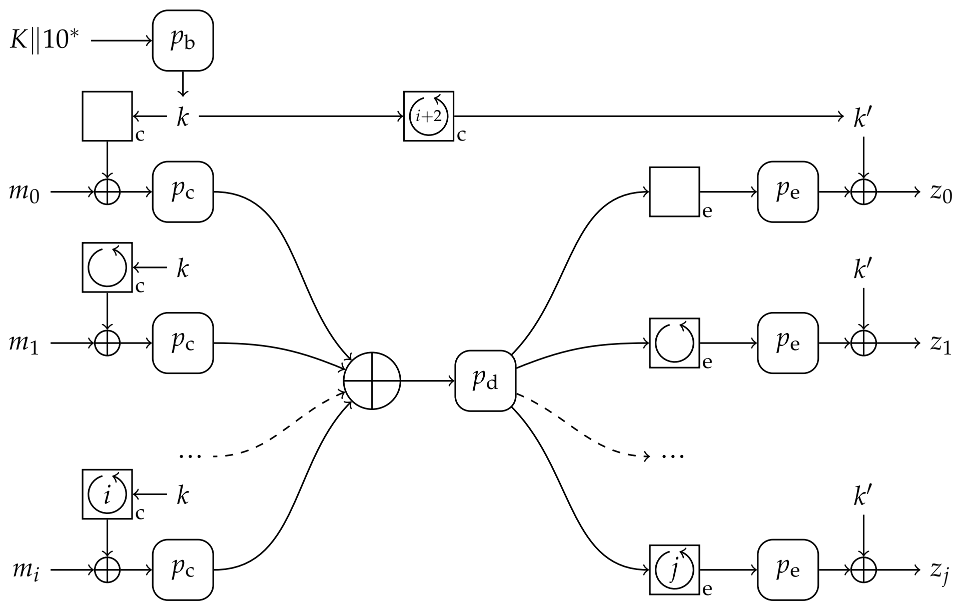


Figure 6.5: The Farfalle construction, taken from [BDH+16].

key derivation layer takes a key and expands it into b-bits mask denoted by k, and then k is rolled $i+2$ times using the $roll_c$ function to k', where i is the total number of blocks a message will have, $i = \lceil \frac{|M|}{b} \rceil$. Then, in the compression layer, a message is split into i b-bits blocks, and each message block is XORed with the result of $roll_c$ applied j times to the generated key bit mask k, where j is the block index. Then, each output is XORed into a final variable called the accumulator and denoted as x. In the expansion layer, we apply function p_d on the accumulator, and then for each output block, we apply the second rolling function $roll_e$ j times, and then we apply the p_e function and XOR with k'. An illustration of the construction is provided in Figure 6.5, as well as a specification in Algorithm 4.

6.2.3 Xoofff

XOOFFF is a deck function that we obtain by applying the Farfalle construction on XOODOO[6] and two rolling functions roll_{Xc} and roll_{Xce}. We define these two rolling functions using the notations from Table 6.1 as follows:

Algorithm 4 Definition of Farfalle$[p_\mathrm{b}, p_\mathrm{c}, p_\mathrm{d}, p_\mathrm{e}, \mathrm{roll}_\mathrm{c}, \mathrm{roll}_\mathrm{e}]$

```
**Parameters:** $b$-bit permutations $p_\mathrm{b}$, $p_\mathrm{c}$, $p_\mathrm{d}$ and$p_\mathrm{e}$ and rolling functions $\mathrm{roll}_\mathrm{c}$and $\mathrm{roll}_\mathrm{e}$.
**Input:**
   key $K \in \mathbb{Z}_2^*$, $|K| \le b - 1$
   input string sequence $M^{(m-1)} \circ \cdots \circ M^{(0)} \in (\mathbb{Z}_2^*)^+$
   requested length $n \in \mathbb{N}$ and offset $q \in \mathbb{N}$
**Output:** string $Z \in \mathbb{Z}_2^n$

$K' = \mathrm{pad}10^*(K)$
$k \leftarrow p_\mathrm{b}(K')$ {*mask derivation*}

$x \leftarrow 0^b$
$I \leftarrow 0$
**for** $j$ running from 0 to $m - 1$ **do**
   $M = \mathrm{pad}10^*(M^{(j)})$
   Split $M$ in $b$-bit blocks $m_I$ to $m_{I+\mu-1}$
   $x \leftarrow x + \sum_{i=I}^{I+\mu-1} p_\mathrm{c}(m_i + \mathrm{roll}_\mathrm{c}^i(k))$
   $I \leftarrow I + \mu + 1$ {*skip the blank index*}
$k' \leftarrow \mathrm{roll}_\mathrm{c}^I(k)$

$y \leftarrow p_\mathrm{d}(x)$
**while** all the requested $n$ bits are not yet produced **do**
   produce $b$-bit blocks as $z_j = p_\mathrm{e}(\mathrm{roll}_\mathrm{e}^j(y)) + k'$
$Z \leftarrow n$ successive bits from concatenation of $z_0||z_1||z_2\ldots$ starting from bit with index $q$.
**return** $Z = 0^n + F_K\left(M^{(m-1)} \circ \cdots \circ M^{(0)}\right) \ll q$
```

roll$_{\text{Xc}}$:

$$\begin{aligned}
&A_{0,0} \leftarrow A_{0,0} + (A_{0,0} \ll 13) + (A_{1,0} \ll 3)\\
&B \leftarrow A_0 \lll (3,0)\\
&A_0 \leftarrow A_1\\
&A_1 \leftarrow A_2\\
&A_2 \leftarrow B
\end{aligned}$$

roll$_{\text{Xe}}$:

$$\begin{aligned}
&A_{0,0} \leftarrow A_{1,0} \cdot A_{2,0} + (A_{0,0} \ll 5) + (A_{1,0} \ll 13) + 0x00000007\\
&B \leftarrow A_0 \lll (3,0)\\
&A_0 \leftarrow A_1\\
&A_1 \leftarrow A_2\\
&A_2 \leftarrow B
\end{aligned}$$

An illustration of Xoofff is given in Figure 6.6, as we can see, we have the Farfalle construction we have seen above with $p_b = p_c = p_e = p_d =$ Xoodoo[6], $roll_c =$ roll$_{\text{Xc}}$ and $roll_e =$ roll$_{\text{Xe}}$.

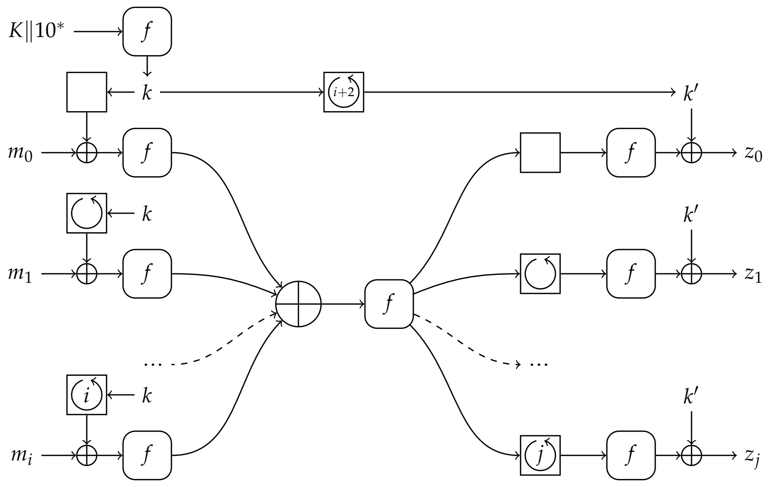


Figure 6.6: Illustration of Xoofff: applying the Farfalle construction on Xoodoo[6]

Chapter 7

Deck-Tink

In this chapter, we explain our proposed mode called Deck-Tink. Similar to the jammin cipher chapter, we start by giving the general design of the mode in section 7.1 and a more in depth explanation of the inner workings in section 7.2.

7.1 Deck-Tink Design

We define in Algorithm 5 a deck function mode for nonce-based AE called Deck-Tink. Deck-Tink and Deck-Plain share a similar base, but their primary goals differ. The most noticeable difference is the presence of history. While in Deck-Plain, we carry a history to authenticate the sequence of all messages up to that point, in Deck-Tink, we do not have that. In other words, Deck-Plain supports authentication sessions while Deck-Tink does not. The main goal of Deck-Tink was to allow online communication sessions, meaning every message can be verified based on the key, the nonce, the plaintext/ciphertext, and a counter. Hence, the scheme will allow additional features that Deck-Plain does not. The most noticeable are out-of-order messages and message dropping during communication. If we keep a history, we bind ourselves to enforce that all messages are properly communicated and that no message is lost. If a message is lost, the decrypter's history will be out of sync, and they will not be able to decrypt any further messages. Furthermore, if a message arrives out of sync, message M_2 arrives before message M_1, again, we get an error as the decrypter history will differ from the sender's. There are some ways of propagating it by adding a buffer, but our scheme aims to solve the issue by design. Another difference in the scheme is the use of one associated data at the initialization of the scheme and not throughout the wrap and unwrap. We follow Google Tink's approach: we only use a header, and after that, each wrap and unwrap uses a specific nonce counter. Similarly to Google Tink, the first associated data/header should have a nonce in it, ensuring the session's uniqueness.

Algorithm 5 Definition of Deck-Tink(F, t)

```
**Parameters:** deck function $F$, expansion length $t$

**Constructor:** $init(K, H)$ taking key $K$ and absorbing header $H$
**return** instance $Q.F_k(H)$ and finish $\leftarrow$ False

**Instance cloner:** inst.$clone()$
**return** new instance inst' with all attributes ($K$, $H$) copied from inst

**Bit String get current squeeze:** inst.getCurrentValue()
**return** the squeeze value of the inst without absorbing any input

**Interface:** $Q.TinkWrap(P$, counter, $a)$ returning $C$
**if** finish **then**
  **return** ERROR: finished instance
$Q' \leftarrow Q.clone$
**if** $|P| > 0$ **then**
  $X \leftarrow P + Q'.F_k(\text{counter}||0)$
  $T \leftarrow 0^t + Q'.F_k(X||1)$
  $C \leftarrow X||T$
**else**
  $T \leftarrow 0^t + Q'.F_k(\text{counter}||1)$
  $C \leftarrow T$
**if** $a > 0$ **then**
  finish $\leftarrow$ True
**return** $C$, counter

**Interface:** $Q.TinkUnwrap(C$, counter, $a)$ returning $P$ or $\perp$
**if** finish **then**
  **return** ERROR: finished instance
**if** $|C| < t$ **then**
  **return** $\perp$
$Q' \leftarrow Q.clone$
**if** $|C| > t$ **then**
  Parse $C$ in $X$ and $T$
  $Q' \leftarrow Q'.F_k(\text{counter}||0)$
  $Q'' \leftarrow Q'.clone$
  $T' \leftarrow 0^t + Q'.F_k(X||1)$
**else**
  $T' \leftarrow 0^t + Q'.F_k(\text{counter}||1)$
**if** $T' \neq T$ **then**
  **return** $\perp$
**if** $a > 0$ **then**
  finish $\leftarrow$ True
**if** $|C| > t$ **then**
  $P \leftarrow X + Q''$.getCurrentValue()
  **return** $P$
**return** $\varnothing$
```

Notice that header H, in this case, is both associated data and nonce, it must be unique per init call for the same key.

7.2 Inner workings

Deck-Tink offers the same interface as Deck-Plain, consisting of init, wrap, and unwrap functions. It has one length parameter, t being the length of the redundancy determining the security level. It supports different key, header, and message sizes and empty messages for authenticated acknowledgments. If a key is used more than once, the header provided in the initialization phase must be a nonce, e.g., a session incremental value.

The initialization call sets the key for our deck function F to K, absorbs the header H, and sets our finish variable to False, which indicates the end of the usability of this session.

Similarly to the Deck-PLAIN, in a wrap call, Deck-Tink encrypts a plaintext by adding a keystream that is the output of the underlying deck function with the context input. The difference is that, in this case, the context is the header and a counter. Then we follow the same encrypt then authenticate structure of Deck-PLAIN.

1. **Encryption**: It absorbs the counter, extracts the keystream from the deck function, and adds it to the plaintext, yielding the ciphertext.

2. **Tag generation**: It appends ciphertext to the original associated data and extracts the tag from the deck function.

The unwrapping follows the same idea as the wrap: we first verify the tag and then decrypt it. For authentication-only messages, Deck-Tink skips the en(de)cryption step and the absorbing of ciphertext, instead, it absorbs the counter and only yields a tag. Deck-Tink appends a frame bit to ciphertext strings for domain separation before absorbing them to create separation between encryption and authentication. In particular, ciphertext strings end with 1 and tag strings with 0.

Chapter 8

Security Analysis of Deck-Tink

In this chapter, we analyze the security of Deck-Tink. Our proof uses the H-coefficient technique from [Pat08] and its adaptation from [CS13]. We first explain the idea of the H-coefficient technique and then provide proof of a bound for Deck-Tink.

8.1 H-coefficient Technique

An adversary $\mathcal{A}$ has access to two AE schemes, one being in the ideal world, denoted as $\mathcal{O}$, and the other in the real world, denoted by $\mathcal{P}$. In our case, $\mathcal{P}$ is Deck-Tink instantiated with an ideal deck function: a random oracle, and $\mathcal{O}$ is the jammin cipher. The adversary creates a transcript, denoted by τ, by interacting with the model. In other words, τ is a list of queries and their responses from the model. The probability of getting a particular response differs between the worlds as they differ. We will denote it as $\mathcal{D}_{\mathcal{O}}$ for the probability distribution of transcripts that can be obtained in the ideal world and $\mathcal{D}_{\mathcal{P}}$ for the probability distribution of transcripts that can be obtained in the real world. Ideally, we would wish to have the statistical distance between the two probability distributions $(\frac{\mathcal{D}_{\mathcal{O}}}{\mathcal{D}_{\mathcal{P}}})$ as close as possible to 1, meaning the two worlds are hardly distinguishable. We calculate this distance over finite domain D as: $\Delta(X,Y) = \frac{1}{2}\sum_{\alpha \in D} |\Pr[X = \alpha] - \Pr[Y = \alpha]|$. However, some transcripts are often better for the adversary than others, in the sense that for a transcript τ, the ratio between the two probability distributions might be smaller. This will allow the adversary to have a better guess about which world they are talking to. We call such transcripts bad transcripts and denote them by $\mathcal{T}_{bad}$. One can divide the set of all transcripts $\mathcal{T}$ into $\mathcal{T} = \mathcal{T}_{bad} \cup \mathcal{T}_{good}$, where $\mathcal{T}_{bad}$ represent the cases in which the ratio is small and $\mathcal{T}_{good}$ when the ratio is close to 1. Below, we are giving the H-coefficient Technique lemma we will use in our proof, the proof for the lemma can be found in [CS13].

Lemma 1 (**H-coefficient Technique**). *Consider a fixed information-theoretic deterministic adversary $\mathcal{A}$ whose goal is to distinguish $\mathcal{O}$ from $\mathcal{P}$. Let ε be such that for all $\tau \in \mathcal{T}_{good} : Pr(D_{\mathcal{O}} = \tau)/Pr(D_{\mathcal{P}} = \tau) \geq 1 - \varepsilon$. Then, $\Delta_{\mathcal{A}}(\mathcal{O};\mathcal{P}) \leq \varepsilon + Pr(D_{\mathcal{P}} \in \mathcal{T}_{bad})$.*

In our proof below, we use the special case where for all $\tau \in \mathcal{T}_{good}$ $\Pr(\mathcal{D}_{\mathcal{O}}) \geq \Pr(\mathcal{D}_{\mathcal{P}})$ so we have $\Delta_{\mathcal{A}}(\mathcal{O};\mathcal{P}) \leq \Pr(D_{\mathcal{P}} \in \mathcal{T}_{bad})$. As expected, we set $\mathcal{O}$ to be the jammin cipher and $\mathcal{P}$ to be Deck-Tink.

8.2 Proof of a Bound for Deck-Tink

The security of Deck-Tink relies on the header to be a nonce. Otherwise, we will leak the difference between two plaintexts. If we assume the encryption context to be a nonce, the only way to distinguish Deck-Tink from the jammin cipher is by a forgery or by distinguishing the deck function from a random function, as captured in the theorem below.

Theorem 2. *Let $\mathcal{D}$ be any fixed deterministic adversary whose goal is to distinguish Deck-Tink(F,t,l) from $\mathcal{J}^{+t}$, the jammin cipher with WrapExpand$(p) = p + t$ in the multi-target settings where there are μ users. Let $N \in \mathbb{N}$ and $N > 0$ represent the amount of queries an adversary performs, and let $k \in \mathbb{N}$ be the length of a key. If in the queries of $\mathcal{D}$ the header is a nonce, there exists an adversary D' using the same resources as $\mathcal{D}$ such that*

$$\Delta_{\mathcal{D}}(\textit{Deck-Tink}(F,t,\ell);\mathcal{J}^{+t}) \leq \frac{q_{unwrap}}{2^t} + Adv_F^{prf}(\mathcal{D}') + \frac{\mu(\mu-1)}{2^{k+1}} + \frac{N\mu}{2^k},$$

$Adv_F^{prf} = |\mathbb{P}[K \xleftarrow{\$} \mathcal{K}^u : \mathcal{D}^{F_{K_1},\ldots,F_{K_u}} = 1] - \mathbb{P}[\mathcal{D}^{\mathcal{RO}_1,\ldots,\mathcal{RO}_u} = 1]|$ where $\mathcal{RO}$ is a random oracle that takes as input a string sequence.

The last two terms in Theorem 2 are related to the key collisions and guesses. When we have μ users with a uniformly randomly picked key, the probability of a collision is the number of users choose 2 divided by the length of the key. If we enforce that the header we use in the init part is unique per user communication session, we can drop the $\frac{\mu(\mu-1)}{2\times 2^k}$ part together as the combination of the key and the header is unique. Therefore, the attributes passed to the init call would never collide, meaning multiple communication sessions will never share the same state after the init call. The uniqueness of the header can come from including additional attributes in the header. These attributes can be anything that would be unique per communication session, such as a combination of a username and a counter, which increases for each init call.

The last term, $\frac{N\mu}{2^k}$, comes from the possibility of guessing a key. Given that there are μ users, the possibility of guessing a key is the computation power of an adversary, N times μ divided by the key length. If one guesses ten keys, the probability of one of those choices being correct is ten times more likely for ten users than one user, therefore, the μ is in the formula. If we follow the idea from above and use a unique header for each communication session, the probability drops to $\frac{N}{2^k}$, as each communication session will have a different state after init, so an attacker would only be able to try and guess a key for one communication session. One can reasonably assume that an adversary has a computational power limited to $N \ll 2^{128}$ [LBD23]. Therefore, a key length greater than 128 bits would make the whole term negligible.

If the user of the scheme follows those two enhancements, the security of the scheme becomes the security of distinguishing Deck-Tink in the single-user scenario:

Theorem 3. *Let $\mathcal{D}$ be any fixed deterministic adversary whose goal is to distinguish Deck-Tink(F, t, l) from $\mathcal{J}^{+t}$, the jammin cipher with WrapExpand$(p) = p + t$. If in the queries of $\mathcal{D}$ the header is a nonce, there exists an adversary D' using the same resources as $\mathcal{D}$ such that*

$$\Delta_{\mathcal{D}}(\textit{Deck-Tink}(F, t, \ell); \mathcal{J}^{+t}) \leq \tfrac{q_{unwrap}}{2^t} + Adv_F^{prf}(\mathcal{D}'),$$

with q_{unwrap} the number of unwrap calls $\mathcal{D}$ makes and Adv_F^{prf} defined as: $Adv_F^{prf} = |\mathbb{P}[K \xleftarrow{\$} \mathcal{K} : \mathcal{D}^{F_K} = 1] - \mathbb{P}[\mathcal{D}^{\mathcal{RO}} = 1]|$ where $\mathcal{RO}$ is a random oracle that takes as input a string sequence.

Proof. We use a hybrid argument and replace the deck function with a random oracle before comparing Deck-Tink with the jammin cipher. This follows from the security claim of the deck function, which should behave like a pseudorandom function (PRF) [BDH+22]:

$$\Delta_{\mathcal{D}}(\text{Deck-Tink}(F, t, \ell); \mathcal{J}^{+t}) \leq \Delta_{\mathcal{D}''}(\text{Deck-Tink}(\mathcal{RO}, t, \ell); \mathcal{J}^{+t}) + Adv_F^{prf}(\mathcal{D}'),$$

where $\mathcal{D}''$ has the same resources as $\mathcal{D}$ and Deck-Tink$(\mathcal{RO}, t, \ell)$ means that our F_K function returns an output of a random Oracle.

We use the Lemma 1, where we take $\mathcal{O} = \mathcal{J}^{+t} \overset{\Delta}{=} \mathcal{J}$ and $\mathcal{P} =$ Deck-Tink$(\mathcal{RO}, t, \ell)$. In this proof, we use the syntax of the jammin cipher, and we can consider Deck-Tinkan instance of jamming with only the key and header as history.

As we have different interfaces for the two schemes, we build a mode on top of the jammin cipher, which makes calls to its inner functions (init, clone, wrap, and unwrap). Following the definition of Deck-Tink, we know that init calls get a key K and a header H, TinkWrap gets plaintext P, counter, and is last attribute a, and TinkUnwrap gets ciphertext C, counter, and is last attribute a. Our mode on top of jammin performs the following operations for the three calls, also visible in Algorithm 6:

- Init: Calls the init function on key K, init(K), which returns Q, and then wraps on header H and empty associated data "", Q.wrap("", H). Set local boolean value finish to 0.
- Wrap: If finish is 0: calls clone on the instance, Q.clone() which returns Q', then calls wrap on counter, and plaintext P, Q'.wrap(counter, P), to get ciphertext C, if a is 1, set local boolean value finish to 1 and returns C to the advisory. Otherwise, if finish is 1: returns an error.
- Unwrap: If finish is 0: calls clone on the instance, Q.clone() which returns Q', then calls unwrap on counter, and ciphertext C, Q'.unwrap(counter, C), to get plaintext P or error $\perp$, if the plaintext is not an error and if a is 1, set local boolean value finish to 1, and returns plaintext to the advisory. Otherwise, if finish is 1: returns an error.

By following this interface, we do not lose any security compared to the jammin cipher because the combination of the header and the counter is unique, which results in a random output from the jammin cipher for each communication session. As the header is a nonce, different communication sessions will never result in the same state for neither our Deck-Tink nor jammin cipher. Then, in each communication session, we have the counter, which is also a nonce per communication session. So, for each call to the jammin cipher clone, the attributes will never be the same, resulting in a different state for the jammin cipher and random ciphertext.

We define a transcript τ as a sequence of records of the form:

$$(\mathbf{wrap/unwrap}, (\mathrm{H}, \mathrm{counter}), P, C),$$

where the first value is the type of call made, and the second is the combination of the header H and counter. The P and the C are either a parameter or a return value depending on the operation, such that if the operation is **wrap**, then P is a parameter and C is the returned value where $C \neq \perp$. Or if the operation is **unwrap**, then C is a parameter, and P is the return value, which might contain an error code $\perp$.

We ignore the transcript records with

- the transcript **wrap** records with tuple $(\mathrm{counter}, P)$ equal **unwrap** records with tuple $(\mathrm{counter}, C)$.
- the transcript **unwrap** records which has the same $(\mathrm{counter}, P, C)$ as **wrap** records.
- the transcript where the instance is finished, there was a call (un)wrap with a equal one.

- Out-of-order messages, as we only consider header H and the given counter for each wrap operation, there is no need of previous communications to unwrap a ciphertext.

Algorithm 6 Definition of interface around jammin cipher

```
Parameters: jammin cipher $\mathcal{J}$

Constructor: $init(K, H)$ taking key $K$ and absorbing header $H$
$Q \leftarrow \mathcal{J}$.init($K$)
$Q$.wrap("", $H$)
return $Q$ and finish $\leftarrow$ False

Instance cloner: inst.$clone()$
return new instance inst' with all attributes (finish) copied from inst

Interface: $Q$.Wrap($P$, counter, $a$) returning $C$
if finish then
    return ERROR: finished instance
$Q' \leftarrow Q.clone$
$C \leftarrow Q'$.wrap(counter, $P$)
if $a > 0$ then
    finish $\leftarrow$ True
return $C$, counter

Interface: $Q$.unwrap($C$, counter, $a$) returning $P$ or $\perp$
if finish then
    return ERROR: finished instance
$Q' \leftarrow Q.clone$
$P \leftarrow Q'$.unwrap(counter, $C$)
if $P \neq \perp$ then
    if $a > 0$ then
        finish $\leftarrow$ True
return $P$
```

We can ignore those cases w.l.o.g. as both worlds act deterministically and would behave consistently in this respect. This gives us a simple definition of forgery, namely the presence of a successful unwrap record in the transcript.

This means that for our **H-coefficient** technique, we have only one type of bad event, namely a successful forgery. To put it formally (**unwrap**, P, C where $P \neq \perp$). As we are using a $\mathcal{RO}$, it means that the tag for a given plaintext is generated at random. A forgery attempt means that for a given input, the adversary picks the correct random generated tag. As $\mathcal{RO}$ generate a t-bit string tag uniformly at random, the probability that they are equal is

2^{-t}, hence $Pr(\mathcal{D}_{\mathcal{P}} \in \mathcal{T}_{\text{bad}}) \leq \frac{q_{\mathbf{unwrap}}}{2^t}$ after $q_{\mathbf{unwrap}}$ calls to **unwrap**.

Next, we need to prove that for all $\tau \in \mathcal{T}_{\text{good}}, Pr(D_{\mathcal{J}} = \tau) \geq pr(D_{\mathcal{P}} = \tau)$ hence $\varepsilon = 0$ in Lemma 1. We know that in both worlds, the cryptogram is generated randomly and independently for different contexts. This means we can use the independent event probability law, e.g. simply split the transaction records per context and multiply their probability. We consider a subset of the transcript for a given context value.

In the case of Deck-Tink, the scheme behaves a bit differently in cases of empty and nonempty messages. Therefore, we need to consider both options. To put it formally, a wrap call can have one of the two forms $(\text{wrap}, \text{counter}, P \neq \epsilon, C)$ or $(\text{wrap}, \text{counter}, \epsilon, C_\epsilon \neq C)$, this will come handy later in the proof. We can consider the record independently as each **wrap** call has a unique counter, which provides us with the desired unique counter.

Next, we compare the possible transactions we have in the two worlds. Upon an unsuccessful unwrap query, the jammin cipher returns $\perp$ as it avoids forgeries, and hence we get a 1 for the probability of $Pr(D_{\mathcal{J}})$. Upon a wrap query, the jammin cipher selects C from a set of cardinality at most $2^{|P|+t}$ (the length of the ciphertext) and hence contributes a factor of at least $2^{-(|P|+t)}$ to $Pr(D_{\mathcal{J}} = \tau)$. It may return an error, but thanks to Proposition 2, this would require $q_{\text{unwrap}} \geq 2^t$.

Upon an unsuccessful unwrap query, $P = \text{Deck-Tink}(\mathcal{RO}, t, \ell)$ returns $\perp$ in a good transcript, and same as above, we have a contribution of at most 1 to $Pr(D_{\mathcal{P}} = \tau)$. Upon a **wrap** query, $\mathcal{P}$ computes the value $C = X||T$ with $X = P + \mathcal{RO}(H; \text{counter})$, and $T = \mathcal{RO}(H; \text{counter}; X)$. Thanks to the fact that upon **wrap** the counter is unique and $\mathcal{P}$ takes tags and keystream in different domains or from different parts of the $\mathcal{RO}$ output stream, it contributes a factor of exactly $2^{-(|P|+t)}$ to $Pr(D_{\mathcal{P}} = \tau)$. In the second case where we have an empty plaintext, we have a **wrap** record with $P = \epsilon$ contributing a factor 2^{-t} to $Pr(D_{\mathcal{P}} = \tau)$.

This means that for $Pr(D_{\mathcal{J}} = \tau)$ we have that $Pr(D_{\mathcal{J}} = \tau) \leq 1 * 2^{-(|P|+t)}$ and $pr(D_{\mathcal{P}} = \tau) \leq 1 * 2^{-(|P|+t)} * 2^{-t}$. This shows that $Pr(D_{\mathcal{J}} = \tau) \geq pr(D_{\mathcal{P}} = \tau)$ and concludes the proof. □

Chapter 9

Xoofff-Tink Implementation

So far, we have discussed our deck function mode construction Deck-Tink for authenticated encryption. Our primitive for the mode is the deck function F. We decided to use the XOOFFF as our deck function, which gives us our Deck-Tink instantiation Xoofff-Tink. As XOOFFF uses the Farfalle construction with XOODOO as its primitive, our underlying primitive is also XOODOO. An illustration of this idea, together with the primitives and their construction, is given in Figure 9.1.

We have decided to implement Xoofff-Tink in two programming languages: C and Rust [Alt24]. Below, we explain the implementation choices.

9.1 C Implementation

We followed the XKCP implementation style to ensure we could simply add Xoofff-Tink to the library. XKCP stands for the eXtended Keccak Code Package or the XOODOO and Keccak Code Package. It is a repository maintained by the Keccak team, and it attempts to gather different free and open-source implementations of the cryptographic schemes defined by the Keccak team [AKC23]. The XKCP style includes three types of libraries: one

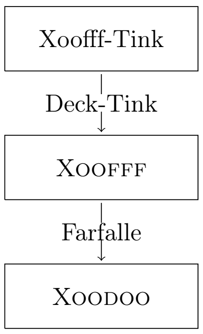


Figure 9.1: Xoofff-Tink dependencies lower level schemes.

for use, another for testing, and a third to test performance (benchmark). By following the XKCP style, we ensure we can use different optimized versions of the underlying primitive XOODOO, increasing performance on different architectures.

When extracting a library from XKCP, one needs to provide three aspects to XKCP. The high-level service, in our case Xoofff-Tink, the lower-level schemes, XOOFFF and XOODOO, and the architecture the library is used on. More about this can be found in the HOWTO-customize.build in [AKC23]. We also need to specify the purpose of our library. If left empty, we will get the library itself, if we give BM, we will get the speed performance (benchmarking), and if we give UT, we will get the unit tests. An example of a choice can be <target name="MyBenchmarks" inherits="Xoofff Xoodoo-SSE2 Xoodoox4-SSSE3 Xoodoox8-AVX2 BM"/>, which will provide us the benchmarking code of XOOFFF with AVX2 optimization. By following this style, we get all primitives' optimization benefits for our scheme, leaving us to implement the mode. This way, we abstract from one specific implementation to a set of implementations that works for different architectures.

We used the XOOFFF instances present in XKCP as inspiration for our code together with the implementation of XOOFFF. The XOOFFF implementation provides three function calls: Xoofff_MaskDerivation, Xoofff_Compress, and Xoofff_Expand. The first function, Xoofff_MaskDerivation, initializes a XOOFFF instance with a given key. The function Xoofff_Compress is the absorb operation used to handle input data. The function Xoofff_Expand is the squeeze operation used to expand output from the absorbed input. Similarly, we have three functions that the user can call: Deck_Tink_Initialize, Deck_Tink_Wrap, and Deck_Tink_Unwrap, as can also be seen in the pseudocode in Algorithm 5. We decided to use the counter as a 64-bit value as we believe 2^{64} messages should be a reasonable amount for a single header. After that, one can change the header and keep using the same key. The scheme user manages the counter, and it is up to them to determine the behavior of a counter reaches the value $2^{64} - 1$. Both Deck_Tink_Wrap, and Deck_Tink_Unwrap make use of Xoofff_Compress and Xoofff_Expand. In the pseudocode (Algorithm 5), whenever we use $Q.F_k(X)$, we make a call to Xoofff_Compress. If we have + before the $Q.F_k(X)$, we call both Xoofff_Compress and Xoofff_Expand, XORing the output of Xoofff_Expand with the value on the other side of the +. The function getCurrentValue also makes a call to Xoofff_Expand to squeeze the input we absorbed before. Finally, we created the addToContext function to handle the addition of the domain separation to the input. The rest of the code aligns with the pseudocode in Algorithm 5. If the wrap or unwrap functions are called on a finished instance, the code will return a value of 1, which means an error, and the input values will remain unchanged.

9.2 Rust Implementation

In this section, we discuss the Rust implementation of Xoofff-Tink. We first motivate our reasoning for choosing Rust as our second programming language for our scheme. Then, we provide more explanations on the implementation in Rust.

9.2.1 Why Rust?

Rust is a relatively new programming language developed by Mozilla. Its main goal is to provide a safe programming environment while allowing control over lower-level resources that languages like C and C++ provide [ESDH21]. It is getting increasingly popular in the cryptography field as it provides the benefits of C and C++ while shielding programmers from command pitfalls and vulnerabilities that C and C++ possess. Some of its most noticeable advantages are:

- Rust has a memory-safe design that prevents common vulnerabilities like buffer overflows and use-after-free. It achieves this by using an ownership model and borrow checkers that enforce strict rules on how memory is accessed and modified at compile time [MKW18].
- Rust's syntax supports functional and concurrent programming paradigms, making programming more accessible and robust for different development styles.
- Rust has a strong and static type system that helps to catch errors at compile time and avoid runtime crashes. Rust also supports generics and traits, which enable code reuse and abstraction [MKW18].
- Rust has a formal and verified subset called RustBelt that provides strong guarantees about the soundness and safety of Rust programs. RustBelt can be used to verify the correctness and security of cryptographic implementations in Rust, adding another layer of safety to the code [JJKD17].
- Rust has a growing ecosystem of libraries that provides developers access to various cryptographic APIs, making it easier to use for safer cryptographic implementation.

As we see it, Rust allows us to obtain some of the benefits of low-level languages like C, while adding additional safety. Therefore, we decided also to implement Xoofff-Tink in Rust. Below, we give some rationale for the decisions made while implementing Xoofff-Tink and its primitives.

9.2.2 Xoofff-Tink Implementation in Rust

Unlike in the C implementation, there is no standard implementation for Xoofff-Tink's primitives, Xoodoo and Xoofff. While working on this thesis, the first Rust implementation for Xoofff [Roy23] came to be, and we decided to use it as a base. We also came across a rust implementation for Xoodyak [Den23], which provided us with a more optimized version of Xoodoo. In this thesis, with the help of Leon Botros from iHub, we combined the two versions mentioned above, as well as added AVX optimization in order to implement our version of Xoofff-Tink. Next, we explain the AVX extension, the main method we used for optimization. Then, we explain the design made in each part of our implementation for two primitives Xoodoo and Xoofff and for the mode we used, Xoofff-Tink.

AVX

There are different approaches to optimizing the performance of a code from a parallelist point of view. Intel Advanced Vector Extensions (AVX) is an extension that implements the Single instruction multiple data (SIMD) vectorization optimization approach. This approach increases the size of a register from the typical 32/64 bits such that instead of containing a single value, it can contain a vector of multiple values. Consider the case where one wishes to add 16 different 32-bit integers. If we do it in the traditional way, it will take us 8 additions. But if we first place 8 of the integers on one 256-bits register and the other 8 on another register, perform the addition, and then split them back to 8 32-bits values, we only need to perform the addition once. This is the core idea of the SIMD vectorization optimization approach and its implementation AVX. Intel introduced three types of AVX extensions. AVX, which increased the register size to 256 bits and introduced a limited Intel operations set [Lom21]. Afterward, Intel introduced AVX-2, which has the same register size of 256 bits but additional operations, allowing us to perform more sophisticated operations on the 256-bits registers [Int21]. Lastly, AVX-512 extended the register size to 512 bits, increased the number of registers from 16 to 32, and added additional instruction sets for more complex operations [Cue21].

Xoodoo Implementation

For Xoodoo, we took the generic code from [Den23] and added the Rust abstraction for AVX. In Rust, it is possible to use AVX optimizations with little effort in implementation. We can design the function such that multiple instances of it can run simultaneously. In this case, the round function is designed to run in parallel multiple states.

Xoofff Implementation

It is slightly more complicated for Xoofff as we defined three possible SIMD vectorization optimization variants: SSE, AVX-2, and AVX-512. For each of them, we implemented a different version that does either 4, 8, or 16 operations of rounds at once. We used the version of Xoofff from [Roy23] and adopted the function to support arrays of 4, 8, or 16 values simultaneously.

We continue optimizing the code by removing the finalize call from the Xoofff implementation. In [Roy23], there are three functions: absorb, finalize, and squeeze. We first absorb all the input, then finalize (the absorption phase), and finally squeeze. The finalize is used in cases where we want to absorb multiple times. Instead of absorbing each time we call the function, we only absorb when the state is full, or we want to squeeze. However, this is not applicable in our case as we always call finalize after absorption. The separation of those two functions included some additional operations, which we removed by combining the two functions. We also included the XOR in the squeeze function, which allowed us to save the allocation of another array in order to XOR the output value with the input value. Together with some small optimizations, such as reusing the same buffer for input and output (similar to what is done in the C code) and reorganizing some functions, we got our final optimized version.

Xoofff-Tink Implementation

The Rust implementation of Xoofff-Tink comes in two flavors, supporting the "bytes in bytes out" approach and the inplace detached mode. In the first option, the functions are called wrap or unwrap. The user passes the plaintext message, and the counter and receives the wrap output in a single variable in the order of ciphertext, counter, and tag, and vice versa for the unwrap. In the second case, the user passes the plaintext, a counter, and an empty variable for the tag and receives the cipher in place of the plaintext and the tag and counter in their matching passed variables. The rest of the Rust implementation of Xoofff-Tink is similar to the C and the pseudocode in Algorithm 5. However, due to the nature of Rust, there are a few differences between the C and Rust implementations. The last attribute (a in the pseudocode) is incorporated into the borrow checkers design of Rust, meaning we do not need to pass the value. Instead, we need to decide if we call the function (un)wrap or (un)wrap_last. The reason for this difference is the ownership over attributes present in Rust. If we do not explicitly specify that we want to get the attribute back after calling a function on it, the attribute's scope ends, and it can no longer be used. We specify whether we want the attribute back by the way we pass the attribute to the function. If we pass it by reference, we will get it back. Otherwise, if we call it by value, the variable will be consumed by the function, and it can no longer be used after the function call. So if we want our instance to be consumed, like in the case of having last, a, equal one, we call it by value and not by reference. Therefore, we have two functions: one that allows call by reference (un)wrap and returns the ownership of the Xoofff-Tink instance and the other (un)wrap_last, called by value and consumes the Xoofff-Tink instance. This also means that the error we will get is different. While in C, we will get one in the return value, in Rust, the compiler will give an error that we try to operate on a variable that is not in scope anymore.

Additionally, the code can also return an error if the tag given is invalid for the given ciphertext. The unwrap would then return WrongTag error.

There are a few more differences between the C and the Rust implementations, which we discuss in the next section 9.3. The rest of the implementation aligns with the C code and the pseudocode from Algorithm 5.

9.3 C and Rust Compression

The main difference between the two implementations is the basic length unit. While the C implementation operates on bits, the Rust implementation operates on bytes. This means that a message of 129 bits in C will result in a 129 ciphertext, while in Rust, it will result in a 136-bit ciphertext. More specifically, it will expect the message itself to be 136 bits as we use a vector of bytes. This comes from the implementation dependencies of our scheme, as we are using existing implementations of XOOFFF in the two languages. We do not believe that this limits our Rust implementation, as today, most online communication is based on bytes rather than bits, and the needed padding is small (at most 7 bits) compared to the big plaintext that would be wrapped. It is important to notice that the domain separation in Rust is one bit, the same as in the C implementation. In this case, we use a mask (which is already implemented in the XOOFFF Rust implementation [Roy23]), which allows us to use one bit for domain separation.

Another difference is that in Rust, we introduce two function call options: one similar to the C, where plaintext, tag, and counter are separated, and another that supports the bytes in bytes out approach. We did that to ease the integration of our scheme with other implementations, as today's bytes in bytes out approach is popular in Rust. It is important to notice that the speed of the bytes in bytes out implementation is lower than the other implementation. This comes from the overhead of extending an existing vector in Rust. As Rust is a memory-safe programming language, extending a vector is expensive due to the memory allocations happening behind the scenes.

To ensure no difference between the two implementations, we tested a large input set that covers all key sizes between 0 bits and 376 bits and nonce sizes between 0 bits and 768 bits, as well as 1584 different sizes of the messages. We called that test KATS, which can be found in the Rust implementation tests. This can also be used as a test reference for future implementation in different languages, as it consists of a list of different keys, nonces, messages, and their expected ciphertexts and tags.

Chapter 10

Xoofff-Tink Performance

This section looks at the speed performance of our Xoofff-Tink implementation. The section is split into four main parts: the benchmark setup, the results of the speed performance of our two implementations for Xoofff-Tink, and finally, some general observations from our benchmarks. We start by comparing our C implementation with Google Tink AEAD using AES-GCM128 as its primitive. Due to the lack of a C implementation for the Google Tink AEAD, at the time of writing this thesis, we use the language closest to C and C++. For the Rust variant, we compare our implementation to some more well-known AE schemes. For that end, we use the Rust Crypto AEADs library [dev23], and we compare our implementation to AES-GCM, AES-GCM-SIV, Ascon, and Chacha20poly1305.

One observation that we can anticipate before diving into the results is that some of the schemes we are comparing against are based on or use AES, which has a hardware implementation in most machines our days, including the one we are using for our benchmarks. This makes the comparison slightly imbalanced, as hardware implementations can achieve a significant speed improvement compared to software implementations [DFY+17]. Also, our implementation has a more optimized version, using AVX-512, but unfortunately, for this thesis, we only had access to a machine with an AVX-2.

10.1 Benchmark Setup

Before we start with reporting on the results, we should mention that all the executions of the different implementations are done in a benchmark environment, where performance boosts are turned off. The processor used for those measures was Tiger Lake processor, Intel Core^{TM} i7-1165G7, which has one physical socket with four cores and 8 CPUs. The code for the benchmarking is in the bench folder [Alt24] and is suitable for any x86 architecture with cycles reporting and for non x86 architectures with (milli/nano)seconds reporting for Rust. For the C code, various benchmark options are available using the XKCP suite, we used the AVX-2 optimized variant.

10.2 Xoofff-Tink vs Google-Tink AEAD

In this section, we analyze the results of benchmarking our C implementation against the results of the C++ implementation of Google-Tink AEAD using AES-GCM128 as its primitive. In Table 10.1, we can see the results of running the wrap and unwrap operation of Xoofff-Tink against the results of running encrypt and decrypt of Google Tink AEAD for plaintexts from size 256 bytes to 64 MiB, where we double the input size in every entry. We have run each operation on the given input size 10,000 times and are reporting the median of our observations.

Input size	Xoofff-Tink Wrap in cycles (cycles/byte)	Xoofff-Tink Unwrap in cycles (cycles/byte)	Google-Tink Encrypt in cycles (cycles/byte)	Google-Tink Decrypt in cycles (cycles/byte)
256 bytes	2550 (9.96)	2564 (10.02)	42580 (166.33)	2558 (9.99)
512 bytes	2939 (5.74)	2976 (5.81)	48549 (94.82)	2303 (4.50)
1 KiB	4635 (4.53)	4663 (4.55)	42106 (41.12)	2638 (2.58)
2 KiB	6819 (3.33)	6865 (3.35)	42364 (20.69)	3533 (1.73)
4 KiB	12326 (3.01)	12357 (3.02)	52511 (12.82)	5238 (1.28)
8 KiB	22149 (2.70)	22203 (2.71)	68853 (8.40)	8468 (1.03)
16 KiB	43100 (2.63)	43120 (2.63)	80947 (4.94)	14549 (0.89)
32 KiB	84395 (2.58)	84580 (2.58)	135038 (4.12)	26915 (0.82)
64 KiB	169533 (2.59)	169544 (2.59)	232108 (3.54)	51792 (0.79)
128 KiB	335631 (2.56)	335680 (2.56)	477596 (3.64)	103146 (0.79)
256 KiB	669306 (2.55)	669345 (2.55)	914823 (3.49)	205815 (0.79)
512 KiB	1337451 (2.55)	1337053 (2.55)	1700908 (3.24)	401139 (0.77)
1 MiB	2727950 (2.60)	2738662 (2.61)	3382167 (3.23)	786847 (0.75)
2 MiB	5520075 (2.63)	5516427 (2.63)	7740249 (3.69)	1587885 (0.76)
4 MiB	11149811 (2.66)	11141568 (2.66)	13471424 (3.21)	7275647 (1.73)
8 MiB	22988220 (2.74)	23204611 (2.77)	27052583 (3.22)	14488353 (1.73)
16 MiB	47021783 (2.80)	47367080 (2.82)	52290045 (3.12)	30238242 (1.80)
32 MiB	95814527 (2.86)	96318254 (2.87)	107438522 (3.20)	60433754 (1.80)
64 MiB	193736922 (2.89)	194124970 (2.89)	216850144 (3.23)	121038927 (1.80)

Table 10.1: Benchmarks of Xoofff-Tink and Google-Tink for different sizes.

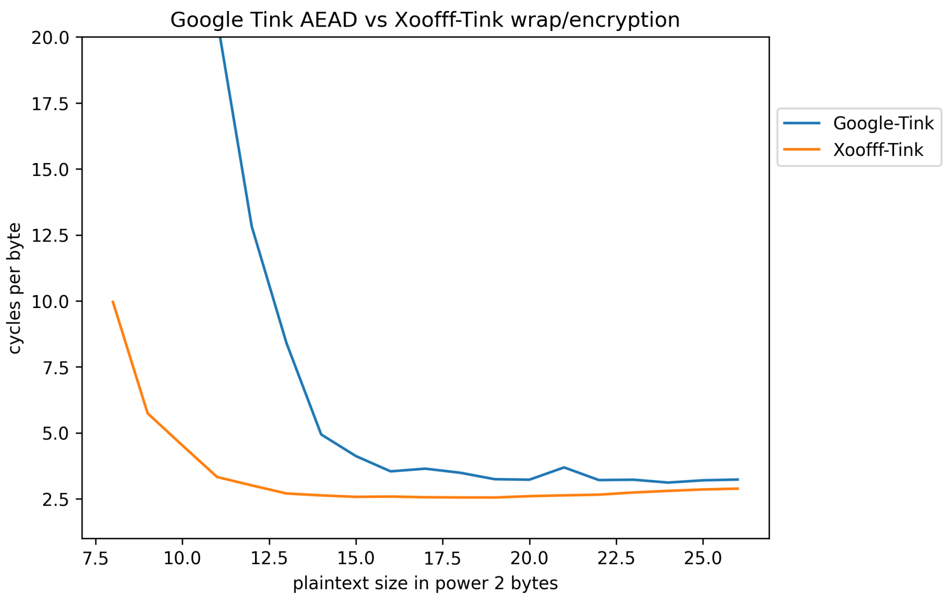


Figure 10.1: Benchmarks of wrap Xoofff-Tink and encrypt Google-Tink for different sizes.

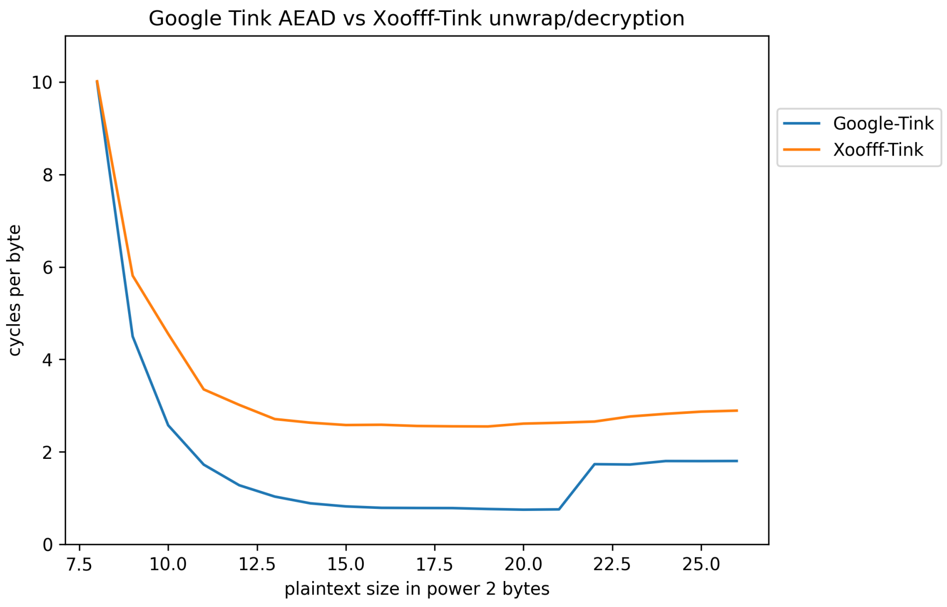


Figure 10.2: Benchmarks of unwrap Xoofff-Tink and decrypt Google-Tink for different sizes.

Discussion

As we see above, our Xoofff-Tink implementation is faster in wrap than Google Tink AEAD encryption and slower in unwrap compared to Google Tink AEAD decryption. For wrap/encryption, we can see that our implementation is at least twice as fast for short plaintext (less than 128 Kib) and between 1.10 and 1.42 faster for longer plaintexts. For unwrap/decryption, we can see that our implementation is around 2 to 3 times slower for medium-size plaintext (more than 1 Kib and less than 4 MiB) and between 1.003 and 1.76 slower for other plaintext sizes.

As we mentioned above, schemes based on or using AES offer some speed performance improvements, and Google Tink AEAD uses AES-GCM, which benefits from the hardware implementation of AES. This makes the comparison slightly imbalanced, as hardware implementation can have a significant speed performance compared to software implementation [DFY+17], and if we had used the AVX-512 variant of our implementation, we could have seen better performance for our scheme. We argue that on machines that do not have AES implemented in hardware, like an IoT device, our scheme can achieve even better performance compared to schemes that use AES, such as Google Tink AEAD.

There are benefits for faster wrap/encryption compared to faster unwrap/decryption. In cases where we have limited resources on a server and can tolerate slower decryption on the receiver side, such speed improvements can improve the server's efficiency. As mentioned above, using lower-level devices, such as IoTs or older devices that might not have AES implemented in hardware, can result in faster implementation when using Xoofff-Tink over Google Tink AEAD.

10.3 Rust Xoofff-Tink vs Rust AEADs Crypto Library

In this section, we analyze the results of our Rust implementation's benchmarking against different schemes in the Rust Crypto AEADs library, namely AES-GCM, AES-GCM-SIV, Ascon, and Chacha20poly1305. In Tables 10.2-10.11, we can see the results of running the wrap and unwrap operation on Xoofff-Tink against the results of running encrypt and decrypt on the other schemes for plaintexts from size 1 KiB to 64 MiB, where we double the input size in every entry. We have used the benchmarking suite of cargo [doc24a] and the microbenchmarking library Criterion [doc24b]. Most of the code for the benchmarking was taken from [dev23] and modified to fit our requirements. In most cases, we only changed the sample size and plaintext sizes. However, for Ascon, we also needed to change the benchmark to use cycles rather than milliseconds/nanoseconds and add a benchmark for decryption.

For our implementation, we report both the results from using the sequential implementation and the results of using AVX2. We run each operation on a given sample size of 10,000, and report the median of our observations and the best estimate provided by the cargo benchmarking suite.

scheme and input size	Wrap in cycles (cycles/byte) as median and best estimate	Unwrap in cycles (cycles/byte) as median and best estimate
Xoofff-Tink 1 KiB	6262 (6.12) 6290 (6.14)	6437 (6.29) 6441 (6.29)
Xoofff-Tink 2 KiB	11447 (5.59) 11476 (5.60)	11863 (5.79) 11894 (5.81)
Xoofff-Tink 4 KiB	21974 (5.36) 21991 (5.37)	22517 (5.50) 22570 (5.51)
Xoofff-Tink 8 KiB	42791 (5.22) 42819 (5.23)	43642 (5.33) 43679 (5.33)
Xoofff-Tink 16 KiB	84658 (5.17) 84712 (5.17)	86296 (5.27) 86354 (5.27)
Xoofff-Tink 32 KiB	168047 (5.13) 168363 (5.14)	172381 (5.26) 172528 (5.27)
Xoofff-Tink 64 KiB	335122 (5.11) 335397 (5.12)	343251 (5.24) 343426 (5.24)
Xoofff-Tink 1 MiB	5374509 (5.13) 5378527 (5.13)	5541107 (5.28) 5544088 (5.29)
Xoofff-Tink 2 MiB	10818048 (5.16) 10819791 (5.16)	11205001 (5.34) 11211408 (5.35)
Xoofff-Tink 4 MiB	21654805 (5.16) 21656638 (5.16)	22514016 (5.37) 22522385 (5.37)
Xoofff-Tink 8 MiB	43408621 (5.17) 43411589 (5.18)	45564554 (5.43) 45580918 (5.43)
Xoofff-Tink 16 MiB	87294028 (5.20) 87302514 (5.20)	91882527 (5.48) 91920874 (5.48)
Xoofff-Tink 32 MiB	175414802 (5.23) 175426993 (5.23)	222001903 (6.62) 222046106 (6.62)
Xoofff-Tink 64 MiB	351078398 (5.23) 351123691 (5.23)	449177066 (6.69) 449192659 (6.69)

Table 10.2: Benchmarks of Rust Xoofff-Tink for different sizes.

scheme and input size	Wrap in cycles (cycles/byte) as median and best estimate	Unwrap in cycles (cycles/byte) as median and best estimate
Xoofff-Tink 1 KiB	4384 (4.28) 4389 (4.29)	4837 (4.72) 4857 (4.74)
Xoofff-Tink 2 KiB	7202 (3.52) 7208 (3.52)	7829 (3.82) 7858 (3.84)
Xoofff-Tink 4 KiB	12794 (3.12) 12809 (3.13)	13814 (3.37) 13852 (3.38)
Xoofff-Tink 8 KiB	23673 (2.89) 23696 (2.89)	25092 (3.06) 25127 (3.07)
Xoofff-Tink 16 KiB	45657 (2.79) 45694 (2.79)	48121 (2.94) 48159 (2.94)
Xoofff-Tink 32 KiB	90416 (2.76) 90524 (2.76)	95502 (2.91) 95841 (2.92)
Xoofff-Tink 64 KiB	181187 (2.76) 181374 (2.77)	189498 (2.89) 189624 (2.89)
Xoofff-Tink 1 MiB	2885718 (2.75) 2893197 (2.76)	3103018 (2.96) 3104623 (2.96)
Xoofff-Tink 2 MiB	5810618 (2.77) 5817611 (2.77)	6364880 (3.04) 6366203 (3.04)
Xoofff-Tink 4 MiB	11684519 (2.79) 11684871 (2.79)	12970993 (3.09) 12972205 (3.09)
Xoofff-Tink 8 MiB	23786073 (2.84) 23795521 (2.84)	25937248 (3.09) 25943416 (3.09)
Xoofff-Tink 16 MiB	48274082 (2.88) 48263583 (2.88)	51868000 (3.09) 51915887 (3.09)
Xoofff-Tink 32 MiB	96646901 (2.88) 96639464 (2.88)	144753123 (4.31) 144770252 (4.31)
Xoofff-Tink 64 MiB	193432819 (2.88) 193443963 (2.88)	292498155 (4.36) 292505479 (4.36)

Table 10.3: Benchmarks of Rust Xoofff-Tink using AVX2 for different sizes.

scheme and input size	Encrypt in cycles (cycles/byte) as median and best estimate	Decrypt in cycles (cycles/byte) as median and best estimate
Ascon-128 (inplace) 1 KiB	15881 (15.51) 15967 (15.59)	15323 (14.96) 15367 (15.01)
Ascon-128 (inplace) 2 KiB	30490 (14.89) 30581 (14.93)	29973 (14.64) 30023 (14.66)
Ascon-128 (inplace) 4 KiB	59815 (14.60) 60251 (14.71)	59261 (14.47) 59334 (14.49)
Ascon-128 (inplace) 8 KiB	118570 (14.47) 118741 (14.49)	117859 (14.39) 118061 (14.41)
Ascon-128 (inplace) 16 KiB	235933 (14.40) 236180 (14.42)	235046 (14.35) 235496 (14.37)
Ascon-128 (inplace) 32 KiB	470354 (14.35) 473413 (14.45)	469385 (14.32) 469877 (14.34)
Ascon-128 (inplace) 64 KiB	939562 (14.34) 941393 (14.36)	938221 (14.32) 939407 (14.33)
Ascon-128 (inplace) 1 MiB	15032714 (14.34) 15052794 (14.36)	15011233 (14.32) 15026187 (14.33)
Ascon-128 (inplace) 2 MiB	30064883 (14.34) 30108121 (14.36)	30015994 (14.31) 30054250 (14.33)
Ascon-128 (inplace) 4 MiB	60167015 (14.34) 60216645 (14.36)	60023967 (14.31) 60085625 (14.33)
Ascon-128 (inplace) 8 MiB	120365084 (14.35) 120408166 (14.35)	120166502 (14.32) 120263010 (14.34)
Ascon-128 (inplace) 16 MiB	265054157 (15.80) 265046095 (15.80)	240721633 (14.35) 240800024 (14.35)
Ascon-128 (inplace) 32 MiB	482512975 (14.38) 482605262 (14.38)	481207857 (14.34) 481333012 (14.34)
Ascon-128 (inplace) 64 MiB	964908471 (14.38) 965169583 (14.38)	962282389 (14.34) 962631420 (14.34)

Table 10.4: Benchmarks of Ascon-128 (inplace) for different sizes.

scheme and input size	Encrypt in cycles (cycles/byte) as median and best estimate	Decrypt in cycles (cycles/byte) as median and best estimate
Ascon-128a (inplace) 1 KiB	12256 (11.97) 12285 (12.00)	10650 (10.40) 10683 (10.43)
Ascon-128a (inplace) 2 KiB	23424 (11.44) 23448 (11.45)	20631 (10.07) 20646 (10.08)
Ascon-128a (inplace) 4 KiB	45958 (11.22) 46046 (11.24)	40598 (9.91) 40631 (9.92)
Ascon-128a (inplace) 8 KiB	91025 (11.11) 91132 (11.12)	80527 (9.83) 80631 (9.84)
Ascon-128a (inplace) 16 KiB	180977 (11.05) 180993 (11.05)	160348 (9.79) 160447 (9.79)
Ascon-128a (inplace) 32 KiB	361847 (11.04) 363494 (11.09)	320107 (9.77) 320316 (9.78)
Ascon-128a (inplace) 64 KiB	723263 (11.04) 723399 (11.04)	639744 (9.76) 640336 (9.77)
Ascon-128a (inplace) 1 MiB	11519338 (10.99) 11516235 (10.98)	10226937 (9.75) 10229626 (9.76)
Ascon-128a (inplace) 2 MiB	23036649 (10.98) 23033042 (10.98)	20464668 (9.76) 20480863 (9.77)
Ascon-128a (inplace) 4 MiB	46065999 (10.98) 46071957 (10.98)	40929240 (9.76) 40957362 (9.76)
Ascon-128a (inplace) 8 MiB	92152360 (10.99) 92161377 (10.99)	81900876 (9.76) 81935902 (9.77)
Ascon-128a (inplace) 16 MiB	208332154 (12.42) 208255281 (12.41)	164186416 (9.79) 164218617 (9.79)
Ascon-128a (inplace) 32 MiB	369555112 (11.01) 369631367 (11.02)	328309786 (9.78) 328372861 (9.79)
Ascon-128a (inplace) 64 MiB	739119163 (11.01) 739150913 (11.01)	656750437 (9.79) 656861681 (9.79)

Table 10.5: Benchmarks of Ascon-128a (inplace) for different sizes.

scheme and input size	Encrypt in cycles (cycles/byte) as median and best estimate	Decrypt in cycles (cycles/byte) as median and best estimate
Ascon-80pq (inplace) 1 KiB	15862 (15.49) 15929 (15.56)	15309 (14.95) 15340 (14.98)
Ascon-80pq (inplace) 2 KiB	30474 (14.88) 30625 (14.95)	29962 (14.63) 29995 (14.65)
Ascon-80pq (inplace) 4 KiB	59894 (14.62) 60082 (14.67)	59270 (14.47) 59339 (14.49)
Ascon-80pq (inplace) 8 KiB	118403 (14.45) 118702 (14.49)	117915 (14.39) 118138 (14.42)
Ascon-80pq (inplace) 16 KiB	235928 (14.40) 236954 (14.46)	235121 (14.35) 235941 (14.40)
Ascon-80pq (inplace) 32 KiB	471561 (14.39) 472151 (14.41)	469593 (14.33) 469938 (14.34)
Ascon-80pq (inplace) 64 KiB	942406 (14.38) 943720 (14.40)	938462 (14.32) 939923 (14.34)
Ascon-80pq (inplace) 1 MiB	15028823 (14.33) 15046308 (14.35)	15007269 (14.31) 15021887 (14.33)
Ascon-80pq (inplace) 2 MiB	30066381 (14.34) 30103732 (14.35)	30016736 (14.31) 30041649 (14.32)
Ascon-80pq (inplace) 4 MiB	60181766 (14.35) 60232006 (14.36)	60040545 (14.31) 60097567 (14.33)
Ascon-80pq (inplace) 8 MiB	120370533 (14.35) 120410836 (14.35)	120154620 (14.32) 120211754 (14.33)
Ascon-80pq (inplace) 16 MiB	264673556 (15.78) 264720630 (15.78)	240592114 (14.34) 240662410 (14.34)
Ascon-80pq (inplace) 32 MiB	482527341 (14.38) 482651279 (14.38)	481033020 (14.34) 481160783 (14.34)
Ascon-80pq (inplace) 64 MiB	964885706 (14.38) 965124033 (14.38)	962102066 (14.34) 962355572 (14.34)

Table 10.6: Benchmarks of Ascon-80pq (inplace) for different sizes.

scheme and input size	Encrypt in cycles (cycles/byte) as median and best estimate	Decrypt in cycles (cycles/byte) as median and best estimate
aes-gcm-128 1 KiB	2852 (2.79) 2853 (2.79)	2326 (2.27) 2329 (2.27)
aes-gcm-128 2 KiB	5423 (2.65) 5427 (2.65)	4592 (2.24) 4600 (2.25)
aes-gcm-128 4 KiB	10545 (2.57) 10552 (2.58)	8907 (2.17) 8914 (2.18)
aes-gcm-128 8 KiB	20638 (2.52) 20657 (2.52)	17426 (2.13) 17438 (2.13)
aes-gcm-128 16 KiB	41418 (2.53) 41494 (2.53)	35285 (2.15) 35364 (2.16)
aes-gcm-128 32 KiB	83161 (2.54) 83263 (2.54)	70738 (2.16) 70928 (2.16)
aes-gcm-128 64 KiB	166081 (2.53) 166438 (2.54)	141287 (2.16) 141486 (2.16)
aes-gcm-128 1 MiB	2730251 (2.60) 2735845 (2.61)	2337858 (2.23) 2341534 (2.23)
aes-gcm-128 2 MiB	5621183 (2.68) 5617846 (2.68)	4790858 (2.28) 4788046 (2.28)
aes-gcm-128 4 MiB	11304316 (2.70) 11301983 (2.69)	9635883 (2.30) 9670106 (2.31)
aes-gcm-128 8 MiB	23116765 (2.76) 23134984 (2.76)	19870215 (2.37) 19827245 (2.36)
aes-gcm-128 16 MiB	46692266 (2.78) 46720339 (2.78)	39948449 (2.38) 39886653 (2.38)
aes-gcm-128 32 MiB	131610298 (3.92) 131613803 (3.92)	117961614 (3.52) 117974282 (3.52)
aes-gcm-128 64 MiB	267018434 (3.98) 266865192 (3.98)	239843578 (3.57) 239740463 (3.57)

Table 10.7: Benchmarks of aes-gcm-128 for different sizes.

scheme and input size	Encrypt in cycles (cycles/byte) as median and best estimate	Decrypt in cycles (cycles/byte) as median and best estimate
aes-gcm-256 1 KiB	3080 (3.01) 3083 (3.01)	2320 (2.27) 2324 (2.27)
aes-gcm-256 2 KiB	5810 (2.84) 5814 (2.84)	4591 (2.24) 4594 (2.24)
aes-gcm-256 4 KiB	11307 (2.76) 11315 (2.76)	8918 (2.18) 8926 (2.18)
aes-gcm-256 8 KiB	22133 (2.70) 22168 (2.71)	17437 (2.13) 17446 (2.13)
aes-gcm-256 16 KiB	44398 (2.71) 44539 (2.72)	35293 (2.15) 35336 (2.16)
aes-gcm-256 32 KiB	89122 (2.72) 89234 (2.72)	70764 (2.16) 70901 (2.16)
aes-gcm-256 64 KiB	177953 (2.72) 178293 (2.72)	141299 (2.16) 141476 (2.16)
aes-gcm-256 1 MiB	2927851 (2.79) 2931483 (2.80)	2332194 (2.22) 2336692 (2.23)
aes-gcm-256 2 MiB	5984155 (2.85) 5985905 (2.85)	4792218 (2.29) 4786159 (2.28)
aes-gcm-256 4 MiB	12108353 (2.89) 12116325 (2.89)	9632384 (2.30) 9620415 (2.29)
aes-gcm-256 8 MiB	25028263 (2.98) 25008787 (2.98)	19514011 (2.33) 19514129 (2.33)
aes-gcm-256 16 MiB	50114932 (2.99) 50124200 (2.99)	39875572 (2.38) 39815639 (2.37)
aes-gcm-256 32 MiB	138912974 (4.14) 138972682 (4.14)	118999690 (3.55) 119052977 (3.55)
aes-gcm-256 64 MiB	277072261 (4.13) 277070911 (4.13)	236672455 (3.53) 236667244 (3.53)

Table 10.8: Benchmarks of aes-gcm-256 for different sizes.

scheme and input size	Encrypt in cycles (cycles/byte) as median and best estimate	Decrypt in cycles (cycles/byte) as median and best estimate
aes-gcm-siv-128 1 KiB	3419 (3.34) 3421 (3.34)	3735 (3.65) 3738 (3.65)
aes-gcm-siv-128 2 KiB	6041 (2.95) 6051 (2.95)	6869 (3.35) 6871 (3.35)
aes-gcm-siv-128 4 KiB	11211 (2.74) 11211 (2.74)	12818 (3.13) 12816 (3.13)
aes-gcm-siv-128 8 KiB	21690 (2.65) 21689 (2.65)	24925 (3.04) 24921 (3.04)
aes-gcm-siv-128 16 KiB	42168 (2.57) 42228 (2.58)	48644 (2.97) 48628 (2.97)
aes-gcm-siv-128 32 KiB	85061 (2.60) 85067 (2.60)	97953 (2.99) 97931 (2.99)
aes-gcm-siv-128 64 KiB	170043 (2.59) 170201 (2.60)	195916 (2.99) 196046 (2.99)
aes-gcm-siv-128 1 MiB	2765764 (2.64) 2767434 (2.64)	3185011 (3.04) 3186508 (3.04)
aes-gcm-siv-128 2 MiB	5715624 (2.73) 5720685 (2.73)	6505373 (3.10) 6513565 (3.11)
aes-gcm-siv-128 4 MiB	11423533 (2.72) 11429209 (2.72)	13039767 (3.11) 13055528 (3.11)
aes-gcm-siv-128 8 MiB	23131546 (2.76) 23184199 (2.76)	27073330 (3.23) 27043401 (3.22)
aes-gcm-siv-128 16 MiB	46729320 (2.79) 46832551 (2.79)	54481752 (3.25) 54431208 (3.24)
aes-gcm-siv-128 32 MiB	132331804 (3.94) 132301051 (3.94)	147959604 (4.41) 148000865 (4.41)
aes-gcm-siv-128 64 MiB	265059624 (3.95) 265068283 (3.95)	297114057 (4.43) 297038637 (4.43)

Table 10.9: Benchmarks of aes-gcm-siv-128 for different sizes.

scheme and input size	Encrypt in cycles (cycles/byte) as median and best estimate	Decrypt in cycles (cycles/byte) as median and best estimate
aes-gcm-siv-256 1 KiB	3987 (3.89) 3998 (3.90)	4463 (4.36) 4468 (4.36)
aes-gcm-siv-256 2 KiB	6792 (3.32) 6795 (3.32)	7990 (3.90) 7992 (3.90)
aes-gcm-siv-256 4 KiB	12312 (3.01) 12314 (3.01)	14658 (3.58) 14692 (3.59)
aes-gcm-siv-256 8 KiB	23477 (2.87) 23481 (2.87)	28059 (3.43) 28069 (3.43)
aes-gcm-siv-256 16 KiB	45369 (2.77) 45417 (2.77)	54878 (3.35) 54909 (3.35)
aes-gcm-siv-256 32 KiB	91109 (2.78) 91124 (2.78)	110195 (3.36) 110450 (3.37)
aes-gcm-siv-256 64 KiB	181571 (2.77) 181707 (2.77)	219604 (3.35) 219606 (3.35)
aes-gcm-siv-256 1 MiB	2943877 (2.81) 2946129 (2.81)	3564010 (3.40) 3566481 (3.40)
aes-gcm-siv-256 2 MiB	6061912 (2.89) 6063739 (2.89)	7258860 (3.46) 7262922 (3.46)
aes-gcm-siv-256 4 MiB	12186716 (2.91) 12193506 (2.91)	14552729 (3.47) 14564510 (3.47)
aes-gcm-siv-256 8 MiB	24674182 (2.94) 24655687 (2.94)	30173715 (3.60) 30169187 (3.60)
aes-gcm-siv-256 16 MiB	49850168 (2.97) 49831467 (2.97)	60421351 (3.60) 60454718 (3.60)
aes-gcm-siv-256 32 MiB	138506160 (4.13) 138518994 (4.13)	159025068 (4.74) 159023460 (4.74)
aes-gcm-siv-256 64 MiB	280312571 (4.18) 280236659 (4.18)	321519161 (4.79) 321465462 (4.79)

Table 10.10: Benchmarks of aes-gcm-siv-256 for different sizes.

scheme and input size	Encrypt in cycles (cycles/byte) as median and best estimate	Decrypt in cycles (cycles/byte) as median and best estimate
chacha20poly1305 1 KiB	3742 (3.65) 3745 (3.66)	1875 (1.83) 1875 (1.83)
chacha20poly1305 2 KiB	6361 (3.11) 6369 (3.11)	2885 (1.41) 2889 (1.41)
chacha20poly1305 4 KiB	11624 (2.84) 11633 (2.84)	4452 (1.09) 4455 (1.09)
chacha20poly1305 8 KiB	21930 (2.68) 21983 (2.68)	7955 (0.97) 7958 (0.97)
chacha20poly1305 16 KiB	42642 (2.60) 42672 (2.60)	14976 (0.91) 14992 (0.92)
chacha20poly1305 32 KiB	85339 (2.60) 85395 (2.61)	30624 (0.93) 30581 (0.93)
chacha20poly1305 64 KiB	166533 (2.54) 166643 (2.54)	56064 (0.86) 56386 (0.86)
chacha20poly1305 1 MiB	2713568 (2.59) 2714895 (2.59)	964584 (0.92) 967601 (0.92)
chacha20poly1305 2 MiB	5607755 (2.67) 5609831 (2.67)	2064648 (0.98) 2066000 (0.99)
chacha20poly1305 4 MiB	11450323 (2.73) 11452020 (2.73)	4229244 (1.01) 4230838 (1.01)
chacha20poly1305 8 MiB	23410702 (2.79) 23391487 (2.79)	8578521 (1.02) 8529849 (1.02)
chacha20poly1305 16 MiB	46879157 (2.79) 46896213 (2.80)	17009536 (1.01) 17068015 (1.02)
chacha20poly1305 32 MiB	132973404 (3.96) 133080652 (3.97)	74170734 (2.21) 74181324 (2.21)
chacha20poly1305 64 MiB	268716076 (4.00) 268582232 (4.00)	148097491 (2.21) 148054463 (2.21)

Table 10.11: Benchmarks of chacha20poly1305 for different sizes.

Discussion

We split the discussion into two sections: the first will be AES-GCM variants and chacha20poly1305 vs. Xoofff-Tink, and the second part will be Ascon vs. Xoofff-Tink. We decided to make this split mainly due to the speed of the results and to keep the graph more coherent. We used the median rather than the cargo's best estimate for comparison.

AES-GCM & ChachaPloy vs Xoofff-Tink

As we can see above, in Figures 10.3, 10.7-10.11 our Xoofff-Tink implementation is slower for shorter size plaintexts and faster for very large message's size (more than 16 MiB) in wrap/encryption and slower for all size messages for unwrap/decryption. For wrap/encryption, we can see that for short plaintexts (less than 1 MiB) Xoofff-Tink is around 1.2-1.5 times slower, and for medium, they are around the same speed, and for longer ones (more than 16 MiB) Xoofff-Tink is faster. For unwrap/decryption, we can see that Xoofff-Tink is around 1.04-2.4 slower for messages less than 8 MiB and around 1-1.2 slower for longer messages, with some cases in which Xoofff-Tink is faster, for example, against AES-GCM-SIV, and Xoofff-Tink is around 1.2 times faster for messages longer than 16 MiB. Notice that we report the differences according to the AVX2 benchmarking.

As we mentioned above, the comparison against AES schemes is imbalanced as AES is implemented in hardware, and again, we used a machine that has AVX2 and is not the most optimized version of our code, which takes advantage of AVX512.

The goal of this benchmark was to place our scheme in performance among the more well-known schemes and provide the reader with a better understanding of the speed of our implementation. While our main focus was to benchmark our implementation against Google Tink AEAD, we recognized the importance of providing a comprehensive comparison. As there are few benchmarks for Google Tink AEAD in the literature, we also decided to report benchmarking against more acknowledged schemes.

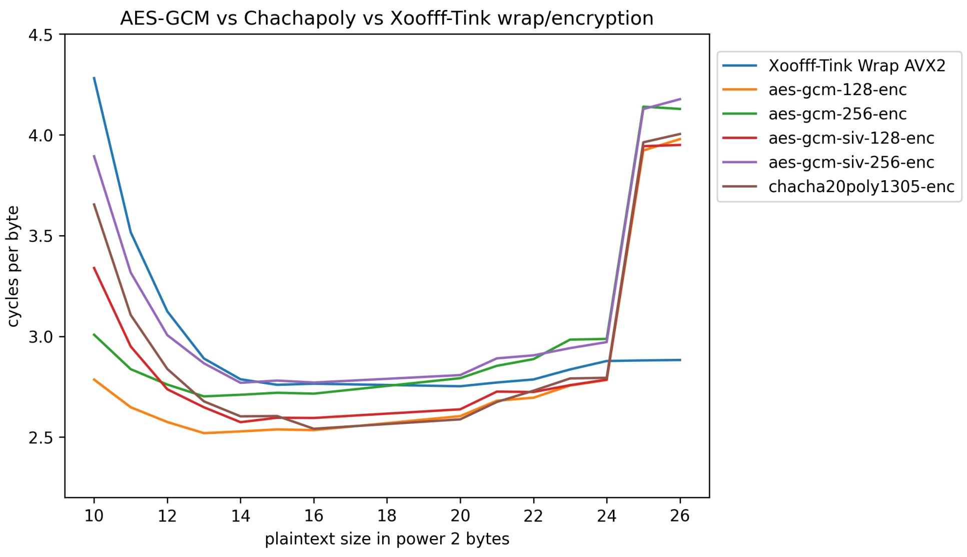


Figure 10.3: Benchmarks of wrap Xoofff-Tink and encrypt of AES-GCM variants and chacha20poly1305 for different sizes.

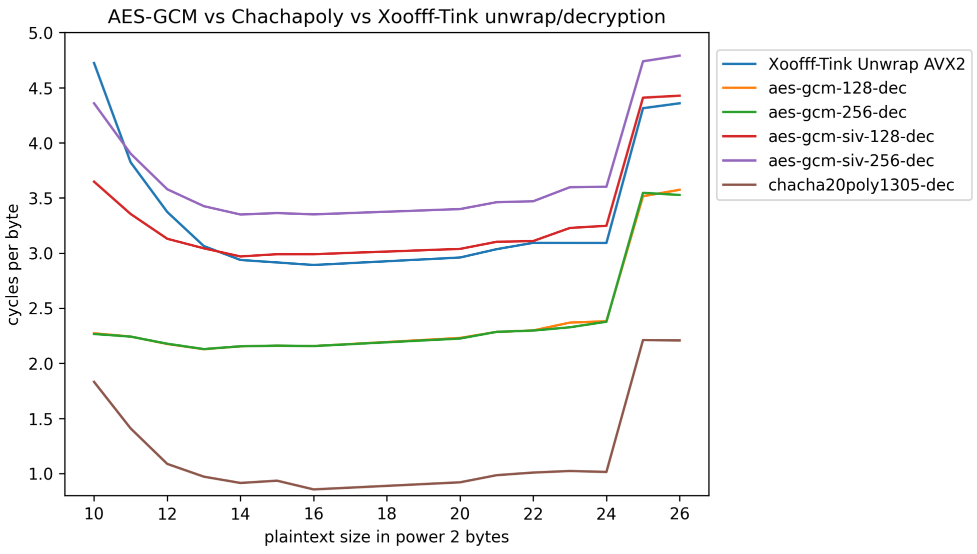


Figure 10.4: Benchmarks of unwrap Xoofff-Tink and decrypt of AES-GCM variants and chacha20poly1305 for different sizes.

Ascon vs Xoofff-Tink

Our final benchmarking is against the newly lightweight cryptography standardized scheme Ascon. As we can see in Figures 10.3-10.6, Xoofff-Tink is around 2-5 times faster in both wrap/encryption and unwrap/decryption for the AVX2 implementation and 2-3 times faster for the non-parallelized version.

This benchmark aims to compare Xoofff-Tink to the new standardized scheme for lightweight cryptography as we argue that our scheme can be used for lightweight cryptography.

We believe the main reason for the speed differences between the two schemes is the difference in their message processing. While Xoofff-Tink processes 384 bits of messages for each permutation call, Ascon processes only 64 bits, resulting in many more permutation calls for each scheme operation. As we are using rather large message sizes, this creates a considerable overhead. Another reason might be the implementation of the scheme. The implementation was done by the RustCrypto library developers and not the Ascon team. Perhaps if the Ascon team had implemented a Rust version, it could have been faster and could have taken advantage of different optimizations, such as the AVX2.

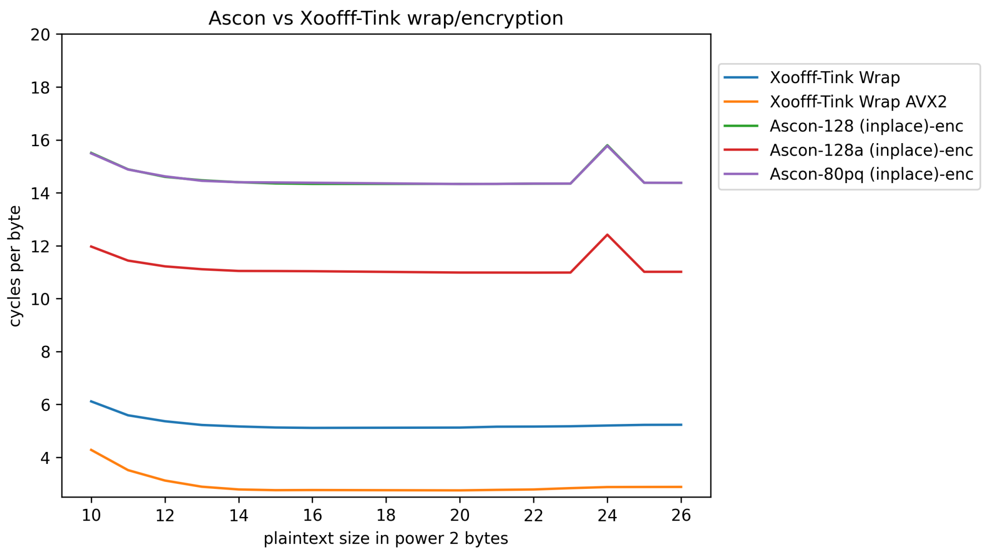


Figure 10.5: Benchmarks of wrap Xoofff-Tink and encrypt of Ascon variants for different sizes.

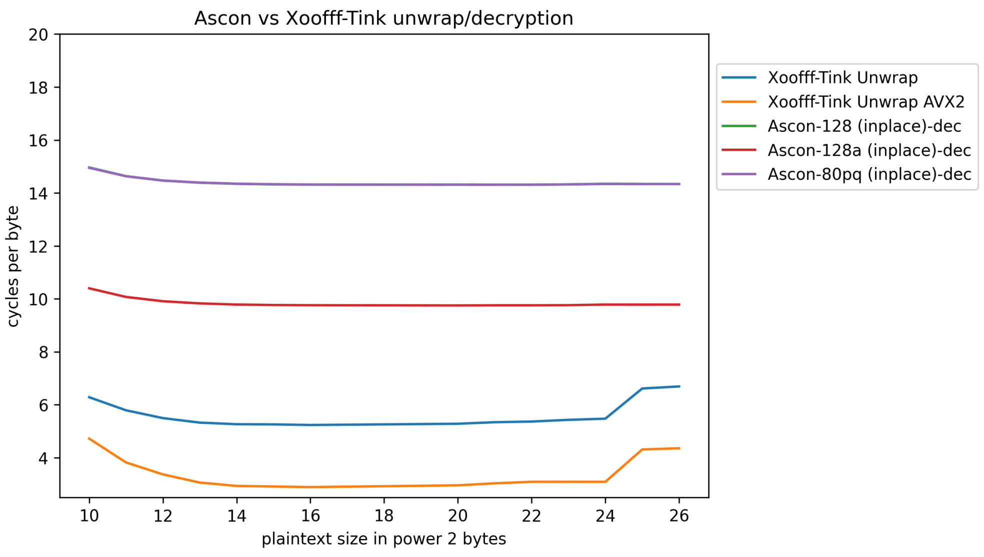


Figure 10.6: Benchmarks of unwrap Xoofff-Tink and decrypt of Ascon for different sizes.

10.4 General Observations

We believe a few additional observations about the benchmarking reported above are important to be mentioned.

We can notice some differences between the speed of Xoofff-Tink wrap and unwrap in some cases. If we examine the two implementations (or the pseudocode), we can see that the wrap and unwrap operations are rather similar in nature. Unwrap has some additional clone operations and some additional if checks, but compared to the absorb and squeeze operations, those do not play a big role in our scheme's speed performance. This is clear in the c code implementation, where we see similar speed performances for wrap and unwrap. We can see a similar pattern in the non-AVX2 implantation of Xoofff-Tink in Rust, although it is not as clear as in the case of c, the wrap and unwrap have similar speeds with the exceptions of very large message sizes, which we will discuss in later in this section. In the case of Xoofff-Tink in Rust using AVX2, we see more significant speed differences between wrap and unwrap. This is due to different optimizations the Rust compiler performs under the hood. The AVX2 part of the code is not explicitly implemented but is left for the compiler to do so while providing the expected amount of values to fit on the 256-bit register. The specific implementation is abstract, and the compiler decides how to implement it. It might perform additional optimization through the code using the AVX2 optimization features. This means that in the code of the wrap, the compiler manages to perform more optimizations compared to the unwrap calls, which explains the differences in speed performances that are not present in the C or non-AVX2 implementation. The C AVX2 implementation is specified and optimized by the Keccak team and is part of the XKCP package that we used for our C implementation.

There seems to be a pick in the speed performance for the Rust implementations for large messages across almost all schemes. We believe this is due to the memory safety checks that Rust has over C. Such observations can be studied further to better understand Rust's behavior, but they are outside the scope of this thesis.

Chapter 11

Xoofff-Tink in PostGuard

In this section, we discuss the PostGuard project, how we included our Xoofff-Tink implementation into the project, and the result of this collaboration in terms of analysis of the ease of integration of our implementation.

11.1 The PostGuard Project

The PostGaurd: encryption for all project is an interdisciplinary collaboration whose goal, as the name suggests, is to provide usable encryption to everyone, whether in the public or private sector. The project aims to create an easy-to-use plugin on top of our traditional email application, which will provide encryption and authentication for the users [BBJ+23]. Encrypted email providers such as PGP have existed for a while [SWR+22]. However, they are not widely used due to the difficulty in setting up and usage, mainly because of challenges in cryptographic key management [BBJ+24]. Meanwhile, encryption is becoming increasingly essential, with legislation, such as the GDPR, enforcing laws that require encryption of communication between different parties (for instance, between doctors and patients) and mass surveillance activities of internationally operating intelligence agencies on the rise. The project battles the biggest difficulties of PGP (key management and user experience) by considering user experience while designing the tool and the cryptography around it. Their novelty lies in combining their cryptography approach, identity-based encryption with an identity wallet. Each individual should possess such a wallet, and using their wallet, they can prove their identity. When a sender sends an email, they should specify who can decrypt it by providing the recipient attributes (such as name, email address, or some other attribute). On the recipient side, the user can use their attributes from the wallet to decrypt the email. Thus, the PostGuard reduces decryption to authentication.

11.2 Goal of our Collaboration

After analyzing the security of our scheme and implementing two instances, in Rust and C, we would also like to check the usability of Xoofff-Tink. To that end, we collaborated with iHub on their PostGuard project. The primary objective of our collaboration with PostGuard is to demonstrate our new cryptographic scheme's practical applicability and ease of integration. By replacing their existing AES-GCM scheme with our scheme, we aimed to illustrate its user-friendliness and seamless adaptability in a real-world scenario.

This collaboration serves as a case study for the broader adoption of our cryptographic scheme. By successfully implementing our system in a live environment, we aim to establish a precedent for its application in various fields, emphasizing its versatility and scalability.

11.3 Results of Collaboration

Currently, PostGuard provides two services: a plugin on top of some traditional email services and an encryption and decryption CLI tool. The cryptography scheme used for those two services is the same, implemented in one place and used in both tools. PostGuard uses the bytes-in-bytes-out approach, the program gives the plaintext to the cryptographic implementation and expects the output to be written to the same input vector, recall that the output, in our case, is a ciphertext, counter, and a tag. For that end, we can use the Rust implementation of our scheme with the bytes-in-bytes-out approach. As we have already implemented it, we only had to change the init, wrap, and unwrap call from AES-GCM to our Xoofff-Tink and introduce a managed counter, which increases after each wrap use. Those changes were fairly simple and involved only modifying a few lines of code. This demonstrates that with minor effort, one can replace one of the most commonly used schemes for authentication encryption with Xoofff-Tink.

Does this mean that from now on, people can use Xoofff-Tink for authentication and encryption on their email providers? Unfortunately, it is not that easy. Although the scheme change in the code is simple, another issue must be considered before changing the scheme. As PostGuasrd is already used, some emails have already been encrypted using AES-GCM. Suppose one will be to change the scheme from AES-GCM to Xoofff-Tink. In that case, all the emails before the change will no longer be accessible, as the scheme applied to the authenticated and encrypted email will provide a non-readable authentication and decryption. There are solutions for this issue. One can use both schemes for a period of time, but the question of when one can drop the older scheme is not easy to answer. Another solution can be to encrypt all emails using the new scheme and save them instead of the old ones. This is a challenging task and can be an expensive one. The issue we are facing is not an implementation one but rather a software engineering one, and a solution for such a problem cannot be answered easily and requires thinking from the developers of PostGuard.

Does this mean our effort to include Xoofff-Tink in PostGuard and collaboration is in vain? Quite the opposite, our goal was not to create a new version for PostGuard, which will replace the old one, but rather to see how difficult it would be to do so. The answer to this question is promising, with minimal effort, highlighting our scheme's ease of usability. Another outcome of this collaboration is the creation of a CLI tool that wraps and unwraps messages. As mentioned above, PostGaurd has two tools: a plugin on top of email providers and a CLI tool that wraps and unwraps files. Our implementation of the second tool works and can be used without any issues.

In conclusion, we believe the collaboration to be successful, which highlights the practical applicability and ease of integration of our new cryptographic scheme and showcases how one can change one of the most commonly used authenticated encryption schemes to our Xoofff-Tink scheme with minimal effort.

Chapter 12

Conclusions and Future Work

In this thesis, we have introduced Deck-Tink, a new authenticated encryption (AE) mode, and analyzed its security in the multi-target scenario. We showed how, with an ideal-world scheme with the same interface as Deck-Tink built on top of the jammin cipher, we can prove a multi-target security bound for our mode without necessitating the construction of a new ideal world, a significant advancement over existing methodologies such as those used in Google Tink AEAD.

Further, we instantiated the Deck-Tink mode using the deck function Xoofff, resulting in our Xoofff-Tink scheme. We implemented Xoofff-Tink in both C and Rust [Alt24], and have benchmarked them against several AE schemes. We report the following findings for our benchmarks:

- Xoofff-Tink C implementation vs Google Tink AEAD C++ implementation: for wrap/encryption, we can see that our implementation is at least twice as fast for short plaintext (less than 128 Kib) and between 1.10 and 1.42 faster for longer plaintexts. For unwrap/decryption, we can see that our implementation is around 2 to 3 times slower for medium-size plaintext (more than 1 Kib and less than 4 MiB) and between 1.003 and 1.76 slower for other plaintext sizes.

- Xoofff-Tink Rust implementation vs AES-GCM & ChachaPoly implementation: for wrap/encryption, we can see that for short plaintexts (less than 1 MiB) Xoofff-Tink is around 1.2-1.5 times slower, and for medium, they are around the same speed, and for longer ones (more than 16 MiB) Xoofff-Tink is faster. For unwrap/decryption, we can see that Xoofff-Tink is around 1.04-2.4 slower for messages less than 8 MiB and around 1-1.2 slower for longer messages, with some cases in which Xoofff-Tink is faster, for example, against AES-GCM-SIV, and Xoofff-Tink is around 1.2 times faster for messages longer than 16 MiB.

- Xoofff-Tink Rust implementation vs Ascon: Xoofff-Tink is around 2-5 times faster in both wrap/encryption and unwrap/decryption for the

AVX2 implementation and 2-3 times faster for the non-parallelized version.

Moreover, our collaboration with iHub on the PostGuard project showcased the practical applicability and seamless integration capabilities of Xoofff-Tink, demonstrating its potential beyond theoretical constructs into real-world utility.

Xoofff-Tink has demonstrated potential, particularly for streaming encryption on platforms without dedicated AES support, as well as in applications requiring online data transmissions, such as email, file sharing, and potentially even TCP communications, due to its support for out-of-order message handling.

We believe that more work can be done to better understand the potential of Xoofff-Tink and Deck-Tink. In this thesis, we implemented Xoofff-Tink in Rust using the Rust AVX high-level operations. However, implementing native AVX instructions could potentially improve performance, resulting in faster wrap and unwrap operations. Additionally, exploring more use cases, such as in low-level devices that do not have AES in hardware, could expand the utility of Xoofff-Tink and show its potential and improvement compared to current schemes like AES-GCM. Furthermore, it would be beneficial to test Xoofff-Tink in different network protocols to evaluate its efficacy and robustness in varied communication scenarios.

We also observed two other areas that might be worth further investigation. The first is the difference in speed between encryption and decryption for Google Tink AEAD. As AES-GCM is used as a primitive, one would expect to see similar speeds for the two operations, but we notice that decryption is almost twice as fast. Secondly, in our unwrap benchmark of Xoofff-Tink in Rust and most of the other AE schemes in Rust, we observed a high pike around messages with size 16 MiB. We expect this pick to originate from safe Rust memory handling, but more investigation into the exact cause can help us better understand Rust and its use in cryptography implementations.

Bibliography

[AKC23] G. Van Assche, R. Van Keer, and Contributors. Extended keccak code package, 2023. https://github.com/XKCP/XKCP.

[Alt24] D. Alter. Xoofff-tink implementation, 2024. https://github.com/DorAlter/Xoofff-Tink/.

[AO16] A.A.M Aliyu and A. Olaniyan. Vigenere cipher: trends, review and possible modifications. *International Journal of Computer Applications*, 135(11):46–50, 2016.

[BBJ⁺23] L. Botros, M. Brandon, B. Jacobs, D. Ostkamp, H. Schraffenberger, and M. Venema. Postguard: Towards easy and secure email communication. In *Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems*, CHI EA '23, New York, NY, USA, 2023. Association for Computing Machinery. https://doi.org/10.1145/3544549.3585622.

[BBJ⁺24] L. Botros, M. Brandon, B. Jacobs, D. Ostkamp, H. Schraffenberger, and M. Venema. Postguard: encryption for all (webpage), 2024. https://ihub.ru.nl/project/postguard.page.

[BDH⁺16] G. Bertoni, J. Daemen, S. Hoffert, M. Peeters, G. Van Assche, and R. Van Keer. Farfalle: parallel permutation-based cryptography. Cryptology ePrint Archive, Paper 2016/1188, 2016. https://eprint.iacr.org/2016/1188.

[BDH⁺22] N. Băcuieți, J. Daemen, S. Hoffert, G. Van Assche, and R. Van Keer. Jammin' on the deck. Cryptology ePrint Archive, Paper 2022/531, 2022. https://eprint.iacr.org/2022/531.

[BN00] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. Cryptology ePrint Archive, Paper 2000/025, 2000. https://eprint.iacr.org/2000/025.

[Boz24] A. Bozhko. Properties of AEAD Algorithms, June 2024. https://datatracker.ietf.org/doc/draft-irtf-cfrg-aead-properties/.

[CS13] S. Chen and J. Steinberger. Tight security bounds for key-alternating ciphers. Cryptology ePrint Archive, Paper 2013/222, 2013. https://eprint.iacr.org/2013/222.

[Cue21] A.V. Cueva. The Intel Advanced Vector Extensions 512 (Intel® AVX-512) Vector Length Extensions Feature on Intel® Xeon® Scalable Processors. *Intel*, 2018 accessed on May 2021. https://software.intel.com/content/www/us/en/develop/articles/the-intel-advanced-vector-extensions-512-feature-on-intel-xeon-scalable.html?wapkw=advanced%20vector%20extensions.

[Den23] F. Denis. rust-xoodyak, Last accessed September 2023. https://github.com/jedisct1/rust-xoodyak.

[dev23] RustCrypto developers. Authenticated encryption with associated data (aead) traits, Last accessed September 2023. https://docs.rs/aead/latest/aead/.

[DFY⁺17] W. Diehl, F. Farahmand, P. Yalla, J.P Kaps, and K. Gaj. Comparison of hardware and software implementations of selected lightweight block ciphers. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, 2017. https://ieeexplore.ieee.org/document/8056808.

[DGP07] L. Dorrendorf, Z. Gutterman, and B. Pinkas. Cryptanalysis of the random number generator of the windows operating system. Cryptology ePrint Archive, Paper 2007/419, 2007. https://eprint.iacr.org/2007/419.

[DHAK18] J. Daemen, S. Hoffert, G. Van Assche, and R. Van Keer. The design of xoodoo and xoofff. *IACR Transactions on Symmetric Cryptology*, 2018(4):1–38, Dec. 2018. https://tosc.iacr.org/index.php/ToSC/article/view/7359.

[doc24a] Rust docs. cargo-bench, Last accessed on March 2024. https://doc.rust-lang.org/cargo/commands/cargo-bench.html.

[doc24b] Rust docs. Crate criterion, Last accessed on March 2024. https://docs.rs/criterion/latest/criterion/.

[ESDH21] M. Emre, R. Schroeder, K. Dewey, and B. Hardekopf. Translating c to safer rust. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021. https://doi.org/10.1145/3485498.

[FFL12] E. Fleischmann, C. Forler, and S. Lucks. Mcoe: A family of almost foolproof on-line authenticated encryption schemes. In

Fast Software Encryption Workshop, 2012. https://www.iacr.org/archive/fse2012/75490200/75490200.pdf.

[Goo23a] Google. Tink cryptographic library (docs), 2023. https://developers.google.com/tink.

[Goo23b] Google. Tink library (code), 2023. https://github.com/google/tink.

[HDWH12] N. Heninger, Z. Durumeric, E. Wustrow, and J.A.. Halderman. Mining your ps and qs: Detection of widespread weak keys in network devices. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 205–220, Bellevue, WA, August 2012. USENIX Association. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/heninger.

[HRRV15] V.T. Hoang, R. Reyhanitabar, P. Rogaway, and D. Vizár. Online authenticated-encryption and its nonce-reuse misuse-resistance. In *Advances in Cryptology – CRYPTO 2015*, page 493–517, Berlin, Heidelberg, 2015. Springer-Verlag. https://eprint.iacr.org/2015/189.pdf.

[HS20] V.T. Hoang and Y. Shen. Security of streaming encryption in google's tink library. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 243–262, New York, NY, USA, 2020. Association for Computing Machinery. https://eprint.iacr.org/2020/1019.pdf.

[Int21] Intel. Intrinsics for Intel Advanced Vector Extensions 2. *Intrinsics for Intel Advanced Vector Extensions 2*, 2013 accessed on April 2021. https://www.cism.ucl.ac.be/Services/Formations/ICS/ics_2013.0.028/composer_xe_2013/Documentation/en_US/compiler_c/main_cls/index.htm#GUID-9E84F9C5-1711-4F59-8742-8F9DF283A472.htm.

[JJKD17] R. Jung, J.H. Jourdan, R. Krebbers, and D. Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), dec 2017. https://doi.org/10.1145/3158154.

[LBD23] C. Lefevre, Y. Belkheyar, and J. Daemen. Kirby: A robust permutation-based PRF construction. Cryptology ePrint Archive, Paper 2023/1520, 2023. https://eprint.iacr.org/2023/1520.

[LHA+12] A.K. Lenstra, J.P. Hughes, M. Augier, J.W. Bos, T. Kleinjung, and Christophe Wachter. Public keys. In *Advances in Cryptology*

- *Crypto 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 626–642. Springer, 2012.

[Lom21] C. Lomont. Introduction to Intel Advanced Vector Extensions. *Intel White Paper*, 2011, accessed on May 2021. https://software.intel.com/content/www/us/en/develop/articles/introduction-to-intel-advanced-vector-extensions.html.

[MKW18] K. Mindermann, P. Keck, and S. Wagner. How usable are rust cryptography apis? In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 143–154, 2018.

[MV04] D.A. McGrew. and J. Viega. The security and performance of the galois/counter mode (gcm) of operation. In Anne Canteaut and Kapaleeswaran Viswanathan, editors, *Progress in Cryptology - INDOCRYPT 2004*, pages 343–355, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007/978-3-540-30556-9_27.

[MV05] D.A. McGrew and J. Viega. The galois/counter mode of operation (gcm), 2005. https://api.semanticscholar.org/CorpusID:6053538.

[NL18] Y. Nir and A. Langley. ChaCha20 and Poly1305 for IETF Protocols. RFC 8439, June 2018. https://www.rfc-editor.org/info/rfc8439.

[Pat08] J. Patarin. The "coefficients h" technique., 08 2008.

[PNT+15] D. Pandya, K.R. Narayan, S. Thakkar, T. Madhekar, and B.S. Thakare. Brief history of encryption. *International Journal of Computer Applications*, 131(9):28–31, 2015.

[Roy23] A. Roy. Xoofff (rust implementation), Last accessed September 2023. https://github.com/itzmeanjan/xoofff.

[SWR+22] C. Stransky, O. Wiese, V. Roth, Y. Acar, and S. Fahl. 27 years and 81 million opportunities later: Investigating the use of email encryption for an entire university. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 860–875, 2022.