Authenticated Encryption with Key Identification

Julia Len[1] Paul Grubbs[2] Thomas Ristenpart[1]

[1] Cornell Tech
[2] University of Michigan

Asiacrypt 2022

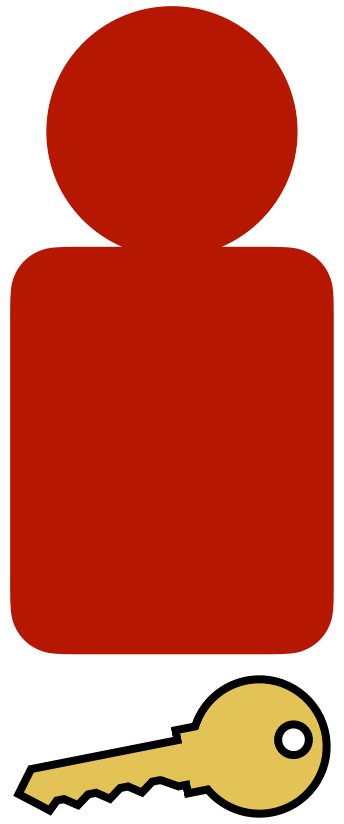
Authenticated Encryption

Nonce N

Associated data AD

Plaintext M

C ← AEAD.Enc(, N, AD, M)



Authenticated Encryption

Nonce N

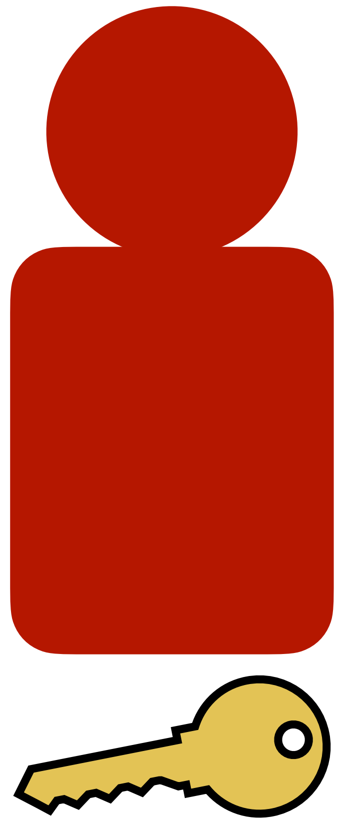
Associated data AD

Plaintext M

C ← AEAD.Enc(, N, AD, M)

N || AD || C

M ← AEAD.Dec(, N, AD, C)



Authenticated Encryption

Nonce N

Associated data AD

Plaintext M

C ← AEAD.Enc(, N, AD, M)

N || AD || C

M ← AEAD.Dec(, N, AD, C)

<u>Assumption #1</u>: The adversary has no way of knowing, or even guessing, the secret key

Authenticated Encryption

Nonce N
Associated data AD
Plaintext M
C ← AEAD.Enc(🔑, N, AD, M)

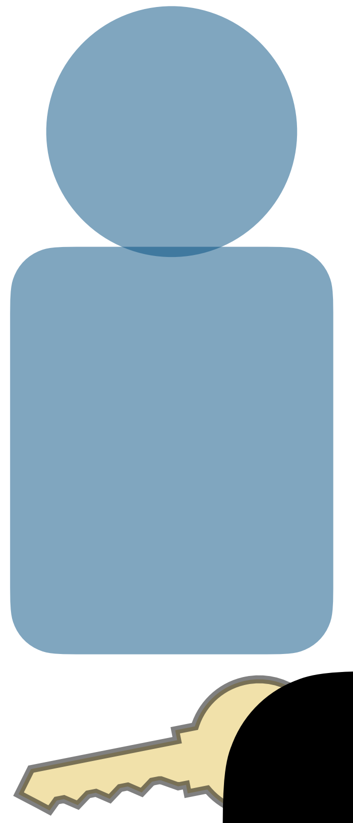
N || AD || C

M ← AEAD.Dec(🔑, N, AD, C)

Assumption #1: The adversary has no way of knowing, or even guessing, the secret key

Assumption #2: Decryption only ever gets a single secret key

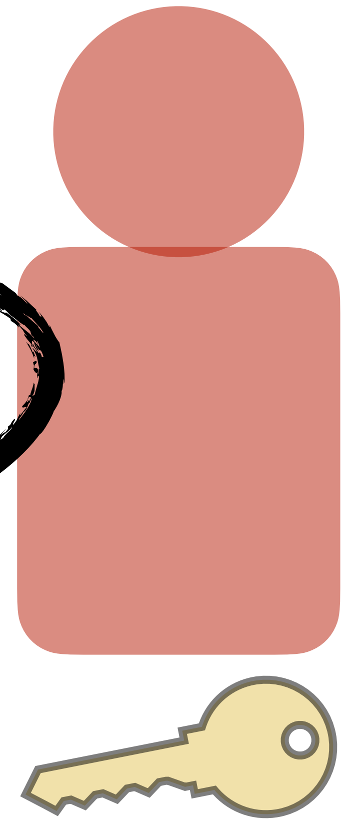
Authenticated Encryption



Nonce N
Associated data AD
Plaintext M
C ← AEAD.Enc(, N, AD, M)

N || AD || C

M ← AEAD.Dec(, N, AD, C)



Assumption #1: The adversary has no way of knowing, or even guessing, the secret key

Assumption #2: Decryption only ever gets a single secret key

! But this model does not always capture how AEAD is used in practice…

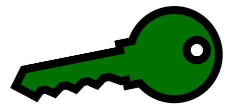
Breaking Assumption #1: Key Robustness

Nonce N′

Associated data AD′

Ciphertext C′

M ← AEAD.Dec( , N′, AD′, C′)

M* ← AEAD.Dec( , N′, AD′, C′)

When Assumptions Fail To Model Practice

<u>Assumption #1</u>: The adversary has no way of knowing, or even guessing, the secret key

Key robustness attacks in practice:

- Facebook Messenger message franking protocol
  [GLR CRYPTO'17], [DGRW CRYPTO'18]
- Partitioning oracle attacks
  [LGR Sec'21]
- Envelope encryption, Subscribe with Google
  [ADGKLS Sec'22]

When Assumptions Fail To Model Practice

<u>Assumption #1</u>: The adversary has no way of knowing, or even guessing, the secret key

Key robustness attacks in practice:

- Facebook Messenger message franking protocol [GLR CRYPTO'17], [DGRW CRYPTO'18]
- Partitioning oracle attacks [LGR Sec'21]
- Envelope encryption, Subscribe with Google [ADGKLS Sec'22]

<u>Assumption #2</u>: Decryption only ever gets a single secret key

Cryptography libraries work with sets of keys:

- Google Tink API

Example: Google Tink

Key Management with Tink

In addition to cryptographic operations Tink provides support for key management features like key versioning, key rotation, and storing keysets or encrypting with master keys in remote key management systems (KMS). To get a quick overview of Tink design, incl. key management features, you can also take a look at slides from a talk about Tink presented at Real World Crypto 2019.

Tinkey is a command-line tool that allows managing Tink's key material. Tink also provides a rich key management API (e.g., see KeysetManager).

Key, Keyset, and KeysetHandle

Tink performs cryptographic tasks via so-called primitives, each of which is defined via a corresponding interface that specifies the functionality of the primitive.

A particular implementation of a *primitive* is identified by a cryptographic **key** structure that contains all key material and parameters needed to provide the functionality of the primitive. The key structure is a *protocol buffer*, whose globally unique name (a.k.a. *type url*) is referred to as **key type**, and is used as an identifier of the corresponding implementation of a primitive. Any particular implementation comes in a form of a **KeyManager** which "understands" the key type: the manager can instantiate the primitive corresponding to a given key, or can generate new keys of the supported key type.

To take advantage of key rotation and other key management features, a Tink user works usually not with single keys, but with **keysets**, which are just sets of keys with some additional parameters and metadata. In particular, this extra information in the keyset determines which key is *primary* (i.e. will be used to create new cryptographic data like ciphertexts, or signatures), which keys are *enabled* (i.e. can be used to process existing cryptographic data, like decrypt ciphertext or verify signatures), and which keys should not be used any more. For more details about the structure of keys, keysets and related protocol buffers see tink.proto.

The keys in a keyset can belong to *different implementations/key types*, but must all implement the *same primitive*. Any given keyset (and any given key) can be used for one primitive only. Moreover, to protect from accidental leakage or corruption, an Tink user doesn't work *directly* with keysets, but rather with **KeysetHandle** objects, which form a wrapper around the keysets. Creation of KeysetHandle objects can be restricted to specific factories (whose visibility can be governed by a white list), to enable control over actual storage of the keys and keysets, and so avoid accidental leakage of secret key material.

https://github.com/google/tink/blob/master/docs/KEY-MANAGEMENT.md

Example: Google Tink

Key Management with Tink

In addition to cryptographic operations Tink provides support for key management features like key versioning, key rotation, and storing keysets or encrypting with master keys in remote key management systems (KMS). To get a quick overview of Tink design, incl. key management features, you can also take a look at slides from a talk about Tink presented at Real World Crypto 2019.

To take advantage of key rotation and other key management features, a Tink user works usually not with single keys, but with **keysets**, which are just sets of keys with some additional parameters and metadata. In particular, this extra information in the keyset determines which key is *primary* (i.e. will be used to create new cryptographic data like ciphertexts, or signatures), which keys are *enabled* (i.e. can be used to process existing cryptographic data, like decrypt ciphertext or verify signatures), and which keys should not be used any more. For more details about the structure of keys, keysets and related protocol buffers see tink.proto.

A particular implementation of a *primitive* is identified by a cryptographic **key** structure that contains all key material and parameters needed to provide the functionality of the primitive. The key structure is a *protocol buffer*, whose globally unique name (a.k.a. *type url*) is referred to as **key type**, and is used as an identifier of the corresponding implementation of a primitive. Any particular implementation comes in a form of a **KeyManager** which "understands" the key type: the manager can instantiate the primitive corresponding to a given key, or can generate new keys of the supported key type.

To take advantage of key rotation and other key management features, a Tink user works usually not with single keys, but with **keysets**, which are just sets of keys with some additional parameters and metadata. In particular, this extra information in the keyset determines which key is *primary* (i.e. will be used to create new cryptographic data like ciphertexts, or signatures), which keys are *enabled* (i.e. can be used to process existing cryptographic data, like decrypt ciphertext or verify signatures), and which keys should not be used any more. For more details about the structure of keys, keysets and related protocol buffers see tink.proto.

The keys in a keyset can belong to *different implementations/key types*, but must all implement the *same primitive*. Any given keyset (and any given key) can be used for one primitive only. Moreover, to protect from accidental leakage or corruption, an Tink user doesn't work *directly* with keysets, but rather with **KeysetHandle** objects, which form a wrapper around the keysets. Creation of KeysetHandle objects can be restricted to specific factories (whose visibility can be governed by a white list), to enable control over actual storage of the keys and keysets, and so avoid accidental leakage of secret key material.

https://github.com/google/tink/blob/master/docs/KEY-MANAGEMENT.md

Example: Google Tink

Key Management with Tink

In addition to cryptographic operations Tink provides support for key management features like key versioning, key rotation, and storing keysets or encrypting with master keys in remote key management systems (KMS). To get a quick overview of Tink design, incl. key management features, you can also take a look at slides from a talk about Tink presented at Real World Crypto 2019.

To take advantage of key rotation and other key management features, a Tink user works usually not with single keys, but with **keysets**, which are just sets of keys with some additional parameters and metadata. In particular, this extra information in the keyset determines which key is *primary* (i.e. will be used to create new cryptographic data like ciphertexts, or signatures), which keys are *enabled* (i.e. can be used to process existing cryptographic data, like decrypt ciphertext or verify signatures), and which keys should not be used any more. For more details about the structure of keys, keysets and related protocol buffers see tink.proto.

Tin

Key

Ke

Tink

func

A particular implementation of a *primitive* is identified by a cryptographic **key** structure that contains all key material and parameters needed to provide the functionality of the primitive. The key structure is a *protocol buffer*, whose globally unique name (a.k.a. *type url*) is referred to as **key type**, and is used as an identifier of the corresponding implementation of a primitive. Any particular implementation comes in a form of a **KeyManager** which "understands" the key type: the manager can instantiate the primitive corresponding to a given key, or can generate new keys of the supported key type.

To take advantage of key rotation and other key management features, a Tink user works usually not with single keys, but with **keysets**, which are just sets of keys with some additional parameters and metadata. In particular, this extra information in the keyset determines which key is *primary* (i.e. will be used to create new cryptographic data like ciphertexts, or signatures), which keys are *enabled* (i.e. can be used to process existing cryptographic data, like decrypt ciphertext or verify signatures), and which keys should not be used any more. For more details about the structure of keys, keysets and related protocol buffers see tink.proto.

The keys in a keyset can belong to *different implementations/key types*, but must all implement the *same primitive*. Any given keyset (and any given key) can be used for one primitive only. Moreover, to protect from accidental leakage or corruption, an Tink user doesn't work *directly* with keysets, but rather with **KeysetHandle** objects, which form a wrapper around the keysets. Creation of KeysetHandle objects can be restricted to specific factories (whose visibility can be governed by a white list), to enable control over actual storage of the keys and keysets, and so avoid accidental leakage of secret key material.

How does decryption know which key to use?

https://github.com/google/tink/blob/master/docs/KEY-MANAGEMENT.md

Example: Google Tink

- Tink adds a 5-byte prefix to each ciphertext which acts as a key identifier
- Tink will try to decrypt the key specified by the identifier first
- If decryption fails, Tink will attempt trial decrypting with "raw" keys (keys without identifiers) until it finds a key that successfully decrypts

https://developers.google.com/tink/wire-format

When Assumptions Fail To Model Practice

Assumption #1: The adversary has no way of knowing, or even guessing, the secret key

Key robustness attacks in practice:

- Facebook Messenger message franking protocol [GLR CRYPTO'17], [DGRW CRYPTO'18]
- Partitioning oracle attacks [LGR Sec'21]
- Envelope encryption, Subscribe with Google [ADGKLS Sec'22]

Assumption #2: Decryption only ever gets a single secret key

Cryptography libraries work with sets of keys:

- Google Tink API

Attacks on sets of keys:

- Multi-user Shadowsocks [LGR Sec'21]
- Key management services [ADGKLS Sec'22]

When Assumptions Fail To Model Practice

<u>Assumption #1</u>: The adversary has no way of knowing, or even guessing, the secret key

Key robustness attacks in practice:

- Facebook Messenger message franking protocol [GLR CRYPTO'17], [DGRW CRYPTO'18]
- Partitioning oracle attacks [LGR Sec'21]
- Envelope encryption, Subscribe with Google [ADGKLS Sec'22]

<u>Assumption #2</u>: Decryption only ever gets a single secret key

Cryptography libraries work with sets of keys:

- Google Tink API

Attacks on sets of keys:

- Multi-user Shadowsocks [LGR Sec'21]
- Key management services [ADGKLS Sec'22]

It's unclear what security properties these approaches achieve

Our Contributions

- Initiate the formal study of AEAD that supports key identification by extending nonce-based AEAD into this setting
- Formalize a new cryptographic primitive called AEAD with key identification (AEAD-KI)
- Introduce new security definitions for the AEAD-KI setting
  - ➡ Our definitions allow an adversary to specify malicious keys to better model possible attacks
- Analyze security of existing key identification approaches and suggest new ones

Our Contributions

- Initiate the formal study of AEAD that supports key identification by extending nonce-based AEAD into this setting
- Formalize a new cryptographic primitive called AEAD with key identification (AEAD-KI)
- Introduce new security definitions for the AEAD-KI setting
  - ➡ Our definitions allow an adversary to specify malicious keys to better model possible attacks
- Analyze security of existing key identification approaches and suggest new ones

Our Contributions

- Initiate the formal study of AEAD that supports key identification by extending nonce-based AEAD into this setting
- Formalize a new cryptographic primitive called AEAD with key identification (AEAD-KI)
- Introduce new security definitions for the AEAD-KI setting
  - ➡ Our definitions allow an adversary to specify malicious keys to better model possible attacks
- Analyze security of existing key identification approaches and suggest new ones

Our Contributions

- Initiate the formal study of AEAD that supports key identification by extending nonce-based AEAD into this setting
- Formalize a new cryptographic primitive called AEAD with key identification (AEAD-KI)
- Introduce new security definitions for the AEAD-KI setting
  - ➡ Our definitions allow an adversary to specify malicious keys to better model possible attacks
- Analyze security of existing key identification approaches and suggest new ones

AEAD with Key Identification

AEAD with Key Identification

K ← AEKI.Kg(kid)

Key generation takes as input a key label
These can be public, application-defined strings (e.g. URIs)

The key is a key label - secret key tuple

AEAD with Key Identification

$K \leftarrow \text{AEKI.Kg}(kid)$

Key generation takes as input a key label
These can be public, application-defined strings (e.g. URIs)

The key is a key label - secret key tuple

$(T_k, C) \leftarrow \text{AEKI.Enc}(K, N, AD, M)$

Encryption can now output a special key tag as part of the ciphertext to help with decryption

AEAD with Key Identification

$K \leftarrow \text{AEKI.Kg}(kid)$

Key generation takes as input a key label
These can be public, application-defined strings (e.g. URIs)

The key is a key label - secret key tuple

$(T_k, C) \leftarrow \text{AEKI.Enc}(K, N, AD, M)$

Encryption can now output a special key tag as part of the ciphertext to help with decryption

$(K, M) / \perp \leftarrow \text{AEKI.Dec}(\mathbb{K}, N, AD, T_k, C)$

Decryption takes in a vector of keys, which preserves information about the order

If decryption succeeds, it returns the key that produced the resulting plaintext, in addition to the plaintext
Otherwise, it returns the special error symbol ⊥

AEAD-KI Correctness

Recall: An AEAD scheme is correct if for any (K, N, AD, M), it holds that

$$\mathsf{Dec}(K, N, AD, \mathsf{Enc}(K, N, AD, M)) = M$$

with probability 1 over the coins used in encryption.

AEAD-KI Correctness

Recall: An AEAD scheme is correct if for any (K, N, AD, M), it holds that

$$\mathsf{Dec}(K, N, AD, \mathsf{Enc}(K, N, AD, M)) = M$$

with probability 1 over the coins used in encryption.

But what does a "correct" AEAD-KI scheme mean?

AEAD-KI Correctness

Recall: An AEAD scheme is correct if for any (K, N, AD, M), it holds that

$$\mathsf{Dec}(K, N, AD, \mathsf{Enc}(K, N, AD, M)) = M$$

with probability 1 over the coins used in encryption.

But what does a "correct" AEAD-KI scheme mean?

First attempt: An AEAD-KI scheme is correct if for any $K, \mathbb{K}, N, AD, M$ where $K \in \mathbb{K}$, it holds that

$$\mathsf{Dec}(\mathbb{K}, N, AD, \mathsf{Enc}(K, N, AD, M)) = (K, M)$$

with probability 1 over the coins used in encryption.

➡ **Problem:** There could be another key in $\mathbb{K}$ that can decrypt the ciphertext so this cannot be an information theoretic guarantee.

AEAD-KI Correctness

Recall: An AEAD scheme is correct if for any (K, N, AD, M), it holds that

with probabil

- We expect an AEAD-KI scheme to function correctly if it returns the correct key but this requires a computational definition
- We therefore provide a simpler, absolute correctness definition and rely on a key robustness definition to model this behavior

But what doe

First attemp , it holds that

with probability 1 over the coins used in encryption.

➡ **Problem:** There could be another key in $\mathbb{K}$ that can decrypt the ciphertext so this cannot be an information theoretic guarantee.

AEAD-KI Correctness

An AEAD-KI scheme is correct if the following hold:

AEAD-KI Correctness

An AEAD-KI scheme is correct if the following hold:

(1) For any (K, N, AD, M) it holds that Pr[$(K', M') = (K, M)$] $= 1$ where

$$(K', M') \leftarrow \mathsf{Dec}([K], N, AD, \mathsf{Enc}(K, N, AD, M))$$

and the probability is over the coins used by encryption

➡ Translates traditional AEAD correctness to syntax of AEAD-KI for decryption with a single key

AEAD-KI Correctness

An AEAD-KI scheme is correct if the following hold:

(1) For any (K, N, AD, M) it holds that Pr[$(K', M') = (K, M)$] $= 1$ where

$$(K', M') \leftarrow \text{Dec}([K], N, AD, \text{Enc}(K, N, AD, M))$$

and the probability is over the coins used by encryption

- ➡ Translates traditional AEAD correctness to syntax of AEAD-KI for decryption with a single key

(2) For any $(\mathbb{K}, N, AD, T_k, C)$ and $(K, M) \leftarrow \text{Dec}(\mathbb{K}, N, AD, T_k, C)$ it must be that either $(K, M) = \bot$ or $K \in \mathbb{K}$

- ➡ Decryption should only output a key that was in the key vector

AEAD-KI Correctness

An AEAD-KI scheme is correct if the following hold:

(1) For any (K, N, AD, M) it holds that Pr[$(K', M') = (K, M)$] $= 1$ where

$$(K', M') \leftarrow \text{Dec}([K], N, AD, \text{Enc}(K, N, AD, M))$$

and the probability is over the coins used by encryption

➡ Translates traditional AEAD correctness to syntax of AEAD-KI for decryption with a single key

(2) For any $(\mathbb{K}, N, AD, T_k, C)$ and $(K, M) \leftarrow \text{Dec}(\mathbb{K}, N, AD, T_k, C)$ it must be that either $(K, M) = \perp$ or $K \in \mathbb{K}$

➡ Decryption should only output a key that was in the key vector

(3) For any $\mathbb{K}, \mathbb{K}'$ and any (N, AD, T_k, C), let

$$(K, M) \leftarrow \text{Dec}(\mathbb{K}, N, AD, T_k, C), (K', M') \leftarrow \text{Dec}(\mathbb{K}', N, AD, T_k, C) .$$

If $(K, M) \neq \perp$ and $K \in \mathbb{K}'$, then $(K', M') \neq \perp$

➡ If decryption of (N, AD, T_k, C) outputs key K, any other key vector containing K should not fail to decrypt (N, AD, T_k, C)

Key Robustness in the AEAD-KI Setting

Full robustness (KI-FROB)

$\mathbb{K}_0$ $\mathbb{K}_1$

N, AD, T_k, C

The adversary wins if:

$((kid_0, K_0), M_0) \leftarrow$ AEKI.Dec($\mathbb{K}_0$, N, AD, T_k, C)

$((kid_1, K_1), M_1) \leftarrow$ AEKI.Dec($\mathbb{K}_1$, N, AD, T_k, C)

such that

- Decryption is successful under both key vectors
- $K_0 \neq K_1$

Key Robustness in the AEAD-KI Setting

Full robustness (KI-FROB)

$\mathbb{K}_0$ $\mathbb{K}_1$

N, AD, T_k, C

The adversary wins if:

$((kid_0, K_0), M_0) \leftarrow \text{AEKI.Dec}(\mathbb{K}_0, N, AD, T_k, C)$

$((kid_1, K_1), M_1) \leftarrow \text{AEKI.Dec}(\mathbb{K}_1, N, AD, T_k, C)$

such that

- Decryption is successful under both key vectors
- $K_0 \neq K_1$

- Key robustness guarantees that only a single key can be used to decrypt a given ciphertext

Key Robustness in the AEAD-KI Setting

Full robustness (KI-FROB)

$\mathbb{K}_0$ $\mathbb{K}_1$

N, AD, T_k, C

The adversary wins if:

$((kid_0, K_0), M_0) \leftarrow \text{AEKI.Dec}(\mathbb{K}_0, N, AD, T_k, C)$

$((kid_1, K_1), M_1) \leftarrow \text{AEKI.Dec}(\mathbb{K}_1, N, AD, T_k, C)$

such that

- Decryption is successful under both key vectors
- $K_0 \neq K_1$

- Key robustness guarantees that only a single key can be used to decrypt a given ciphertext
- Functions partly as *correctness* in this setting: we expect that the key used to encrypt a plaintext should be the only one to correctly decrypt the resulting ciphertext

Key Robustness in the AEAD-KI Setting

Full robustness (KI-FROB)

$\mathbb{K}_0$ $\mathbb{K}_1$

N, AD, T_k, C

The adversary wins if:

$((kid_0, K_0), M_0) \leftarrow$ AEKI.Dec($\mathbb{K}_0$, N, AD, T_k, C)

$((kid_1, K_1), M_1) \leftarrow$ AEKI.Dec($\mathbb{K}_1$, N, AD, T_k, C)

such that

- Decryption is successful under both key vectors
- $K_0 \neq K_1$

- Key robustness guarantees that only a single key can be used to decrypt a given ciphertext
- Functions partly as *correctness* in this setting: we expect that the key used to encrypt a plaintext should be the only one to correctly decrypt the resulting ciphertext
- If different orderings of the same key vector cause different keys to be output, these two key vectors would give a KI-FROB win

All-in-one confidentiality and integrity (KI-nAE)

- We extend indistinguishability style security definitions for AEAD to AEAD-KI
- Our KI-nAE definition captures confidentiality and integrity in this setting
- We allow adversaries to query the decryption oracle with a key vector that includes honest or malicious keys, in any order, to better capture the setting

  → We use a simulator-based definition to model this

- We also specify a key-anonymous version called KI-nAE-KA

All-in-one confidentiality and integrity (KI-nAE)

- **Adversary goal**: distinguish between real and simulated world
- Adversary has access to oracles to generate honest keys, encrypt, and decrypt

KI-nAE1: Real world game

models interactions with the real scheme

KI-nAE0: Ideal world game

uses stateful simulator to generate oracle outputs

Ideal game KI-nAE0

Encryption

- Non-key anonymous leakage L^{id}: simulator receives both the game-generated key identifier and the plaintext size
- Key anonymous leakage L^{anon}: simulator receives only the plaintext size

Ideal game KI-nAE0

Encryption

- Non-key anonymous leakage L^{id}: simulator receives both the game-generated key identifier and the plaintext size
- Key anonymous leakage L^{anon}: simulator receives only the plaintext size

Decryption

- First scans through *honest keys* in queried key vector
  - ➡ If the key and ciphertext were output from a prior call to the encryption oracle, then the associated plaintext is returned

Ideal game KI-nAE0

Encryption

- Non-key anonymous leakage L^{id}: simulator receives both the game-generated key identifier and the plaintext size
- Key anonymous leakage L^{anon}: simulator receives only the plaintext size

Decryption

- First scans through *honest keys* in queried key vector
  - ➡ If the key and ciphertext were output from a prior call to the encryption oracle, then the associated plaintext is returned
- Otherwise, if there are *malicious keys* in the key vector, the simulator is given the ciphertext and remaining malicious keys to decrypt
- This allows our definition to imply a variant of INT-CTXT for this setting

Approaches to AEAD-KI

We divide key identification into several categories and analyze their security

Approach	Description
Key labels	
Trial decryption	
Static key hint	
Static key commitment	
Dynamic key hint	
Dynamic key commitment	

Key Labels

- **Approach**: Assign each key a static label, then prepend the label to each ciphertext it produces
- Parameterized by AEAD scheme
- Examples of labels
  - Google Tink: 1-byte library version + 4-byte randomly generated string
  - AWS KMS: URL indicating where to fetch the key (URI)

Enc(K, N, AD, M)
(kid, K*) ← K
C ← AEAD.Enc(K*, N, AD || kid, M)
Return (kid, C)

Dec($\mathbb{K}$, N, AD, T_k, C)
For (kid, K) in $\mathbb{K}$:
  If kid = T_k:
    M ← AEAD.Dec(K, N, AD || kid, C)
    If M ≠ ⊥: Return ((kid, K), M)
Return ⊥

Key Labels: Analyzing security

Enc(K, N, AD, M)
(kid, K*) ← K
C ← AEAD.Enc(K*, N, AD || kid, M)
Return (kid, C)

Dec($\mathbb{K}$, N, AD, T_k, C)
For (kid, K) in $\mathbb{K}$:
  If kid = T_k:
    M ← AEAD.Dec(K, N, AD || kid, C)
    If M ≠ ⊥: Return ((kid, K), M)
Return ⊥

- Key labels are KI-FROB-secure iff AEAD is key robust
  - **Note**: Key labels might not be unique and cannot be used for key commitment
- Key labels are KI-nAE-secure if AEAD is key robust and multi-user AE-secure
  - **Note**: Ciphertexts are prepended with static strings, so there is no key anonymity
  - Trial decryption, a scheme where key labels are empty, is a key anonymous alternative

Static Key Identifiers

- **Approach**: Compute a static identifier from the key and use this with the key label as the key tag
- Also known as a *key check value* in practice
- Parameterized by key check value function F_{kcv}, encryption key derivation function KDF (where KDF $\neq F_{kcv}$), and an AEAD scheme

Enc(K, N, AD, M)

(kid, K^*) $\leftarrow$ K

kcv $\leftarrow F_{kcv}(K^*)$; $K_e \leftarrow$ KDF(K^*)

$T_k \leftarrow$ kid || kcv

C $\leftarrow$ AEAD.Enc(K_e, N, AD || T_k, M)

Return (T_k, C)

Dec($\mathbb{K}$, N, AD, T_k, C)

For (kid, K) in $\mathbb{K}$:

  kcv $\leftarrow F_{kcv}$(K) ; $K_e \leftarrow$ KDF(K)

  If kid || kcv = T_k:

    M $\leftarrow$ AEAD.Dec(K_e, N, AD || T_k, C)

    If M $\neq \perp$: Return ((kid, K), M)

Return $\perp$

Static Key Hints vs. Static Key Commitments

Static Key Hints

- Key check value computed using PRF
- Can be short and efficiently computed
- **<u>Cannot</u>** be used to commit to a key, so they need to be used with an key robust AEAD scheme to achieve KI-FROB
- No key anonymity since it is static
- Examples
  - GlobalPlatform: $msb_{24}(AES_K(([1]_8)^{16}))$
  - Telegram: $lsb_{64}(SHA1(K))$
  - PKCS#11: $msb_{24}(AES_K(0^{128}))$

Static Key Commitments

- Key check value computed using collision-resistant PRF
- Longer and less efficient to compute than key hints
- **<u>Can</u>** be used to commit to a key, so they can be used with non-key robust AEAD schemes to achieve KI-FROB
- No key anonymity since it is static
- Example
  - AWS Encryption SDK:
    SHA256(K || `0x436f6d6d69740102`)

Dynamic Key Identifiers

- **Approach**: Compute a dynamic identifier from the key and a nonce and use this as the key tag
- This is the *key anonymous* counterpart to Static Key Identifiers
- To preserve anonymity, key labels are empty
- Parameterized by key check value function F_{kcv}, encryption key derivation function KDF, and an AEAD scheme

Enc(K, N, AD, M)

$(\varepsilon, K^*) \leftarrow K$; $(N_0, N_1) \leftarrow N$

$kcv \leftarrow F_{kcv}(K^*, N_0)$; $K_e \leftarrow KDF(K^*)$

$T_k \leftarrow kcv$

$C \leftarrow AEAD.Enc(K_e, N_1, AD \| T_k, M)$

Return (T_k, C)

Dec($\mathbb{K}$, N, AD, T_k, C)

$(N_0, N_1) \leftarrow N$

For (ε, K) in $\mathbb{K}$:

  $kcv \leftarrow F_{kcv}(K, N_0)$; $K_e \leftarrow KDF(K)$

  If $kcv = T_k$:

    $M \leftarrow AEAD.Dec(K_e, N_1, AD \| T_k, C)$

    If $M \neq \perp$: Return $((\varepsilon, K), M)$

Return $\perp$

Dynamic Key Hints vs. Dynamic Key Commitments

Dynamic Key Hints

- Computed using PRF
- Can be short and efficiently computed
- **Cannot** be used to commit to a key, so they need to be used with a key robust AEAD scheme to achieve KI-FROB
- Key anonymity!

Dynamic Key Commitments

- Computed using collision-resistant PRF
- Longer and less efficient to compute than key hints
- **Can** be used to commit to a key, so they can be used with non-key robust AEAD schemes to achieve KI-FROB
- Key anonymity!

FROB = full key robustness

Approaches to AEAD-KI

We divide key identification into several categories and analyze their security

Approach	Description	AEAD FROB?	Key anonymous?
Key labels	Key generation labels each key, sent as part of ciphertext; brute-force decrypt w/ all keys matching key label in ciphertext	✓	✗
Trial decryption	Special case of key labels where all labels are empty	✓	✓
Static key hint	Ciphertext includes deterministic non-CR hash of key	✓	✗
Static key commitment	Ciphertext includes deterministic CR hash of key	✗	✗
Dynamic key hint	Ciphertext includes PRF of key & nonce	✓	✓
Dynamic key commitment	Ciphertext includes CR PRF of key & nonce	✗	✓

Conclusion

jlen@cs.cornell.edu

Full version: eprint 2022/1680

- The current model for AEAD does not always capture how AEAD is used in practice
- Introduce Authenticated Encryption with Key Identification, which allows key identification during the decryption step
- Introduce new security definitions for the AEAD-KI setting
- Analyze security of existing key identification approaches and suggest new ones

References

[ADGKLS Sec’22] Ange Albertini, Thai Duong, Shay Gueron, Stefan Kölbl, Atul Luykx, Sophie Schmieg. How to abuse and fix authenticated encryption without key commitment. USENIX Security, 2022.

[DGRW CRYPTO’18] Yevgeniy Dodis, Paul Grubbs, Thomas Ristenpart, Joanne Woodage. Fast message franking: From invisible salamanders to encryptment. CRYPTO, 2018.

[FOR FSE’17] Pooya Farshim, Claudio Orlandi, Răzvan Roşie. Security of symmetric primitives under incorrect usage of keys. FSE, 2017.

[GLR CRYPTO’17] Paul Grubbs, Jiahui Lu, Thomas Ristenpart. Message franking via committing authenticated encryption. CRYPTO, 2017.

[LGR Sec’21] Julia Len, Paul Grubbs, Thomas Ristenpart. Partitioning Oracle Attacks. USENIX Security, 2021.