Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Actually Good Encryption? Confusing Users by Changing Nonces

Semester Project

Mirco Stäuble

July 12, 2022

Advisors: Prof. Dr. Kenny Paterson

Applied Cryptography Group
Institute of Information Security
Department of Computer Science, ETH Zürich

Abstract

Age is a command line encryption tool designed by Filippo Valsorda, which is gaining popularity. Using Age a user can encrypt arbitrary files into so called Age-files. Thereby, each file can be encrypted for multiple parties. Each party is then able to decrypt the file.

In this report we provide an overview of the encryption and decryption process performed in Age. We take a closer look at the AEAD scheme ChaCha20-Poly1305, which is used in Age as a Stream cipher.

We describe why ChaCha20-Poly1305 is not a robust encryption scheme by showing how we can compute an ambiguous ciphertext which decrypts correctly under two different keys.

Making use of this property we describe an attack against Age, in which an adversary creates two Age-files, differing in as little as one bit, which will decrypt to two different valid plaintexts.

We explore different scenarios under which this attack can be exploited by a malicious party to create confusion over the contents of Age-files.

Contents

Chapter 1

Introduction

Keeping data secure is a common need in today's world. One way to secure confidential data is to encrypt it. There exist many different tools, such as PGP, GPG, or OpenSSL, which allow users to encrypt their files such that they can be safely stored on device or in the cloud, or be safely sent over an untrusted channel.

One such command line encryption tool is Age, designed by Filippo Valsorda [2]. Age encrypts files using the Authenticated Encryption with Associated Data (AEAD) scheme ChaCha20-Poly1305 [20]. A file encrypted using Age is stored in a custom file format, so called Age-files. Each file can be encrypted to multiple parties, called recipients, such that each of them can decrypt the resulting encrypted file.

Age aims to improve over the current command line encryption tools by providing smaller key components. PGP supports key lengths up to 4096 bits, whereby most commonly 1024-bit keys are used. Similarly, OpenSSL uses 1024-bit keys per default but also supports 2048 or 4096-bit keys. Similar key sizes are used in GPG. Age on the other hand makes use of 256-bit keys per default, which is significantly smaller. Additionally, Age allows to encrypt to SSH keys with a built-in support for GitHub keys. Age is becoming more and more popular. Up until the time of writing this report it managed to collect over 10800 GitHub stars.

In this report we take a closer look at the cryptographic functionality of Age. We investigate how Age uses different cryptographic components for payload encryption, authentication of data and key derivation and their effect on the security of Age.

The ultimate goal we tried to achieve is to break the robustness of Age. Robustness is a concept usually applied only to Authenticated Encryption with Associated Data or Public Key Encryption (PKE) schemes [4, 9]. An AEAD or PKE scheme is called robust, if it is hard to produce a ciphertext which decrypts to valid plaintexts under multiple keys. We apply this concept to the entire construction making up Age, combining key derivation, PKE and Key Encapsulation Mechanisms (KEMs) with AEAD. This means, we try to create an Age-file, which can be decrypted to two different plaintexts. This would allow an adversary to craft ciphertexts validly decrypting to a malicious and harmless payload, thereby allowing the attacker to always claim that the original data encrypted is non-malicious.

However, we were not able to break the robustness of Age. Instead, we found that the way in which Age uses the different cryptographic components allows us to craft nearly identical Age-files, which decrypt to different valid plaintexts. This allows an adversary to minimally alter an Age-file on transit from on recipient to another such that the two parties decrypt the seemingly identical file to different plaintexts.[1] This can be dangerous in scenarios like reporting a suspicious file to a supervisor for examination or completing an abuse report in a messenger system. In these cases, an attacker can craft an Age-file such that the first recipient receives an abusive image and the second one decrypts it to a harmless picture of a dog.[2]

The attack makes use of the fact that we can interleave two different file formats into one single file without them interfering with each other [8, 5] and combines it with the non-robustness of ChaCha20-Poly1305 to create an ambiguous ciphertext, which decrypts to different plaintexts using different keys. Additionally, it abuses a lack of authentication of certain bits in the Age-file format, which therefore can be altered without detection.[3]

Before describing the attack in detail (see Chapter 6), we provide the relevant background on Age's design and the involved cryptographic components, such as ChaCha20-Poly1305, HMAC [13], and HKDF [14].

We start by taking a closer look at the internal functionality of Age, i.e. we provide a detailed analysis of the encryption and decryption process of Age-files, in Chapter 2. This includes a brief overview of the AEAD scheme ChaCha20-Poly1305 and a detailed description of the Age-file format. In Chapter 3, we describe how we can abuse the lack of key commitment in ChaCha20-Poly1305 to create so called ambiguous ciphertexts, which can be decrypted to valid plaintexts using different keys. Chapter 4 discusses how two different file formats can be interleaved into one single file, called a polyglot file, and how an ambiguous ciphertext can be constructed from such a polyglot file. In Chapter 5, we take a closer look at the key deriva-

[1]The files are in fact not identical but differ in at least one bit.

[2]Of course the attack is possible in the both directions, i.e. the first recipient decrypts the file to a harmless payload whilst the second party decrypts it to a malicious file.

[3]Only by comparing the original with the altered file two users could detect the change. However, without additional information they can not deduce which of the two versions is the original and which the altered file.

tion function HKDF and the MAC algorithm HMAC. We describe how an adversary having control over the key used in an HMAC computation can efficiently create collisions and how this translates to HKDF. We combine our insights from all these chapters in Chapter 6, where we describe an attack against Age. In Chapter 7, we present two possible solutions, which when implemented will prevent the attack we found. Finally, we present our conclusion in Chapter 8.

1.1 Notation

Throughout this report we will use "$||$" to denote the concatenation of either byte- or character-strings. Using the symbol "$\oplus$" we denote the bit-wise xor operation between two given bit- or byte-strings. By making use of an array-like notation, $x[n-1]$, we refer to the n-th byte of a given byte array x. We use index 0 to refer to the first byte of an array. We slightly deviate from this notation in Chapter 3, where $x[n-1]$ does not refer to the n-th byte of x but instead denotes the n-th 16-byte block of the byte-string.

In all presented pseudocodes, the statement $a \leftarrow b$ denotes the assignment of value b to the variable a. We denote random sampling of a variable x from a set $\mathcal{Y}$ by $x \leftarrow\!\!\$\, \mathcal{Y}$ and use // to indicate a comment.

Chapter 2

How Age Works

Age being a command line encryption tool allows a user to encrypt arbitrary files for one or multiple parties, called recipients. Each recipient is then able to decrypt the file to get access to the original plaintext. Each Age-file has a header, storing data for the selected recipients. In Age, a 128 bit master key, called file key, is used, from which all further keys for authentication of the header and encryption of the plaintext are derived. The user has no control over the file key used, instead it is selected randomly in the process of creating the Age-file.

The header of an Age-file contains the encryption of the file key for the respective recipient(s). Depending on which recipient type is used, from which there exist four in total, the key is encrypted in a different way. This process is called wrapping and the final wrapped file key, including a recipient type identifier, is called a recipient stanza.

For derivation of the different keys, the key derivation function HKDF [14] is used. The derivation of the actual encryption key involves a randomly sampled nonce, which is placed inside an Age-file directly after the header, such that the correct key can be derived in the decryption process.

The final part of an Age-file consists of the actual encryption of the payload data. The Authenticated Encryption with Associated Data scheme ChaCha20-Poly1305 [20] is used for this process.

We will now present a detailed overview of the key derivations in Age, the file format and its various components, and the use of ChaCha20-Poly1305 for encryption and decryption of the payload. The observations are based on the meanwhile superseded as well as the new Age documentation [24, 1] and the actual source code [2].

2.1 Key Derivations

Recall, the key used to authenticate the header is derived from the file key using HKDF [14], using SHA256 [11] as the hash function. We will call the key used in this process MAC-Key and it is derived as follows:

$$\text{MAC-K}{\scriptstyle\text{EY}} = \text{HKDF}(\text{SHA256}, \text{ikm} = \mathit{file\,key}, \text{salt} = \mathit{none}, \text{info} = \text{"payload"})\,.$$

The salt value being set to 'none' results in an all-zero byte-string being used as specified in the HKDF RFC [14].

Similarly, the key provided to ChaCha20-Poly1305 for payload encryption, which we will call payload key, is derived from the file key using the sampled nonce as follows:

$$\mathit{payload\,key} = \text{HKDF}(\text{SHA256}, \text{ikm} = \mathit{file\,key}, \text{salt} = \mathit{nonce}, \text{info} = \text{"payload"})\,.$$

The key derivations allow Age to achieve proper key separation between payload encryption and header authentication.

2.2 File Format

An Age-file consists of a triple of header, nonce, and encrypted payload. In this section we focus on the format of the file header, which contains the so called recipient stanzas. The nonce, as already mentioned, is sampled randomly and is 16 bytes long. The encrypted payload consists of the ChaCha20-Poly1305 ciphertext generated using the payload key by encrypting the desired plaintext. We will discuss the encryption process in more detail in later sections. See Section 2.3 for a high level overview of ChaCha20-Poly1305 and Section 2.4 for an overview of the encryption process in Age making use of ChaCha20-Poly1305.

An abstracted illustration of an Age-file can be seen in Figure 2.1.

2.2.1 Header Format

The header starts with a fixed string "*age-encryption.org/*" followed by a string indicating the version used. The current version of Age at the point of writing this report is version "v1".

The rest of the header consists of one or more recipient stanzas. Each stanza starts with the indicator `"->"` at the start of the line. It is followed by a string identifying the stanza type, which is one of the four "X25519", "scrypt", "ssh-rsa", or "ssh-ed25519". Following the identifier there is stanza-dependent information, like public-key components, in base64 encoded format according to RFC 4648 [12]. Finally, each stanza is concluded with the canonical base64 encoding of the encrypted file key.

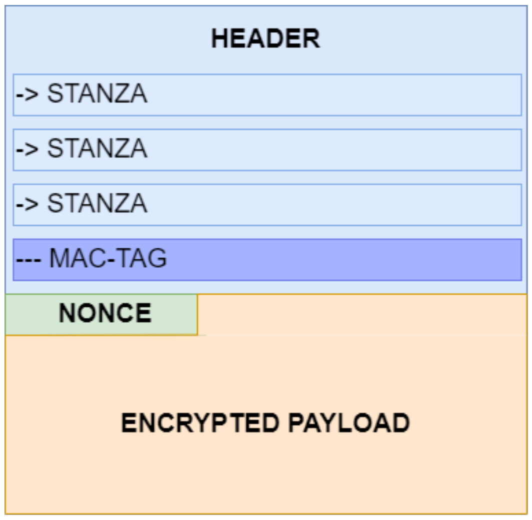


Figure 2.1: Abstracted illustration of an Age-file consisting of a header, containing different recipient stanzas, a nonce, and the encrypted payload.

```
age-encryption.org/v1
-> X25519 yOonyHn7BtHqEnrM7GhSC9r1iXvA3+Xv5LP1O6R/M0g
EULvz7nIydiOafyareYPqsgyvIztiIu/9jj4Gcv8ed4
-> X25519 yOonyHn7BtHqEnrM7GhSC9r1iXvA3+Xv5LP1O6R/M0g
2kfiNEfWKXX0StNwZRTi7OGeEqjOTDmenh7jZWXL1Wk
--- JTv1JdUpYSrX5WvJwyCvbdlBNCk6hGTtfiAoBRZfvvQ
```

Figure 2.2: Illustration of an exmaple Age-file-header containing recipient stanzas for two different X25519 recipients.

Note that the encryption (or wrapping) of the file key is dependent on the stanza type. In the following subsection we discuss each of them in more detail.

The last part of the header consists of an authentication tag, computed over the entire header using HMAC [13] making use of the MAC-key derived from the original file key. The start of the MAC-tag is indicated by the string "---", followed by the base64 encoding of the computed tag. Note that everything up until, and including, the indication string "---" is authenticated. Therefore, no recipient can be added nor removed from the header without recomputation of the header MAC.

An illustration of an Age-file-header containing two X25519 recipient stanzans up until the corresponding MAC-tag is shown in Figure 2.2.

2.2.2 Recipient Stanzas

Recall, for each recipient the file key is wrapped in one recipient stanza. Different types of stanzas can be present in the same file, except for scrypt recipients. As in that case the file key is encrypted using a key derived from a password [21], a stronger notion of authentication is assumed. Therefore, if an scrypt recipient is present, it must be the only recipient in the Age file. Trying to encrypt a file for multiple recipients, one of which being an scrypt recipient, will fail.

This restriction makes sense, as every recipient, for which a file is encrypted, can tamper with the file. Assuming an adversary Eve is together with Bob part of the recipient list of a file Alice encrypted using Age, then, if Eve can intercept the file sent to Bob, she can decrypt it, change its content, encrypt it again, attach the header of the original file and forward it to Bob, who will successfully decrypt the file and see the modified plaintext chosen by Eve.[1]

We will use the following functions in the description of the recipient stanzas as they are defined in the Age documentation [24, 1].

- ENCODE(DATA) performes canonical base64 encoding without padding as defined in RFC 4648 [12].
- ENCRYPT[KEY](PTXT) encrypts the plaintext under the provided key using ChaCha20-Poly1305 encryption as defined in RFC 7539 [20].
- `X25519(secret, point)` as used for ECDH (Elliptic Curve Diffie-Hellman) from RFC 7748 [16].
- `HKDF[salt, label](key)` returns 32 bytes of HKDF with SHA-256 as defined in RFC 5869 [14].
- `HMAC[key](message)` computes the HMAC of the given message under the provided key using SHA-256.
- `scrypt[salt, N](password)` outputs 32 bytes of scrypt using $r = 8$ and $P = 1$ as it is defined in RFC 7914 [21].
- `RSAES-OAEP[key label](plaintext)` using SHA-256 and MFG1, from RFC 8017 [19].
- `random`(n) outputs a string of n random bytes.

X25519 Recipient Stanza

The general structure of an X25519 recipient stanza is illustrated in Figure 2.3.

[1]Note the same can be done by any party knowing the password used for an scrypt recipient.

```
-> X25519 ENCODE(X25519(ephemeral secret, basepoint))
ENCRYPT[HKDF[salt, label](X25519(ephemeral secret, public
key))](file key)
```

Figure 2.3: Generall structure of an X25519 recipient stanza as used in an Age-file-header.

```
-> scrypt ENCODE(salt) log2(N)
ENCRYPT[scrypt["age-encryption.org/v1/scrypt" + salt,
N](password)](file key)
```

Figure 2.4: Generall structure of an scrypt recipient stanza as used in an Age-file-header.

```
-> ssh-rsa ENCODE(SHA256(SSH key)[:4])
RSAES-OAEP[public key, "age-encryption.org/v1/ssh-rsa"](file key)
```

Figure 2.5: Generall structure of an ssh-rsa recipient stanza as used in an Age-file-header.

The *ephermal secret* is set to be a random 32 byte value and must be new for every new file key. For the HKDF used, the *salt* is defined to be

```
X25519(ephemeral secret, basepoint) || public key,
```

and the *label* is specified to be the string "age-encryption.org/v1/X25519".

Scrypt Recipient Stanza

Figure 2.4 illustrates the general format of an scrypt recipient stanza.

N is defined as the scrypt cost parameter in decimal form, salt is a random 16 byte string. A new salt needs to be generated for every new file key.

SSH-RSA Recipient Stanza

An ssh-rsa recipient stores the encoding of the SHA256 hash of a SSH public key according to RFC 8332 [7], called SSH key. Thereby, the public key must be at least 2048 bits long. Figure 2.5 illustrates the general format of an ssh-rsa recipient stanza.

SSH-ED25519 Recipient Stanza

Illustrated in Figure 2.6 is the general structure of an ssh-ed25519 recipient stanza as used in Age.

The `tag` corresponds to the base64 encoding of the hashed ssh public key as done for ssh-rsa recipients, e.g.

```
tag = ENCODE(SHA256(SSH key)[:4]).
```

```
-> ssh-ed25519 tag ENCODE(X25519(ephemeral secret, basepoint))
ENCRYPT[HKDF[salt, label](X25519(ephemeral secret, tweaked
key))](file key)
```

Figure 2.6: Generall structure of an ssh-ed25519 recipient stanza as used in an Age-file-header.

For every new file key, a new `ephemeral secret` must be sampled, which is a random 32 byte string. The `salt` used in HKDF corresponds to

`X25519(ephemeral secret, basepoint)` || `converted key`,

whereby `converted key` is the ED25519 public key converted to the Montgomery curve.[2] The `label` corresponds to the string

"`age-encryption.org/v1/ssh-ed25519`"

and the `SSH key` is the binary representation of the ssh public key. The `tweaked key` is defined as `X25519(tweak, converted key)`, whereby `tweak` is computed as

`HKDF[SSH key, "age-encryption.org/v1/ssh-ed25519"]("")`.

2.2.3 File Key Restrictions

Recall, the file key wrapped in the recipient stanzas is a randomly sampled 128 bit value. However, Age does not check the length of the file key for all recipient types. In fact, only for X25519 and scrypt recipients is a length check on the unwrapped file key performed in the decryption process of the Age-file. This is especially interesting as Age uses HKDF for key derivations and HMAC for header authentication. By having control over the key length, it is easy to create collisions in HMAC (see Chapter 5). We will later on come back to this property when we try to break the robustness of Age.

2.3 ChaCha20-Poly1305

For payload encryption Age uses the AEAD scheme ChaCha20-Poly1305 [20]. It is a combination of the ChaCha20 stream cipher and the polynomial MAC Poly1305.

In this section we present a high level overview of the scheme to give relevant background on the payload encryption done in Age.

[2]The procedure of using ED25519 signing keys for encryption is described by the Age author, Filippo Valsorda, in one of his blog posts [25] in more detail.

2.3.1 ChaCha20

ChaCha20 is a 20-round stream cipher. Compared to other ciphers, such as the block cipher AES [18], it is faster and easier to implement in regards to power or timing side-channel attacks [6], as the round function involved takes constant time to execute for each round, independent of the provided input.

The cipher takes on input a 32-byte key k and an arbitrary byte sequence, $ptxt$, representing the plaintext to be encrypted. Using the initial key, a 12-byte nonce, a 4-byte block counter and a fixed 16-byte constant value, ChaCha20 computes 20 iterations of its round function to expand k to a new 64-byte key. The result is then used to encrypt one 64-byte block of the given plaintext in one-time-pad fashion, by xoring the ptxt bytes with the key bytes. After one block of data has been encrypted, the block counter is incremented by one and the process is repeated for the next block until the entire plaintext is encrypted.

Note that ChaCha20 does not perform any padding on the provided input plaintext, i.e. the outputted ciphertext does leak the length of the underlying plaintext.

2.3.2 Poly1305

Poly1305 is a polynomial message authentication code. It takes on input a 32-byte key k and a message m, interpreted as a sequence of bytes, of arbitrary length and computes a 16-byte tag, which is used to authenticate the message.

When used in ChaCha20-Poly1305 AEAD, the key used for tag computation is normally generated using ChaCha20. To be precise, the key used for the MAC computation corresponds to the first 32 bytes of the key-stream generated by ChaCha20. The next 32 bytes of the key-stream are thrown away and plaintext encryption starts using the next 64 bytes, now generated using a block counter equal to 1. This mechanism is used in the ChaCha20-Poly1305 implementation used in Age and is formally defined in RFC 7539 [20, Appendix A.4].

Regardless of the key generation method used, the key is partitioned into two 16-byte keys r and s. Normally r and s correspond to the first and second 16 bytes of the provided key, such that $r||s = k$. Before being used, r needs to be preprocessed, also called clamped. Treating r as a 16-octet little-endian number, the following conditions have to be fulfilled:

- $r[3]$, $r[7]$, $r[11]$, and $r[15]$ are required to be smaller than 16 (top four bits need to be 0)

- $r[4]$, $r[8]$, and $r[12]$ are required to be divisible by 4 (bottom two bits need to be 0)

This can be achieved by computing an AND operation between r and an appropriate mask as follows:

$$\textsc{clamp}(r) = r \text{ AND } 0x0ffffffc0ffffffc0ffffffc0fffffff\,.$$

The provided message than is processed in 16-byte blocks, where the last block might be shorter. Each is interpreted as a little-endian number and one bit is added beyond the number of octets, i.e. for a full 16-byte block, this corresponds to adding 2^{128} to the number. For the last block the bit added beyond the number of octets can correspond to any power of two that is evenly divisible by 8, depending on the length of the block. Note that this is always possible as we assume a message to be non-empty. Therefore, the minimum number added is 2^8, in the case were the entire message fits into one octet, i.e. is not more than one byte long. The resulting number is added to an accumulator variable, initialized to be 0 at the start of the computation. The accumulator is then multiplied with r and the result is taken modulo P for $P = 2^{130} - 5$.

After performing this computation for the entire message, the second part of the key, s, is added to the result. The final outputted tag then consists of the 128 least significant bits of the result interpreted as a little-endian integer. Taking the 128 least significant bits is equivalent to performing a mod 2^{128} operation.

In Figure 2.7 we show a pseudocode implementation of the Poly1305 tag computation.

In ChaCha20-Poly1305 the message authenticated using Poly1305 consists of the associated data, AD, the ChaCha20 ciphertext C, and the lengths of AD and C interpreted as eight byte little-endian integers respectively. The final output corresponds to the concatenation of the ciphertext and the computed Poly1305 tag.

2.4 Encryption in Age

Having covered the relevant background, we are now able to present a high level overview of the encryption procedure in Age.

The encryption function takes as input an input file, i.e. the data to be encrypted, an output destination, another file or the command terminal if none is specified, and a list of recipients. The output is either the encrypted data, if no output file is specified or nothing, i.e. the encrypted plaintext is written into the specified output file.

```
Poly1305(K, M)
1 :  $(r, s) \leftarrow K$
2 :  $r \leftarrow \textsc{clamp}(r)$
3 :  $acc \leftarrow 0$
4 :  $P \leftarrow 2^{130} - 5$
5 :  // Iterate over the message in 16-byte blocks
6 :  for $block$ in $M$ do
7 :      $acc \leftarrow acc + block$
8 :      $acc \leftarrow (r * acc) \bmod P$
9 :  endfor
10 : $acc \leftarrow acc + s$
11 : return $\textsc{tagBytes}(acc)$
```

Figure 2.7: Pseudocode implementation of the Poly1305 algorithm. Given a 32-byte key K and an arbitrary message M, it outputs a 16-byte tag. The function $\textsc{clamp}(\cdot)$ performs the necessary precomputation on r, $\textsc{tagBytes}(\cdot)$ outputs the first 128 least significant bits of the number given to it as input, interpreted as a little-endian integer. For all computations, the key and message bytes are interpreted as little-endian integers.

Recall, the user has no control over the file key used to encrypt a file, as it is chosen at random at the start of the encryption procedure. After sampling a file key, the file header is generated. This process involves the wrapping of the file key for each recipient specified in the given recipient list (as described in Section 2.2.2). This process can be costly, as it involves asymmetric cryptography. Also recall that if an scrypt recipient is present, it must be the only one. If this restriction is not met, the program returns an error and terminates.

As soon as the complete header is generated, it is authenticated using HMAC, which uses a MAC-key derived from the file key using HKDF (as described in Section 2.1). The resulting tag then is append to the header and the result is written to the specified output.

In a next step, the function samples a random 16-byte nonce value and uses it to derive the payload key used for ChaCha20-Poly1305. Using the derived key the payload is encrypted in chunks of 2^{16} bytes. Thereby, each chunk uses a different ChaCha20-Poly1305 instance, i.e. they use the same key but a different AEAD nonce, which is defined to be a counter starting at zero and which is incremented by one for each new chunk being encrypted. This also means that each chunk is authenticated independently from all other chunks. To prevent truncation attacks, the eight least significant bits of the AEAD nonce used for ChaCha20-Poly1305 are set to be equal to $0x00$ for all chunks except the last one, for which they are set to $0x01$. Therefore, the AEAD nonce has the following format: $\textsc{ctr}||0x0b$ for $b \in \{0,1\}$. This design implicitly prevents truncation attacks, as if the last chunk would be missing, the decryption procedure will use the incorrect AEAD nonce, ending in $0x01$ instead of $0x00$, to decrypt the last chunk of the truncated payload, which results in a decryption failure.[3]

Note that truncations smaller than the size of a chunk are prevented by the design of ChaCha20-Poly1305, as in such a case, the Poly1305 verification of the truncated chunk will fail by design.

The first encrypted payload chunk is prepended by the nonce used to derive the payload key, such that the same key can be derived in the decryption process. Finally, each encrypted chunk is written to the designated output.

Note that each chunk of 2^{16} bytes is immediately written to the designated output file once it is encrypted. As a result, if an error occurs during encryption of chunk i, all chunks j for $j < i$ will already be written to the output file. Using the command line, a user will be notified that the encryption of a chunk has failed. However, if a user ignores the error message or an application using Age does not properly handle the error, it could be possible that a user believes that the output file contains the encryption of the entire payload although this might not be the case.

Figure 2.8 shows a pseudocode implementation of the Age encryption function.

2.5 Decryption in Age

The Age decryption function reverses the steps performed during encryption. It takes as input an input file, now assumed to be an Age-file generated using the encryption function, an optional output destination, and a list of identities. If no output destination is specified, the output is written directly into the terminal. An identity is the complement to a recipient, i.e. only using the identity matching to a recipient the corresponding recipient stanza can be unwrapped to retrieve the file key. Typically, an identity contains the private key components matching the public key components of the respective recipient.

In a first step, the function parses the input file into the file header and the payload. In this step, the nonce, which lies between the header and the encrypted data is considered to be part of the payload. The function then reads out the stanzas from the header and tries to find a matching stanza

[3]To be precise, the Poly1305 authentication of the last chunk for the truncated payload will fail.

```
ENCRYPTION(input, out, recipients)

// Generation of the Age-file header
$filekey \leftarrow\!\!\$ \{0,1\}^{128}$
if $\textsc{onlyOneScrypt}(recipients) = False$ do
   return $\bot$
endif
$header \leftarrow "age-encryption.org/v1"$
for $recipient$ in $recipients$ do
   $stanza \leftarrow \textsc{Wrap}(filekey, recipient)$
   $header \leftarrow header||stanza$
endfor
$MACKey \leftarrow \text{HKDF}(SHA256, fileKey, none, "header")$
$tag \leftarrow \text{HMAC}(MACKey, header)$
$header \leftarrow header||tag$
$nonce \leftarrow\!\!\$ \{0,1\}^{128}$
$out.\textsc{write}(header||nonce)$
$payloadkey \leftarrow \text{HKDF}(SHA256, filekey, nonce, "payload")$
$chachaCtr \leftarrow 0$   // 11 bytes
// Encryption of Payload
for $chunk$ in $input$ do   // Iterate over input in $2^{16}$ byte chunks
   $chachaNonce \leftarrow chachaCtr||0x00$
   if $chunk$ is last chunk do   // Check if chunk is the last chunk to be encrypted
      $chachaNonce \leftarrow chachaCtr||0x01$
   endif
   $c \leftarrow \textsc{ChaCha20-Poly1305Enc}(payloadkey, chachaNonce, chunk)$
   $out.\textsc{write}(c)$
   $chachaCtr \leftarrow chachaCtr + 1$
endfor
```

Figure 2.8: Pseudocode, describing the encryption process in Age on a high-level, using ChaCha20-Poly1305. The function .WRITE(·) writes the given byte sequence to the designated file. The WRAP(·,·) function wraps the file key for the corresponding recipient as defined in Section 2.2.2. The function ONLYONESCRYPT(·) returns true, iff when an scrypt recipient is present it is the only one in the provided recipient list. Using ⊥, we denote a general error.

for one of the identities provided. If no match can be found, the program terminates with an error.

Once a match has been found, the info provided from the identity is used to unwrap the file key from the stanza.

Using the same methods as for encryption, the MAC-key is derived from the file key. Using the MAC-key, the MAC-tag authenticating the header is verified. If verification fails, the program terminates with an error.

After successful MAC verification, the nonce, used to derive the payload key, is read out as the first 16 bytes of the payload. Using it, the function derives the payload key and starts decrypting the encrypted payload using ChaCha20-Poly1305.

As for encryption, decryption operates in chunks, now of size $2^{16} + 16$ bytes to account for the additional 16 bytes taken up by the Poly1305 tag. Each chunk is decrypted using the same key but a different nonce. As for encryption, the nonce consists of an 11-byte counter and an extra byte, indicating if the current chunk is the last payload-chunk. If decryption of a chunk is successful, it is written directly to the specified output. If an error occurs during the decryption process, the function returns an error and terminates.

Note that if an error occurs whilst decrypting chunk i, for example tag verification fails, the decryption of all chunks j with $j < i$ will already be written to the output file.

As it is the case during encryption, also during decryption a user is informed with an error being displayed in the command line if decryption or authentication of a chunk fails. If a user ignores the error or an application using Age does not properly handle it, truncation of Age-files is possible. Especially, if an application does not correctly handle a decryption error, a user could decrypt a truncated Age-file and believe the resulting output corresponds to the non-truncated payload.

Recall that the last payload-chunk is encrypted using a nonce ending in $0x01$ to prevent truncation attacks. As there is no explicit length field in an Age-file, the decryption function has no way of knowing if the current chunk is the last one to be decrypted. To be precise, if the current chunk is not full, i.e. less than $2^{16} + 16$ bytes long, the function assumes it is the last chunk and directly tries decryption using the nonce ending in $0x01$. If however the last chunk is a full chunk, i.e. exactly $2^{16} + 16$ bytes long, decryption fails, as the wrong nonce, ending in $0x00$ will be used for its decryption.

To account for this problem, Age retries decryption for the first chunk, for which decryption returns an error, with the nonce ending in $0x01$, assuming it must be the last chunk of payload. Therefore, in the case the last chunk is a full chunk, Age performs one additional decryption operation of a $2^{16} + 16$ byte chunk than would be necessary.[4]

For ease of understanding, we abstracted this functionality into a black-box *"is last chunk"* if-statement in the pseudocode describing the decryption procedure in Age shown in Figure 2.9.

[4]Note that after trying decryption with a nonce ending in $0x01$, i.e. assuming to decrypt the last chunk, no additional decryption will be performed no matter if the attempted decryption succeeded.

```
DECRYPTION(input, out, identities)

// Parse the Age-file header
$hdr, payload \leftarrow$ ParseInput($input$)   // separate header from payload
$stanzas, MAC \leftarrow$ ParseHeader($header$)   // read stanzas, MAC-tag from header
$filekey \leftarrow \perp$
for $identity$ in $identities$ do
  $filekey \leftarrow$ Unwrap($stanzas, identity$)
endfor
if $filekey = \perp$ do   // No matching identity found
  return $\perp$
endif
$MACKey \leftarrow \text{HKDF}(SHA256, fileKey, none, "header")$
$tag \leftarrow \text{HMAC}(MACKey, header)$
if $tag \neq MAC$ do   // check validity of MAC
  return $\perp$
endif
$nonce||payload \leftarrow payload$   // $|nonce| = 12$ bytes
$payloadkey \leftarrow \text{HKDF}(SHA256, filekey, nonce, "payload")$
$chachaCtr \leftarrow 0$   // 11 bytes
// Decryption of Payload
for $chunk$ in $payload$ do   // Iterate over input in $2^{16} + 16$ byte chunks
  $chachaNonce \leftarrow chachaCtr||0x00$
  if $chunk$ is last chunk do   // Check if chunk is the last chunk to be decrypted
    $chachaNonce \leftarrow chachaCtr||0x01$
  endif
  $p \leftarrow$ ChaCha20-Poly1305Dec($payloadkey, chachaNonce, chunk$)
  if $p = \perp$ do   // Check for decryption error
    return $\perp$
  endif
  $out$.write($p$)
  $chachaCtr \leftarrow chachaCtr + 1$
endfor
```

Figure 2.9: Pseudocode, describing the decryption process in Age on a high-level, using ChaCha20-Poly1305. The ParseInput$(\cdot)$ function parses the given input into the header and payload section. The ParseHeader$(\cdot)$ function splits a given Age-file-header into the individual stanzas and the MAC-tag. The function .write$(\cdot)$ writes the given byte sequence to the designated file. The Unwarp$(\cdot, \cdot)$ function tries to unwrap the given stanzas using the provided identity and returns the unwrapped file key on success or $\perp$. Using $\perp$, we denote a general error.

Chapter 3

Collisions in ChaCha20-Poly1305

Several widely used authenticated encryption (AE) schemes, such as AES-GCM [22], AES-GCM-SIV [10], ChaCha20-Poly1305 and OCB3 [15] suffer from a lack of key commitment. This has been shown by Len et al. [17] and Albertini et al. [5]. Not committing to a key means that it is easy to come up with a ciphertext C, which decrypts correctly under two different keys K_1 and K_2.

In this chapter we will show in detail how one can create such a ciphertext for ChaCha20-Poly1305. We will use this later on in an attack against Age, where this property allows us to encrypt a polyglot file (see Chapter 4) with two different keys such that using either one of the keys the ciphertext is decrypted to one of the two valid files making up the polyglot file.

Our computation is inspired by the general construction for polynomial MAC schemes given by Albertini et al. [5, Section 3.4.1].

3.1 High-Level Overview of Poly1305

The one-time authenticator Poly1305 was designed by Bernstein and is formally specified in RFC 7539 [20]. It takes as input a 256-bit key, K, and a message, M, of arbitrary length and outputs a 128-bit tag authenticating the message. For simplicity we assume that all the messages M have a length, which is a multiple of 16 bytes.[1] This allows us to simplify the computation. Additionally, in Age there is never a case in which the Poly1305 algorithm will receive a message input which is not a multiple of 16 bytes. This is due to the way in which ChaCha20-Poly1305 is defined. The additional data and ChaCha20 cipherext each are padded using zeros until their lengths are a

[1]If a block is shorter, it will be padded using zeros after one bit is added beyond the number of octets. I.e. in Step 4. below, $M'[j] = M[j] + 2^x$, where $x \in \{1 \leq x < 128, x \bmod 8 = 0\}$. We refer to the RFC for more details.

multiple of 16 bytes. An additional 16-byte block is added, which contains the encoding of the lengths of the additional data and the ChaCha20 ciphertext (see Section 3.4 for more details on the structure of the authenticated data).

The tag is computed using the following steps:

1. Generate two 128-bit keys r and s from K
2. Clamp r such that it has the appropriate format:[2]
$$r = r \text{ AND } 0x0ffffffc0ffffffc0ffffffc0fffffff$$
3. Divide the message into 16-byte blocks: $M[0], \ldots, M[n-1]$
4. Compute the tag τ' as follows:
$$\tau' = s + \left[\sum_{i=0}^{n-1} M'[i] \cdot r^{n-i}\right] \text{ MOD } 2^{130} - 5$$
where $M'[j] = M[j] + 2^{128}$
5. Compute final 128-bit tag τ:
$$\tau = \tau' \text{ MOD } 2^{128}$$

Each message block is interpreted as a little-endian number. In all of the following we use $P = 2^{130} - 5$.

3.2 Creating Collisions in ChaCha20-Poly1305

To guarantee that a ciphertext C decrypts correctly under two different keys K_1 and K_2 in ChaCha20-Poly1305, we have to ensure that the authenticator τ, computed over C, is the same using either of the keys K_1 or K_2. This is required, as decryption will fail if verification of the authenticity of the ciphertext does not succeed. Assuming the Poly1305 instance derives keys r_1, s_1 and r_2, s_2 from the keys K_1, K_2 respectively, and $C = C[0], \ldots, C[n-1]$, $|C[i]| = 16$ bytes, this gives us the following equation which needs to be satisfied:

$$\begin{aligned} &\left[s_1 + \left[\sum_{i=0}^{n-1} C'[i] \cdot r_1^{n-i}\right] \text{ MOD } P\right] \text{ MOD } 2^{128} \\ = &\left[s_2 + \left[\sum_{i=0}^{n-1} C'[i] \cdot r_2^{n-i}\right] \text{ MOD } P\right] \text{ MOD } 2^{128}, \end{aligned} \tag{3.1}$$

[2]We will not discuss the reasons behind the clamping, as it does not affect the computations we will do. We refer the interested reader to the RFC or Bernstein's original article to learn more about this design decision.

where $C'[j] = C[j] + 2^{128}$.

We can now fix all but one block of the ciphertext, $C[t]$. This is not a problem in the context of creating meaningful plaintexts. Most file formats allow for appended data, can start at different offsets or have special blocks, which will be ignored by the parser. As such blocks do not affect the representation of the file, modification of blocks in those positions do not impose a problem, as although they will decrypt to random strings, the parser will ignore them (see Chapter 4 for more details).

Fixing ciphertext block $C[t]$ gives us the following equation:

$$\begin{aligned}&\left[s_1 + \left[C'[t] \cdot r_1^{n-t} + \sum_{i=0, i \neq t}^{n-1} C'[i] \cdot r_1^{n-i}\right] \bmod P\right] \bmod 2^{128} \\ = &\left[s_2 + \left[C'[t] \cdot r_2^{n-t} + \sum_{i=0, i \neq t}^{n-1} C'[i] \cdot r_2^{n-i}\right] \bmod P\right] \bmod 2^{128}\end{aligned} \tag{3.2}$$

As all the variables above are known, we would like to rearrange the equations to find the value of the target ciphertext block $C'[t]$. To make it easier to work with the equations, we drop the $\bmod 2^{128}$ reduction and consider the equations to hold over the integers. As a result we may lose some solutions. However, if the equations hold over the integers, they are guaranteed to hold over the original equations including the $\bmod 2^{128}$ reduction.

$$\begin{aligned}&\left[C'[t] \cdot r_1^{n-t} - C'[t] \cdot r_2^{n-t}\right] \bmod P \\ =&s_2 - s_1 + \left[\sum_{i=0}^{n-1} C'[i] \cdot r_2^{n-i} - C'[i] \cdot r_1^{n-i}\right] \bmod P\end{aligned} \tag{3.3}$$

$$\begin{aligned}&\left[C'[t] \cdot (r_1^{n-t} - r_2^{n-t})\right] \bmod P \\ =&s_2 - s_1 + \left[\sum_{i=0}^{n-1} C'[i] \cdot r_2^{n-i} - C'[i] \cdot r_1^{n-i}\right] \bmod P\end{aligned} \tag{3.4}$$

To extract $C'[t]$ we need to get rid of the term $(r_1^{n-t} - r_2^{n-t})$. This cannot easily be done, as we cannot multiply both sides of the equation with the modulo P inverse of $(r_1^{n-t} - r_2^{n-t})$. To solve this problem, we consider a reduction $\bmod P$ for both sides of the equation, to arrive at:

$$\begin{aligned}&\left[C'[t] \cdot (r_1^{n-t} - r_2^{n-t})\right] \bmod P \\ = &\left[s_2 - s_1 + \sum_{i=0}^{n-1} C'[i] \cdot r_2^{n-i} - C'[i] \cdot r_1^{n-i}\right] \bmod P\end{aligned} \tag{3.5}$$

The resulting Equation 3.5 is not 'correct' in all cases. With a small probability, the right-hand-side of Equation 3.5 will either be greater than $P - 1$ or smaller than 0. In both cases the resulting ciphertext block $C[t]$ will not correct the tag correctly and the resulting tags computed using either of the keys K_1 or K_2 will not collide.

In Figure 6.1 we analysed our proof-of-concept implementation and found that in 100000 runs we encountered the right-hand-side of Equation 3.5 being negative 4518 times and 4549 times it was greater than $P - 1$. This resulted in a total of 2.5% of all retry attempts being caused by our incorrect transformation.

As only a small percentage of all retry attempts are caused by our incorrect transformation, we accepted the additional overhead and extract $C'[t]$ from Equations 3.5 as follows:

$$\begin{aligned} &C'[t] \bmod P \\ &= \left[\left[s_2 - s_1 + \sum_{i=0}^{n-1} C'[i] \cdot r_2^{n-i} - C'[i] \cdot r_1^{n-i}\right] \cdot (r_1^{n-t} - r_2^{n-t})^{-1}\right] \bmod P \end{aligned} \tag{3.6}$$

Equation 3.6 fully determines $C'[t]$. But, as in ChaCha20-Poly1305 $C'[t] = C[t] + 2^{128}$, we need to additionally check if the new ciphertext block we computed is in the range $[2^{128}, 2^{129} - 1]$. If this is not the case, we need to try again, using either a new target block or new keys. Our analysis shown in Section 6.6 shows that an average of roughly 3 retries are required due to this. Otherwise, the new ciphertext $\hat{C} = C[0], \dots C[t-1], C'[t] - 2^{128}, C[t+1], \dots, C[n-1]$ will result in the same tag τ being computed using either of the keys K_1 or K_2.

Note that the original ciphertext block at position t will be replaced by the newly computed block $C'[t] - 2^{128}$ and therefore the decryption of this block, under either of the keys, will be random.

3.3 Pseudocode Implementation

In Figure 3.1 we present a pseudocode implementation, which given on input a ciphertext C, a multiple of 16 bytes long, two 32-byte keys K_1 and K_2, and an integer t, indicating the ciphertext block to be modified, outputs a new ciphrtext $\hat{C}$, such that the Poly1305-tag computed over $\hat{C}$ is the same for both keys K_1 and K_2 and $\hat{C}$ is identical to C except for block t.

If the result of the computation does not fulfil the requirement of being in range $[2^{128}, 2^{129} - 1]$ or the computed block does not succeed in correcting the tag due to an incorrect calculation coming from our mod P transformation, the function returns an error denoted as $\perp$.

```
Collide(C, $K_1$, $K_2$, t)
if $|C| \bmod 16 \neq 0$ do
    return $\perp$
endif
$N \leftarrow |C|/16$
if $N = 1$ or $t < 0$ or $t > N - 1$ do
    return $\perp$
endif
$P \leftarrow 2^{130} - 5$
$r_1, s_1 \leftarrow$ GeneratePoly1305Keys($K_1$)
$r_2, s_2 \leftarrow$ GeneratePoly1305Keys($K_2$)
$counter \leftarrow 0$
$sum_1, sum_2 \leftarrow 0$
$inv \leftarrow (r_1^{N-t} - r_2^{N-t})^{-1_P}$
// Iterate over ciphertext in 16 byte blocks
for $block$ in $C$ do
    if $counter = t$ do    // Skip block to be modified
        $counter \leftarrow counter + 1$
        continue
    endif
    $c \leftarrow$ LEBytesToNum($block$)
    $c \leftarrow c + 2^{128}$    // Equivalent to appending 0x01 to a full block
    $sum_1 \leftarrow sum_1 + c \cdot r_1^{N-counter} \bmod P$
    $sum_2 \leftarrow sum_2 + c \cdot r_2^{N-counter} \bmod P$
    $counter \leftarrow counter + 1$
endfor
$temp \leftarrow (s_2 - s_1 + sum_2 - sum_1) \cdot inv$
$newCt \leftarrow temp$
$newCt \leftarrow newCt \bmod P$
// Check if result is in range $[2^{128}, 2^{129} - 1]$ and correct
if InRange($newCt$) and $0 \leq temp < P$ do
    $newC \leftarrow$ ReconstructNewC($C, newCt, t$)
    return $newC$
endif
return $\perp$
```

Figure 3.1: Pseudocode given on input a ciphertext C, two keys K_1 and K_2 and an integer t, outputs a new ciphertext $newC$ identical to C except for block t such that the Poly1305 tags computed over $newC$ are identical for both keys K_1 and K_2. The notation x^{-1_P} denotes the inverse of x mod P. The function GeneratePoly1305Keys($\cdot$) computes the two Poly1305-key-components r and s given a 32-byte key. LEBytesToNum($\cdot$) returns the little-endian integer interpretation of a given byte sequence, InRange($\cdot$) returns True, if the given integer is in the range $[2^{128}, 2^{129} - 1]$, and ReconstructNewC($\cdot,\cdot,\cdot$) generates the new ciphertext by replacing block t in C with $newCt - 2^{128}$.

3.4 Remarks

ChaCha20-Poly1305 being an authenticated encryption scheme also allows to authenticate additional data, which will not be encrypted. We ignored this fact in the description and construction of our code as in the use-case of Age command line encryption, no associated data will be used. However, our code can easily be adapted to also allow the presence of associated data.

As described in RFC 7539 [20, Section 2.8], the data authenticated in ChaCha20-Poly1305 is constructed as follows:

$$macData = aad||pad16(aad)||ctxt||pad16(ciphertext)||len(aad)||len(ctxt)\,,$$

where pad16($\cdot$) is a function, which returns between zero and fifteen 0-bytes, such that $len(x||pad16(x)) \bmod 16 = 0$, where '||' denotes the concatenation of bytes.

Replacing the input C of our Collide($\cdot$,$\cdot$,$\cdot$,$\cdot$) function with *macData* allows the computation of a collision even when associated data is present.

Note that the last authenticated block consists of the lengths of associated data and the ciphertext, both interpreted as eight byte little-endian integers. In consequence, the last 16-byte block of input to the Poly1305 algorithm can not be modified to create collisions, as doing so will change the "authenticated lengths" and would therefore cause authentication to fail once trying to decrypt the new ciphertext with overwhelmingly high probability.

Recall that to create a collision we need at least one block of freedom in either the ciphertext or associated data, i.e. one block of 16 bytes, which can be arbitrarily changed.

Using the same high-level technique we showed here, it is also possible to create ciphertexts which decrypt correctly, i.e. without an error, for more than two keys. This however requires multiple blocks of freedom for tag correction. We refer to Len et al. [17] for more details on such an approach.

Chapter 4

Polyglot Files

In Chapter 3 we described how an attacker can create a ciphertext C which decrypts correctly under two different keys K_1 and K_2 using ChaCha20-Poly1305. We will call ciphertexts with this property *ambiguous* ciphertexts.

The pseudocode described in Figure 3.1 shows how we can create an ambiguous ciphertext from a given ciphertext and two keys. However, we did not address the problem of ensuring that the ciphertext decrypts to meaningful plaintexts under both keys. Crafting ambiguous ciphertexts, whilst making certain that the decryption under both keys is meaningful, requires controlling the bits in the resulting plaintexts.

In this chapter we will show how to construct ambiguous ciphertexts, which decrypt to different valid file formats. The main idea behind the approach is to combine two different files into one single file such that the respective parsers ignore the additional content of the other file. Such a construct then allows us to encrypt each file with its own key such that on decryption only one of the two files will be decrypted correctly, whilst the decryption of the relevant parts corresponding to the other file result in random looking values. As we made sure that the content of the other file is ignored by the respective parser, this imposes no problem, as the random values will be ignored and therefore have no effect on the correctly decrypted file. This requires understanding the file formats used, their relation to one another and the constraints enforced by the computation of the tag-collision.

We will describe the high-level idea behind the approach and refer the interested reader to Dodis et al. [8] and Albertini et al. [5] for more details and insights into specific file format combinations.

4.1 Constraints

Random blocks for tag correction

As shown in Chapter 3, for ChaCha20-Poly1305 we require a single block of 16 bytes to "correct" the tag computed under the two keys such that they collide. The position of this block can be chosen by the attacker but is the same for both keys. Its decryption will be random, i.e. can not be controlled by the adversary, but different for each of the keys used.

The two resulting plaintexts therefore need to be able to handle a random block at the same position, i.e. both file formats need to be able to "ignore" the random block while parsing the file.

Fixing parts of the plaintext(s)

If the decryption of an ambiguous ciphertext C under K_1 results in P_1 and the decryption under K_2 in P_2, the following relation between P_1 and P_2 holds: $P_1 \oplus S_1 = P_2 \oplus S_2 = C$, where S_1, S_2 are the key-streams used to en-/decrypt C under K_1, K_2 respectively, whereby $\oplus$ denotes the 'xor' operation. As the attacker selects the keys K_1 and K_2 in our attack scenario, both S_1 and S_2 are known to the adversary.

Fixing a part of the plaintext P_1 determines the corresponding bits in the ciphertext and therefore the analogous part in P_2. If we have no requirement on the resulting bits in P_2 in that part, we can just fix the bits in P_1 and determine the corresponding part of C (and therefore the part in P_2). However, if we need to control a bit at the same position in P_1 and P_2, we are required to find two key-streams such that if p_1, p_2 denote the bit in the plaintexts P_1 and P_2, s_1, s_2 the bit in the key-streams S_1 and S_2 and c the corresponding bit in the ciphertext C, then $p_1 \oplus s_1 = p_2 \oplus s_2 = c$.

The chosen file formats will impose constraints on the structure of the plaintexts P_1 and P_2. If the overlap of bits which are constrained for the plaintexts becomes larger, attacks become more infeasible, as there is no efficient way of computing keys which satisfy constraints on the resulting key-streams, i.e. they need to be brute-forced. Assuming key-streams are uniformly random, if we are required to satisfy constraints for x bits, the probability that a second key-stream fulfills the constraints with respect to a first key-stream will be 2^{-x}. The constraints imposed by the file formats therefore need to be minimized.[1]

[1]In fact, it is possible to have co-existing files without any overlap. We will use such a file in our proof-of-concept implementation of an attack against Age, as without the need of finding collisions in the key-streams the attack becomes a lot more efficient.

4.2 Polyglot Files

A binary string which is valid when it it is interpreted as two different file types is called a polyglot file. We use the terms introduced by Albertini et al. [5] and distinguish between *binary* polyglots and *near* polyglots.

If two file formats can co-exist without any overlap, for example by starting at different offsets, we call the resulting file a *binary* polyglot file.

Not all combinations of file formats are able to co-exist in one file in such a way. For example, changing some bits of the header of a file may be required to turn it from one to another file format. We call a pair of files, describing different file formats, which differ only in a few bits, *near* polyglots. We will use the term overlap to refer to the bits/bytes for which both files impose different constraints, i.e. the bits which need to be a fixed, but different, value in both files at the same positions.

An ambiguous ciphertext can be created from a binary polyglot by encrypting the ranges corresponding to one of the files using one of the key-streams, S_1 resulting from key, K_1, chosen by the attacker and encrypting the parts corresponding to the second file using the second key-stream, S_2 resulting from the chosen key K_2. The same can be done for near polyglots, where now the creation becomes harder as we are required to find key-streams resulting in the correct bits for the overlap of the files.

Figure 4.1 shows an abstracted illustration of the process of creating an ambiguous ciphertext from a polyglot file without any overlap. In the first step, the two files are 'combined' into a single byte-string, which is valid under both formats. Interpreting the resulting byte-string as one of the file types involved in its creation will display the respective file. The byte-string is constructed in such a way that the parser ignores the content making up the second file contained within the byte-string. Next, the key-streams S_1 and S_2 are computed, such that each stream is long enough to encrypt the entire polyglot file. The final ambiguous ciphertext consists of the encryption of the parts corresponding to file 1 and file 2 using the respective key-stream S_1 or S_2.

Decrypting the ambiguous ciphertext using either of the key-streams S_1 or S_2 will not result in a polyglot file. Only the parts corresponding to one of the files will be decrypted correctly, depending on the key-stream used. Therefore, the resulting byte-string is only valid for one file type.

Albertini et al. [5] found more than 280 working combinations to create binary polyglots and over 50 combinations for near polyglots, such that the key-streams can be brute-forced with a reasonable amount of effort. We refer to their work for more details.

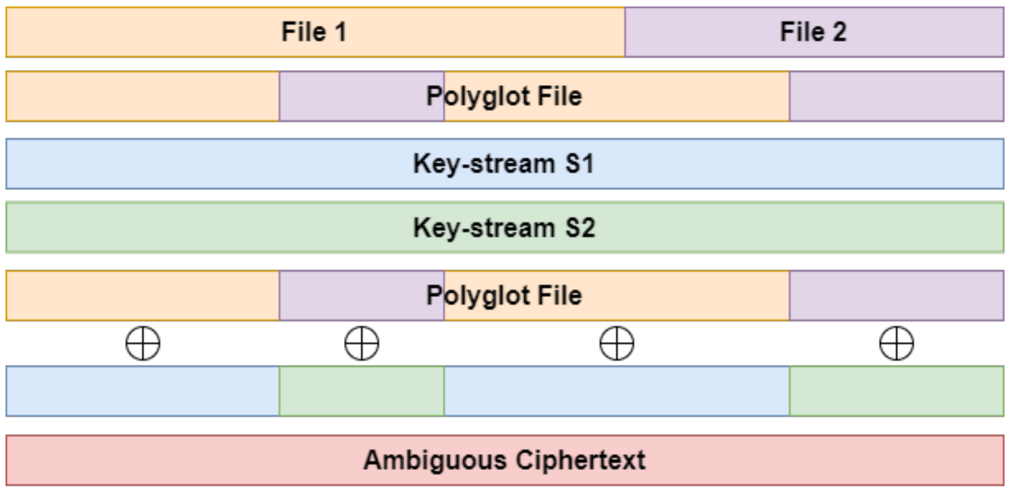


Figure 4.1: Abstract illustration of the process of creating an ambiguous ciphertext from a polyglot file.

4.3 Characteristics of File Formats

The creation of binary and near polyglots heavily relies on the different characteristics of various file formats. In this section we briefly discuss the main features used in the creation of polyglot files and refer to the paper by Albertini et al. [5, Section 4] for more insights in the creation of binary and near polyglots.

Offsets

Most file formats enforce the file structure to start at offset zero. However, there exist file formats which allow the file structure to start at any offset. This feature can be use to prepend one file in front of another.

Appending Data

If a parser determines that the file has ended, for example by reading a file specific end-of-file (EOF) sequence, any possibly appended data will be ignored. Therefore it is possible to append a file, allowing the file structure to start at any offset, to a file tolerating appended data.

Parasites

In most file formats we can include ”parasitic data”, which will be ignored by the parser. As an example, the reader may consider comments in a python script, which will be ignored by the compiler. This can be used to interleave one file as parasitic data into another one to create a polyglot file.

4.4 Remarks

In this chapter we briefly described the high-level idea behind polyglot files and how they could be constructed. It is not meant to serve as documentation on how to create polyglot files but should convey intuition on what a polyglot file is, how it could be constructed, which problems may arise in the process of their creation and how they can be used to create ambiguous ciphertexts.

We will use a binary polyglot file in the proof-of-concept implementation of our attack in Chapter 6. The file we will use is a binary polyglot combining a small .jpg and a small .pdf file. We created the file making use of the Mitra tool [3] developed by Albertini et al.. For more information on the creation of polyglot files and the implementation of the tool, we refer to the original paper by Albertini et al. [5] and the Mitra GitHub page [3].

Chapter 5

HKDF and HMAC

We have seen in Chapter 2 that Age uses HKDF for key derivations and HMAC for the MAC computation on the file header. In this chapter we will show that it is easy to compute collisions in HMAC if we have control over the size of the key used. Recall from Section 2.2.3 that the size of the file key is not checked for all recipient types in the decryption process of an Age-file.

Having control over the size of the file key could allow us to create collisions in HMAC. However, the file key is not directly used as a key in HMAC. Instead, it is used in HKDF, which internally uses HMAC to derive different keys. We explore the possibility of creating collisions in HKDF, by having control over the size of the used keys.

In the context of Age, being able to create collisions in HKDF would allow an adversary, who has control over the file keys, to force different recipients of the same Age-file to derive the same MAC-key but different payload keys. In consequence, such an adversary could create a single Age-file which decrypts to different plaintexts for different recipients, thereby breaking the robustness of Age.

5.1 HMAC

HMAC is a message authentication code (MAC) making use of a cryptographic hash function, H. It is formally specified in RFC 2104 [13]. Given a secret key K and a message M, it computes an output τ, which can be used to authenticate the message.

In this section we do not describe the functionality of HMAC but rather take a closer look at how it treats the provided secret key K. HMAC is defined to accept keys of any length up to B bytes, where B denotes the block-length of the used hash function H. If a key is shorter than B bytes, it will be padded with zeros until it is B bytes long.

This directly leads to a possible collision attack. Assume we have control over the secret key K used for the HMAC computation on some message M. We can set K to be random, such that $|K| < B$, i.e. K will be padded with zeros before use. It should be clear that by setting $K' = K||0$ the result of the HMAC computation using keys K and K' will be equal, i.e. $\text{HMAC}(H, K, M) = \text{HMAC}(H, K', M)$, due to HMAC being deterministic.

On the other hand, if an application uses keys that are longer than B bytes, then RFC 2104 states that the key is hashed using the hash function H before being used as the secret key in HMAC.[1] This again leads to a trivial collision attack. Choosing a random key K with $|K| > B$ we can compute $K' = H(K)$. As HMAC is defined, all computations done using K result in the same outputs as if they are done using K', i.e. $\text{HMAC}(H, K, M) = \text{HMAC}(H, K', M)$ for all M.[2]

We conclude that, having control over the size of the key used for an HMAC computation, it is easy to create collisions in HMAC. Note that these collisions are on keys, not on messages - messages are always the same.

5.2 HKDF

HKDF is a key derivation function based on HMAC, which is formally specified in RFC 5869 [14]. It is split up into two main steps, the 'Extract' step and the 'Expand' step. We focus on the extract step in this section as it derives a key used in the expand step from the initially provided key material using HMAC. Therefore, if we are able to create a collision in this step, we have crated a collision in HKDF as the the same key results in the same output due to the deterministic nature of the expand step.

The extract step takes as input a salt value and the initially provided key material, called IKM in all of the following. It outputs a pseudorandom key, called PRK, by performing the following HMAC computation:

$$PRK = \text{HMAC}(H = H, K = \text{salt}, M = IKM)\,,$$

where H is a secure hash function.

Observe that the initial key material IKM is not used as the key input of the HMAC computation but rather as the message input. Therefore, having control over the key material provided to HKDF does not lead to a collision attack in the same way it is possible for HMAC. However, if the salt value

[1]Depending on the block size B and the output length of the hash function H, the hash of the key might require padding with zeros to get to a length of B bytes.

[2]Note that this is independent of the hash function used and the detailed functionality of HMAC.

provided to HKDF is under adversarial control, a collision attack becomes possible again, as the salt is used as the key input for the HMAC computation and therefore the same techniques can be applied to create a collision.[3]

5.3 Implications for Age

Sections 5.1 and 5.2 show that it is easy to compute collisions in HMAC and HKDF if we can control the key or salt respectively, i.e. not only the value but also the length of the key or salt.

Recall that if we are able to create collisions in HKDF we could break the robustness of Age. Wrapping different file keys, which result in HKDF collisions for the derivation of the MAC-key, for different recipients would allow all recipients to verify the authenticity of the file header but derive different payload keys. By using an ambiguous ciphertext as the payload of such an Age-file all recipients would derive a different plaintext from the same payload. Therefore, the same Age-file would be decrypted to different plaintexts by different recipients, breaking the robustness of Age.

Although we are able to control the file key used in HKDF in Age, we can not perform a collision attack as described. Recall that HKDF uses the initial key material, in the case of Age the file key, as the message input for HMAC and not as the HMAC-key. As the salt value for the derivations is fixed to be the all-zero string or a random nonce, creating collisions using the described methods is impossible.

5.3.1 Changing Nonces

However, recall from Chapter 2 that the payload key is derived from the file key using the following HKDF computation:

$$payload\,key = \text{HKDF}(\text{SHA256}, \text{ikm} = file\,key, \text{salt} = nonce, \text{info} = \text{"payload"})\,.$$

As one can see, the salt value is set to the randomly sampled nonce value. As the nonce is placed between the header and the encrypted payload of an Age-file, it is not authenticated. Therefore, any change to the nonce results in a different payload key being derived, whilst still allowing to correctly authenticate the header. This is because neither the file key nor the derived MAC-key are affected by a change in the nonce.

If Age is used normally, an accidental change of the nonce would not be detected directly as such. Rather, the decryption of the first payload block will fail, as the Poly1305 authentication done in ChaCha20-Poly1305 will not succeed with high probability.

[3]Note that the attacker needs to be able to insert different sized salts to create a collision.

An attacker can however carefully craft a ciphertext, which will correctly verify for two different payload keys used due to the possibility of calculating collisions in ChaCha20-Poly1305 as we showed in Chapter 3. It is therefore easy for an adversary to create multiple Age-files, which only differ in the nonce values used, such that each recipient decrypts the payload to a different plaintext. We discuss the details of such an attack in the next chapter.

Chapter 6

Attacking Age

Combining our insights on Age (see Chapter 2), the absence of key commitment using ChaCha20-Poly1305 (see Chapter 3) and the ability to create ambiguous ciphertexts from polyglot files (see Chapter 4), we are able to perform an attack which allows an adversary to create two nearly identical Age-files for (at least) two recipients such that each recipient decrypts the payload to a different valid file format, i.e. they end up with different plaintexts.

The attack also works if only one recipient is present. However, the most realistic attack scenarios involve at least two different parties, as the goal of the attack is to confuse the users into believing they have decrypted the same file but ended up with different plaintexts.

We consider an attack scenario, where an adversary Mallory creates an Age-file intended for two recipients Alice and Bob and sends it to one of them. The first recipient, Alice, will decrypt the payload to a (malicious) plaintext P_1. If Alice forwards the file to the second recipient, Bob, Mallory can flip one bit in the Age-file sent by Alice to force Bob to decrypt the payload to another (non malicious) plaintext P_2.

Think of Alice submitting an abuse report and Bob needing to verify the reported message. In such a scenario Mallory could send malicious data to Alice without Bob ever witnessing the malicious payload.

6.1 Threat Model

In its current specification, the nonce value used for the derivation of the payload key is a randomly selected 16-byte string and is stored in-between the header and the encrypted payload of an Age-file. Recall, this means that the nonce is not integrity protected and any change to it, malicious or accidental, will not be detected as such on payload decryption

We consider an active man-in-the-middle (MitM) attacker who is able to intercept, alter and delay all traffic. As for our attack the adversary knows the involved keys, it would also be possible that the attacker completely replaces a sent Age-file with a different one. Although this possibility exists for this threat model, our attack has certain benefits over a simple replacement of the sent file.

The two Age-files created in our attack are confusingly similar. As the difference between the files can be as small as one bit, it can easily get unnoticed by an unsophisticated user inspecting the two files. Note however that advanced users most probably will compare fingerprints of the files after noticing the difference in the plaintexts. In this case, the one bit difference between the files will be amplified and the fingerprints of the two files will be completely different.

More importantly, an active MitM attacker can not replace an Age-file if it is sent over an encrypted channel, as the adversary does not have access to the encryption key. On the other hand, it can be possible, depending on the encryption scheme used, that an active MitM adversary can flip bits of encrypted Age-file sent over the channel. In such a case, our attack can still succeed whilst a simple replacement can not.

6.1.1 Attack Scenario

We briefly describe a possible attack scenario to provide more intuition on where our attack could be applied.

Assume a company 'Age-Shopping' uses Age-files for incoming orders sent by customers and uses the same Age-files internally for further processing of the order. If Mallory is an active MitM attacker on the internal system used by Age-Shopping, she could perform the following attack.

Mallory can craft two Age-files, differing in exactly one bit, containing two different orders for some products offered by Age-Shopping. Assume the first file contains an order for cheap products and the second file corresponds to an order for more expensive products. Mallory can now send the first Age-file as an order to Age-Shopping. Assume the employees of Age-Shopping decrypt the incoming order, prepare the bill and relay the Age-file to an internal server for delivery preparation.

Mallory being an active MitM attacker on Age-Shopping's internal network can flip the correct bit in the sent Age-file to turn it into the second version she prepared, now containing the order for the more expensive products. The employees decrypting the file to prepare the delivery of the products will now send Mallory the expensive products whilst she only gets a bill for the cheap ones.

This is only a toy example illustrating how the attack could be applied. It should help the reader to imagine similar scenarios where an adversary can fool an abuse reporting system into believing an original abusive message is harmless, or how an attacker could embed malicious code into one of the Age-files created, potentially allowing installation of an adversarial controlled program on user devices.

6.2 Key Derivations in Age

Recall that all keys used in Age are derived from a randomly selected 16-byte file key in the following way:

- Header-MAC-Key, 32-bytes: HKDF(SHA256, None, file key, ”header”)
- Payload key, 32-bytes: HKDF(SHA256, Nonce, file key, ”payload”)
- Poly1305 Key(s), two times 16-bytes: First 32-bytes of ChaCha20 keystream generated using the payload key, as described in RFC 7539 [20, Section 2.6].

For two parties to decrypt the payload of an Age file to two different plaintexts, the derived payload keys must be different. However, both parties need to compute the same header-MAC-key to verify the integrity of the file header. Due to the specification of HKDF, we can not efficiently find collisions for the derivation of the header-MAC-key and therefore both recipients must have access to the same file key, i.e. in the header the same key is encapsulated in the recipient stanzas. We described a method of efficiently computing collisions in HKDF and the underlying HMAC function and why it cannot be applied for Age in Chapter 5.

This restriction only leaves the option of providing a different nonce value to each recipient. A different nonce will guarantee that both parties derive a different payload key from the same file key. However, this leads to the situation where the two Age-files differ in the nonce and are therefore not identical.

6.3 File Preparation

Our attack makes use of a binary polyglot file, interleaving a small .pdf (482 bytes) and .jpg file (238 bytes) into a single polyglot file. The file formats are able to co-exist in the same file without any overlapping bits, simplifying the attack implementation.

We created the file using the Mitra tool [3]. Given the two files as input, it outputs a polyglot file named *P(6-1e8)-JPG[PDF].6b90e1cc.jpg.pdf*. In-between the brackets ”( )” are hex-encoded byte positions, indicating at which byte offsets the content switches form one file to the other. In the example we used, we only have two switches at offsets $0x6$ and $0x1e8$. We will extract this information from the file name and use it in our attack code to create an ambiguous ciphertext from the polyglot file, by encrypting bytes 6 up to 488 with key K_2 and all other bytes with key K_1.

In the following we will refer to the positions extracted out of the file name as switching positions and assume they are available to us in an array called SwitchingPositions.

6.4 Attack Implementation

In this section we describe our proof-of-concept implementation of the attack against Age. We split up the code into four main parts, File Encryption, Tag Collision, Header Generation and Final File Construction. For each part of the attack we briefly describe the main functionality of our code and provide a pseudocode interpretation of its functionality.

6.4.1 File Encryption

Recall that the payload of an Age-file is encrypted with ChaCha20-Poly1305 using a payload key derived from the file key using HKDF. To be able to create the ambiguous ciphertext from a polyglot file we first need to derive the two different payload keys that the recipients will derive during the decryption process of the final Age-file.

Key Derivation

In a first step, we select a random 16-byte file key, FK, and a 16-byte random nonce, N_1. By flipping the last bit of N_1 we create a second 16-byte nonce N_2. This bit-flip simulates the affect of an adversary modifying the nonce value in the final file once it is sent from one recipient to another. Note that it does not matter which bit of the 16-byte nonce is flipped and it is also possible to alter more than one bit of the nonce. However, it is important that the bit(s) altered are at a well known position to the adversary such that the correct nonce N_2 can be generated for payload key derivation.

Using HKDF, we derive the two different payload keys as follows:

1. $SK_1 = \text{HKDF}(SHA256,\ N_1,\ FK,\ "payload")$
2. $SK_2 = \text{HKDF}(SHA256,\ N_2,\ FK,\ "payload")$

The derived keys allow us to compute the ChaCha20 key-streams which will be used to en-/decrypt a file using the corresponding key. For both SK_1 and SK_2 we compute the resulting ChaCha20 key-stream, such that each stream is long enough to encrypt the entire file. We will refer to them as $ChaChaStream_1$ and $ChaChaStream_2$ respectively.

File Encryption

Now, we can encrypt the payload. To do so, we iterate over the plaintext byte by byte and start encrypting the payload using $ChaChaStream_1$. If we arrive at a switiching position, i.e. an offset at which the contents switches from one file to the other, we switch the key-stream used for encryption from $ChaChaStream_1$ to $ChaChaStream_2$ and vice versa. After iterating over all plaintext bytes, we have an ambiguous ciphertext, which we will refer to as $Ctxt$.

A pseudocode implementation of the key derivation and file encryption process is shown in Figure 6.1.

6.4.2 Tag Collisions

Recall that to guarantee that both recipients are able to decrypt the ambiguous ciphertext correctly, i.e. without an error, we need to add an additional block, or modify an existing one, to "correct" the tag computed under both Poly1305 keys such that they collide.

We use the same techniques as described in Chapter 3. The pseudocode implementation computing the new ciphertext block and outputing the new ciphertext is illustrated in Figure 3.1.

The binary polyglot file we selected for the attack allows for appended data, i.e. any additional data added to the end of the file will be ignored by the parsers. This allows us to insert the "correction block" directly after the last encrypted plaintext block. Note that this is not possible for all combinations of file formats. If no additional block can be appended for the selected file formats, the block for tag correction must be inserted in another position such that it will be ignored by the respective parsers.[1]

6.4.3 Header Generation

Recall from Chapter 2 that the encrypted payload of an Age-file is prepended by a randomly selected nonce value and a header, containing the recipient stanzas, i.e. the wrapped file key.

In the process of our attack, we select a file key ourself and therefore need to generate the corresponding recipient stanzas for the victims.

[1]For example, the block could be prepended or inserted as parasitic data into one of the files making up the polyglot file.

```
FileEncryption(Ptxt, SwitchingPositions)
$FK \leftarrow\$ \textsc{randBytes}(16)$
$N_1 \leftarrow\$ \textsc{randBytes}(16)$
$Mask \leftarrow 0x00000000000000000000000000000001$
$N_2 \leftarrow N_1 \oplus Mask$   // Flip last bit of $N_1$
$SK_1 \leftarrow \text{HKDF}(SHA256,\ N_1,\ FK,\ "payload")$
$SK_2 \leftarrow \text{HKDF}(SHA256,\ N_2,\ FK,\ "payload")$
$ChaChaStream_1 \leftarrow \textsc{GetChaCha20KeyStream}(SK_1, \textsc{len}(Ptxt) + 1)$
$ChaChaStream_2 \leftarrow \textsc{GetChaCha20KeyStream}(SK_2, \textsc{len}(Ptxt) + 1)$
$counter \leftarrow 0$
$stream \leftarrow ChaChaStream_1$
$Ctxt \leftarrow b""$   // Initialize ciphertext as empty byte string
// Iterate over the plaintext byte by byte
for $byte$ in $Ptxt$ do
  if $counter$ in $SwitchingPositions$ do   // Switch key-streams
    $stream \leftarrow \textsc{SwitchStreams}(stream, ChaChaStream_1, ChaChaStream_2)$
  endif
  // Encrypt plaintext-byte with current stream-byte
  $Ctxt \leftarrow Ctxt||(byte \oplus stream[counter])$
  $counter \leftarrow counter + 1$
endfor
return $Ctxt, FK, N_1, N_2$
```

Figure 6.1: Pseudocode encrypting a given plaintext under two keys, switching between the keys at the positions indicated by SwitchingPositions. $\textsc{randBytes}(n)$ is a function, given an integer n, returning n random bytes, $\textsc{len}(x)$, returns the length in bytes of its input x, $\textsc{GetChaCha20KeyStream}(n,\ K)$ returns the first n bytes of the ChaCha20 key-stream generated using key K, $\textsc{SwitchStream}(x, y, z)$ returns y if $x = z$ and x if $y = z$.

Following the Age specification [1, 24], we were able to create valid Age-file-headers containing a wrapped file key of our choosing, where we used the Header-MAC-key, derived from the file key (see Section 6.2), to compute the MAC over the header.

6.4.4 Final File Construction

The final Age-file consists of a header, containing the recipient stanzas for the selected file key, followed by the MAC-tag authenticating it. The next 16 bytes are occupied by the normally randomly chosen nonce value, which in our case is replaced by the nonce(s) we selected.[2] Finally, the encrypted payload data, in our case the ambiguous ciphertext is appended, after computing the now colliding Poly1305 tag.

Note that as the files used in our proof-of-concept implementation are small, we can compute the single Poly1305 tag after performing the necessary changes to the ciphertext.

A pseudocode descryption of the entire attack code is presented in Figure 6.2

6.5 Remarks

Key Derivation

Note that in our proof-of-concept implementation we did not care about which recipient will derive which file key. In a real attack scenario, an adversary would need to carefully select which nonce will be delivered to which recipient such that the correct file can be encrypted using the corresponding key-stream. In most cases, an attacker would want the first recipient to decrypt the payload to the malicious file. This also requires careful monitoring whilst creating the polylgot file to be able to identify which parts of the polyglot correspond to which file.

Header Generation

We only considered X25519 recipients in the scope of our attack. Note the same techniques can be used against ssh-rsa and ssh-ed25519 recipients, or combinations of the three respectively.

In the case of an scrypt recipient, the Age-files can only contain one recipient stanza. This does not prevent our attack, as changing the nonce will still result in the derivation of a different payload key, therefore a second user,

[2] Once the file is relayed to the second party, the mask changing the nonce must be applied to guarantee decryption to the second plaintext.

```
Attack(Ptxt, SwitchingPositions, Recipients)

1 :  $NewCtxt \leftarrow \perp$
2 :  // Iterate until collision succeeds
3 :  while $NewCtxt = \perp$ do
4 :      $Ctxt, FK, N_1, N_2 \leftarrow \textsc{FileEncryption}(Ptxt, SwitchingPositions)$
5 :      $pos \leftarrow |Ctxt|/16$   // Modify second to last block
6 :      $Ctxt \leftarrow Ctxt||0^{16}$   // Append 16 0-bytes for tag correction
7 :      $Ctxt \leftarrow Ctxt||\textsc{LengthEncoding}(Null, ptxt)$
8 :      $NewCtxt \leftarrow \textsc{Collide}(Ctxt, K_1, K_2, pos)$
9 :  endwhile
10 : $tag \leftarrow \textsc{Poly1305Tag}(Ctxt, K_1)$
11 : $Header \leftarrow \textsc{GenerateHeader}(FK, Recipients)$
12 : $file_1 \leftarrow Header||N_1||NewCtxt||tag$
13 : $file_2 \leftarrow Header||N_2||NewCtxt||tag$
14 : return $file_1, file_2$
```

Figure 6.2: Pseudocode describing the creation of the two different Age files used in the attack. $\textsc{FileEncryption}(\cdot,\cdot)$ is as described in Figure 6.1, $\textsc{LengthEncoding}(\cdot,\cdot)$ returns the encoded length of plaintext and associated data as described in Section 3.4, $\textsc{Collide}(\cdot,\cdot,\cdot,\cdot)$ as in Figure 3.1, $\textsc{GenerateHeader}(\cdot,\cdot)$ generates the Age-file header and header MAC-tag, $\textsc{Poly1305Tag}(\cdot,\cdot)$ computes the Poly1305 tag over a given message unsing the provided key.

or the same user for that matter, knowing the password can be tricked into decrypting the file to a different plaintext.

In Appendix A.1, we provide two example Age-files, differing in exactly one nonce-bit, and an X25519 identity, which illustrate the attack described in this chapter.

6.6 Attack Statistics

We analysed the performance of our proof-of-concept implementation to show the practicality of the attack. All calculations were done on a Windows 10 machine using an Intel Core i9-10900K processor. The code is written in Python and compiled using Python 3.10.4.

First, only considering the generation of an ambiguous ciphertext from a polyglot file, i.e. the computation of a collision for ChaCha20-Poly1305. Figure 6.1 summaries the statistics after running the attack 100000 times. Shown are the minimum and maximum runtime in milliseconds as well as the mean runtime and its standard deviation. Recall, that the computation of the 'tag-correction block' can fail, if the result is not in the correct range or due to the

ChaCha20-Poly1305 Collision Attack Statistics				
	Min	Max	Mean	Std
Runtime (ms)	0.51212	66.82038	5.34495	4.68059
Total Retry Attempts	0	54	3.3644	3.8396
Range Errors	0	50	3.2737	3.7622
Remapping Errors	0	4	0.0907	0.3150

Table 6.1: Performance statistics of our attack code only considering the generation of the ambiguous ciphertext, i.e. computing a collision in ChaCha20-Poly1305. The initial plaintext file is 724 bytes in size. The runtime is given in milliseconds.

Attack Statistics				
	Min	Max	Mean	Std
Runtime (ms)	92.51	437.81	109.19	9.63

Table 6.2: Performance statistics of our entire attack code including output Age-file generation. The initial plaintext file is 724 bytes in size. The runtime is given in milliseconds.

mod P remapping resulting in an incorrect result. Similarly to the runtime, we analysed the minimum and maximum number of retry attempts needed to find a collision as well as its mean and standard deviation. The polyglot file used for the analysis is 724 bytes in size.

We made a similar analysis for the entire proof-of-concept implementation of our attack. This includes the computation of the ChaCha20-Poly1305 collision, as well as the construction and writing of the final Age-file(s). This time, we ran the attack 10000 times and only measured its runtime in milliseconds. The results are presented in Figure 6.2.

The analysis shows that the attack is indeed practical for small enough files. An average runtime of roughly 110 milliseconds for the entire (unoptimised) code is fast. Doing some rough estimation, this results in an average time of close to 0.15 milliseconds per byte of plaintext.[3]

However, if files become larger, which in consequence means that the polyglot file will be bigger, the attack becomes less practical, as it will require more runtime. Doing a rough estimation, an attack involving a 65 kB polyglot file will take around 10 seconds to complete. As each chunk is authenticated on its own, the code can be parallelized to speed up computation.

Our analysis shows that we need between 3 and 4 retry attempts before the

[3]The plaintext here is the original polyglot file given to the code as input.

attack succeeds for one-chunk-payloads. Recall that this means we need to select a different block for tag correction or select new keys to get new key-streams. Assuming each chunk is independent of all other chunks, and we only have one block of freedom in each chunk for tag correction, then the success probability rapidly decreases the larger the file becomes. For a file consisting of n chunks, i.e. a file of size $n \cdot 2^{16}$ bytes, the success probability will be somewhere between 3^{-n} and 4^{-n}. This means that already for a file consisting of 4 chunks, the success probability will be around 1%. However, multiple blocks of freedom per chunk, i.e. more flexibility in the polyglot file creation, can significantly improve the success probability.

Chapter 7

Fixes

We came up with two possible solutions to adapt Age in a minimal way to mitigate the attack shown in Chapter 6.

The first approach would integrate the selected nonce into the header, therefore it would be authenticated by the MAC generated over the header and the attack, as it is described, will not work any more, as a change in the nonce would require a MAC tag forgery to result in valid decryption.

The second solution would be to set the nonce to a fixed value, hardcoded into Age. This is already done for the generation of the header MAC key. Recall that in this case the nonce is set to 'none', which results in the all-zero-string being used as the nonce value for the HKDF key derivation. In a similar way, the nonce value for the derivation of the payload key can be set to a constant value, other than the all-zero-string, to mitigate the attack.

Note that both solutions could be implemented in a next version of Age, but would not be backward compatible with the current 'v1' version.

7.1 Why Age is Designed in this Way

We informed Filippo Valsorda, the author of Age, about the attack we found, proposed our two solution ideas and asked what led to the design decision of placing the nonce between the header and the encrypted payload, in an unauthenticated manner.

In his response [23], he mentioned that for our second solution, having a fixed nonce value, an attacker, having access to 2^{64} Age-files with a known prefix, could find one decryption in an average time of 2^{64}. Having a longer file key would increase the number of needed files as well as the average time required to find a decryption, but a larger file key would increase the overhead on a per-recipient basis instead of per-file, as the file key is encrypted for each recipient in its own recipient stanza.

Our first solution, placing the nonce inside the header, maybe should have been the way to go [23]. There were two main reasons which led to the decision of not to do so.

First, the nonce can be seen as part of the output of a stream encryption, similarly to how a nonce is part of the output of an AEAD encryption operation. Secondly, users may reuse the same header to encrypt multiple files, although this is discouraged in the specification, to save on the expensive asymmetric computations done in header generation. By placing the nonce outside the header, such a re-use can safely work, as a new nonce will be selected for each file, while if the nonce is part of the header, the same payload key will be derived for all files using the same header, which leads to a serious vulnerability as it allows an attacker having access to two files to recover the key-stream used to encrypt the files.

Regarding the second point, one could allow users to re-use headers, by sampling a random nonce and recomputing the header MAC value. This would allow users to still save on the expensive operations done for header generation and only add a small overhead, as the HMAC computation for the new MAC-tag is cheap.

Chapter 8

Conclusion

We presented a high level overview of Age in Chapter 2, showed a possible attack in Chapter 6 and proposed solutions to mitigate the problem in Chapter 7.

Although we were able to find an attack against Age, it is only applicable in a limited attack scenario. An adversary needs to know the file key in order to perform the attack and additionally needs to be able to force a second recipient to receive an altered version of the Age-file sent to the first recipient.

The files constructed in our attack are not identical. To trick two different users into decrypting the seemingly same file to different plaintexts, we needed to change the nonce in the files. The required difference in the nonces can be as small as one bit. Therefore, we were not able to break the robustness of Age.

This project shows that one needs to take great care designing encryption protocols. Every design decision needs to be thought through carefully as every single bit can be the root of a potential vulnerability against the entire design.

In conclusion, Age seems to be a very well designed command line encryption tool. As a continuation of this project, a formal security analysis of Age can be done in an appropriate security model, to either prove Age secure or to discover new vulnerabilities. For this we suggest moddeling Age as a public key encryption scheme and try proving that the modelled scheme is robust.

Appendix A

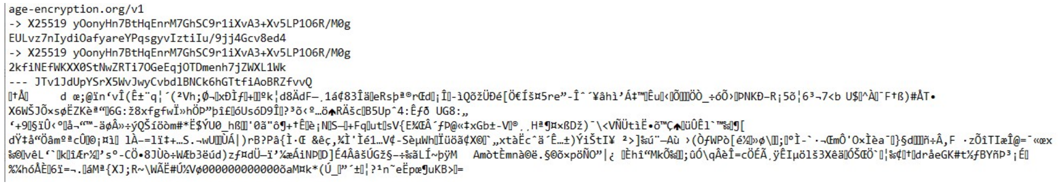
Appendix

A.1 Attack Example

In this section we briefly present two Age-files, which we generated using our proof-of-concept implementation described in Chapter 6.

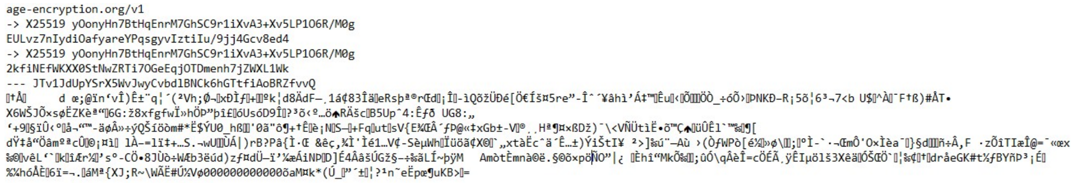
Figure A.1 shows the two Age-files output by our implementation. The two files differ only in the last bit of the nonce. Decrypting the original or modified version of the files results in the output either being a valid .jpg file displaying the word 'JPG' or the output corresponding to a .pdf file showing the word 'PDF' as shown in Figure A.2.

To verify that our attack works, we provide the identity corresponding to one of the recipients of the files shown, in Figure A.3. This identity can be used to decrypt the provided Age-files and can therefore be used to verify that the files decrypt to the claimed outputs.



```
age-encryption.org/v1
-> X25519 yOonyHn7BtHqEnrM7GhSC9r1iXvA3+Xv5LP1O6R/M0g
EULvz7nIydiOafyareYPqsgyvIztiIu/9jj4Gcv8ed4
-> X25519 yOonyHn7BtHqEnrM7GhSC9r1iXvA3+Xv5LP1O6R/M0g
2kfiNEfWKXX0StNwZRTi7OGeEqjOTDmenh7jZWXL1Wk
--- JTv1JdUpYSrX5WvJwyCvbdlBNCk6hGTtfiAoBRZfvvQ
▯†Å▯      d œ;@ïn‘vÎ(Ê±¨q¦´(²Vh;Ø¬▯xÐÌƒ▯+▯▯ºk¦d8ÄdF–¸1á¢83Îä▯eRsþª®rŒd▯¡Î▯-ìQõžÜÐé[Ö€Íš¤5re”-Î^´¥âhì’Á‡™▯Êu▯‹▯Õ▯▯▯ÖÒ_÷óÕ›▯ÞNKÐ–R¡5õ¦6³¬7<b U$▯^À▯¯F†ß)#ÅT•
X6WŠJÕ×søËZKèª“▯6G:ž8xfgfwÏ»hÖÞ”þî£▯óUsóD9Î▯?³õ‹º…ö⬆RÄšc▯B5Up^4:Êƒð UG8:„
‘+9▯§ïÛ‹º▯å¬“™-äøÂ»÷ýQŠíöòm#*Ë$ÝU0_hß▯▯'0ã"ô¶+†Ê▯è¡N▯S–▯+Fq▯ut▯sV{E¾ŒÂ´ƒÞ@«‡xGb±-V▯®¸¸Hª¶¤×ßDž)¯\<VÑÜtìË•õ™Ç⬆▯üÛÊ1`™‰▯¶[
dŸ‡å“Öâmºªcû▯©¡¤ì▯ 1À–=1ï‡+…S.¬wU▯▯ÙÁ|)rB?Pâ{Ì·Œ &êç,¾Ì'Ìé1…V¢-SèµWh▯Ïüöã¢X©▯ˆ„xtàËc^ä´Ê…±)ÝiŠtI¥ ²>]‰ú¨—Aù ›(ÒƒWPò[é¼▯»ø\▯▯;▯°Ì-`·¬ŒmÔ'O×Ìèa¯▯}§d▯▯▯ñ÷Â,F ·zÕîTIæÎ@=¯«œx
‰©▯vêL‘`▯k▯îÆr¼▯’sº-CÖ•8JÙò÷WÆb3ëúd)zƒ¤dÜ–ï’¼æÁiNÞ▯D]É4ÂâšÚGž§–÷‰ãLÍ~þÿM   AmòtÈmnà©ë.§©õ×pöÑO”|¿ ▯Èhî“MkÕ‰▯▯;ûÓ\qÂèÎ=cÖÉÃ¸ÿÊIµöl š3Xêä▯ÓŠŒÖ`▯¦‰¢▯†▯dråeGK#t½ƒBYñÞ³¡É▯
%¼hóÅÈ▯6ï=¬.▯áMª{XJ;R~\WÃË#Ú½Vø0000000000000õaM¤k*(Ú_▯”´±▯¦?¹n˜eËpœ¶uKB>▯=
```

(a) Original Age-file.



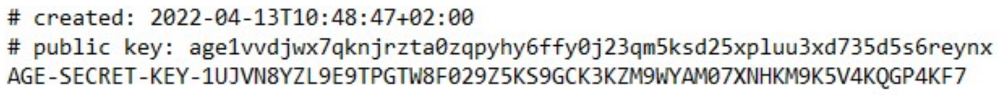
```
age-encryption.org/v1
-> X25519 yOonyHn7BtHqEnrM7GhSC9r1iXvA3+Xv5LP1O6R/M0g
EULvz7nIydiOafyareYPqsgyvIztiIu/9jj4Gcv8ed4
-> X25519 yOonyHn7BtHqEnrM7GhSC9r1iXvA3+Xv5LP1O6R/M0g
2kfiNEfWKXX0StNwZRTi7OGeEqjOTDmenh7jZWXL1Wk
--- JTv1JdUpYSrX5WvJwyCvbdlBNCk6hGTtfiAoBRZfvvQ
▯†Å▯      d œ;@ïn‘vÎ)Ê±¨q¦´(²Vh;Ø¬▯xÐÌƒ▯+▯▯ºk¦d8ÄdF–¸1á¢83Îä▯eRsþª®rŒd▯¡Î▯-ìQõžÜÐé[Ö€Íš¤5re”-Î^´¥âhì’Á‡™▯Êu▯‹▯Õ▯▯▯ÖÒ_÷óÕ›▯ÞNKÐ–R¡5õ¦6³¬7<b U$▯^À▯¯F†ß)#ÅT•
X6WŠJÕ×søËZKèª“▯6G:ž8xfgfwÏ»hÖÞ”þî£▯óUsóD9Î▯?³õ‹º…ö⬆RÄšc▯B5Up^4:Êƒð UG8:„
‘+9▯§ïÛ‹º▯å¬“™-äøÂ»÷ýQŠíöòm#*Ë$ÝU0_hß▯▯'0ã"ô¶+†Ê▯è¡N▯S–▯+Fq▯ut▯sV{E¾ŒÂ´ƒÞ@«‡xGb±-V▯®¸¸Hª¶¤×ßDž)¯\<VÑÜtìË•õ™Ç⬆▯üÛÊ1`™‰▯¶[
dŸ‡å“Öâmºªcû▯©¡¤ì▯ 1À–=1ï‡+…S.¬wU▯▯ÙÁ|)rB?Pâ{Ì·Œ &êç,¾Ì'Ìé1…V¢-SèµWh▯Ïüöã¢X©▯ˆ„xtàËc^ä´Ê…±)ÝiŠtI¥ ²>]‰ú¨—Aù ›(ÒƒWPò[é¼▯»ø\▯▯;▯°Ì-`·¬ŒmÔ'O×Ìèa¯▯}§d▯▯▯ñ÷Â,F ·zÕîTIæÎ@=¯«œx
‰©▯vêL‘`▯k▯îÆr¼▯’sº-CÖ•8JÙò÷WÆb3ëúd)zƒ¤dÜ–ï’¼æÁiNÞ▯D]É4ÂâšÚGž§–÷‰ãLÍ~þÿM   AmòtÈmnà©ë.§©õ×pöÑO”|¿ ▯Èhî“MkÕ‰▯▯;ûÓ\qÂèÎ=cÖÉÃ¸ÿÊIµöl š3Xêä▯ÓŠŒÖ`▯¦‰¢▯†▯dråeGK#t½ƒBYñÞ³¡É▯
%¼hóÅÈ▯6ï=¬.▯áMª{XJ;R~\WÃË#Ú½Vø0000000000000õaM¤k*(Ú_▯”´±▯¦?¹n˜eËpœ¶uKB>▯=
```

(b) Modified Age-file, differing in exactly one bit of the nonce.

Figure A.1: The two different Age-files generated by our attack code, differing in exactly one nonce-bit and decrypting to two different valid files, a .jpg file showing the word 'JPG' and a .pdf file showing the word 'PDF'.

(a) A .jpg file showing the word 'JPG'. **(b)** A .pdf file showing the word 'PDF'.

Figure A.2: The two different output files resulting from decrypting the original and modified Age-file.



```
# created: 2022-04-13T10:48:47+02:00
# public key: age1vvdjwx7qknjrzta0zqpyhy6ffy0j23qm5ksd25xpluu3xd735d5s6reynx
AGE-SECRET-KEY-1UJVN8YZL9E9TPGTW8F029Z5KS9GCK3KZM9WYAM07XNHKM9K5V4KQGP4KF7
```

Figure A.3: The identity corresponding to one of the recipients of the shown Age-files, which can be used to decrypt the files and verify their outputs.

Bibliography

[1] Age Format Specification. https://github.com/C2SP/C2SP/blob/main/age.md. Accessed: 2022-12-05.

[2] Age Sourcecode. https://github.com/FiloSottile/age. Accessed: 2022-23-02.

[3] Mitra: A tool to generate binary polyglots. https://github.com/corkami/mitra. Accessed: 2022-14-04.

[4] Michel Abdalla, Mihir Bellare, and Gregory Neven. Robust encryption. *Journal of Cryptology*, 31(2):307–350, apr 2018.

[5] Ange Albertini, Thai Duong, Shay Gueron, Stefan Kölbl, Atul Luykx, and Sophie Schmieg. How to abuse and fix authenticated encryption without key commitment. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, August 2022. USENIX Association.

[6] Swarup Bhunia and Mark Tehranipoor. Chapter 8 - Side-Channel Attacks. In Swarup Bhunia and Mark Tehranipoor, editors, *Hardware Security*, pages 193–218. Morgan Kaufmann, 2019.

[7] Denis Bider. Use of RSA Keys with SHA-256 and SHA-512 in the Secure Shell (SSH) Protocol. RFC 8332, March 2018.

[8] Yevgeniy Dodis, Paul Grubbs, Thomas Ristenpart, and Joanne Woodage. Fast message franking: From invisible salamanders to encryptment. In *Advances in Cryptology – CRYPTO 2018*, volume 10991 of *Lecture Notes in Computer Science*, pages 155–186. Springer, 2018.

[9] Pooya Farshim, Benoît Libert, Kenneth G. Paterson, and Elizabeth A. Quaglia. Robust encryption, revisited. Cryptology ePrint Archive, Paper 2012/673, 2012. https://eprint.iacr.org/2012/673.

[10] Shay Gueron, Adam Langley, and Yehuda Lindell. AES-GCM-SIV: Nonce Misuse-Resistant Authenticated Encryption. RFC 8452, April 2019.

[11] Tony Hansen and Donald E. Eastlake 3rd. US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF). RFC 6234, May 2011.

[12] Simon Josefsson. The Base16, Base32, and Base64 Data Encodings. RFC 4648, October 2006.

[13] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, February 1997.

[14] Hugo Krawczyk and Pasi Eronen. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869, May 2010.

[15] Ted Krovetz and Phillip Rogaway. The OCB Authenticated-Encryption Algorithm. RFC 7253, May 2014.

[16] Adam Langley, Mike Hamburg, and Sean Turner. Elliptic Curves for Security. RFC 7748, January 2016.

[17] Julia Len, Paul Grubbs, and Thomas Ristenpart. Partitioning oracle attacks. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 195–212. USENIX Association, August 2021.

[18] Fabio Maino, Uri Blumenthal, and Keith McCloghrie. The Advanced Encryption Standard (AES) Cipher Algorithm in the SNMP User-based Security Model. RFC 3826, June 2004.

[19] Kathleen Moriarty, Burt Kaliski, Jakob Jonsson, and Andreas Rusch. PKCS #1: RSA Cryptography Specifications Version 2.2. RFC 8017, November 2016.

[20] Yoav Nir and Adam Langley. ChaCha20 and Poly1305 for IETF Protocols. RFC 7539, May 2015.

[21] Colin Percival and Simon Josefsson. The scrypt Password-Based Key Derivation Function. RFC 7914, August 2016.

[22] Joseph A. Salowey, David McGrew, and Abhijit Choudhury. AES Galois Counter Mode (GCM) Cipher Suites for TLS. RFC 5288, August 2008.

[23] Filippo Valsorda. Personal Communication. 25 May 2022.

[24] Filippo Valsorda. Superseded Age Documentation. https://docs.google.com/document/d/11yHom20CrsuX8KQJXBBw04s80Unjv8zCg_A7sPAX_9Y/preview. Accessed: 2022-14-04.

[25] Filippo Valsorda. Using ed25519 signing keys for encryption. Blog Post, 2019, Accessed: 2022-12-05. https://words.filippo.io/using-ed25519-keys-for-encryption/.