Google

ISE-Crypto

Moreno Ambrosin
ambrosin@google.com

March, 2023

Motivation: Cryptographic libraries are tricky to use

- Often expose low-level APIs that require in-depth expertise
    - Developers shouldn't need to focus on cryptography...
- Simple mistakes can have serious consequences

```
int EVP_EncryptInit_ex(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
                       ENGINE *impl, const unsigned char *key, const unsigned char *iv);
int EVP_EncryptUpdate(EVP_CIPHER_CTX *ctx, unsigned char *out,
                      int *outl, const unsigned char *in, int inl);
int EVP_EncryptFinal_ex(EVP_CIPHER_CTX *ctx, unsigned char *out, int *outl);
int EVP_CIPHER_CTX_ctrl(EVP_CIPHER_CTX *ctx, int type, int arg, void *ptr);
```


Google

Outline

01 What is Tink?

02 Tink concepts

03 Key management

04 Next Steps

05 Q&A

What is Tink?

- A **multi language** and **multi-platform open source** cryptography library
  - github.com/google/tink
  - Documentation: developers.google.com/tink
  - Used by Google Cloud customer, Jetpack Security library, etc.
- **Design goals**:
  - Secure and easy to use APIs
    - Hard to misuse, hide low-level details
  - Support for key management
  - Extensible

What is Tink? (Cont.)

- Currently implemented in
  - Java, C++, Python, Go, Obj-C
- Built on top of standard and/or established crypto libraries
  - BoringSSL/OpenSSL (C++)
  - BoringSSL (Python, Obj-C)
  - Java JCE/Conscrypt
  - crypto and x/crypto (Go)

Tink concepts - Primitive

- Abstract **cryptographic functionality**
- Defines the functionality at a high-level and its security properties

```
class Aead(metaclass=abc.ABCMeta):

  @abc.abstractmethod
  def encrypt(self, plaintext: bytes, associated_data: bytes) -> bytes:
    # ...

  @abc.abstractmethod
  def decrypt(self, ciphertext: bytes, associated_data: bytes) -> bytes:
    # ...
```


Google

Example: AEAD encrypt

```
import tink
from tink import aead

# Read or create a key material.
keyset_handle = ...

# Obtain an AEAD primitive.
aead_primitive = keyset_handle.primitive(aead.Aead)

# Use the primitive to encrypt.
ciphertext = aead_primitive.encrypt(plaintext, associated_data)
```


Google

Tink concepts - Key

- **Key material** and **metadata** (parameters and algorithm)
  - Identified by a **type URL**, e.g.,

    `type.googleapis.com/google.crypto.tink.AesGcmKey`
  - E.g., a Tink AEAD key specifies:
    - How the plaintext is encrypted and encoded
    - How a ciphertext is decrypted
- In Tink an AES-EAX key != AES-GCM key

Google

Tink concepts - Key Manager

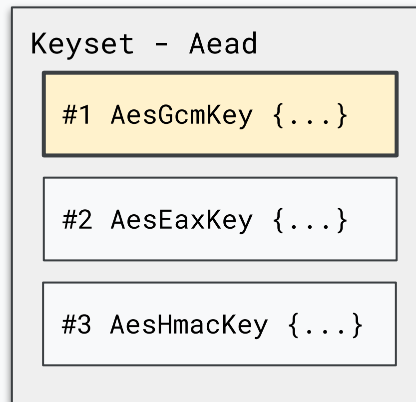
- A **key manager** is a class that creates primitives from keys
- Tink uses a **registry** to store available key managers
  - Users must initialize it with built-in key managers and/or add custom ones

```
class AesGcmKeyManager(core.KeyManager[aead.Aead]):
  def primitive(self, key_data: tink_pb2.AesGcmKey) -> aead.Aead:
    # Create primitive that implements AES-GCM with the given key.

  def key_type(self) -> str:
    return "type.googleapis.com/google.crypto.tink.AesGcmKey"

  # ...
```


Google

Tink concepts - Keyset

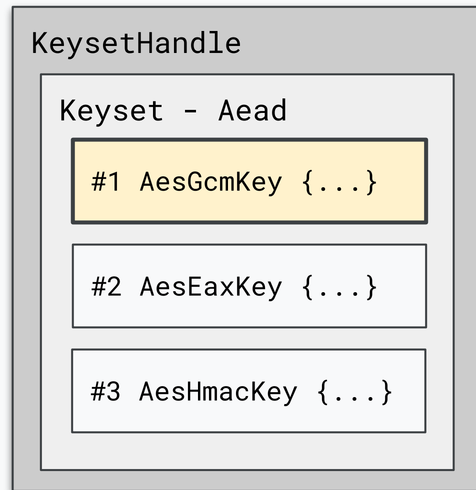
- A set of keys that implements **the same primitive**
- It facilitates **key rotation**
- Each key has a **unique ID** (within a keyset)
    - Usually prefix to produced ciphertexts, signatures, tags
- Only one key at a time is **primary**
    - Used to e.g., encrypt or sign



Google

Tink concepts - Keyset handle

- A **keyset handle** is a wrapper around a keyset
- Restricts access to sensitive data
- Provides APIs to obtain a “wrapping” primitive for the keyset
- E.g., for `Aead`:
  - `encrypt(...)` uses the primary key
  - `decrypt(...)` uses the key whose ID is in the ciphertext



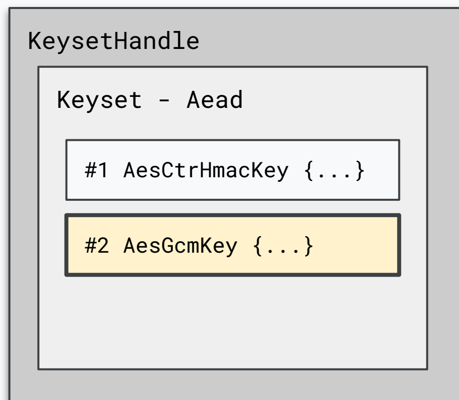
Google

Example: AEAD encrypt

```
import tink
from tink import aead

# Make all the AEAD primitives available.
aead.register()

# Create a keyset with a single key and get a handle to it.
keyset_handle = tink.new_keyset_handle(aead.aead_key_templates.AES128_GCM)

# Wrap the keyset into an AEAD primitive.
aead_primitive = keyset_handle.primitive(aead.Aead)

# Use the primitive to encrypt (uses the primary key!).
ciphertext = aead_primitive.encrypt(plaintext, associated_data)
```


Google

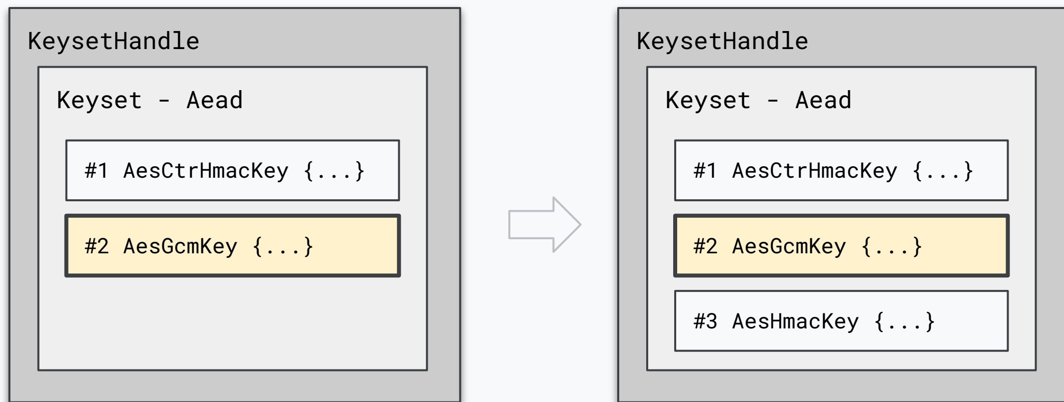
Key management - Key rotation with keysets



Key #2 is primary key

Google

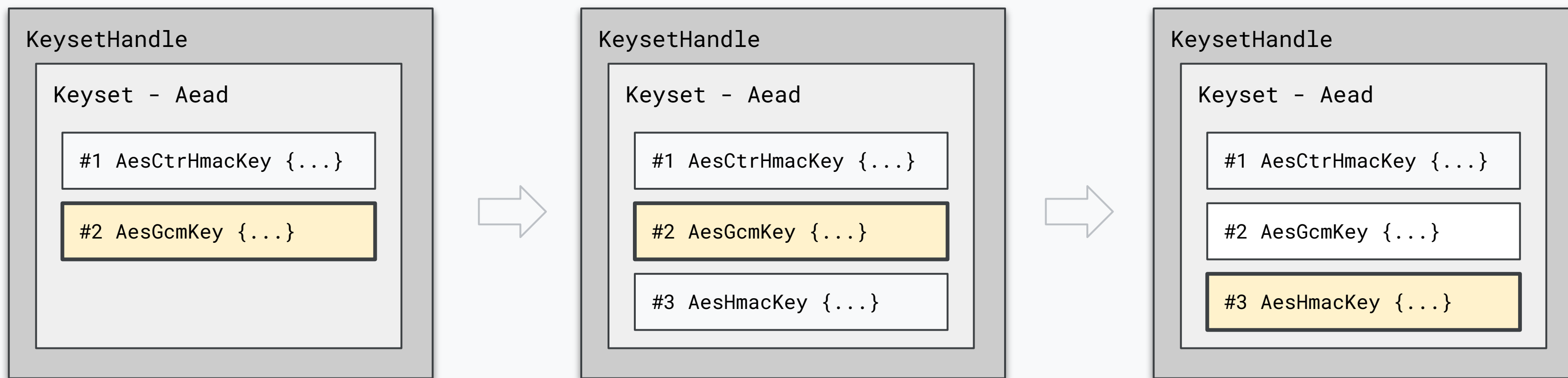
Key management - Key rotation with keysets



Key #2 is primary key

Key #2 is primary key
Key #3 is added

Key management - Key rotation with keysets



Key #2 is primary key

Key #2 is primary key
Key #3 is added

Key #3 is primary key

Google

Key management - KMS support

- Tink uniformly handles external keys
- For example, Tink allows getting AEAD primitive form a KMS key
  - KMS AEAD key `KmsAeadKey` is “just another key type”
    - Simply “points” to the KMS key with its URI
  - `KmsAeadKeyManager` construct AEAD from the key URI
    - Using KMS-specific clients, such as `GcpKmsClient`.

Google

Example: AEAD Encrypt with KMS

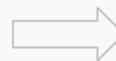
```
import tink
from tink import aead
from tink.integration import gcpkms

# Register a KMS client that is bound to kms_key.
gcpkms.GcpKmsClient.register_client(kms_key, credential_path)

# Key template for the key URI.
kms_key_template = aead.aead_key_templates.create_kms_aead_key_template(kek_uri)
# Create a keyset with a single KMS key and get a handle to it.
keyset_handle = tink.new_keyset_handle(kms_key_template)

# Wrap the keyset into an AEAD primitive.
aead_primitive = keyset_handle.primitive(aead.Aead)
# Use the KMS key to encrypt.
ciphertext = aead_primitive.encrypt(plaintext, associated_data)
```

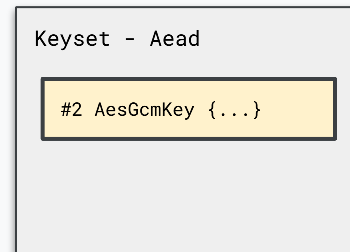

Google

Example: Encrypt keyset with KMS and serialize it

```
import tink
from tink import aead
from tink.integration import gcpkms

# Register a KMS client that is bound to kms_key.
gcpkms.GcpKmsClient.register_client(kms_key, credential_path)

# Key template for the key URI.
kms_key_template = aead.aead_key_templates.create_kms_aead_key_template(kek_uri)
# Create a keyset with a single KMS key and get a handle to it.
keyset_handle = tink.new_keyset_handle(kms_key_template)

# Wrap the keyset into an AEAD primitive.
aead_primitive = keyset_handle.primitive(aead.Aead)
# Encrypt the keyset with the KMS key and serialize as JSON.
keyset_handle_to_encrypt.write_with_associated_data(
    tink.JsonKeysetWriter(text_io_stream), aead_primitive, associated_data)
```

Key management - The Tinkey CLI tool



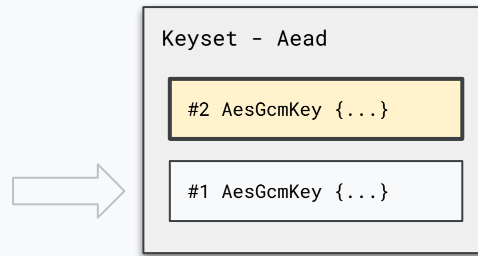


- CLI tool for managing keysets w/ KMS integration

```
readonly KEK_KMS_KEY_URI="gcp-kms://..."
readonly KMS_CREDENTIALS_FILE_PATH="credentials.json"

# Create a keyset with one AES128-GCM key, encrypts it with a KMS key and outputs to file.
tinkey create-keyset --key-template AES128_GCM --out encrypted-keyset.json \
  --master-key-uri "${KEK_KMS_KEY_URI}" --credential "${KMS_CREDENTIALS_FILE_PATH}"
```


Google

Key management - The Tinkey CLI tool



- CLI tool for managing keysets w/ KMS integration

```
readonly KEK_KMS_KEY_URI="gcp-kms://..."
readonly KMS_CREDENTIALS_FILE_PATH="credentials.json"

# Create a keyset with one AES128-GCM key, encrypts it with a KMS key and outputs to file.
tinkey create-keyset --key-template AES128_GCM --out encrypted-keyset.json \
  --master-key-uri "${KEK_KMS_KEY_URI}" --credential "${KMS_CREDENTIALS_FILE_PATH}"

# Add non-primary key to the keyset; outputs encrypted with the KMS key to a new file as JSON.
tinkey add-key --key-template AES256_GCM \
  --in encrypted-keyset.json --out new-encrypted-keyset.json \
  --master-key-uri "${KEK_KMS_KEY_URI}" --credential "${KMS_CREDENTIALS_FILE_PATH}"
```

Next steps

- Splitting into multiple repos and migrate to github.com/tink-crypto
  - Decouple versions
  - Cross-language compatibility documented
- New APIs (WIP)
  - Access to individual keys
  - Improve configurability
  - Monitoring hooks
- Overhaul documentation

Takeaways

- Tink provides high-level easy to use API
- Tink is multi-language and multi platform
- Tink provides support/tooling for key management
- We use Tink internally at Google but is also open source
  - github.com/google/tink
  - Contributions are welcome!