An Empirical Study of Cryptographic Misuse In Android Applications

Manuel Egele, David Brumley
Davidson Mellon University
{mepol,ebrumley}@cmu.edu

Yariv Fanta-Tonia, Christopher Kruegel
University of California, Santa Barbara
{yariv, kruegel}@cs.ucsb.edu

ABSTRACT
Developers use cryptographic APIs in Android with the intent of securing data such as passwords and personal information in mobile devices. In this paper, we ask whether developers use the cryptographic APIs as intended and provide insights into the ways cryptographic libraries are used, e.g., IND-CPA security. We develop program analysis techniques that automatically check programs on the Google Play marketplace, and find that 10,307 out of 11,746 applications use cryptographic APIs incorrectly, with at least one instance. These numbers show that applications do not use cryptographic APIs in a fashion that maximizes overall security. We then suggest specific remediation based on our analysis to improve overall cryptographic security in Android applications.

Categories and Subject Descriptors
D.7 [Software Engineering]: Distribution, Maintenance, and Enhancement—Reengineering, reverse engineering, and reengineering

General Terms
Android program store, misuse of cryptographic primitive

Keywords
Software Security, Program Analysis

1 Introduction
Developers use cryptographic primitives like block ciphers and message authentication codes (MACs) to secure data, communications. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.

In this paper, we ask whether developers know how to use cryptographic APIs correctly. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way practically leads to breaches.
CRYPTO MISUSE IN ANDROID APPS

15,134 apps from Google play used crypto;
Analyzed 11,748 of them
CRYPTO MISUSE IN ANDROID APPS

15,134 apps from Google play used crypto;
Analyzed 11,748 of them

<table>
<thead>
<tr>
<th># apps</th>
<th>violated rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>5,656</td>
<td>Uses ECB (BouncyCastle default) (R1)</td>
</tr>
<tr>
<td>3,644</td>
<td>Uses constant symmetric key (R3)</td>
</tr>
<tr>
<td>2,000</td>
<td>Uses ECB (Explicit use) (R1)</td>
</tr>
<tr>
<td>1,932</td>
<td>Uses constant IV (R2)</td>
</tr>
<tr>
<td>1,636</td>
<td>Used iteration count &lt; 1,000 for PBE(R5)</td>
</tr>
<tr>
<td>1,629</td>
<td>Seeds SecureRandom with static (R6)</td>
</tr>
<tr>
<td>1,574</td>
<td>Uses static salt for PBE (R4)</td>
</tr>
<tr>
<td>1,421</td>
<td>No violation</td>
</tr>
</tbody>
</table>
**CRYPTO MISUSE IN ANDROID APPS**

15,134 apps from Google play used crypto; Analyzed **11,748** of them

<table>
<thead>
<tr>
<th># apps</th>
<th>violated rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>5,656</td>
<td>Uses ECB (BouncyCastle default) (R1)</td>
</tr>
<tr>
<td>3,644</td>
<td>Uses constant symmetric key (R3)</td>
</tr>
<tr>
<td>2,000</td>
<td>Uses ECB (Explicit use) (R1)</td>
</tr>
<tr>
<td>1,932</td>
<td>Uses constant IV (R2)</td>
</tr>
<tr>
<td>1,636</td>
<td>Used iteration count &lt; 1,000 for PBE(R5)</td>
</tr>
<tr>
<td>1,629</td>
<td>Seeds SecureRandom with static (R6)</td>
</tr>
<tr>
<td>1,574</td>
<td>Uses static salt for PBE (R4)</td>
</tr>
<tr>
<td>1,421</td>
<td>No violation</td>
</tr>
</tbody>
</table>
NEVER use ECB
(but over 50% of Android apps do)
BouncyCastle is a library that conforms to Java’s `Cipher` interface:

```java
Cipher c =
    Cipher.getInstance("AES/CBC/PKCS5Padding");

// Ultimately end up wrapping a ByteArrayOutputStream
// in a CipherOutputStream
```

Java documentation specifies:

If no mode or padding is specified, **provider-specific default values for the mode and padding scheme are used. For example, the SunJCE provider uses ECB as the default mode, and PKCS5Padding as the default padding scheme for DES, DES-EDE and Blowfish ciphers**
<table>
<thead>
<tr>
<th>#Occurences</th>
<th>Symmetric encryption scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>5878</td>
<td>AES/CBC/PKCS5Padding</td>
</tr>
<tr>
<td>4803</td>
<td>AES *</td>
</tr>
<tr>
<td>1151</td>
<td>DES/ECB/NoPadding</td>
</tr>
<tr>
<td>741</td>
<td>DES *</td>
</tr>
<tr>
<td>501</td>
<td>DESede *</td>
</tr>
<tr>
<td>473</td>
<td>DESede/ECB/PKCS5Padding</td>
</tr>
<tr>
<td>468</td>
<td>AES/CBC/NoPadding</td>
</tr>
<tr>
<td>443</td>
<td>AES/ECB/PKCS5Padding</td>
</tr>
<tr>
<td>235</td>
<td>AES/CBC/PKCS7Padding</td>
</tr>
<tr>
<td>221</td>
<td>DES/ECB/PKCS5Padding</td>
</tr>
<tr>
<td>220</td>
<td>AES/ECB/NoPadding</td>
</tr>
<tr>
<td>205</td>
<td>DES/CBC/PKCS5Padding</td>
</tr>
<tr>
<td>155</td>
<td>AES/ECB/PKCS7Padding</td>
</tr>
<tr>
<td>104</td>
<td>AES/CFB8/NoPadding</td>
</tr>
</tbody>
</table>

Table 4: Distribution of frequently used symmetric encryption schemes. Schemes marked with * are used in ECB mode by default.
<table>
<thead>
<tr>
<th>#Occurences</th>
<th>Symmetric encryption scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>5878</td>
<td>AES/CBC/PKCS5Padding</td>
</tr>
<tr>
<td>4803</td>
<td>AES *</td>
</tr>
<tr>
<td>1151</td>
<td>DES/ECB/NoPadding</td>
</tr>
<tr>
<td>741</td>
<td>DES *</td>
</tr>
<tr>
<td>501</td>
<td>DESede *</td>
</tr>
<tr>
<td>473</td>
<td>DESede/ECB/PKCS5Padding</td>
</tr>
<tr>
<td>468</td>
<td>AES/CBC/NoPadding</td>
</tr>
<tr>
<td>443</td>
<td>AES/ECB/PKCS5Padding</td>
</tr>
<tr>
<td>235</td>
<td>AES/CBC/PKCS7Padding</td>
</tr>
<tr>
<td>221</td>
<td>DES/ECB/PKCS5Padding</td>
</tr>
<tr>
<td>220</td>
<td>AES/ECB/NoPadding</td>
</tr>
<tr>
<td>205</td>
<td>DES/CBC/PKCS5Padding</td>
</tr>
<tr>
<td>155</td>
<td>AES/ECB/PKCS7Padding</td>
</tr>
<tr>
<td>104</td>
<td>AES/CFB8/NoPadding</td>
</tr>
</tbody>
</table>

Table 4: Distribution of frequently used symmetric encryption schemes. Schemes marked with * are used in ECB mode by default.
CRYPTO MISUSE IN ANDROID APPS

15,134 apps from Google play used crypto;
Analyzed 11,748 of them

<table>
<thead>
<tr>
<th># apps</th>
<th>violated rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>5,656</td>
<td>Uses ECB (BouncyCastle default) (R1)</td>
</tr>
<tr>
<td>3,644</td>
<td>Uses constant symmetric key (R3)</td>
</tr>
<tr>
<td>2,000</td>
<td>Uses ECB (Explicit use) (R1)</td>
</tr>
<tr>
<td>1,932</td>
<td>Uses constant IV (R2)</td>
</tr>
<tr>
<td>1,636</td>
<td>Used iteration count &lt; 1,000 for PBE(R5)</td>
</tr>
<tr>
<td>1,629</td>
<td>Seeds SecureRandom with static (R6)</td>
</tr>
<tr>
<td>1,574</td>
<td>Uses static salt for PBE (R4)</td>
</tr>
<tr>
<td>1,421</td>
<td>No violation</td>
</tr>
</tbody>
</table>
CRYPTO MISUSE IN ANDROID APPS

15,134 apps from Google play used crypto; Analyzed **11,748** of them

<table>
<thead>
<tr>
<th># apps</th>
<th>violated rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>5,656</td>
<td>Uses ECB (BouncyCastle default) (R1)</td>
</tr>
<tr>
<td>3,644</td>
<td>Uses constant symmetric key (R3)</td>
</tr>
<tr>
<td>2,000</td>
<td>Uses ECB (Explicit use) (R1)</td>
</tr>
<tr>
<td>1,932</td>
<td>Uses constant IV (R2)</td>
</tr>
<tr>
<td>1,636</td>
<td>Used iteration count &lt; 1,000 for PBE(R5)</td>
</tr>
<tr>
<td>1,629</td>
<td>Seeds SecureRandom with static (R6)</td>
</tr>
<tr>
<td>1,574</td>
<td>Uses static salt for PBE (R4)</td>
</tr>
<tr>
<td>1,421</td>
<td>No violation</td>
</tr>
</tbody>
</table>

A failure of the programmers to **know the tools** they use
A failure of library writers to **provide safe defaults**
MISUSING CRYPTO

Avoid shooting yourself in the foot:

• Do not **roll your own** cryptographic mechanisms
  • Takes peer review
  • Apply Kerkhoff’s principle

• Do not **misuse** existing crypto

• Do not even **implement** the underlying crypto
WHY NOT IMPLEMENT AES/RSA YOURSELF?

• Not talking about creating a brand new crypto scheme, just implementing one that’s already widely accepted and used.

• Kerkhoff’s principle: these are all open standards; should be implementable.

• Potentially buggy/incorrect code, but so might be others’ implementations (viz. OpenSSL bugs, poor defaults in Bouncy castles, etc.)

• So why not implement it yourself?
SIDE-CHANNEL ATTACKS

- Cryptography concerns the *theoretical* difficulty in breaking a cipher
Cryptography concerns the theoretical difficulty in breaking a cipher.

But what about the information that a particular implementation could leak?

Attacks based on these are "side-channel attacks"
Cryptography concerns the *theoretical* difficulty in breaking a cipher.

**SIDE-CHANNEL ATTACKS**

- But what about the information that a particular *implementation* could leak?
- Attacks based on these are “*side-channel attacks*”
Interpret *power traces* taken during a cryptographic operation

Simple power analysis can reveal the sequence of instructions executed
Overall operation clearly visible:
Can identify the 16 rounds of DES
Figure 1: SPA trace showing an entire DES operation.

Overall operation clearly visible:
Can identify the **16 rounds of DES**
Specific instructions are also discernible
SPA ON DES

**Figure 3:** SPA trace showing individual clock cycles.

Specific **instructions** are also discernible.

Jump taken

No jump taken
HypotheticalEncrypt(msg, key) {
    for(int i=0; i < key.len(); i++) {
        if(key[i] == 0)
            // branch 0
        else
            // branch 1
    }
}
HypotheticalEncrypt(msg, key) {
  for(int i=0; i < key.len(); i++) {
    if(key[i] == 0) {
      // branch 0
    }
    else {
      // branch 1
    }
  }
}

What if branch 0 had, e.g., a jmp that branch 1 didn’t?
HypotheticalEncrypt(msg, key) {
    for(int i=0; i < key.len(); i++) {
        if(key[i] == 0) // branch 0
            // branch 0
        else // branch 1
            // branch 1
    }
}

Implementation issue: If the execution path depends on the inputs (key/data), then SPA can reveal keys.
HypotheticalEncrypt(msg, key) {
    for(int i=0; i < key.len(); i++) {
        if(key[i] == 0) {
            // branch 0
        } else {
            // branch 1
        }
    }
}

What if branch 0 had, e.g., a jmp that branch 1 didn’t?

What if branch 0

Implementation issue: If the execution path depends on the inputs (key/data), then SPA can reveal keys
HypotheticalEncrypt(msg, key) {
    for(int i=0; i < key.len(); i++) {
        if(key[i] == 0) {
            // branch 0
        } else {
            // branch 1
        }
    }
}

Implementation issue: If the execution path depends on the inputs (key/data), then SPA can reveal keys.
HypotheticalEncrypt(msg, key) {
    for(int i=0; i < key.len(); i++) {
        if(key[i] == 0) // branch 0
            // branch 0
        else // branch 1
            // branch 1
    }
}

What if branch 0 had, e.g., a jmp that branch 1 didn’t?
What if branch 0
- took longer? (timing attacks)
- gave off more heat?

Implementation issue: If the execution path depends on the inputs (key/data), then SPA can reveal keys
HypotheticalEncrypt(msg, key) {
    for(int i=0; i < key.len(); i++) {
        if(key[i] == 0) // branch 0
            What if branch 0 had, e.g.,
            a jmp that branch 1 didn’t?
        else // branch 1
            What if branch 0
            - took longer? (timing attacks)
            - gave off more heat?
            - made more noise?
            - …
    }
}

Implementation issue: If the execution path depends
on the inputs (key/data), then SPA can reveal keys
• SPA just visually inspects a single run

• DPA runs iteratively and reactively
  • Get multiple samples
  • Based on these, construct new plaintext messages as inputs, and repeat
MITIGATING SUCH ATTACKS

• Hide information by making the execution paths depend on the inputs as little as possible
  • Have to *give up some optimizations* that depend on particular bit values in keys
    - Some Chinese Remainder Theorem (CRT) optimizations permitted remote timing attacks on SSL servers

• The crypto community should seek to design cryptosystems under the assumption that some information is going to leak
POOR POLICIES FROM GOVERNMENTS

Exploits export-grade encryption

Figure 4: NSA’s VPN decryption infrastructure. This classified illustration published by Der Spiegel [67] shows captured IKE handshake messages being passed to a high-performance computing system, which returns the symmetric keys for ESP session traffic. The details of this attack are consistent with an efficient break for 1024-bit Diffie-Hellman.

1024-bit and smaller feasibly broken

Logjam downgrades to export-grade (512)
Clipper chip
A lesson in poorly designed protocols

**Goal:**
**Confidentiality**
Support encrypted communication between devices

**Goal:**
**Key escrow**
Permit law enforcement to obtain “session keys” with a warrant
Clipper chip: Design

Tamper-proof hardware

Skipjack
- encryption algorithm

Skipjack Keys
- Unit key
- Global family key

Diffie-Hellman
- key exchange

LEAF
- generation
- & validation

Hardware that is difficult to introspect (e.g., extract keys), alter (change the algorithms), or impersonate
Clipper chip: Design

Tamper-proof hardware

**Skipjack**
encryption algorithm

**Skipjack Keys**
- Unit key
- Global family key

**Diffie-Hellman**
key exchange

**LEAF**
generation & validation

---

Block cipher designed by the NSA, originally classified SECRET.

(Violates Kirchhoff’s principle)

Broken within one day of declassification.

80-bit key; similar algorithm to DES (also broken)
Clipper chip: Design

Tamper-proof hardware

- **Skipjack** encryption algorithm
  - Assigned when the hardware is manufactured.

- **Skipjack Keys**
  - Unit key
  - Global family key
  - Unit key is unique to this unit in particular (each Clipper chip also has a *unit ID*).

- **Diffie-Hellman** key exchange

- **LEAF** generation & validation
  - Global family key is the same across many units.
Clipper chip: Design

Tamper-proof hardware

Skipjack
encryption algorithm

Skipjack Keys
Unit key
Global family key

Diffie-Hellman
key exchange

LEAF generation & validation

Used for establishing a (symmetric) session key

Session keys are ephemeral (e.g., last only for a given connection, transaction, etc.)

General properties about session keys:
- Compromising one session key does not compromise others
- Compromising a long-term key should not compromise past session keys (forward secrecy)
Clipper chip: Design

Tamper-proof hardware

**Skipjack**
- Encryption algorithm

**Skipjack Keys**
- Unit key
- Global family key

**Diffie-Hellman**
- Key exchange

**LEAF**
- Generation & validation

**LEAF** (Law Enforcement Access Field)

To permit wiretapping, law enforcement needs to be able to extract session keys, but only has access to what is sent during communication.

**Idea**: send data that has enough info to allow law enforcement to extract keys (but not any other eavesdropper).
LEAF protocol design

1. DH key exchange
2. Each send LEAF packet
3. Send data encrypted with the session key

The Clipper chips will not decrypt until it has received a valid LEAF packet.

Law enforcement sees all packets.
- Cannot infer key from DH key exchange
- *Can* infer it from the LEAF packet
LEAF message structure

- **Session key**: 80 bits
- **Hash algorithm**: 16 bits
- **Unit Key**
- **Global family key**
- **Unit ID**
- **Encrypted session key**
- **Hash**
- **Other variables**
LEAF message structure

The other Clipper chip also has the Global Family key

=> Can decrypt the LEAF to obtain this triple

- Unit ID
- Encrypted session key
- Hash

Global family key $\rightarrow$ Skipjack $\rightarrow$ LEAF
The other Clipper chip "verifies" the LEAF by making sure that the hash is correct.
LEAF message structure

Law enforcement also has the Global Family Key

=> Can decrypt the LEAF to obtain this triple

Global family key \rightarrow \text{Skipjack} \rightarrow \text{LEAF}
LEAF message structure

- **Session key**: 80 bits
- **Unit Key**: Skipjack
- **Hash algorithm**: 16 bits

**Other variables**

Law enforcement *does not* have direct access to all unit keys; needs a **warrant** to get them.

Unit keys are split across two locations (one location gets a OTP, the other gets the XOR).
LEAF: failure

To verify the LEAF, the other Clipper chip *only* checks the hash.

Clipper chips also allow you to test a LEAF locally.
LEAF: failure

- Session key: 80 bits
- Hash algorithm: 16 bits
- Unit Key -> Skipjack
- Other variables

Generate a random LEAF => $\frac{1}{2^{16}}$ chance of a valid hash

Unit ID | Encrypted session key | Hash

But law enforcement will just see random ID & key

Validates at the other Clipper chip (so it will decrypt messages)
POOR CERTIFICATE MANAGEMENT

Websites aren’t properly revoking their certificates

Browsers aren’t properly checking for revocations

Websites aren’t keeping their secret keys secret
POOR CERTIFICATE MANAGEMENT

Websites aren’t properly revoking their certificates

Browsers aren’t properly checking for revocations

Websites aren’t keeping their secret keys secret

Why?

CAs have incentive to introduce disincentives (bandwidth costs)

Websites have disincentive to do the right thing (CAs charge; key management hard)

Browsers have a disincentive to do the right thing (page load times)