iOS Security

iOS 9.0 or later

September 2015

Security architecture diagram of iOS provides a visual overview of the different technologies discussed in this document.

**Secure Enclave**: carries out onboard crypto operations

**Secure Element**: responsible for Apple Pay

**Crypto engine**: AES-256 in hardware (on DMA path between flash storage and main system memory = fast file encryption.)
Secure Enclave

The Secure Enclave is a coprocessor fabricated in the Apple A7 or later A-series processor. It utilizes its own secure boot and personalized software update separate from the application processor. It provides all cryptographic operations for Data Protection key management and maintains the integrity of Data Protection even if the kernel has been compromised.

The Secure Enclave uses encrypted memory and includes a hardware random number generator. Its microkernel is based on the L4 family, with modifications by Apple. Communication between the Secure Enclave and the application processor is isolated to an interrupt-driven mailbox and shared memory data buffers.

Each Secure Enclave is provisioned during fabrication with its own UID (Unique ID) that is not accessible to other parts of the system and is not known to Apple. When the device starts up, an ephemeral key is created, entangled with its UID, and used to encrypt the Secure Enclave’s portion of the device’s memory space.

Additionally, data that is saved to the file system by the Secure Enclave is encrypted with a key entangled with the UID and an anti-replay counter.

The Secure Enclave is responsible for processing fingerprint data from the Touch ID sensor, determining if there is a match against registered fingerprints, and then enabling access or purchases on behalf of the user. Communication between the processor and the Touch ID sensor takes place over a serial peripheral interface bus. The processor forwards the data to the Secure Enclave but cannot read it. It’s encrypted and authenticated with a session key that is negotiated using the device’s shared key that is provisioned for the Touch ID sensor and the Secure Enclave. The session key exchange uses AES key wrapping with both sides providing a random key that establishes the session key and uses AES-CCM transport encryption.

Key management, probably some sort of hardware-security module (HSM) functionalities (“key wrapping”, encryption, etc.)
Secure Enclave

The Secure Enclave is a coprocessor fabricated in the Apple A7 or later A-series processor. It utilizes its own secure boot and personalized software update separate from the application processor. It provides all cryptographic operations for Data Protection key management and maintains the integrity of Data Protection even if the kernel has been compromised.

The Secure Enclave uses **encrypted memory** and includes a **hardware random number generator**. Its microkernel is based on the L4 family, with modifications by Apple. Communication between the Secure Enclave and the application processor is isolated to an interrupt-driven mailbox and shared memory data buffers.

Each Secure Enclave is provisioned during fabrication with its own UID (Unique ID) that is not accessible to other parts of the system and is not known to Apple. When the device starts up, an ephemeral key is created, entangled with its UID, and used to encrypt the Secure Enclave's portion of the device's memory space.

Additionally, data that is saved to the file system by the Secure Enclave is encrypted with a key entangled with the UID and an anti-replay counter.

The Secure Enclave is responsible for processing fingerprint data from the Touch ID sensor, determining if there is a match against registered fingerprints, and then enabling access or purchases on behalf of the user. Communication between the processor and the Touch ID sensor takes place over a serial peripheral interface bus. The processor forwards the data to the Secure Enclave but cannot read it. It's encrypted and authenticated with a session key that is negotiated using the device's shared key that is provisioned for the Touch ID sensor and the Secure Enclave. The session key exchange uses AES key wrapping with both sides providing a random key that establishes the session key and uses AES-CCM transport encryption.

Key management, probably some sort of hardware-security module (HSM) functionalities ("key wrapping", encryption, etc.)

Has its own RNG... (What kind? How implemented?)

Encrypted memory (What kind of encryption?)
Secure Enclave

The Secure Enclave is a coprocessor fabricated in the Apple A7 or later A-series processor. It utilizes its own secure boot and personalized software update separate from the application processor. It provides all cryptographic operations for Data Protection key management and maintains the integrity of Data Protection even if the kernel has been compromised.

The Secure Enclave uses encrypted memory and includes a hardware random number generator. Its microkernel is based on the L4 family, with modifications by Apple. Communication between the Secure Enclave and the application processor is isolated to an interrupt-driven mailbox and shared memory data buffers.

Each Secure Enclave is provisioned during fabrication with its own UID (Unique ID) that is not accessible to other parts of the system and is not known to Apple. When the device starts up, an ephemeral key is created, entangled with its UID, and used to encrypt the Secure Enclave’s portion of the device’s memory space.

Additionally, data that is saved to the file system by the Secure Enclave is encrypted with a key entangled with the UID and an anti-replay counter.

The Secure Enclave is responsible for processing fingerprint data from the Touch ID sensor, determining if there is a match against registered fingerprints, and then enabling access or purchases on behalf of the user. Communication between the processor and the Touch ID sensor takes place over a serial peripheral interface bus. The processor forwards the data to the Secure Enclave but cannot read it. It’s encrypted and authenticated with a session key that is negotiated using the device’s shared key that is provisioned for the Touch ID sensor and the Secure Enclave. The session key exchange uses AES key wrapping with both sides providing a random key that establishes the session key and uses AES-CCM transport encryption.

Key management, probably some sort of hardware-security module (HSM) functionalities (“key wrapping”, encryption, etc.)

Has its own RNG… (What kind? How implemented?)

Encrypted memory (What kind of encryption?)

Has its own secret. (UID = key?)
Secure Enclave

The Secure Enclave is a coprocessor fabricated in the Apple A7 or later A-series processor. It utilizes its own secure boot and personalized software update separate from the application processor. It provides all cryptographic operations for Data Protection key management and maintains the integrity of Data Protection even if the kernel has been compromised.

The Secure Enclave uses encrypted memory and includes a hardware random number generator. Its microkernel is based on the L4 family, with modifications by Apple. Communication between the Secure Enclave and the application processor is isolated to an interrupt-driven mailbox and shared memory data buffers.

Each Secure Enclave is provisioned during fabrication with its own UID (Unique ID) that is not accessible to other parts of the system and is not known to Apple. When the device starts up, an ephemeral key is created, entangled with its UID, and used to encrypt the Secure Enclave’s portion of the device’s memory space.

Additionally, data that is saved to the file system by the Secure Enclave is encrypted with a key entangled with the UID and an anti-replay counter.

The Secure Enclave is responsible for processing fingerprint data from the Touch ID sensor, determining if there is a match against registered fingerprints, and then enabling access or purchases on behalf of the user. Communication between the processor and the Touch ID sensor takes place over a serial peripheral interface bus. The processor forwards the data to the Secure Enclave but cannot read it. It’s encrypted and authenticated with a session key that is negotiated using the device’s shared key that is provisioned for the Touch ID sensor and the Secure Enclave. The session key exchange uses AES key wrapping with both sides providing a random key that establishes the session key and uses AES-CCM transport encryption.

Key management, probably some sort of hardware-security module (HSM) functionalities (“key wrapping”, encryption, etc.)

Has its own RNG… (What kind? How implemented?)

Encrypted memory (What kind of encryption?)

Has its own secret. (UID = key?)

How is this “ephemeral key” created?

How is it “entangled” with the secret UID?
(Probably some kind of KDF…)

Touch ID

Touch ID is the fingerprint sensing system that makes secure access to the device without asking for the device passcode. The passcode can always be used instead of a passcode-based lock, not by replacing it but by securely providing access to the users won’t have to enter it as frequently. Touch ID also overcomes the inconvenience of a fingerprint map as additional overlapping nodes are identified with each use.
Secure Enclave

The Secure Enclave is a coprocessor fabricated in the Apple A7 or later A-series processor. It utilizes its own secure boot and personalized software update separate from the application processor. It provides all cryptographic operations for Data Protection key management and maintains the integrity of Data Protection even if the kernel has been compromised.

The Secure Enclave uses encrypted memory and includes a hardware random number generator. Its microkernel is based on the L4 family, with modifications by Apple. Communication between the Secure Enclave and the application processor is isolated to an interrupt-driven mailbox and shared memory data buffers.

Each Secure Enclave is provisioned during fabrication with its own UID (Unique ID) that is not accessible to other parts of the system and is not known to Apple. When the device starts up, an ephemeral key is created, entangled with its UID, and used to encrypt the Secure Enclave's portion of the device's memory space.

Additionally, data that is saved to the file system by the Secure Enclave is encrypted with a key entangled with the UID and an anti-replay counter.

The Secure Enclave is responsible for processing fingerprint data from the Touch ID sensor, determining if there is a match against registered fingerprints, and then enabling access or purchases on behalf of the user. Communication between the processor and the Touch ID sensor takes place over a serial peripheral interface bus. The processor forwards the data to the Secure Enclave but cannot read it. It's encrypted and authenticated with a session key that is negotiated using the device's shared key that is provisioned for the Touch ID sensor and the Secure Enclave. The session key exchange uses AES key wrapping with both sides providing a random key that establishes the session key and uses AES-CCM transport encryption.

Key management, probably some sort of hardware-security module (HSM) functionalities ("key wrapping", encryption, etc.)

Has its own RNG... (What kind? How implemented?)

Encrypted memory  (What kind of encryption?)

Has its own secret. (UID = key?)

How is this "ephemeral key" created?

How is it "entangled" with the secret UID? (Probably some kind of KDF...)

SE encrypts data stored in file system, protects against replay (probably stores counter in file metadata?)
Secure Enclave

The Secure Enclave is a coprocessor fabricated in the Apple A7 or later A-series processor. It utilizes its own secure boot and personalized software update separate from the application processor. It provides all cryptographic operations for Data Protection key management and maintains the integrity of Data Protection even if the kernel has been compromised.

The Secure Enclave uses encrypted memory and includes a hardware random number generator. Its microkernel is based on the L4 family, with modifications by Apple. Communication between the Secure Enclave and the application processor is isolated to an interrupt-driven mailbox and shared memory data buffers.

Each Secure Enclave is provisioned during fabrication with its own UID (Unique ID) that is not accessible to other parts of the system and is not known to Apple. When the device starts up, an ephemeral key is created, entangled with its UID, and used to encrypt the Secure Enclave's portion of the device's memory space.

Additionally, data that is saved to the file system by the Secure Enclave is encrypted with a key entangled with the UID and an anti-replay counter.

The Secure Enclave is responsible for processing fingerprint data from the Touch ID sensor, determining if there is a match against registered fingerprints, and then enabling access or purchases on behalf of the user. Communication between the processor and the Touch ID sensor takes place over a serial peripheral interface bus. The processor forwards the data to the Secure Enclave but cannot read it. It's encrypted and authenticated with a session key that is negotiated using the device's shared key that is provisioned for the Touch ID sensor and the Secure Enclave. The session key exchange uses AES key wrapping with both sides providing a random key that establishes the session key and uses AES-CCM transport encryption.

Key management, probably some sort of hardware-security module (HSM) functionalities ("key wrapping", encryption, etc.)

Has its own RNG… (What kind? How implemented?)

Encrypted memory (What kind of encryption?)

Has its own secret. (UID = key?)

How is this “ephemeral key” created?
How is it “entangled” with the secret UID?
(Probably some kind of KDF…)

SE encrypts data stored in file system, protects against replay (probably stores counter in file metadata?)

Authenticated encryption… SE and Touch ID share a key, then used to set up a session key? (HOW?)
Secure Enclave

The Secure Enclave is a coprocessor fabricated in the Apple A7 or later A-series processor. It utilizes its own secure boot and personalized software update separate from the application processor. It provides all cryptographic operations for Data Protection key management and maintains the integrity of Data Protection even if the kernel has been compromised.

The Secure Enclave uses encrypted memory and includes a hardware random number generator. Its microkernel is based on the L4 family, with modifications by Apple. Communication between the Secure Enclave and the application processor is isolated to an interrupt-driven mailbox and shared memory data buffers.

Each Secure Enclave is provisioned during fabrication with its own UID (Unique ID) that is not accessible to other parts of the system and is not known to Apple. When the device starts up, an ephemeral key is created, entangled with its UID, and used to encrypt the Secure Enclave’s portion of the device’s memory space.

Additionally, data that is saved to the file system by the Secure Enclave is encrypted with a key entangled with the UID and an anti-replay counter.

The Secure Enclave is responsible for processing fingerprint data from the Touch ID sensor, determining if there is a match against registered fingerprints, and then enabling access or purchases on behalf of the user. Communication between the processor and the Touch ID sensor takes place over a serial peripheral interface bus. The processor forwards the data to the Secure Enclave but cannot read it. It’s encrypted and authenticated with a session key that is negotiated using the device’s shared key that is provisioned for the Touch ID sensor and the Secure Enclave. The session key exchange uses AES key wrapping with both sides providing a random key that establishes the session key and uses AES-CCM transport encryption.

Key management, probably some sort of hardware-security module (HSM) functionalities (“key wrapping”, encryption, etc.)

Has its own RNG… (What kind? How implemented?)

Encrypted memory… (What kind of encryption?)

Has its own secret. (UID = key?)

How is this “ephemeral key” created?
How is it “entangled” with the secret UID?
(Probably some kind of KDF…)

SE encrypts data stored in file system, protects against replay (probably stores counter in file metadata?)

Authenticated encryption… SE and Touch ID share a key, then used to set up a session key? (HOW?)

Probably some kind of two-way key-transport using AESKW, followed by AES-CCM for bulk encryption of Touch ID data
This specification is intended to satisfy the NIST Key Wrap requirement to: Design a cryptographic algorithm called a Key Wrap that uses the Advanced Encryption Standard (AES) as a primitive to securely encrypt a plaintext key(s) with any associated integrity information and data, such that the combination could be longer than the width of the AES blocksize (128-bits). Each ciphertext bit should be a highly non-linear function of each plaintext bit, and (when unwrapping) each plaintext bit should be a highly non-linear function of each ciphertext bit. It is sufficient to approximate an ideal pseudorandom permutation to the degree that exploitation of undesirable phenomena is as unlikely as guessing the AES engine key. This key wrap algorithm needs to provide ample security to protect keys in the context of a prudently designed key management architecture.
AES “key wrap” (NIST) [RFC 3394]

This specification is intended to satisfy the NIST Key Wrap requirement to: Design a cryptographic algorithm called a Key Wrap that uses the Advanced Encryption Standard (AES) as a primitive to securely encrypt a plaintext key(s) with any associated integrity information and data, such that the combination could be longer than the width of the AES blocksize (128-bits). Each ciphertext bit should be a highly non-linear function of each plaintext bit, and (when unwrapping) each plaintext bit should be a highly non-linear function of each ciphertext bit. It is sufficient to approximate an ideal pseudorandom permutation to the degree that exploitation of undesirable phenomena is as unlikely as guessing the AES engine key. This key wrap algorithm needs to provide ample security to protect keys in the context of a prudently designed key management architecture.

Encryption with associated data
Designed for high-entropy plaintexts

Uh… what? Sounds SPRP-ish…
AES “key wrap” (NIST) [RFC 3394]

This specification is intended to satisfy the NIST Key Wrap requirement to: Design a cryptographic algorithm called a Key Wrap that uses the Advanced Encryption Standard (AES) as a primitive to securely encrypt a plaintext key(s) with any associated integrity information and data, such that the combination could be longer than the width of the AES blocksize (128-bits). Each ciphertext bit should be a highly non-linear function of each plaintext bit, and (when unwrapping) each plaintext bit should be a highly non-linear function of each ciphertext bit. It is sufficient to approximate an ideal pseudorandom permutation to the degree that exploitation of undesirable phenomena is as unlikely as guessing the AES engine key. This key wrap algorithm needs to provide ample security to protect keys in the context of a prudently designed key management architecture.

Encryption with associated data
Designed for high-entropy plaintexts

Uh... what? Sounds SPRP-ish...

Ideal PRP?
Security on the order of guessing the key...
AES “key wrap” (NIST) [RFC 3394]

This specification is intended to satisfy the NIST Key Wrap requirement to: Design a cryptographic algorithm called a Key Wrap that uses the Advanced Encryption Standard (AES) as a primitive to securely encrypt a plaintext key(s) with any associated integrity information and data, such that the combination could be longer than the width of the AES blocksize (128-bits). Each ciphertext bit should be a highly non-linear function of each plaintext bit, and (when unwrapping) each plaintext bit should be a highly non-linear function of each ciphertext bit. It is sufficient to approximate an ideal pseudorandom permutation to the degree that exploitation of undesirable phenomena is as unlikely as guessing the AES engine key. This key wrap algorithm needs to provide ample security to protect keys in the context of a prudently designed key management architecture.
AES “key wrap” (NIST) [RFC 3394]

Encryption with AD, SPRP-like behavior, security on the order of guessing the AES key.

Inputs: Plaintext, $n$ 64-bit values $\{P_1, P_2, \ldots, P_n\}$,

key, $K$ (the KEK).

Outputs: Ciphertext, $(n+1)$ 64-bit values $\{C_0, C_1, \ldots, C_n\}$.

1) Initialize variables
   
   Set $A^0 = IV$, an initial value (see 2.2.3)
   
   For $i = 1, \ldots, n$
   
   $R_i^0 = P_i$

2) Calculate intermediate values
   
   For $t = 1, \ldots, s$, where $s = 6n$
   
   $A^t = \text{MSB}_{64}\left(\text{AES}_K\left(A^{t-1} \mid R_i^{t-1}\right)\right) \oplus t$
   
   For $i = 1, \ldots, n - 1$
   
   $R_i^t = R_i^{t-1}$
   
   $R_n^t = \text{LSB}_{64}\left(\text{AES}_K\left(A^{t-1} \mid R_i^{t-1}\right)\right)$

3) Output the results
   
   Set $C_0 = A^t$
   
   For $i = 1, \ldots, n$
   
   $C_i = R_i^t$

---

Initialize Rs to ptxt blocks

For each round $t$...

1. consume the first R block in line with the current “round IV”
2. shift all remaining R blocks up one position in line
3. push a new R block at the end
AES “key wrap” (NIST) [RFC 3394]

Encryption with AD, SPRP-like behavior, security on the order of guessing the AES key.

**Inputs**: Plaintext, \( n \) 64-bit values \( \{P_1, P_2, \ldots, P_n\} \),
Key, \( K \) (the KEK).

**Outputs**: Ciphertext, \((n+1)\) 64-bit values \( \{C_0, C_1, \ldots, C_n\} \).

1. Initialize variables
   - Set \( A^0 = IV \), an initial value (see 2.2.3)
     - For \( i = 1, \ldots, n \) \( R^0_i = P_i \)
2. Calculate intermediate values
   - For \( t = 1, \ldots, s \), where \( s = \frac{64}{K} \)
     - \( A^t = \text{MSB}_{64} \left( \text{AES}_{K} \left( A^{t-1} | R^{t-1}_s \right) \right) + t \)
     - For \( i = 1, \ldots, n - 1 \)
     - \( R^t_i = R^{t-1}_{i+1} \)
     - \( R^t_n = \text{LSB}_{64} \left( \text{AES}_{K} \left( A^{t-1} | R^{t-1}_1 \right) \right) \)
3. Output the results
   - Set \( C_0 = A' \)
   - For \( i = 1, \ldots, n \)
     - \( C_i = R^t_i \)

The initial value (IV) refers to the value assigned to \( A_0 \) in the first step of the wrapping process. This value is used to obtain an integrity check on the key data. In the final step of the unwrapping process, the recovered value of \( A_n \) is compared to the expected value of \( A_0 \). If there is a match, the key is accepted as valid, and it is returned by the unwrapping algorithm. If there is not a match, then the key is not accepted as valid, and the unwrapping algorithm returns an error.

**Output results**

\[ A_0 = IV = A6A6A6A6A6A6A6A6A6. \]
AES “key wrap” (NIST) [RFC 3394]

Encryption with AD, SPRP-like behavior, security on the order of guessing the AES key.

Deterministic AEAD with tight reduction to underlying SPRP security of blockcipher

Efficient?

one 128-bit AES key + 128-bit of AD = 24 AES calls

Secure?

No attacks known... But no proof either.

(Efficient and provably secure options available, see Rogaway, S’ Eurocrypt 2006)
Secure Enclave

The Secure Enclave is a coprocessor fabricated in the Apple A7 or later A-series processor. It utilizes its own secure boot and personalized software update separate from the application processor. It provides all cryptographic operations for Data Protection key management and maintains the integrity of Data Protection even if the kernel has been compromised.

The Secure Enclave uses encrypted memory and includes a hardware random number generator. Its microkernel is based on the L4 family, with modifications by Apple. Communication between the Secure Enclave and the application processor is isolated to an interrupt-driven mailbox and shared memory data buffers.

Each Secure Enclave is provisioned during fabrication with its own UID (Unique ID) that is not accessible to other parts of the system and is not known to Apple. When the device starts up, an ephemeral key is created, entangled with its UID, and used to encrypt the Secure Enclave’s portion of the device’s memory space.

Additionally, data that is saved to the file system by the Secure Enclave is encrypted with a key entangled with the UID and an anti-replay counter.

The Secure Enclave is responsible for processing fingerprint data from the Touch ID sensor, determining if there is a match against registered fingerprints, and then enabling access or purchases on behalf of the user. Communication between the processor and the Touch ID sensor takes place over a serial peripheral interface bus. The processor forwards the data to the Secure Enclave but cannot read it. It’s encrypted and authenticated with a session key that is negotiated using the device’s shared key that is provisioned for the Touch ID sensor and the Secure Enclave. The session key exchange uses AES key wrapping with both sides providing a random key that establishes the session key and uses AES-CCM transport encryption.

Key management, probably some sort of hardware-security module (HSM) functionalities (“key wrapping”, encryption, etc.)

Has its own RNG… (What kind? How implemented?)

Encrypted memory  (What kind of encryption?)

Has its own secret. (UID = key?)

How is this “ephemeral key” created?

How is it “entangled” with the secret UID? (Probably some kind of KDF…)

SE encrypts data stored in file system, protects against replay (probably stores counter in file metadata?)

Authenticated encryption… SE and Touch ID share a key, then used to set up a session key? (HOW?)

Probably some kind of two-way key-transport using AESKW, followed by AES-CCM for bulk encryption of Touch ID data
The following is a specification of the generation-encryption process of CCM:

**Prerequisites:**
- block cipher algorithm;
- key $K$;
- counter generation function;
- formatting function;
- MAC length $Tlen$.

**Input:**
- valid nonce $N$;
- valid payload $P$ of length $Plen$ bits;
- valid associated data $A$;

**Output:**
ciphertext $C$. 
The following is a specification of the generation-encryption process of CCM:

**Prerequisites:**
- block cipher algorithm;
- key $K$;
- counter generation function;
- formatting function;
- MAC length $Tlen$.

**Input:**
- valid nonce $N$;
- valid payload $P$ of length $Plen$ bits;
- valid associated data $A$;

**Output:**
- ciphertext $C$.

IV-based encryption with support for associated data (probably with security when IV is a nonce)
The following is a specification of the generation-encryption process of CCM:

**Prerequisites:**
- block cipher algorithm;
- key $K$;
- counter generation function;
- formatting function;
- MAC length $Tlen$.

**Input:**
- valid nonce $N$;
- valid payload $P$ of length $Plen$ bits;
- valid associated data $A$;

**Output:**
ciphertext $C$.

**Steps:**
1. Apply the formatting function to $(N, A, P)$ to produce the blocks $B_0, B_1, \ldots, B_r$.
2. Set $Y_0 = \text{CIPH}_K(B_0)$.
3. For $i = 1$ to $r$, do $Y_i = \text{CIPH}_K(B_i \oplus Y_{i-1})$.
4. Set $T = \text{MSB}_{Tlen}(Y_r)$.
5. Apply the counter generation function to generate the counter blocks $Ctr_0, Ctr_1, \ldots, Ctr_m$, where $m = \lceil Plen/128 \rceil$.
6. For $j = 0$ to $m$, do $S_j = \text{CIPH}_K(Ctr_j)$.
7. Set $S = S_1 \| S_2 \| \ldots \| S_m$.
8. Return $C = (P \oplus \text{MSB}_{Plen}(S)) \| (T \oplus \text{MSB}_{Tlen}(S_0))$.

**IV-based encryption with support for associated data**
(probably with security when IV is a nonce)
The following is a specification of the generation-encryption process of CCM:

**Prerequisites:**
- block cipher algorithm;
- key $K$;
- counter generation function;
- formatting function;
- MAC length $Tlen$.

**Input:**
- valid nonce $N$;
- valid payload $P$ of length $Plen$ bits;
- valid associated data $A$;

**Output:**
- ciphertext $C$.

---

**IV-based encryption with support for associated data**
(probably with security when IV is a nonce)

**Steps:**

1. Apply the formatting function to $(N, A, P)$ to produce the blocks $B_0, B_1, \ldots, B_r$.
2. Set $Y_0 = \text{CIPH}_K(B_0)$.
3. For $i = 1$ to $r$, do $Y_i = \text{CIPH}_K(B_i \oplus Y_{i-1})$.
4. Set $T = \text{MSB}_{Tlen}(Y_r)$.
5. Apply the counter generation function to generate the counter blocks $Ctr_0, Ctr_1, \ldots, Ctr_m$, where $m = \lfloor Plen/128 \rfloor$.
6. For $j = 0$ to $m$, do $S_j = \text{CIPH}_K(Ctr_j)$.
7. Set $S = S_1 \parallel S_2 \parallel \ldots \parallel S_m$.
8. Return $C = (P \oplus \text{MSB}_{Plen}(S)) \parallel (T \oplus \text{MSB}_{Tlen}(S_0))$.

---

**CBC-MAC with encrypted-nonce IV**

---

NIST Special Publication 800-38C

Recommendation for Block Cipher Modes of Operation:
The CCM Mode for Authentication and Confidentiality

Morris Dworkin
The following is a specification of the generation-encryption process of CCM:

**Prerequisites:**
- block cipher algorithm;
- key \( K \);
- counter generation function;
- formatting function;
- MAC length \( Tlen \).

**Input:**
- valid nonce \( N \);
- valid payload \( P \) of length \( Plen \) bits;
- valid associated data \( A \);

**Output:**
- ciphertext \( C \).

**Steps:**
1. Apply the formatting function to \((N, A, P)\) to produce the blocks \( B_0, B_1, \ldots, B_r \).
2. Set \( Y_0 = \text{CIPH}_K(B_0) \).
3. For \( i = 1 \) to \( r \), do \( Y_i = \text{CIPH}_K(B_i \oplus Y_{i-1}) \).
4. Set \( T = \text{MSB}_{Tlen}(Y_r) \).
5. Apply the counter generation function to generate the counter blocks \( Ctr_0, Ctr_1, \ldots, Ctr_m \), where \( m = \lceil \frac{Plen}{128} \rceil \).
6. For \( j = 0 \) to \( m \), do \( S_j = \text{CIPH}_K(Ctr_j) \).
7. Set \( S = S_1 \parallel S_2 \parallel \ldots \parallel S_m \).
8. Return \( C = (P \oplus \text{MSB}_{Plen}(S)) \parallel (T \oplus \text{MSB}_{Tlen}(S_0)) \).
The following is a specification of the generation-encryption process of CCM:

**Prerequisites:**
- block cipher algorithm;
- key $K$;
- counter generation function;
- formatting function;
- MAC length $Tlen$.

**Input:**
- valid nonce $N$;
- valid payload $P$ of length $Plen$ bits;
- valid associated data $A$;

**Output:**
- ciphertext $C$.

**CBC-MAC with encrypted-nonce IV**

**Counter-mode with**

$ctr_i = N \ || \ <i>$

### IV-based AEAD scheme, "PRF-then-Encrypt"

**VII-PRF: CBC-MAC with encrypted-nonce IV (birthday-bound PRF security, reduction to PRF security of underlying blockcipher)**

**Encryption: CTR mode (tight IND-CPA security bound, reduction to PRF security of blockcipher)**

**Steps:**
1. Apply the formatting function to $(N, A, P)$ to produce the blocks $B_0, B_1, \ldots, B_r$.
2. Set $Y_0 = \text{CIPH}_K(B_0)$.
3. For $i = 1$ to $r$, do $Y_i = \text{CIPH}_K(B_i \oplus Y_{i-1})$.
4. Set $T = \text{MSB}_{Tlen}(Y_r)$.
5. Apply the counter generation function to generate the counter blocks $Ctr_0, Ctr_1, \ldots, Ctr_m$, where $m = \lceil Plen/128 \rceil$.
6. For $j = 0$ to $m$, do $S_j = \text{CIPH}_K(Ctr_j)$.
7. Set $S = S_1 \ || \ S_2 \ || \ \ldots \ || \ S_m$.
8. Return $C = (P \oplus \text{MSB}_{Plen}(S)) \ || \ (T \oplus \text{MSB}_{Tlen}(S_0))$. 
Secure Enclave

The Secure Enclave is a coprocessor fabricated in the Apple A7 or later A-series processor. It utilizes its own secure boot and personalized software update separate from the application processor. It provides all cryptographic operations for Data Protection key management and maintains the integrity of Data Protection even if the kernel has been compromised.

The Secure Enclave uses encrypted memory and includes a hardware random number generator. Its microkernel is based on the L4 family, with modifications by Apple. Communication between the Secure Enclave and the application processor is isolated to an interrupt-driven mailbox and shared memory data buffers.

Each Secure Enclave is provisioned during fabrication with its own UID (Unique ID) that is not accessible to other parts of the system and is not known to Apple. When the device starts up, an ephemeral key is created, entangled with its UID, and used to encrypt the Secure Enclave's portion of the device's memory space.

Additionally, data that is saved to the file system by the Secure Enclave is encrypted with a key entangled with the UID and an anti-replay counter.

The Secure Enclave is responsible for processing fingerprint data from the Touch ID sensor, determining if there is a match against registered fingerprints, and then enabling access or purchases on behalf of the user. Communication between the processor and the Touch ID sensor takes place over a serial peripheral interface bus. The processor forwards the data to the Secure Enclave but cannot read it. It's encrypted and authenticated with a session key that is negotiated using the device's shared key that is provisioned for the Touch ID sensor and the Secure Enclave. The session key exchange uses AES key wrapping with both sides providing a random key that establishes the session key and uses AES-CCM transport encryption.

Key management, probably some sort of hardware-security module (HSM) functionalities ("key wrapping", encryption, etc.)

Has its own RNG… (What kind? How implemented?)

Encrypted memory (What kind of encryption?)

Has its own secret. (UID = key?)

How is this “ephemeral key” created?

How is it “entangled” with the secret UID?

(Probably some kind of KDF…)

SE encrypts data stored in file system, protects against replay (probably stores counter in file metadata?)

Authenticated encryption… SE and Touch ID share a key, then used to set up a session key? (HOW?)

Probably some kind of two-way key-transport using AESKW, followed by AES-CCM for bulk encryption of Touch ID data
Touch ID: $K_0$

Secure Enclave: $K_0$

Generate $K_T$

$AES_{K_0}(K_T)$

Generate $K_{SE}$

$AES_{K_0}(K_{SE})$

Session key

$K \leftarrow KDF(K_T, K_{SE}, params)$

$AES_{CCM_K}(data)$

(This is my guess)
What’s the device UID?

The device’s unique ID (UID) and a device group ID (GID) are AES 256-bit keys fused (UID) or compiled (GID) into the application processor and Secure Enclave during manufacturing. No software or firmware can read them directly; they can see only the

- Key management, probably some sort of hardware-security module (HSM) functionalities ("key wrapping", encryption, etc.)
- Has its own RNG… (What kind? How implemented?)
- Encrypted memory (What kind of encryption?)
- Has its own secret. (UID = key?)
- How is this “ephemeral key” created?
  How is it “entangled” with the secret UID?
  (Probably some kind of KDF…)
- SE encrypts data stored in file system, protects against replay (probably stores counter in file metadata?)
- Authenticated encryption… SE and Touch ID share a key, then used to set up a session key? (HOW?)
- Probably some kind of two-way key-transport using AESKW, followed by AES-CCM for bulk encryption of Touch ID data

Apart from the UID and GID, all other cryptographic keys are created by the system’s random number generator (RNG) using an algorithm based on CTR_DRBG. System entropy is generated from timing variations during boot, and additionally from interrupt timing once the device has booted. Keys generated inside the Secure Enclave use its true hardware random number generator based on multiple ring oscillators post processed with CTR_DRBG.

Securely erasing saved keys is just as important as generating them. It’s especially challenging to do so on flash storage, where wear-leveling might mean multiple copies of data need to be erased. To address this issue, iOS devices include a feature dedicated to secure data erasure called Effaceable Storage. This feature accesses the underlying storage technology (for example, NAND) to directly address and erase a small number of blocks at a very low level.

The UID allows data to be cryptographically tied to a particular device. For example, the key hierarchy protecting the file system includes the UID, so if the memory chips are physically moved from one device to another, the files are inaccessible. The UID is not related to any other identifier on the device.

Erase all content and settings

The “Erase all content and settings” option in Settings obliterates all the keys in Effaceable Storage, rendering all user data on the device cryptographically inaccessible. Therefore, it’s an ideal way to be sure all personal information is removed from a device before giving it to somebody else or returning it for service. Important: Do not use the “Erase all content and settings” option until the device has been backed up, as there is no way to recover the erased data.
How is “entangling” with the UID performed?

**Tangling**

The process by which a user’s passcode is turned into a cryptographic key and strengthened with the device’s UID. This ensures that a brute-force attack must be performed on a given device, and thus is rate limited and cannot be performed in parallel. The tangling algorithm is PBKDF2, which uses AES keyed with the device UID as the pseudorandom function (PRF) for each iteration.

How many PBKDF2 iterations? 10000 I think…

Key management, probably some sort of hardware-security module (HSM) functionalities (“key wrapping”, encryption, etc.)

Has its own RNG… (What kind? How implemented?)

Encrypted memory (What kind of encryption?)

Has its own secret. (UID = key?)

How is this “ephemeral key” created?

How is it “entangled” with the secret UID? (Probably some kind of KDF…)

SE encrypts data stored in file system, protects against replay (probably stores counter in file metadata?)

Authenticated encryption… SE and Touch ID share a key, then used to set up a session key? (HOW?)

Probably some kind of two-way key-transport using AESKW, followed by AES-CCM for bulk encryption of Touch ID data
Random Number Generators

Apart from the UID and GID, all other cryptographic keys are created by the system's random number generator (RNG) using an algorithm based on CTR_DRBG. System entropy is generated from timing variations during boot, and additionally from interrupt timing once the device has booted. Keys generated inside the Secure Enclave use its true hardware random number generator based on multiple ring oscillators post processed with CTR_DRBG.

Two kinds of RNGs:
1. "entropy" from timing/interrupts + CTR_DRBG
2. true hardware entropy + CTR_DRBG

Key management, probably some sort of hardware-security module (HSM) functionalities ("key wrapping", encryption, etc.)

Has its own RNG… (What kind? How implemented?)

Encrypted memory (What kind of encryption?)

Has its own secret. (UID = key?)

How is this "ephemeral key" created?
How is it “entangled” with the secret UID?
(Probably some kind of KDF…)

SE encrypts data stored in file system, protects against replay (probably stores counter in file metadata?)

Authenticated encryption… SE and Touch ID share a key, then used to set up a session key? (HOW?)

Probably some kind of two-way key-transport using AESKW, followed by AES-CCM for bulk encryption of Touch ID data
"Leftover hash lemma":
If $H: \mathcal{K} \times \{0,1\}^N \rightarrow \{0,1\}^n$ is $\epsilon$-Au,
then no adversary can distinguish between
$(K, H_K(S))$ and $(K, U)$, where $U \leftarrow \{0,1\}^n$,
with prob. more than $0.5 \sqrt{2^n (2^{-m} + \epsilon)}$.

(CBCMAC)

(This is a simplified version of the real specified algorithm. Generally referred to as "Extract-then-expand".)

(CTR mode)
BTW, Intel’s hardware RNG does something very similar...

---

**Fig. 2.** Block diagram for Intel’s RDRAND implementation. The CBCMAC computation uses AES-128 with the fixed key $K' = \text{AES}_0(1)$. The DRBG runs AES-128 in counter mode to produce $\{0,1\}^{128-3}$ bits of output; the first 256 bits are used to update the key $K$ and IV; the final 128 bits are sent to the output buffer, which is read by the RDRAND instruction.

[S', Terashima 2013]
Architecture overview

Every time a file on the data partition is created, Data Protection creates a new 256-bit key (the “per-file” key) and gives it to the hardware AES engine, which uses the key to encrypt the file as it is written to flash memory using AES CBC mode. (On devices with an A8 processor, AES-XTS is used.) The initialization vector (IV) is calculated with the block offset into the file, encrypted with the SHA-1 hash of the per-file key.

The per-file key is wrapped with one of several class keys, depending on the circumstances under which the file should be accessible. Like all other wrappings, this is performed using NIST AES key wrapping, per RFC 3394. The wrapped per-file key is stored in the file’s metadata.

When a file is opened, its metadata is decrypted with the file system key, revealing the wrapped per-file key and a notation on which class protects it. The per-file key is unwrapped with the class key, then supplied to the hardware AES engine, which decrypts the file as it is read from flash memory. All wrapped file key handling occurs in the Secure Enclave; the file key is never directly exposed to the application processor. At boot, the Secure Enclave negotiates an ephemeral key with the AES engine. When the Secure Enclave unwraps a file’s keys, they are rewrapped with the ephemeral key and sent back to the application processor.

The metadata of all files in the file system is encrypted with a random key, which is created when iOS is first installed or when the device is wiped by a user. The file system key is stored in Effaceable Storage. Since it’s stored on the device, this key is not used to maintain the confidentiality of data; instead, it’s designed to be quickly erased on demand (by the user, with the “Erase all content and settings” option, or
File system encryption

Architecture overview

Every time a file on the data partition is created, Data Protection creates a new 256-bit key (the “per-file” key) and gives it to the hardware AES engine, which uses the key to encrypt the file as it is written to flash memory using AES CBC mode. (On devices with an A8 processor, AES-XTS is used.) The initialization vector (IV) is calculated with the block offset into the file, encrypted with the SHA-1 hash of the per-file key.

AES-XTS: The AES-XTS is based on Rogaway's XEX construction, which is in turn based on the idea of a tweaked block cipher as described by Liskov, Rivest and Wagner. To allow for sectors that do not contain a number of bytes equal to an integer multiple of the AES block size, ciphertext stealing is also used.

Essentially this, with:

- \( H \) implemented over AES (and made fast)
- \( T = (\text{sector} \ #, \ \text{block} \ #) \), roughly
- \( Y \) = data block

and some tricks for the last block if the sector isn’t a multiple of 128-bits wide
File system encryption

Architecture overview

Every time a file on the data partition is created, Data Protection creates a new 256-bit key (the “per-file” key) and gives it to the hardware AES engine, which uses the key to encrypt the file as it is written to flash memory using AES CBC mode. (On devices with an A8 processor, AES-XTS is used.) The initialization vector (IV) is calculated with the block offset into the file, encrypted with the SHA-1 hash of the per-file key.

The per-file key is wrapped with one of several class keys, depending on the circumstances under which the file should be accessible. Like all other wrappings, this is performed using NIST AES key wrapping, per RFC 3394. The wrapped per-file key is stored in the file’s metadata.

When a file is opened, its metadata is decrypted with the file system key, revealing the wrapped per-file key and a notation on which class protects it. The per-file key is unwrapped with the class key, then supplied to the hardware AES engine, which decrypts the file as it is read from flash memory. All wrapped file key handling occurs in the Secure Enclave; the file key is never directly exposed to the application processor. At boot, the Secure Enclave negotiates an ephemeral key with the AES engine. When the Secure Enclave unwraps a file’s keys, they are rewrapped with the ephemeral key and sent back to the application processor.

The metadata of all files in the file system is encrypted with a random key, which is created when iOS is first installed or when the device is wiped by a user. The file system key is stored in Effaceable Storage. Since it’s stored on the device, this key is not used to maintain the confidentiality of data; instead, it’s designed to be quickly erased on demand (by the user, with the “Erase all content and settings” option, or by a user or administrator issuing a remote wipe command from a mobile device management (MDM) server, Exchange ActiveSync, or iCloud). Erasing the key in this manner renders all files cryptographically inaccessible.

In addition to the hardware encryption features built into iOS devices, Apple uses a technology called Data Protection to further protect data stored in flash memory on system apps, such as Messages, Mail, Calendar, Contacts, Photos, and Health data values and builds on the hardware encryption technologies built into each iOS device. Data Protection is implemented by constructing and managing a hierarchy of keys, as illustrated in the diagram below.

The diagram shows the encryption process:

1. The file key is generated with a random number (`K_f`) and an initialization vector (`IV`), encrypted with the SHA-1 of the file key (`AES_{SHA1}(K_f)`).
2. The class key is generated with a random number and encrypted with the SHA-1 of the file key (`AES_{SHA1}(K_f)`).
3. The file metadata is encrypted with the class key (`AES_{class key}(W_f)`).
4. The file contents are encrypted with the file system key (`AES_{file system key}(M_f)`).

The diagram also illustrates the key hierarchy:

- **File Key**
- **Class Key**
- **File System Key**
- **Passcode Key**
- **Hardware Key**

The keys are managed by the Secure Enclave, which is certified by Apple and can be configured to allow access to that device’s data in a secure manner.
How iMessage sends and receives messages

The user’s outgoing message is individually encrypted for each of the receiver’s devices. The public RSA encryption keys of the receiving devices are retrieved from IDS. For each receiving device, the sending device generates a random 128-bit key and encrypts the message with it using AES in CTR mode. This per-message AES key is encrypted using RSA-OAEP to the public key of the receiving device. The combination of the encrypted message text and the encrypted message key is then hashed with SHA-1, and the hash is signed with ECDSA using the sending device’s private signing key. The resulting messages, one for each receiving device, consist of the encrypted message text, the encrypted message key, and the sender’s digital signature. They are then dispatched to the APNs for delivery. Metadata, such as the timestamp and APNs routing information, is not encrypted. Communication with APNs is encrypted using a forward-secret TLS channel.

Multi-receiver KEM-DEM, using:

CCA-secure PKE (RSA-OAEP) for the KEM
IND$-CPA-secure encryption (CTR) for the DEM

Plus a signature over both outputs for authenticity protection