

---

## What is Conceal? build unknown

Conceal provides a set of Java APIs to perform cryptography on Android. It was designed to be able to encrypt large files on disk in a fast and memory efficient manner.

The major target for this project is typical Android devices which run old Android versions, have low memory and slower processors.

Unlike other libraries, which provide a Smorgasbord of encryption algorithms and options, Conceal prefers to abstract this choice and use sane defaults. Thus Conceal is not a general purpose crypto library, however it aims to provide useful functionality.

**Upgrading version?** Check the Upgrade notes for key compatibility!

**IMPORTANT: Initializing the library loader** Since v2.0.+ (2017-06-27) you will need to initialize the native library loader. This step is needed because the library loader uses the context. The highly suggested way to do it is in the application class onCreate method like this:

```
1 import com.facebook.soloader.SoLoader;
2 public class MyApplication extends Application {
3     @Override
4     public void onCreate() {
5         super.onCreate();
6         SoLoader.init(this, false);
7     }
8 }
```

## Quick start

### Setup options

1. **Use Maven Central:** Available on maven central under **com.facebook.conceal:conceal:2.0.1@aar** as an AAR package. If you use Android Studio and select the library using the UI, **make sure** to change **build.gradle to include the @aar suffix**. Otherwise the library won't be included.
2. **Build using gradle**

```
1 ./gradlew build
```

It uses gradlew so it takes care of downloading Gradle and all the dependencies it needs. Output will be in `/build/outputs/aar/` directory.

3. **Use prebuilt binaries:** <http://facebook.github.io/conceal/documentation/>. (linked documentation needs update)

---

## An aside on KitKat

Conceal predates Jellybean 4.3. On KitKat, Android changed the provider for cryptographic algorithms to OpenSSL. The default Cipher stream however still does not perform well. When replaced with our Cipher stream (see `BetterCipherInputStream`), the default implementation is competitive against Conceal. On older phones, Conceal is faster than the system provided libraries.

**Re-build OpenSSL library** You can run make from the openssl directory. It will download the code and compile the libraries for each architecture.

```
1 # go to /third-party/openssl
2 make
```

**Before running any test!** Test uses BUCK build tool. BUCK uses the source code for OpenSSL. If you didn't already rebuilt OpenSSL from scratch (previous item) then run this:

```
1 # go to /third-party/openssl
2 make clone
```

That will download the OpenSSL code to a subdirectory.

### Running unit tests

```
1 # C++ tests
2 buck test :cpp
```

### Running integration tests

```
1 # Emulator/device tests
2 ./instrumentTest/crypto/run
```

Since Conceal uses native libraries, the only way to run a test on the entire encryption process is using integration tests.

### Running Benchmarks

```
1 ./benchmarks/run \
2   benchmarks/src/com/facebook/crypto/benchmarks/CipherReadBenchmark.
   java \
3   -- -Dsize=102400
```

This script runs vogar with caliper benchmarks. You can also specify all the options caliper provides.

---

## Usage

**Entity and keys** **Entity:** this is a not-secret identifier of your data. It's used for integrity check purposes (to know that the content has not been tampered) and also to verify it was not swapped with another valid encrypted content/file.

**Key:** the key is provided by the KeyChain implementation passed to the Crypto object. So each time a new encryption is requested, the key is requested to the KeyChain. The key is generated randomly the first time on demand. You might change the implementation by we strongly suggest to generate a random value. If the encryption key needs for some reason to be based on a text password, you can try using the PasswordBasedKeyGenerator object.

### Encryption

```
1 // Creates a new Crypto object with default implementations of a key
  chain
2 KeyChain keyChain = new SharedPrefsBackedKeyChain(context, CryptoConfig
  .KEY_256);
3 Crypto crypto = AndroidConceal.get().createDefaultCrypto(keyChain);
4
5 // Check for whether the crypto functionality is available
6 // This might fail if Android does not load libraries correctly.
7 if (!crypto.isAvailable()) {
8     return;
9 }
10
11 OutputStream fileStream = new BufferedOutputStream(
12     new FileOutputStream(file));
13
14 // Creates an output stream which encrypts the data as
15 // it is written to it and writes it out to the file.
16 OutputStream outputStream = crypto.getCipherOutputStream(
17     fileStream,
18     Entity.create("entity_id"));
19
20 // Write plaintext to it.
21 outputStream.write(plainText);
22 outputStream.close();
```

### Decryption

```
1 // Get the file to which ciphertext has been written.
2 FileInputStream fileStream = new FileInputStream(file);
3
4 // Creates an input stream which decrypts the data as
5 // it is read from it.
6 InputStream inputStream = crypto.getCipherInputStream(
7     fileStream,
8     Entity.create("entity_id"));
```

---

```
9
10 // Read into a byte array.
11 int read;
12 byte[] buffer = new byte[1024];
13
14 // You must read the entire stream to completion.
15 // The verification is done at the end of the stream.
16 // Thus not reading till the end of the stream will cause
17 // a security bug. For safety, you should not
18 // use any of the data until it's been fully read or throw
19 // away the data if an exception occurs.
20 while ((read = inputStream.read(buffer)) != -1) {
21     out.write(buffer, 0, read);
22 }
23
24 inputStream.close();
```

If you don't have a lot of data to encrypt, you could use the convenience functions:

```
1 byte[] cipherText = crypto.encrypt(plainText, Entity.create("mytext"));
2
3 byte[] plainText = crypto.decrypt(cipherText, Entity.create("mytext"));
```

### Integrity

```
1 OutputStream outputStream = crypto.getMacOutputStream(fileStream,
    entity);
2 outputStream.write(plainTextBytes);
3 outputStream.close();
4
5 InputStream inputStream = crypto.getMacInputStream(fileStream, entity);
6
7 // Will throw an exception if mac verification fails.
8 // You must read the entire stream to completion.
9 // The verification is done at the end of the stream.
10 // Thus not reading till the end of the stream will cause
11 // a security bug. For safety, you should not
12 // use any of the data until it's been fully read or throw
13 // away the data if an exception occurs.
14 while((read = inputStream.read(buffer)) != -1) {
15     out.write(buffer, 0, read);
16 }
17 inputStream.close();
```

### Upgrade notes

Starting with v1.1 recommended encryption will use a 256-bit key (instead of 128-bit). This means stronger security. You should use this default.

---

If you need to read from an existing file, you still will need 128-bit encryption. You can use the old way of creating `Crypto` objects as it preserves its 128-bit behavior. Although ideally you should re-encrypt that content with a 256-bit key.

Also there's an improved way of creating `Entity` object which is platform independent. It's strongly recommended for new encrypted items although you need to stick to the old way for already encrypted content.

#### Existing code still with 128-bit keys and old `Entity` (deprecated)

```
1 // this constructor creates a key chain that produces 128-bit keys
2 KeyChain keyChain = new SharedPrefsBackedKeyChain(context);
3 // this constructor creates a crypto that uses 128-bit keys
4 Crypto crypto = new Crypto(keyChain, library);
5 Entity entity = new Entity(someStringId);
```

**New code using 256-keys and `Entity.create`** We recommend the use of the factory class `AndroidConceal`.

```
1 // explicitly create 256-bit key chain
2 KeyChain keyChain = new SharedPrefsBackedKeyChain(context, CryptoConfig
    .KEY_256);
3 // create the default crypto (expects 256-bit key)
4 AndroidConceal.get().createDefaultCrypto(keyChain);
5 // factory class also has explicit methods: createCrypto128Bits and
    ceateCrypto256Bits if desired.
6 Entity entity = Entity.create(someStringId);
```

## Troubleshooting

**I'm getting `NoSuchFieldError` on runtime** If you hit an error on runtime and it says something similar to:

```
1 java.lang.NoSuchFieldError: no field with name='mCtxPtr' signature='J'
    in class Lcom/facebook/crypto/cipher/NativeGCMCipher;
```

This happens because native code needs to refer to Java fields/methods. For doing so it uses typical JNI functions which receive the name and signature. At the same time tools like proguard trim off or rename class members in order to get smaller executables. Normally this process is run on release versions. When native code request the member, it's not present anymore.

To avoid this kind of problems exceptions can be defined. You will need to configure proguard with the rules defined in `proguard_annotations.pro`. You can use the file as is, or you can include its content in your own proguard configuration file.