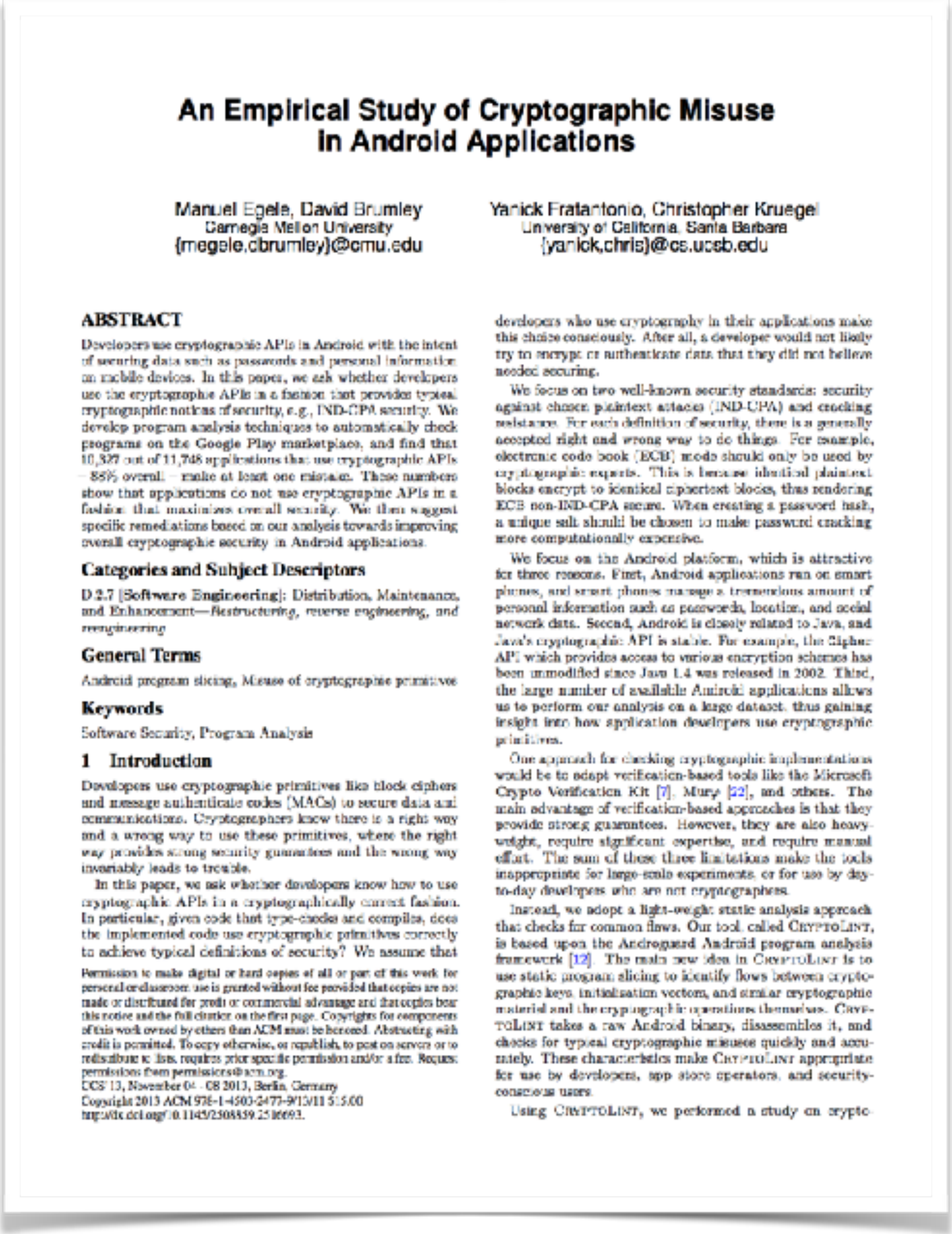
HOW CRYPTO FAILS
IN PRACTICE

CMSC 414

APR 3 2018

POOR PROGRAMING



An Empirical Study of Cryptographic Misuse in Android Applications

Manuel Egele, David Brumley
Carnegie Mellon University
{megele,dbrumley}@cmu.edu

Yanick Fratantonio, Christopher Kruegel
University of California, Santa Barbara
{yanick,chris}@cs.ucsb.edu

ABSTRACT

Developers use cryptographic APIs in Android with the intent of securing data such as passwords and personal information on mobile devices. In this paper, we ask whether developers use the cryptographic APIs in a fashion that provides typical cryptographic notions of security, e.g., IND-CPA security. We develop program analysis techniques to automatically check programs on the Google Play marketplace, and find that 10,327 out of 11,748 applications that use cryptographic APIs – 88% overall – make at least one mistake. These numbers show that applications do not use cryptographic APIs in a fashion that maximizes overall security. We then suggest specific remediations based on our analysis towards improving overall cryptographic security in Android applications.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

General Terms

Android program slicing, Misuse of cryptographic primitives

Keywords

Software Security, Program Analysis

1 Introduction

Developers use cryptographic primitives like block ciphers and message authenticate codes (MACs) to secure data and 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 invariably leads to trouble.

In this paper, we ask whether developers know how to use cryptographic APIs in a cryptographically correct fashion. In particular, given code that type-checks and compiles, does the implemented code use cryptographic primitives correctly to achieve typical definitions of security? We assume that developers who use cryptography in their applications make this choice consciously. After all, a developer would not likely try to encrypt or authenticate data that they did not believe needed securing.

We focus on two well-known security standards: security against chosen plaintext attacks (IND-CPA) and cracking resistance. For each definition of security, there is a generally accepted right and wrong way to do things. For example, electronic code book (ECB) mode should only be used by cryptographic experts. This is because identical plaintext blocks encrypt to identical ciphertext blocks, thus rendering ECB non-IND-CPA secure. When creating a password hash, a unique salt should be chosen to make password cracking more computationally expensive.

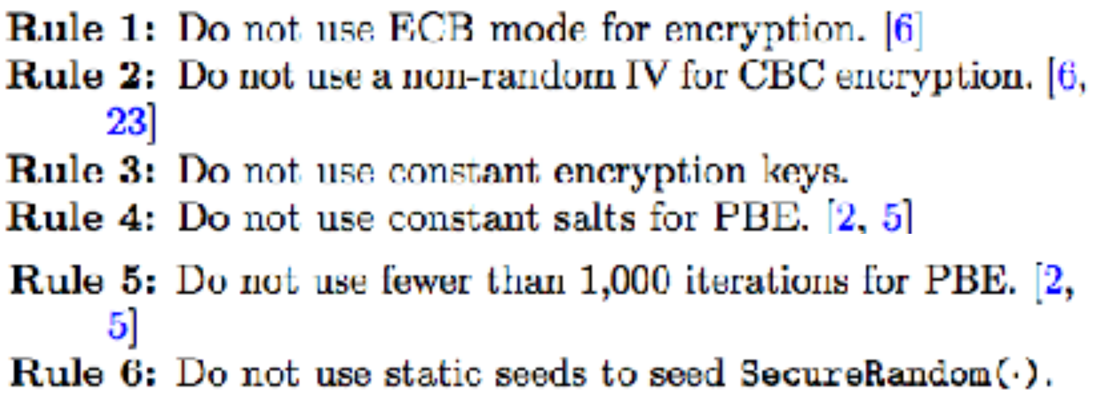
We focus on the Android platform, which is attractive for three reasons. First, Android applications run on smart phones, and smart phones manage a tremendous amount of personal information such as passwords, location, and social network data. Second, Android is closely related to Java, and Java's cryptographic API is stable. For example, the Cipher API which provides access to various encryption schemes has been unmodified since Java 1.4 was released in 2002. Third, the large number of available Android applications allows us to perform our analysis on a large dataset, thus gaining insight into how application developers use cryptographic primitives.

One approach for checking cryptographic implementations would be to adapt verification-based tools like the Microsoft Crypto Verification Kit [7], Murφ [22], and others. The main advantage of verification-based approaches is that they provide strong guarantees. However, they are also heavyweight, require significant expertise, and require manual effort. The sum of these three limitations make the tools inappropriate for large-scale experiments, or for use by day-to-day developers who are not cryptographers.

Instead, we adopt a light-weight static analysis approach that checks for common flaws. Our tool, called CRYPTOLINT, is based upon the Androguard Android program analysis framework [12]. The main new idea in CRYPTOLINT is to use static program slicing to identify flows between cryptographic keys, initialization vectors, and similar cryptographic material and the cryptographic operations themselves. CRYPTOLINT takes a raw Android binary, disassembles it, and checks for typical cryptographic misuses quickly and accurately. These characteristics make CRYPTOLINT appropriate for use by developers, app store operators, and security-conscious users.

Using CRYPTOLINT, we performed a study on crypto-

CCS'13, November 04 - 08 2013, Berlin, Germany
Copyright 2013 ACM 978-1-4503-2477-9/13/11 $15.00
http://dx.doi.org/10.1145/2508859.2516693.



Rule 1: Do not use ECB mode for encryption. [6]
Rule 2: Do not use a non-random IV for CBC encryption. [6, 23]
Rule 3: Do not use constant encryption keys.
Rule 4: Do not use constant salts for PBE. [2, 5]
Rule 5: Do not use fewer than 1,000 iterations for PBE. [2, 5]
Rule 6: Do not use static seeds to seed `SecureRandom(·)`.

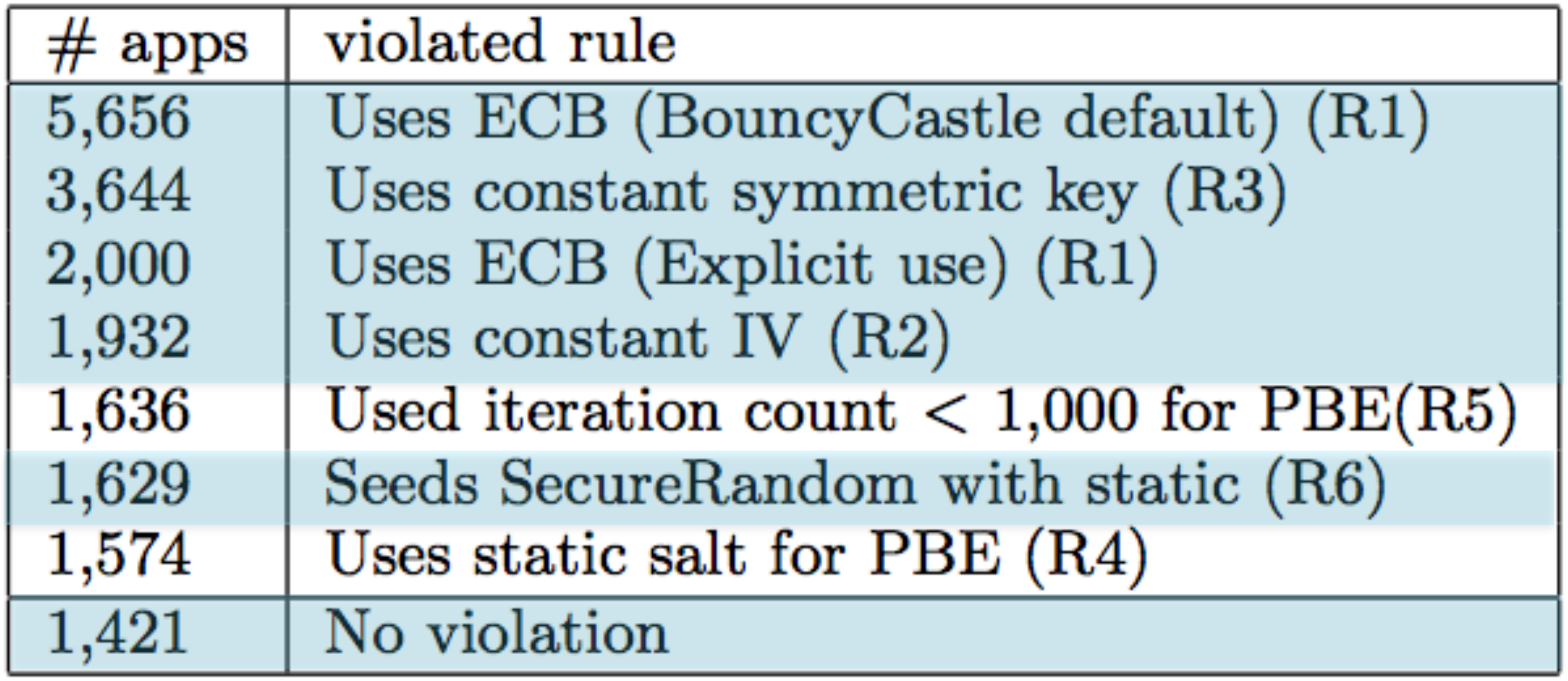
CryptoLint tool to perform static analysis on Android apps to detect how they are using crypto libraries

CRYPTO MISUSE IN ANDROID APPS

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

CRYPTO MISUSE IN ANDROID APPS

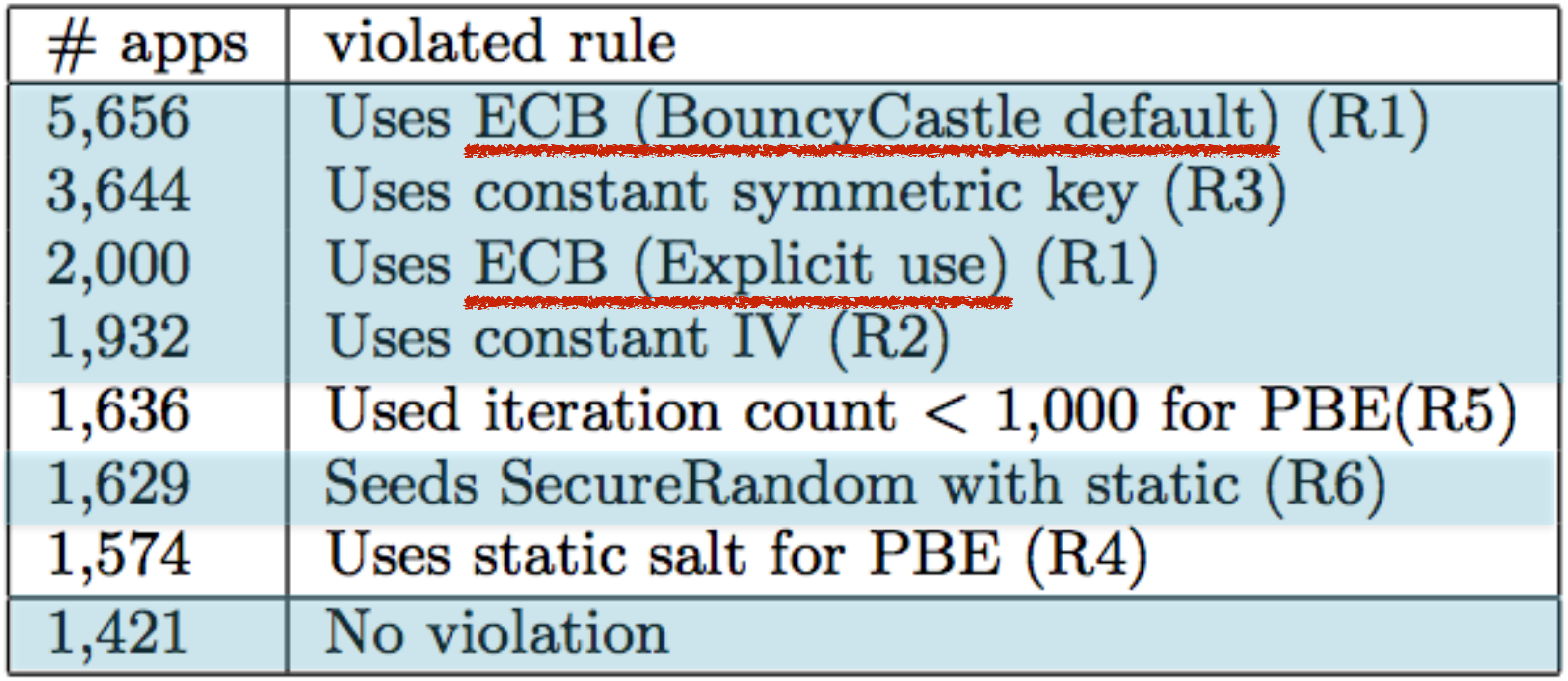
15,134 apps from Google play used crypto;
Analyzed **11,748** of them



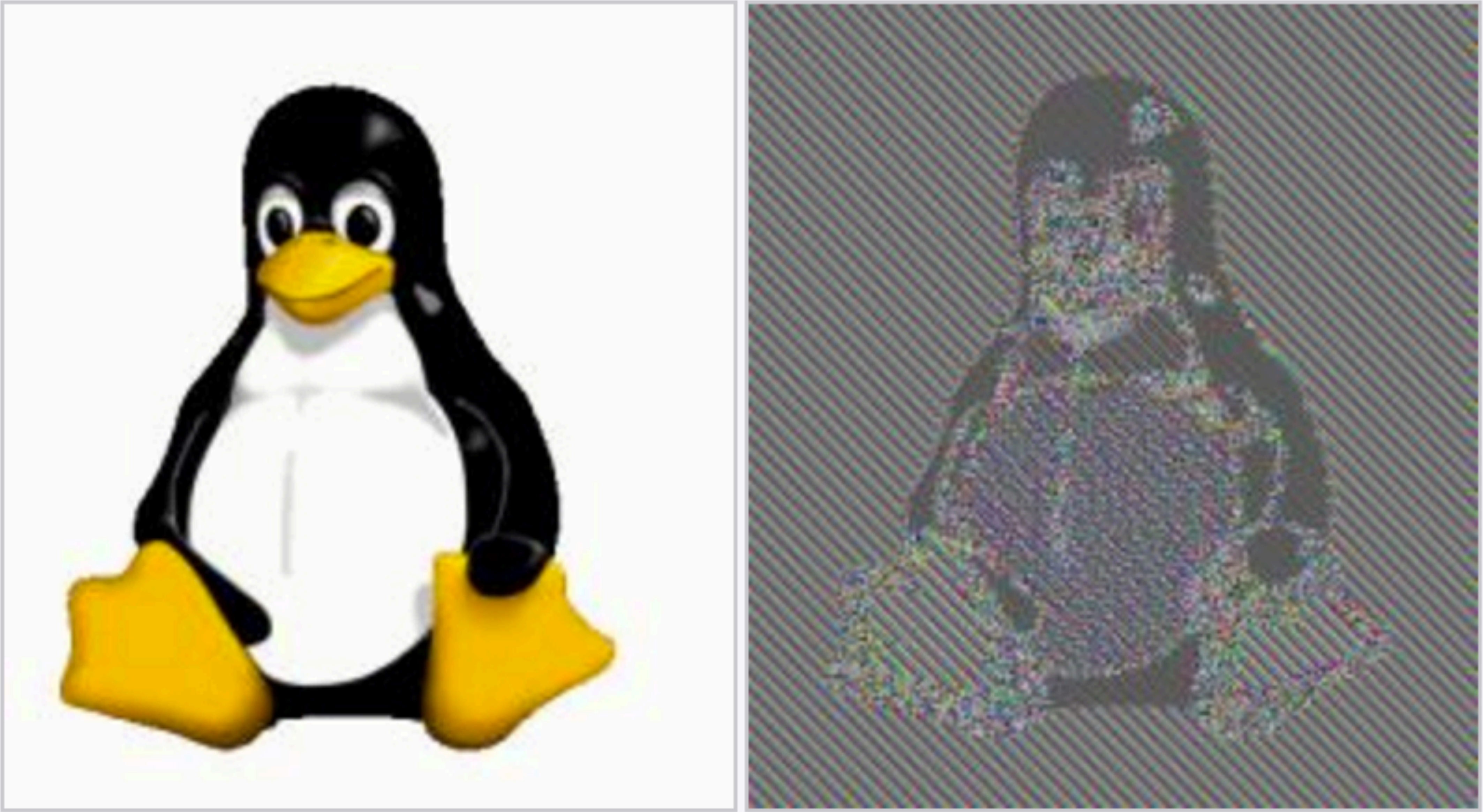
	# apps	violated rule
48%	5,656	Uses ECB (BouncyCastle default) (R1)
31%	3,644	Uses constant symmetric key (R3)
17%	2,000	Uses ECB (Explicit use) (R1)
16%	1,932	Uses constant IV (R2)
	1,636	Used iteration count $< 1{,}000$ for PBE(R5)
14%	1,629	Seeds SecureRandom with static (R6)
	1,574	Uses static salt for PBE (R4)
12%	1,421	No violation

CRYPTO MISUSE IN ANDROID APPS

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



	# apps	violated rule
48%	5,656	Uses ECB (BouncyCastle default) (R1)
31%	3,644	Uses constant symmetric key (R3)
17%	2,000	Uses ECB (Explicit use) (R1)
16%	1,932	Uses constant IV (R2)
	1,636	Used iteration count < 1,000 for PBE(R5)
14%	1,629	Seeds SecureRandom with static (R6)
	1,574	Uses static salt for PBE (R4)
12%	1,421	No violation



Original image

Encrypted using ECB mode

NEVER use ECB
(but over 50% of Android apps do)

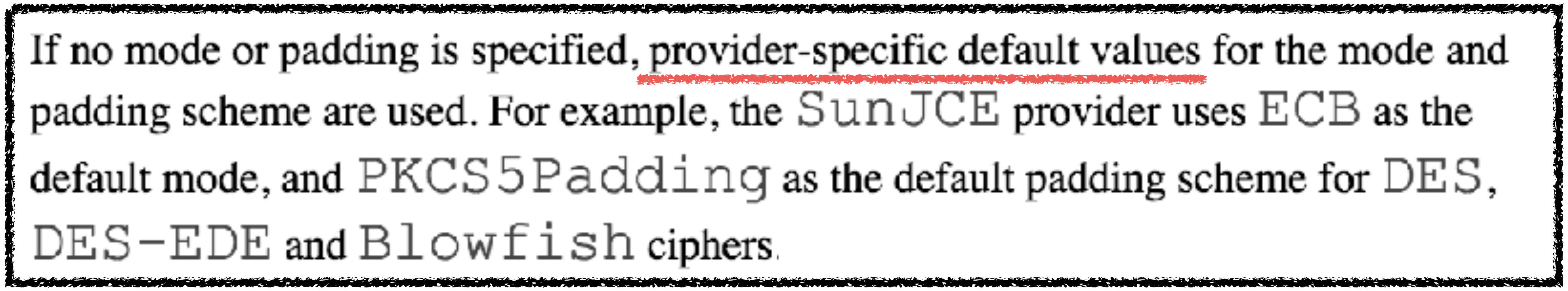
BOUNCYCASTLE DEFAULTS

- BouncyCastle is a library that conforms to Java’s `Cipher` interface:

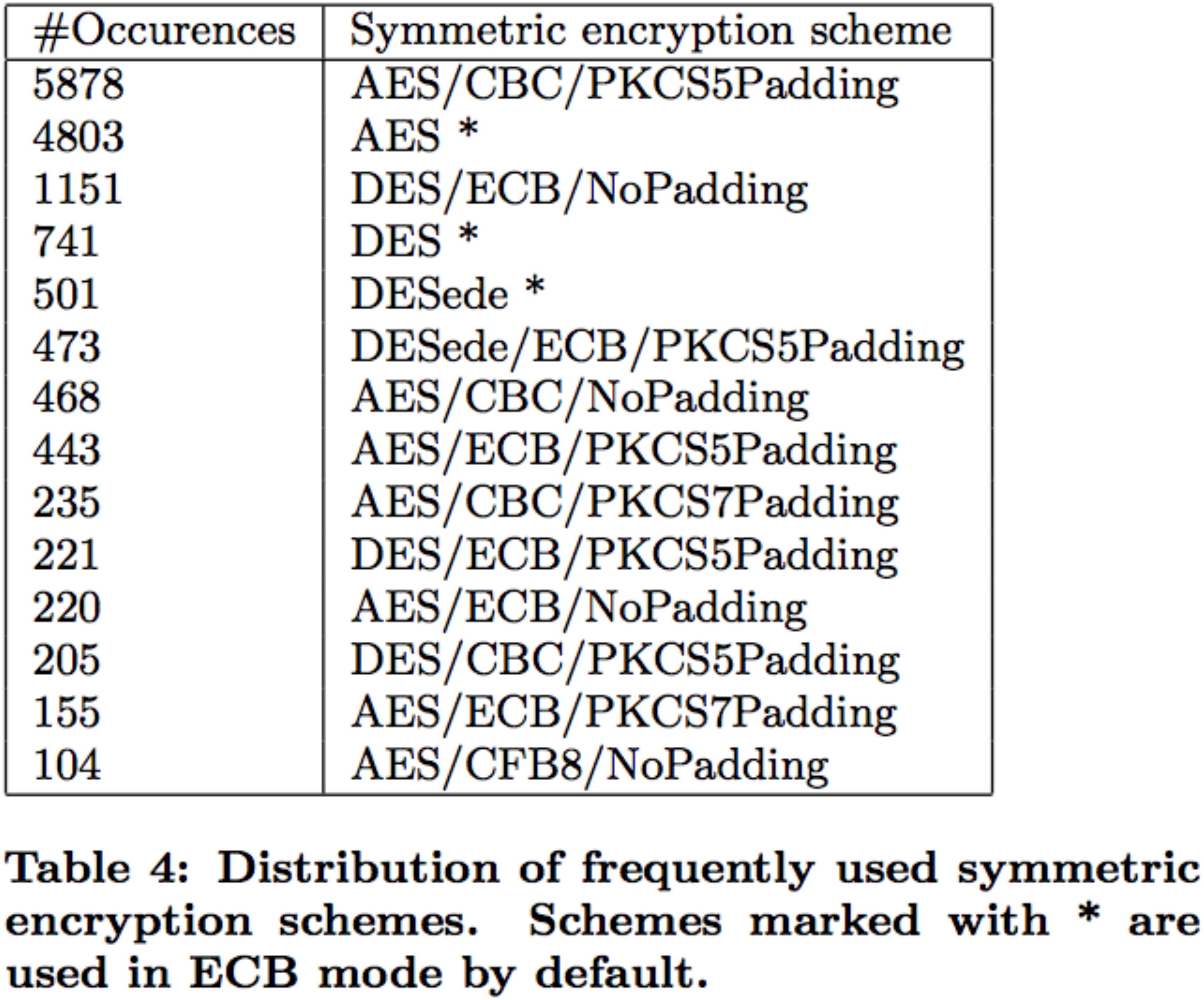
```
Cipher c =
   Cipher.getInstance(“AES/CBC/PKCS5Padding”);

// 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.



#Occurences	Symmetric encryption scheme
5878	AES/CBC/PKCS5Padding
4803	AES *
1151	DES/ECB/NoPadding
741	DES *
501	DESede *
473	DESede/ECB/PKCS5Padding
468	AES/CBC/NoPadding
443	AES/ECB/PKCS5Padding
235	AES/CBC/PKCS7Padding
221	DES/ECB/PKCS5Padding
220	AES/ECB/NoPadding
205	DES/CBC/PKCS5Padding
155	AES/ECB/PKCS7Padding
104	AES/CFB8/NoPadding

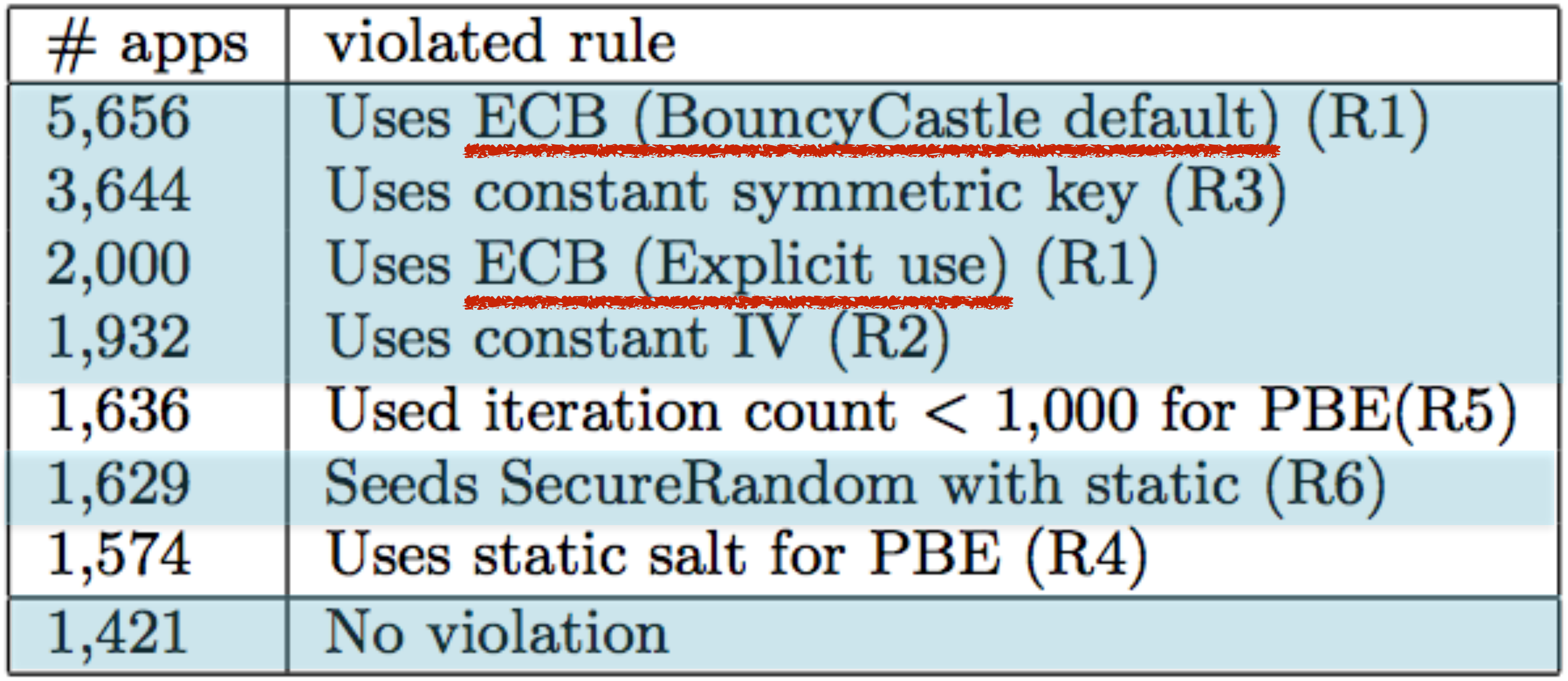
Table 4: Distribution of frequently used symmetric encryption schemes. Schemes marked with * are used in ECB mode by default.

#Occurences	Symmetric encryption scheme
5878	AES/CBC/PKCS5Padding
4803	AES *
1151	DES/ECB/NoPadding
741	DES *
501	DESede *
473	DESede/ECB/PKCS5Padding
468	AES/CBC/NoPadding
443	AES/ECB/PKCS5Padding
235	AES/CBC/PKCS7Padding
221	DES/ECB/PKCS5Padding
220	AES/ECB/NoPadding
205	DES/CBC/PKCS5Padding
155	AES/ECB/PKCS7Padding
104	AES/CFB8/NoPadding

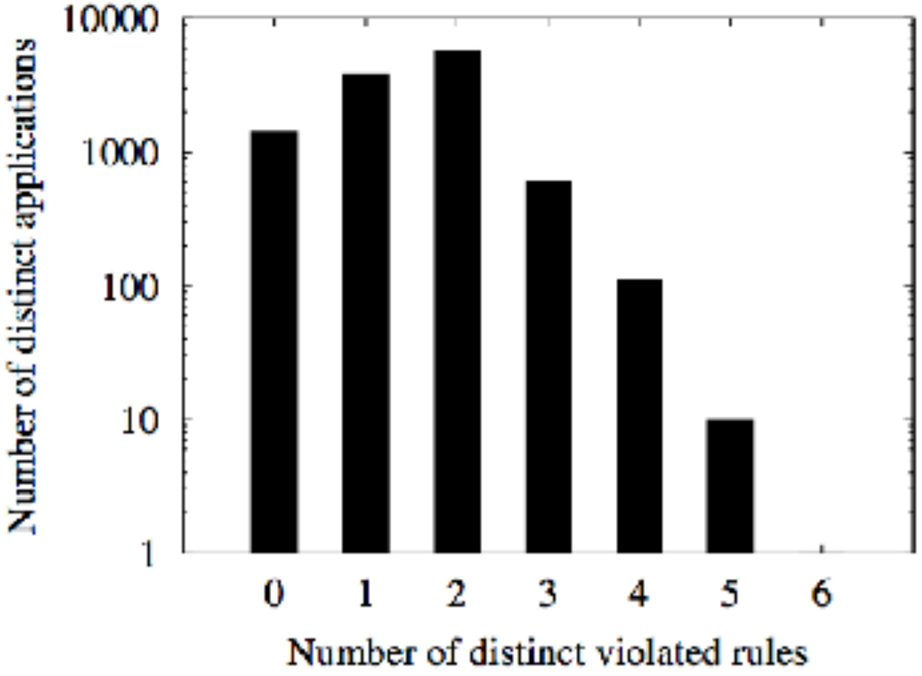
Table 4: Distribution of frequently used symmetric encryption schemes. Schemes marked with * are used in ECB mode by default.

CRYPTO MISUSE IN ANDROID APPS

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

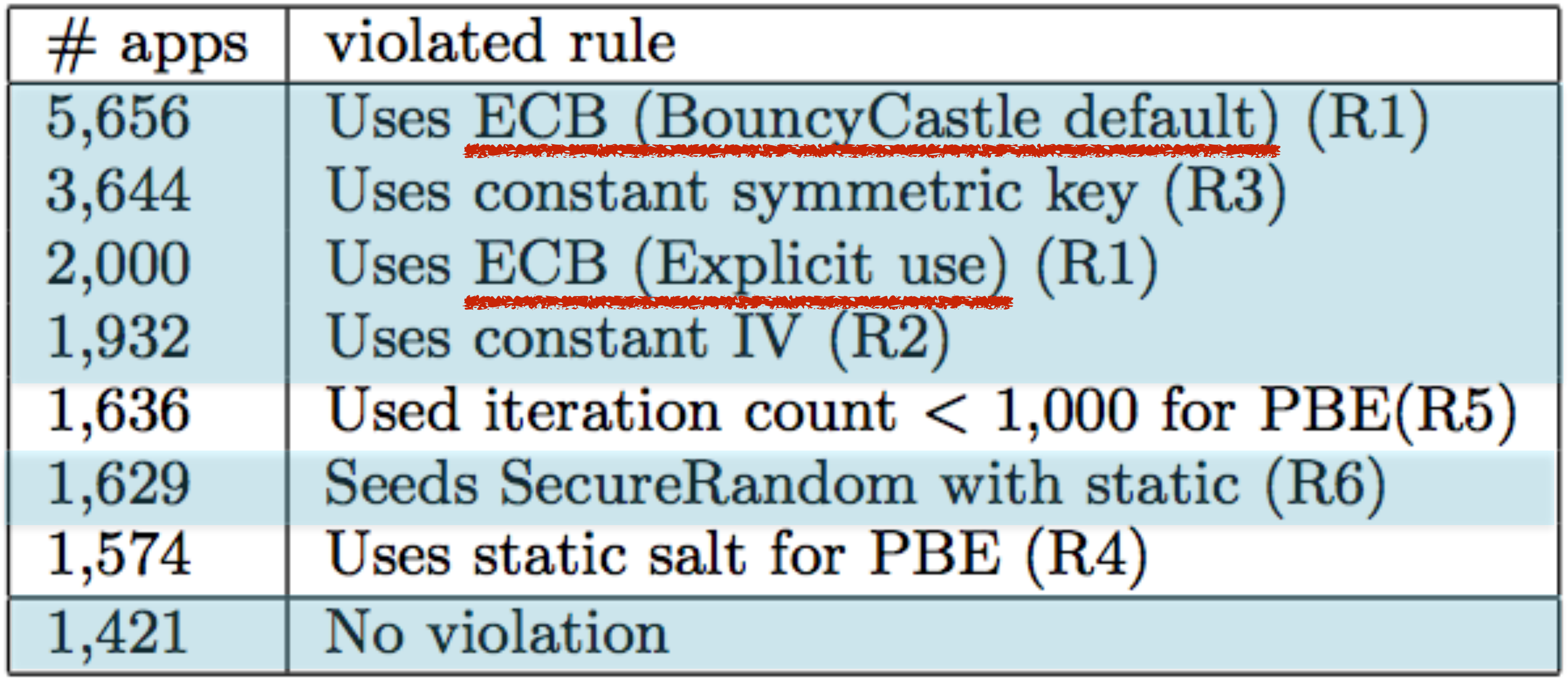


	# apps	violated rule
48%	5,656	Uses ECB (BouncyCastle default) (R1)
31%	3,644	Uses constant symmetric key (R3)
17%	2,000	Uses ECB (Explicit use) (R1)
16%	1,932	Uses constant IV (R2)
	1,636	Used iteration count < 1,000 for PBE(R5)
14%	1,629	Seeds SecureRandom with static (R6)
	1,574	Uses static salt for PBE (R4)
12%	1,421	No violation

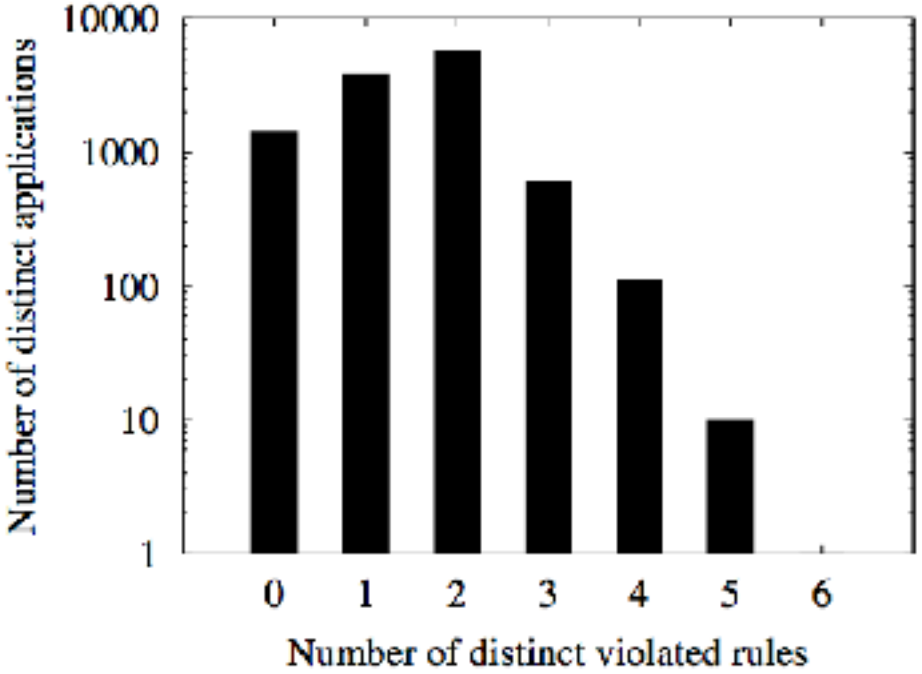


CRYPTO MISUSE IN ANDROID APPS

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



	# apps	violated rule
48%	5,656	Uses ECB (BouncyCastle default) (R1)
31%	3,644	Uses constant symmetric key (R3)
17%	2,000	Uses ECB (Explicit use) (R1)
16%	1,932	Uses constant IV (R2)
	1,636	Used iteration count < 1,000 for PBE(R5)
14%	1,629	Seeds SecureRandom with static (R6)
	1,574	Uses static salt for PBE (R4)
12%	1,421	No violation



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.

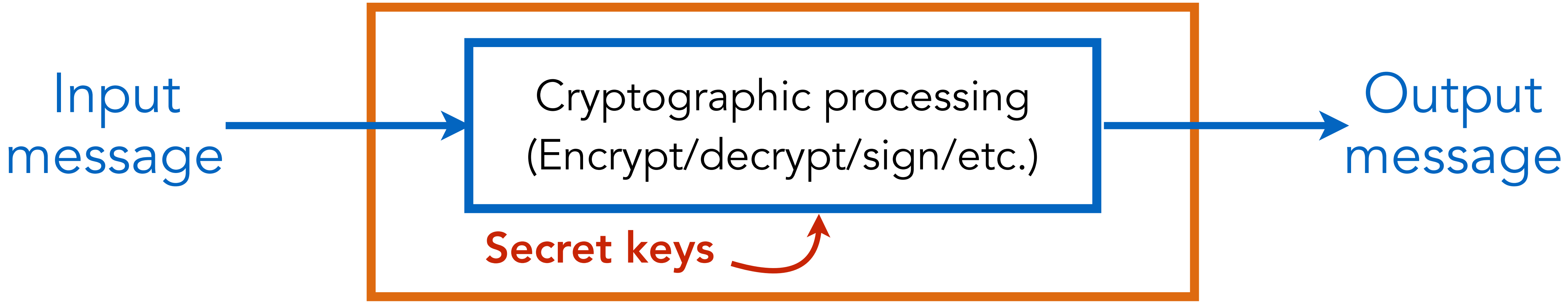
- Kerkhoff's principle: these are all open standards; should be implementable.

- 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



Input message

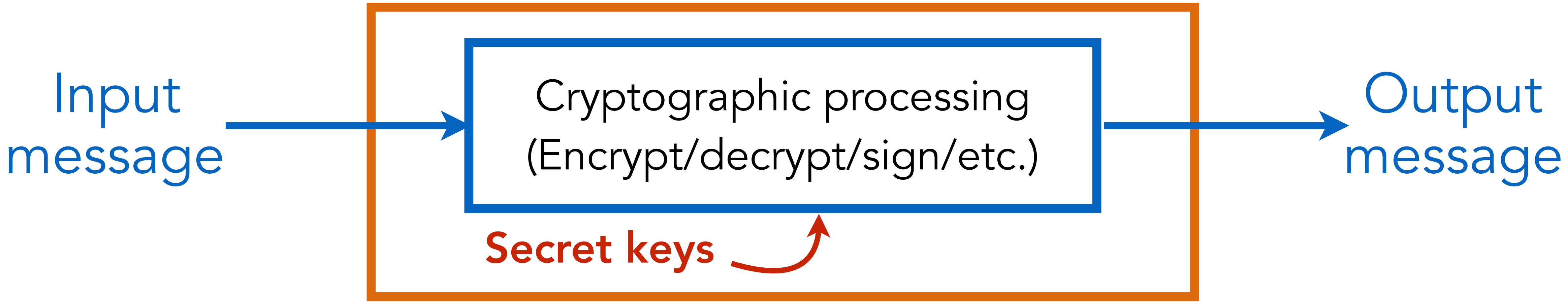
Cryptographic processing (Encrypt/decrypt/sign/etc.)

Secret keys

Output message

SIDE-CHANNEL ATTACKS

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



Input message

Cryptographic processing (Encrypt/decrypt/sign/etc.)

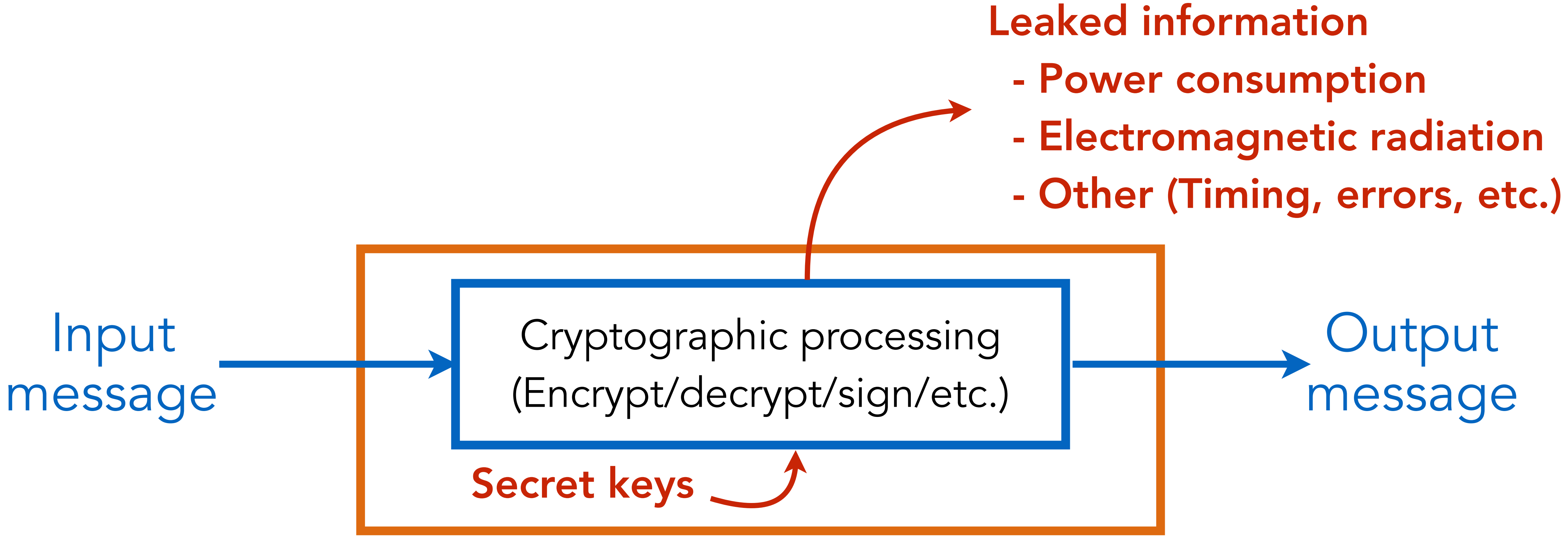
Output message

Secret keys

- But what about the information that a particular *implementation* could leak?
  - Attacks based on these are "**side-channel attacks**"

SIDE-CHANNEL ATTACKS

- 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**"

SIMPLE POWER ANALYSIS (SPA)

- Interpret *power traces* taken during a cryptographic operation
- Simple power analysis can reveal the sequence of instructions executed

SPA ON DES

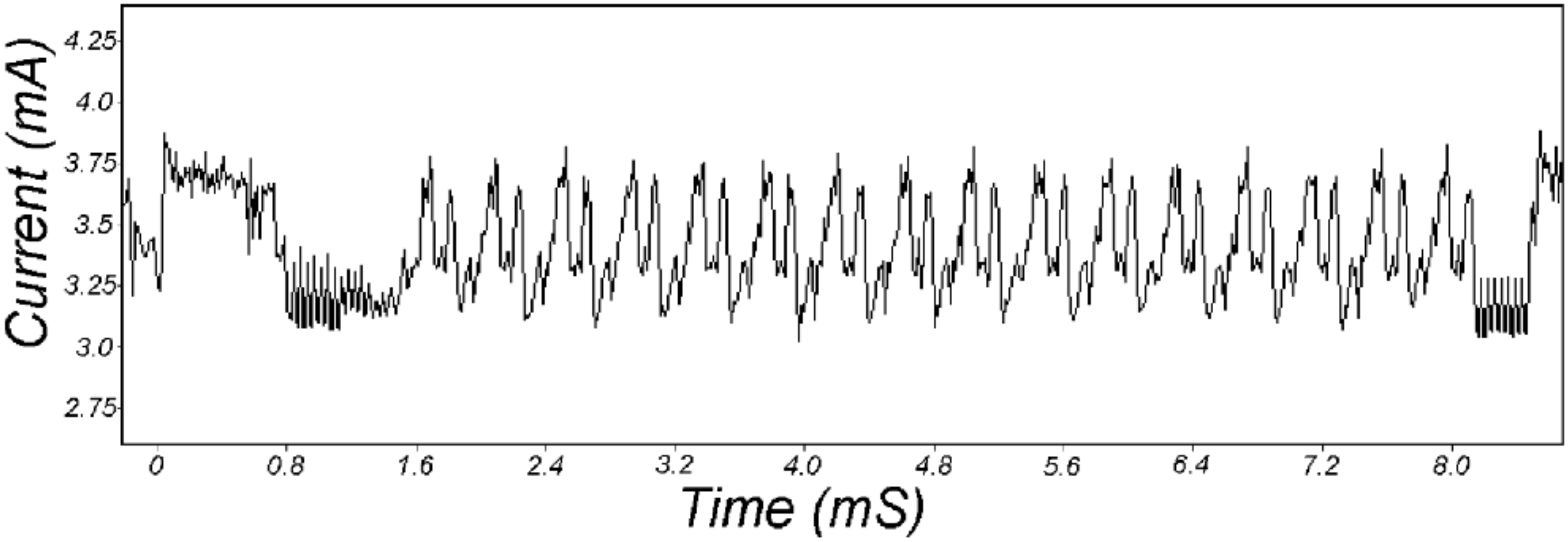


Figure 1: SPA trace showing an entire DES operation.

Overall operation clearly visible:
Can identify the **16 rounds of DES**

SPA ON DES

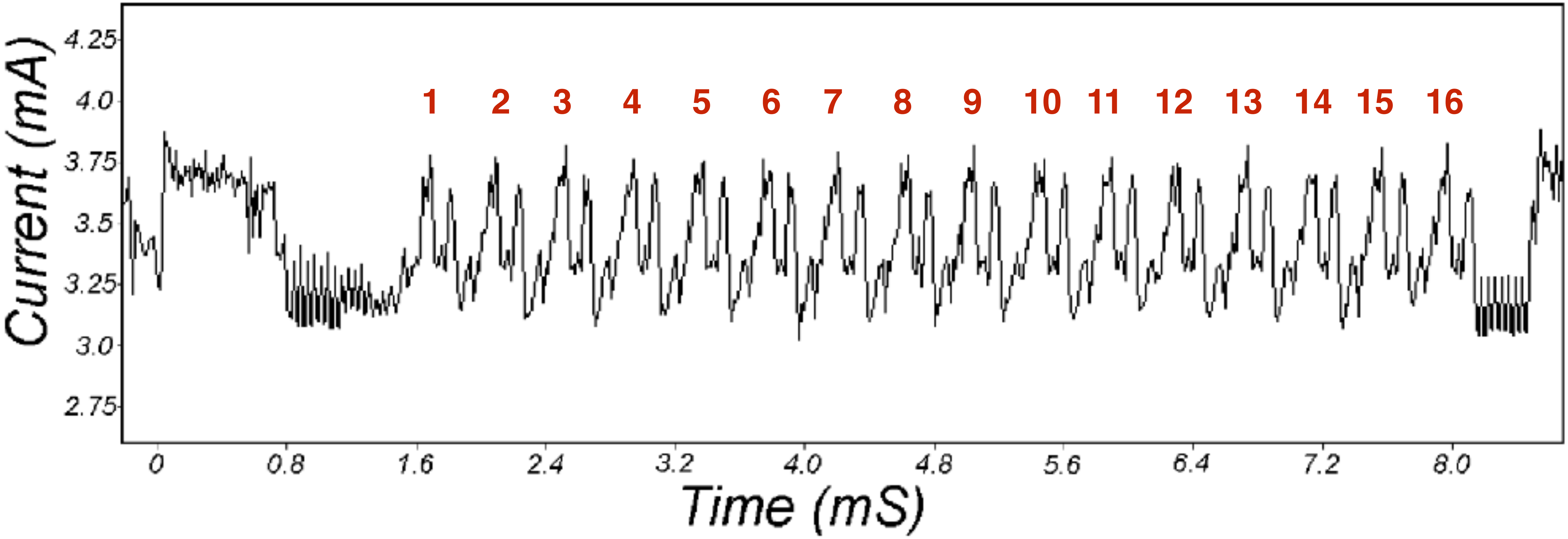


Figure 1: SPA trace showing an entire DES operation.

Overall operation clearly visible:
Can identify the **16 rounds of DES**

SPA ON DES

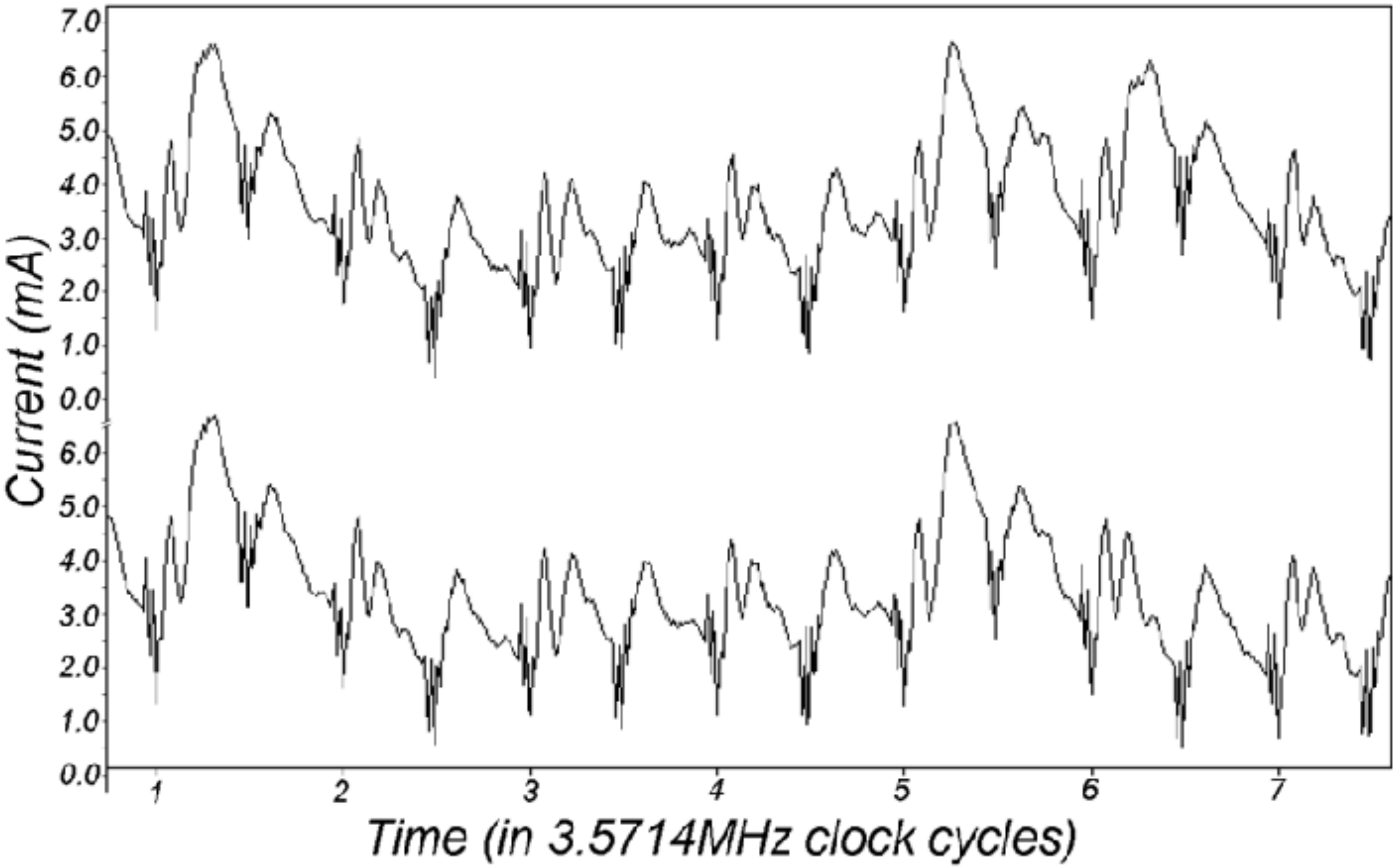
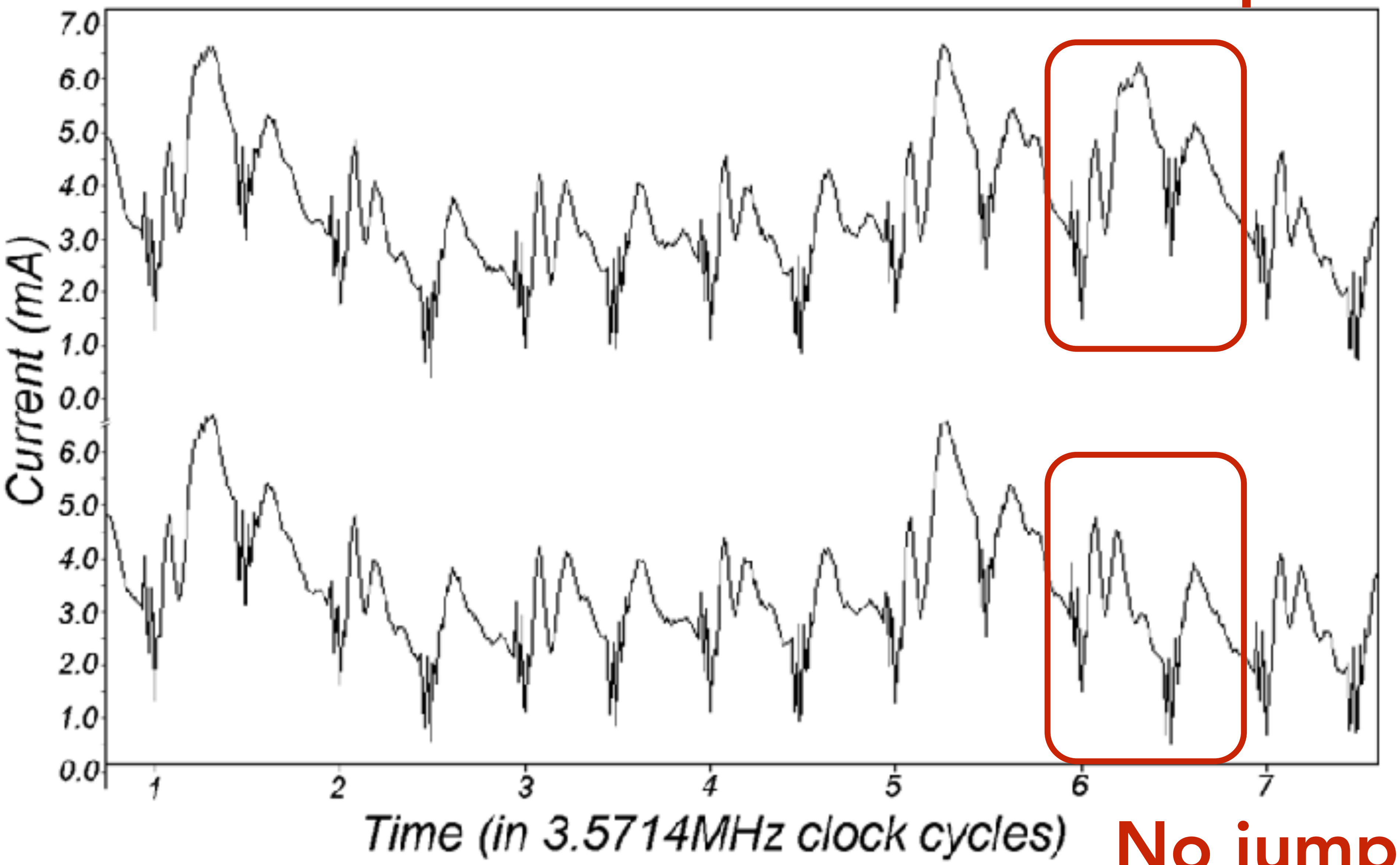


Figure 3: SPA trace showing individual clock cycles.

Specific **instructions** are also discernible

SPA ON DES



Jump taken

No jump taken

Figure 3: SPA trace showing individual clock cycles.

Specific **instructions** are also discernible

HIGH-LEVEL IDEA

```
HypotheticalEncrypt(msg, key) {
  for(int i=0; i < key.len(); i++) {
    if(key[i] == 0)
      // branch 0
    else
      // branch 1
   }
}
```

HIGH-LEVEL IDEA

```
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 brand 1 didn’t?

HIGH-LEVEL IDEA

```
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 brand 1 didn't?

Implementation issue: If the execution path depends on the inputs (key/data), then *SPA can reveal keys*

HIGH-LEVEL IDEA

```
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 brand 1 didn't?

What if branch 0

Implementation issue: If the execution path depends on the inputs (key/data), then *SPA can reveal keys*

HIGH-LEVEL IDEA

```
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 brand 1 didn't?

What if branch 0
- took longer? (timing attacks)

Implementation issue: If the execution path depends on the inputs (key/data), then *SPA can reveal keys*

HIGH-LEVEL IDEA

```
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 brand 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*

HIGH-LEVEL IDEA

```
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 brand 1 didn’t?

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*

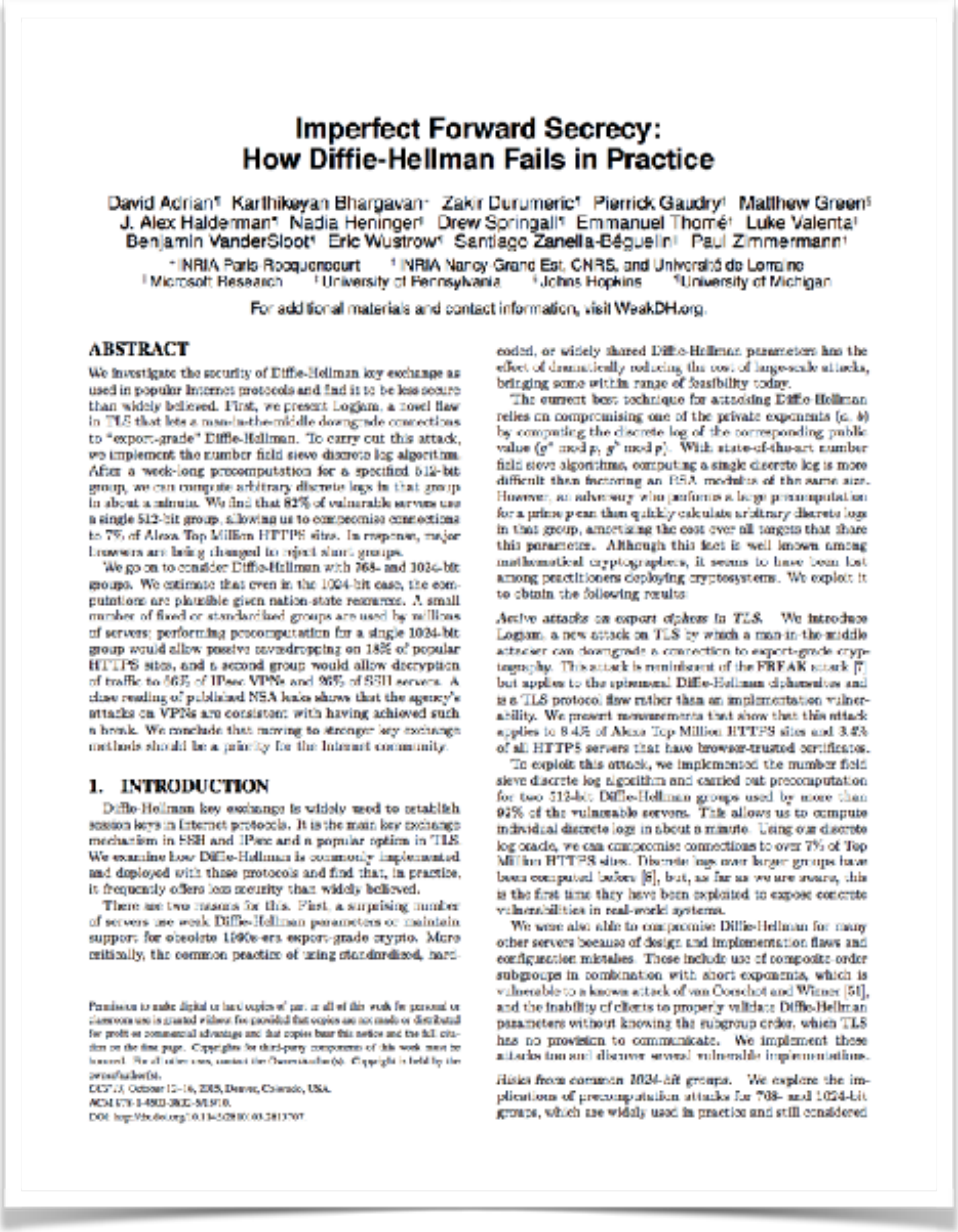
DIFFERENTIAL POWER ANALYSIS (DPA)

- 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



Imperfect Forward Secrecy:
How Diffie-Hellman Fails in Practice

David Adrian¶ Karthikeyan Bhargavan* Zakir Durumeric¶ Pierrick Gaudry† Matthew Green§
J. Alex Halderman¶ Nadia Heninger‡ Drew Springall¶ Emmanuel Thomé† Luke Valenta‡
Benjamin VanderSloot¶ Eric Wustrow¶ Santiago Zanella-Béguelin|| Paul Zimmermann†
*INRIA Paris-Rocquencourt †INRIA Nancy-Grand Est, CNRS, and Université de Lorraine
||Microsoft Research ‡University of Pennsylvania §Johns Hopkins ¶University of Michigan
For additional materials and contact information, visit WeakDH.org.

ABSTRACT

We investigate the security of Diffie-Hellman key exchange as used in popular Internet protocols and find it to be less secure than widely believed. First, we present Logjam, a novel flaw in TLS that lets a man-in-the-middle downgrade connections to "export-grade" Diffie-Hellman. To carry out this attack, we implement the number field sieve discrete log algorithm. After a week-long precomputation for a specified 512-bit group, we can compute arbitrary discrete logs in that group in about a minute. We find that 82% of vulnerable servers use a single 512-bit group, allowing us to compromise connections to 7% of Alexa Top Million HTTPS sites. In response, major browsers are being changed to reject short groups.

We go on to consider Diffie-Hellman with 768- and 1024-bit groups. We estimate that even in the 1024-bit case, the computations are plausible given nation-state resources. A small number of fixed or standardized groups are used by millions of servers; performing precomputation for a single 1024-bit group would allow passive eavesdropping on 18% of popular HTTPS sites, and a second group would allow decryption of traffic to 66% of IPsec VPNs and 26% of SSH servers. A close reading of published NSA leaks shows that the agency's attacks on VPNs are consistent with having achieved such a break. We conclude that moving to stronger key exchange methods should be a priority for the Internet community.

1. INTRODUCTION

Diffie-Hellman key exchange is widely used to establish session keys in Internet protocols. It is the main key exchange mechanism in SSH and IPsec and a popular option in TLS. We examine how Diffie-Hellman is commonly implemented and deployed with these protocols and find that, in practice, it frequently offers less security than widely believed.

There are two reasons for this. First, a surprising number of servers use weak Diffie-Hellman parameters or maintain support for obsolete 1990s-era export-grade crypto. More critically, the common practice of using standardized, hard-coded, or widely shared Diffie-Hellman parameters has the effect of dramatically reducing the cost of large-scale attacks, bringing some within range of feasibility today.

The current best technique for attacking Diffie-Hellman relies on compromising one of the private exponents (a, b) by computing the discrete log of the corresponding public value ($g^a \bmod p$, $g^b \bmod p$). With state-of-the-art number field sieve algorithms, computing a single discrete log is more difficult than factoring an RSA modulus of the same size. However, an adversary who performs a large precomputation for a prime p can then quickly calculate arbitrary discrete logs in that group, amortizing the cost over all targets that share this parameter. Although this fact is well known among mathematical cryptographers, it seems to have been lost among practitioners deploying cryptosystems. We exploit it to obtain the following results:

Active attacks on export ciphers in TLS. We introduce Logjam, a new attack on TLS by which a man-in-the-middle attacker can downgrade a connection to export-grade cryptography. This attack is reminiscent of the FREAK attack [7] but applies to the ephemeral Diffie-Hellman ciphersuites and is a TLS protocol flaw rather than an implementation vulnerability. We present measurements that show that this attack applies to 8.4% of Alexa Top Million HTTPS sites and 3.4% of all HTTPS servers that have browser-trusted certificates.

To exploit this attack, we implemented the number field sieve discrete log algorithm and carried out precomputation for two 512-bit Diffie-Hellman groups used by more than 92% of the vulnerable servers. This allows us to compute individual discrete logs in about a minute. Using our discrete log oracle, we can compromise connections to over 7% of Top Million HTTPS sites. Discrete logs over larger groups have been computed before [8], but, as far as we are aware, this is the first time they have been exploited to expose concrete vulnerabilities in real-world systems.

We were also able to compromise Diffie-Hellman for many other servers because of design and implementation flaws and configuration mistakes. These include use of composite-order subgroups in combination with short exponents, which is vulnerable to a known attack of van Oorschot and Wiener [51], and the inability of clients to properly validate Diffie-Hellman parameters without knowing the subgroup order, which TLS has no provision to communicate. We implement these attacks too and discover several vulnerable implementations.

Risks from common 1024-bit groups. We explore the implications of precomputation attacks for 768- and 1024-bit groups, which are widely used in practice and still considered

CCS'15, October 12–16, 2015, Denver, Colorado, USA.
ACM 978-1-4503-3832-5/15/10.
DOI: http://dx.doi.org/10.1145/2810103.2813707.

Exploits export-grade encryption

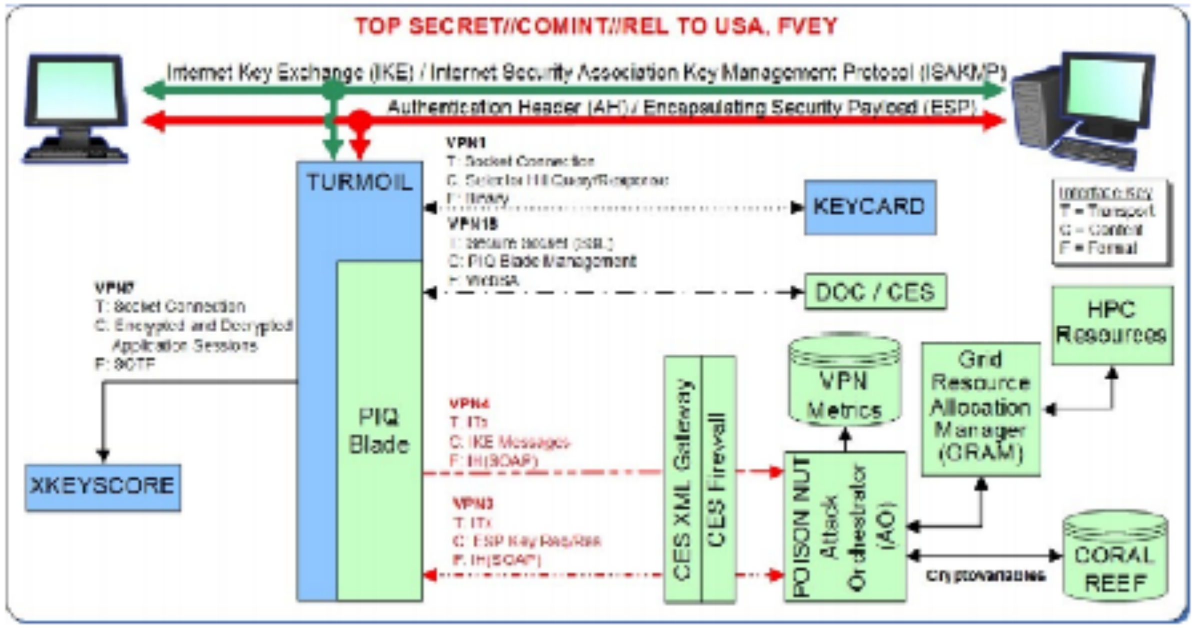


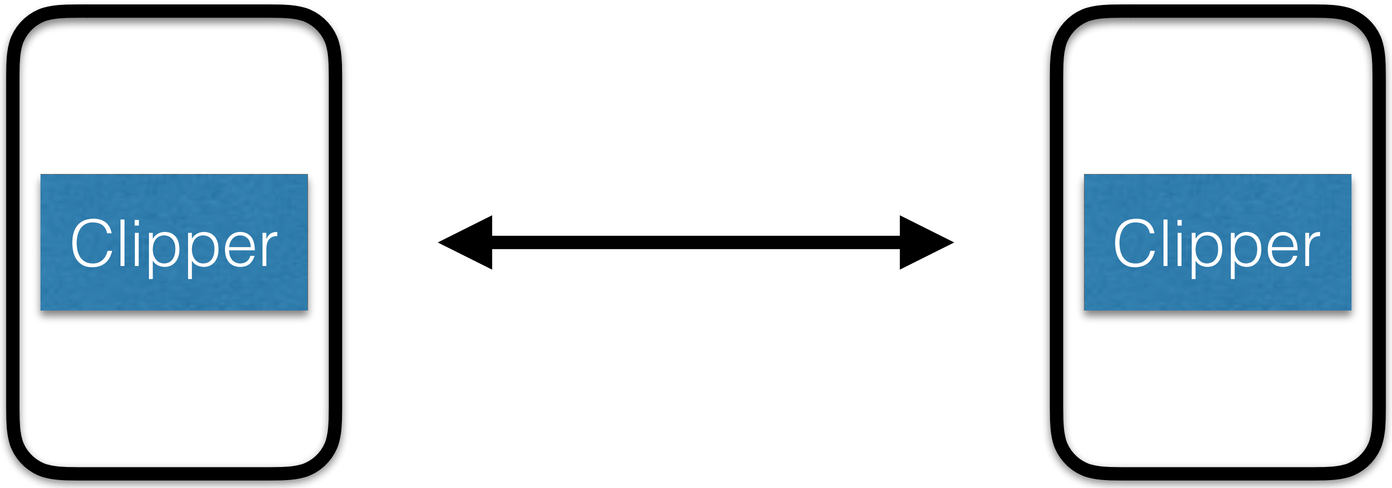
Figure 4: **NSA's VPN decryption infrastructure.** This classified illustration published by Der Spiegel [67] shows captured IKE handshake messages being passed to a high-performance computing system, which returns the symmetric keys for ESP session traffic. The details of this attack are consistent 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 ← Hardware that is difficult to introspect (e.g., extract keys), alter (change the algorithms), or impersonate

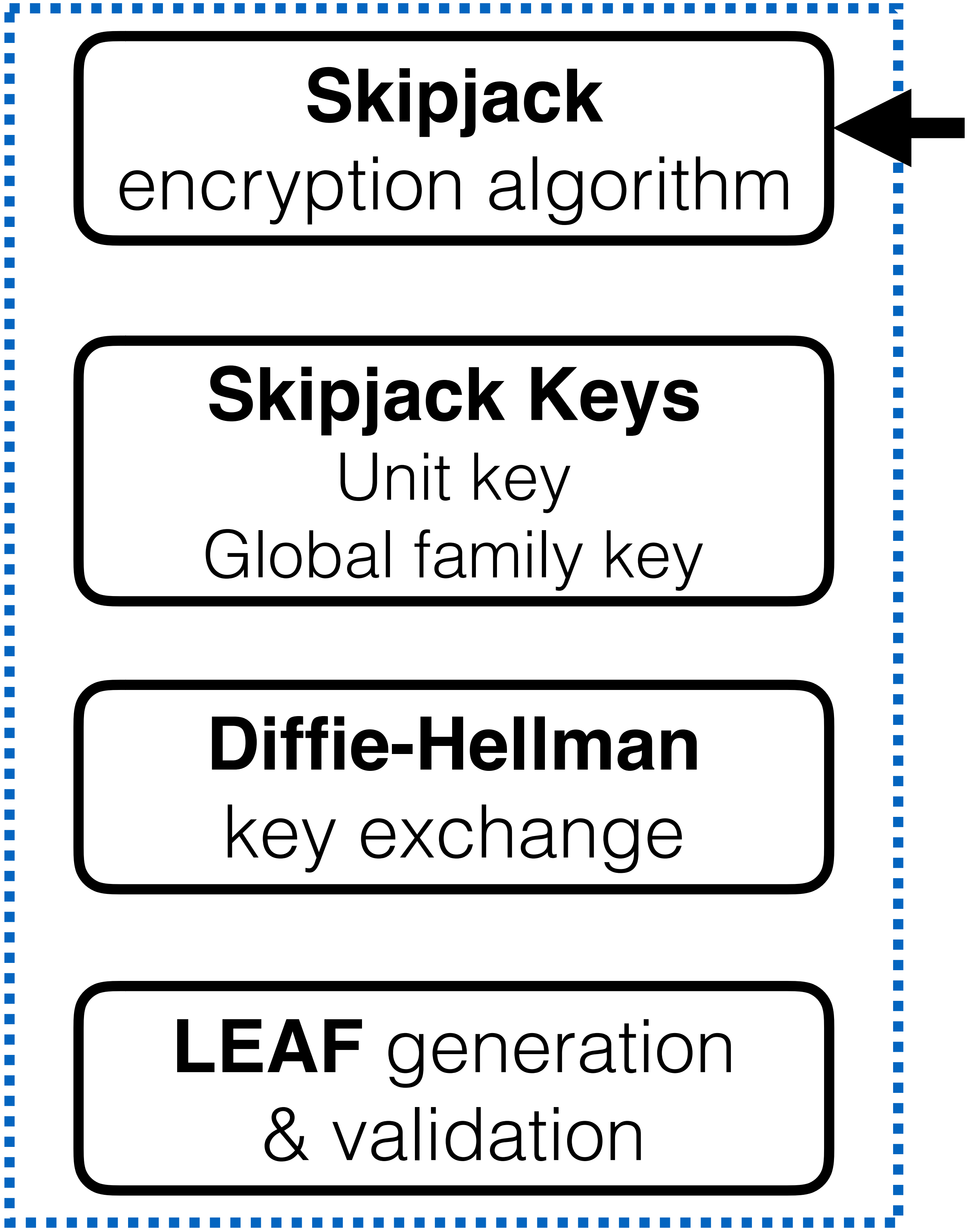
Skipjack
encryption algorithm

Skipjack Keys
Unit key
Global family key

Diffie-Hellman
key exchange

LEAF generation
& validation

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.

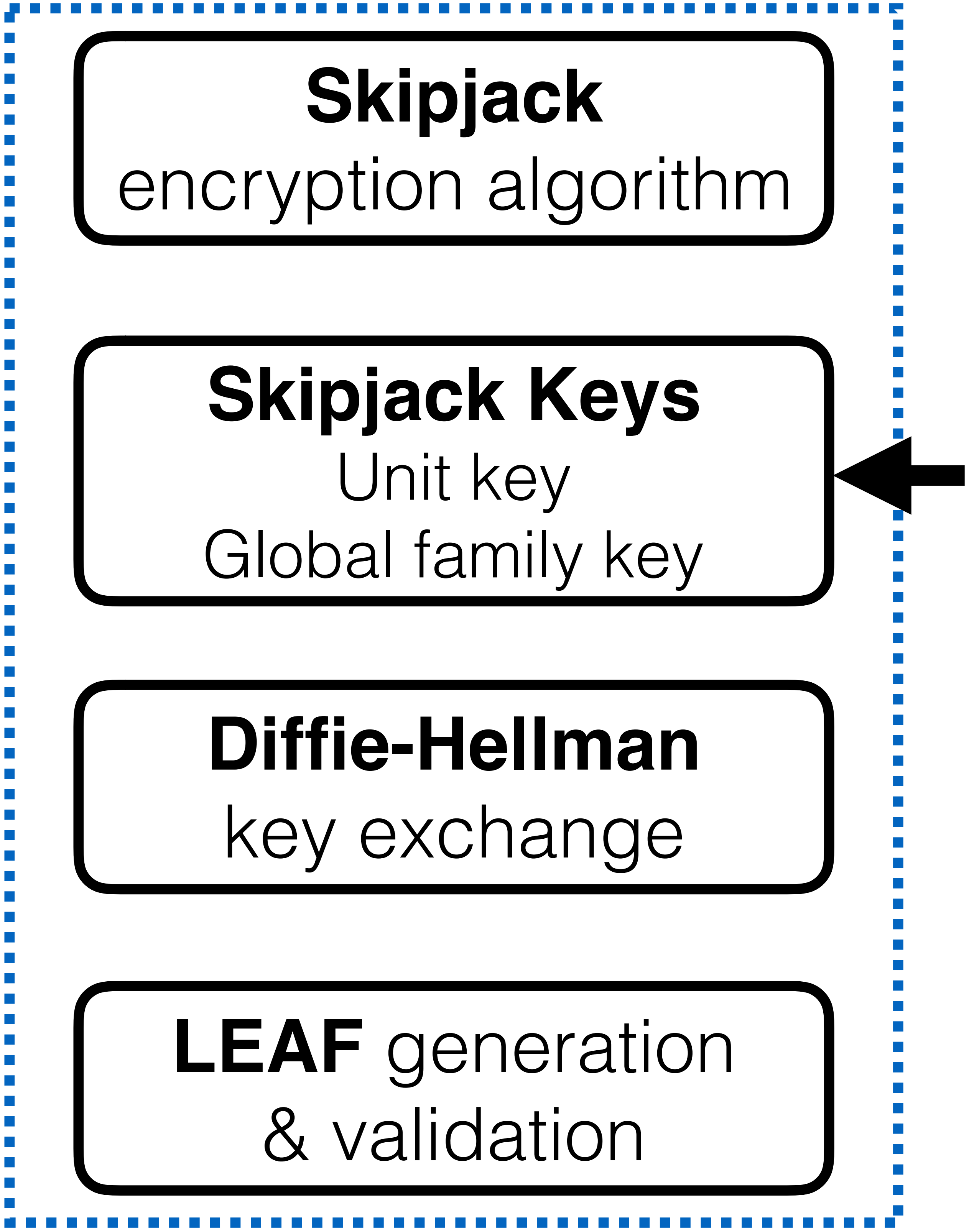
(Violates Kirchhoff's principle)

Broken within *one day* of declassification.

80-bit key; similar algorithm to DES (also broken)

Clipper chip: Design

Tamper-proof hardware



Skipjack
encryption algorithm

Skipjack Keys
Unit key
Global family key

Diffie-Hellman
key exchange

LEAF generation
& validation

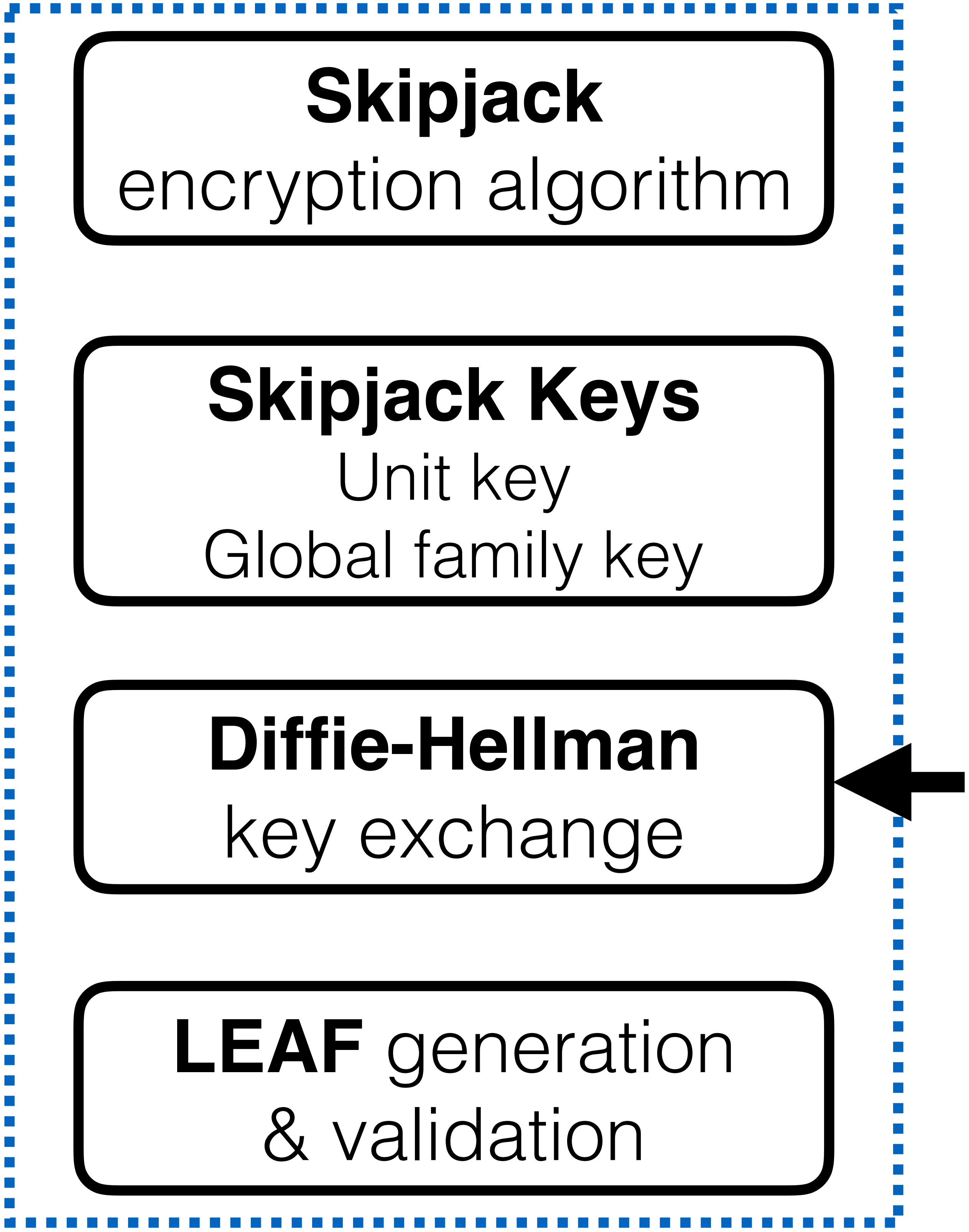
Assigned when the hardware is manufactured.

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

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

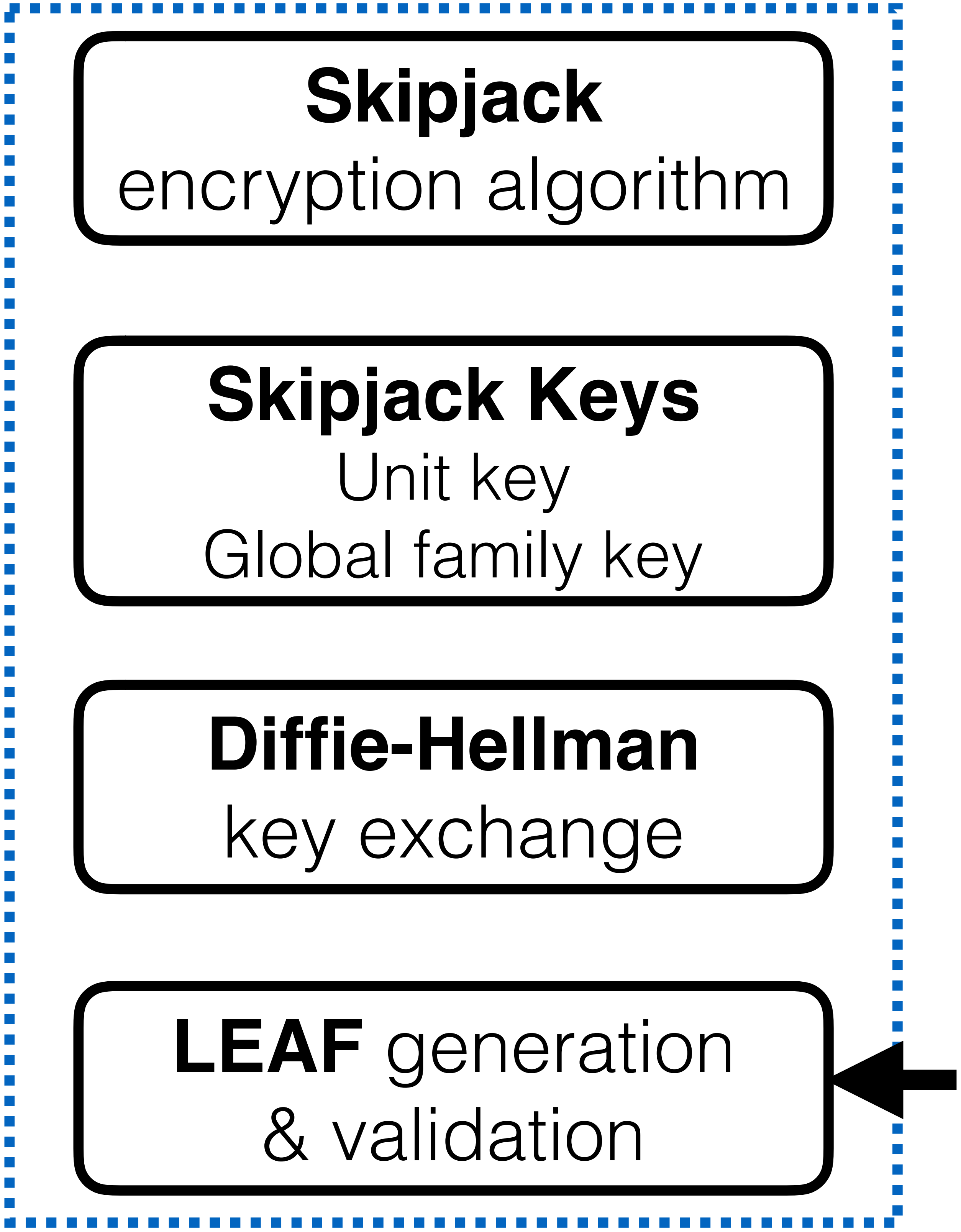
Used for establishing a (symmetric) ***session key***

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

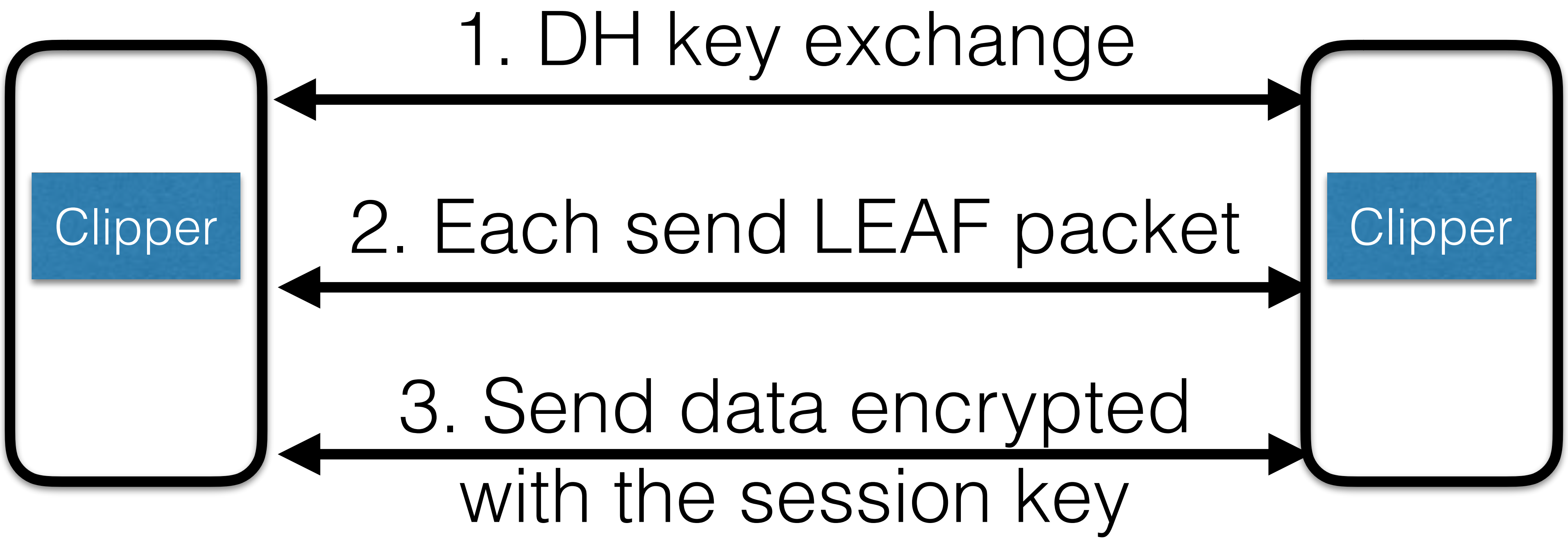
LEAF generation
& validation

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



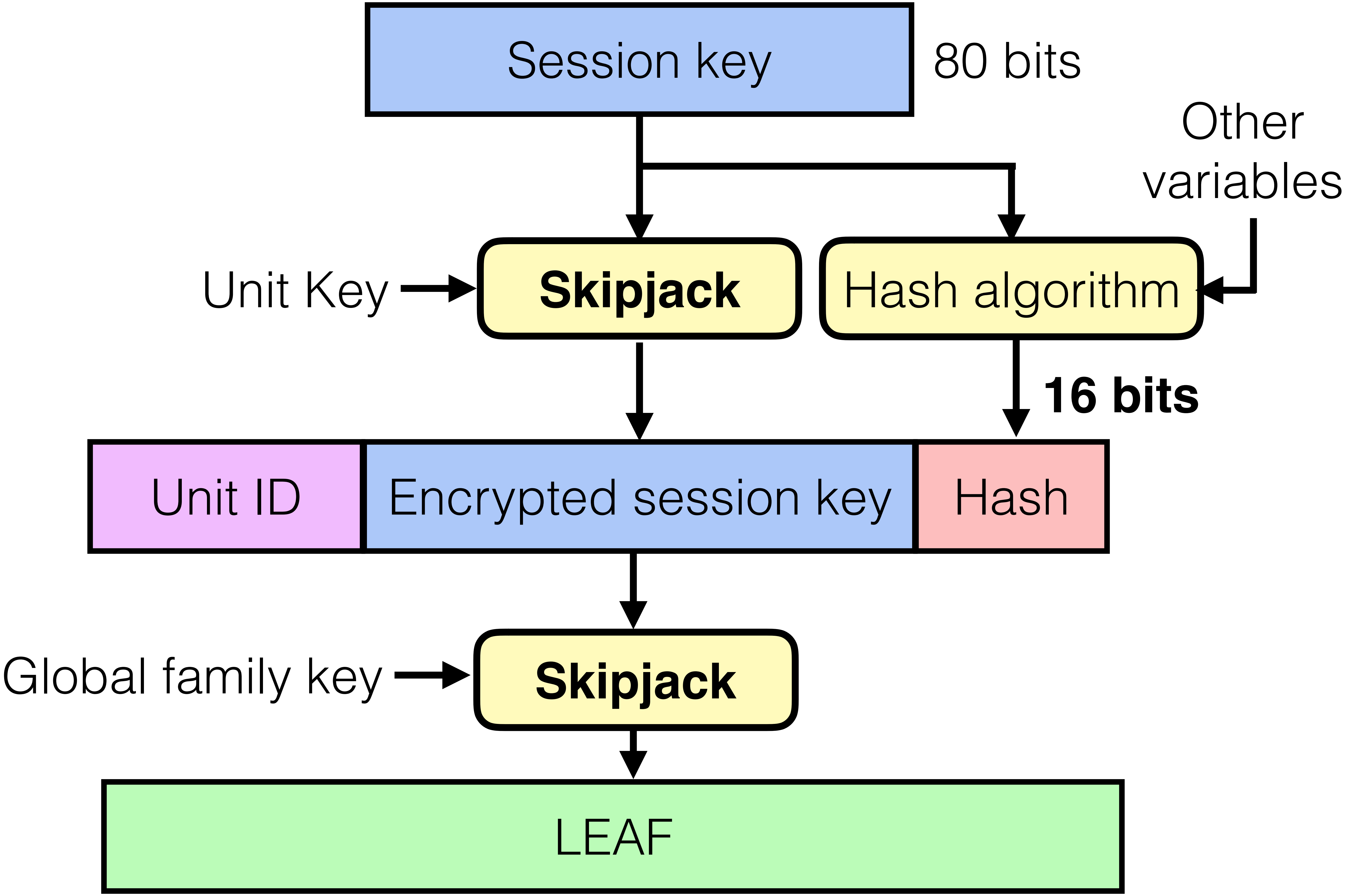
1. DH key exchange
2. Each send LEAF packet
3. Send data encrypted with the session key

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

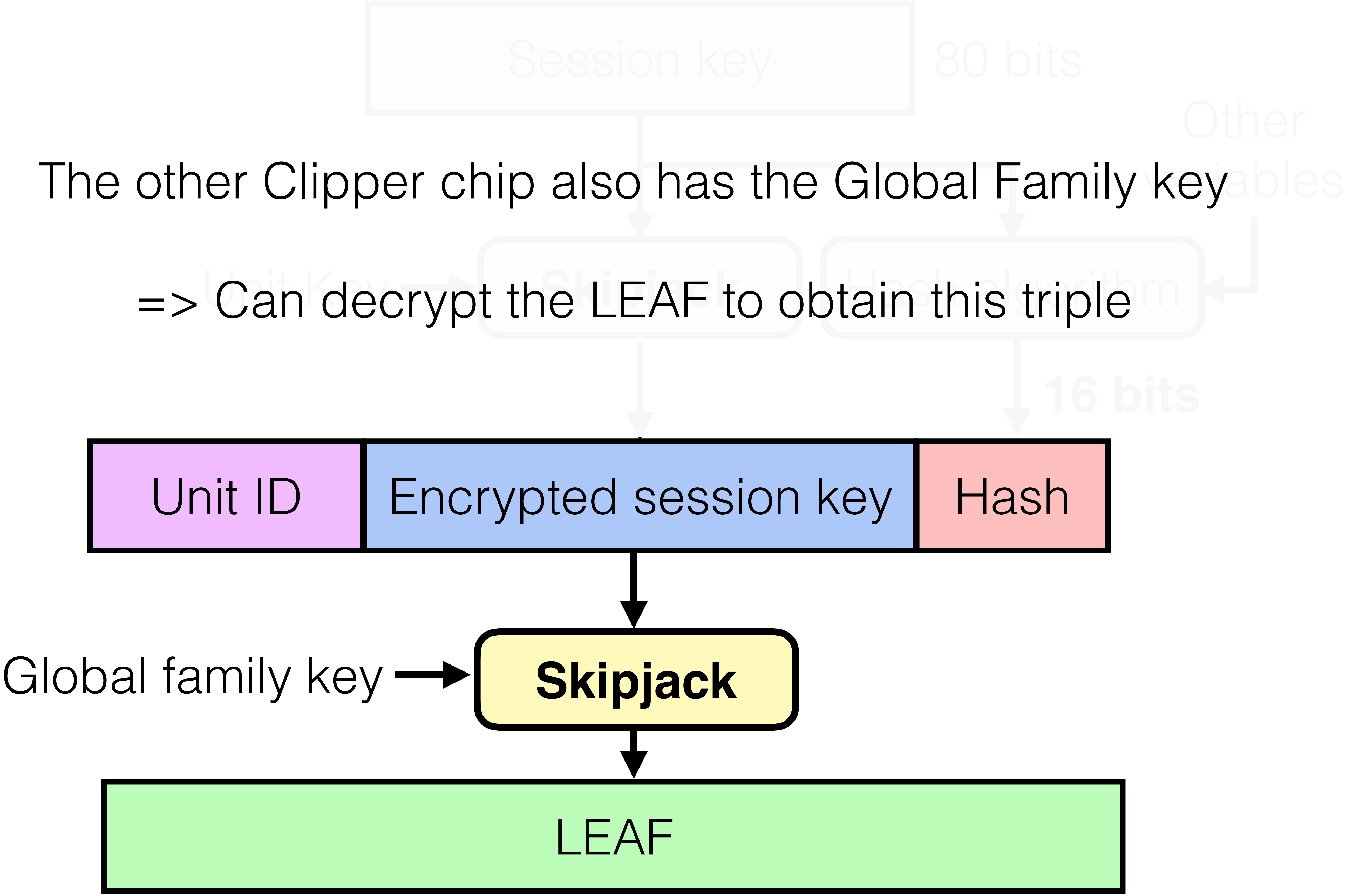
LEAF message structure



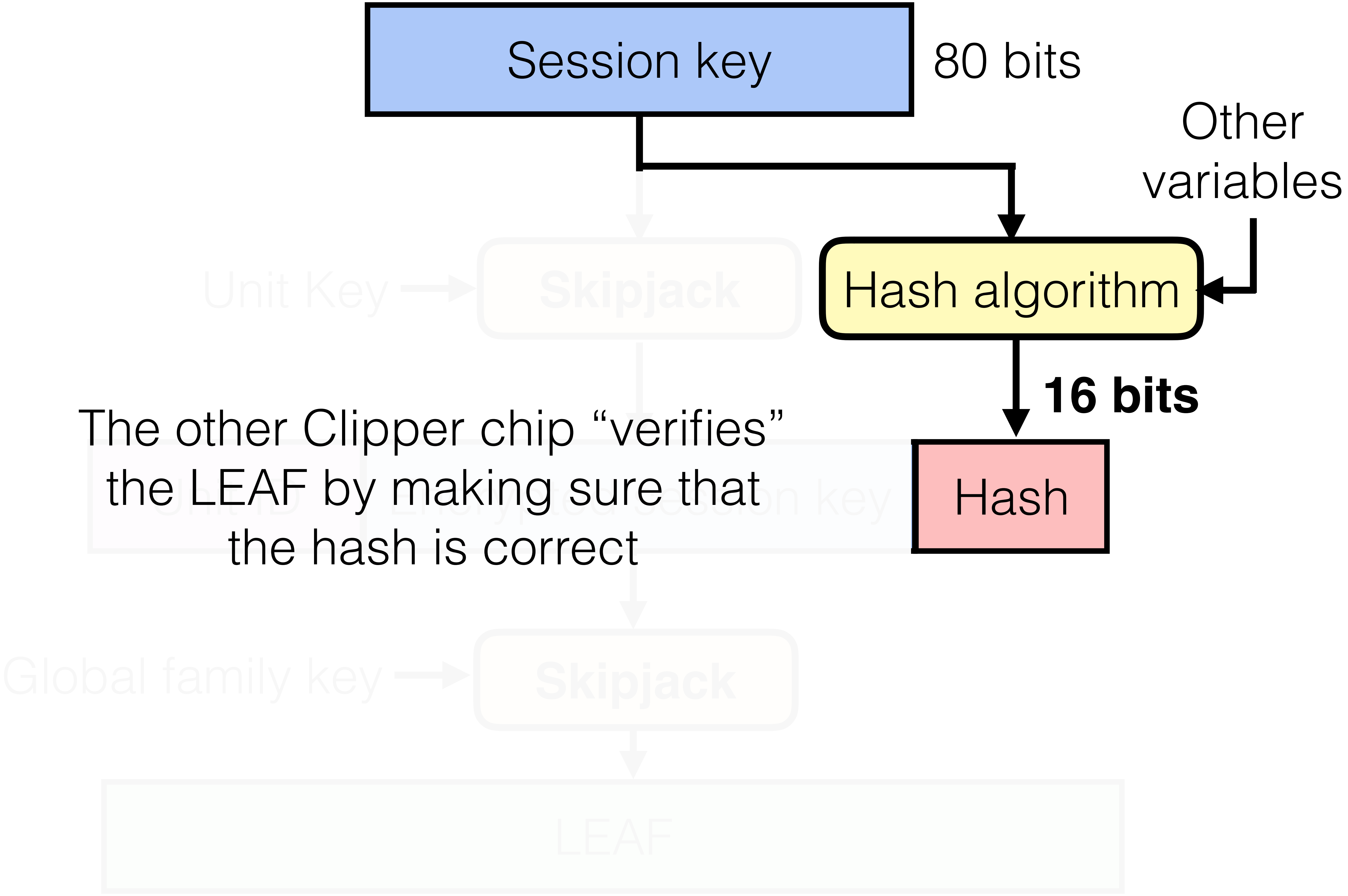
LEAF message structure

The other Clipper chip also has the Global Family key

=> Can decrypt the LEAF to obtain this triple



LEAF message structure



Session key

80 bits

Other variables

Unit Key

Skipjack

Hash algorithm

16 bits

The other Clipper chip “verifies” the LEAF by making sure that the hash is correct

Unit ID

Encrypted session key

Hash

Global family key

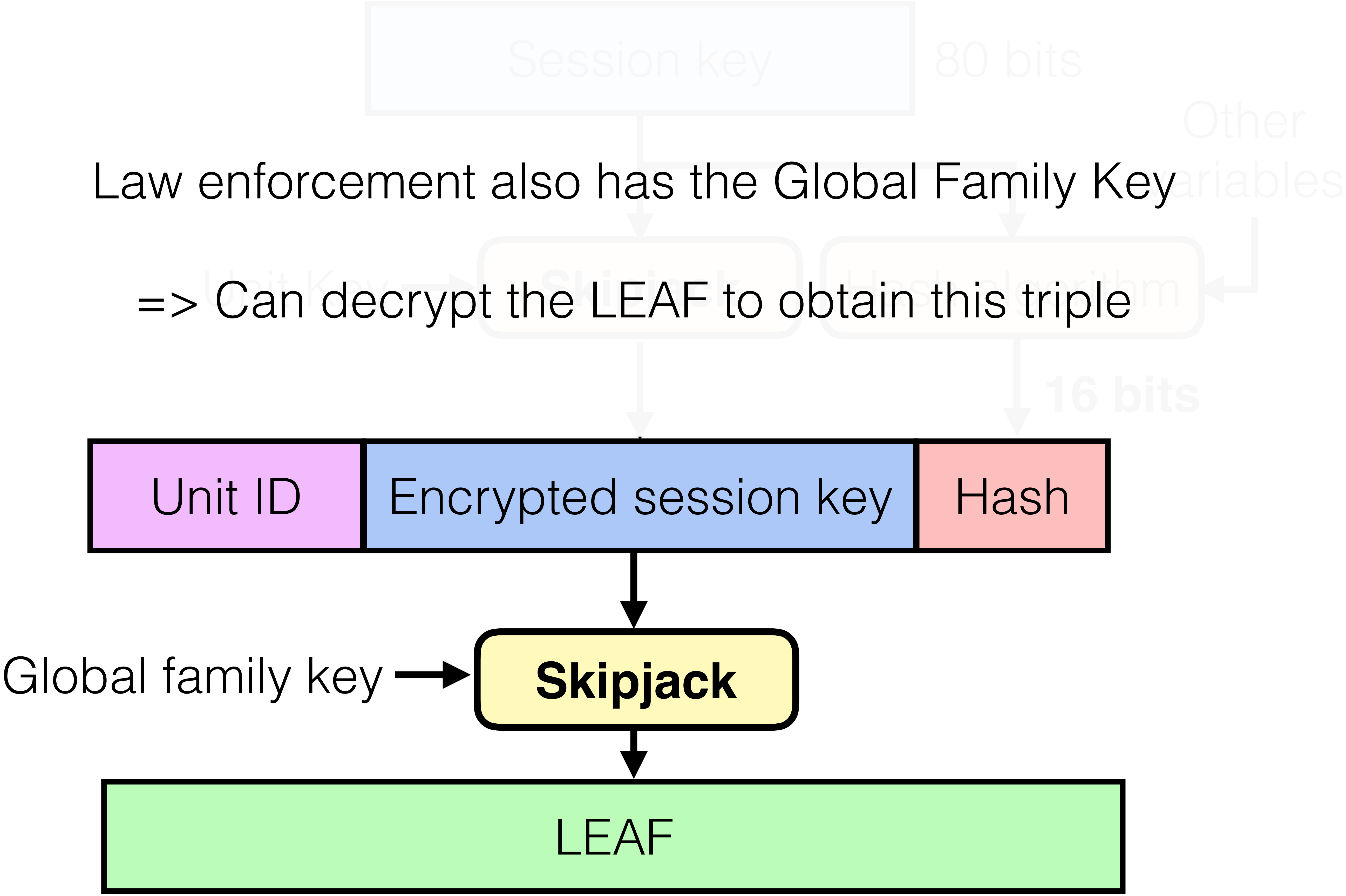
Skipjack

LEAF

LEAF message structure

Law enforcement also has the Global Family Key

=> Can decrypt the LEAF to obtain this triple



Unit ID

Encrypted session key

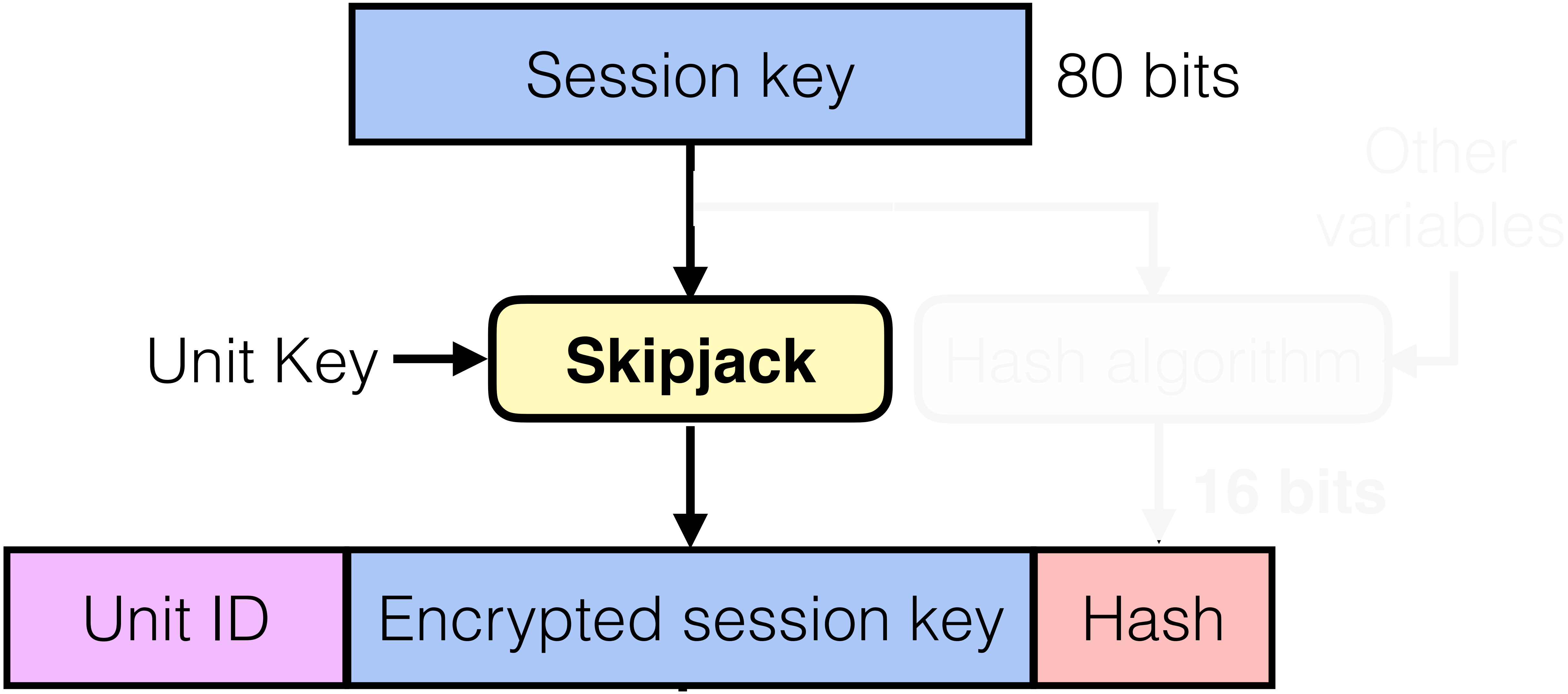
Hash

Global family key

Skipjack

LEAF

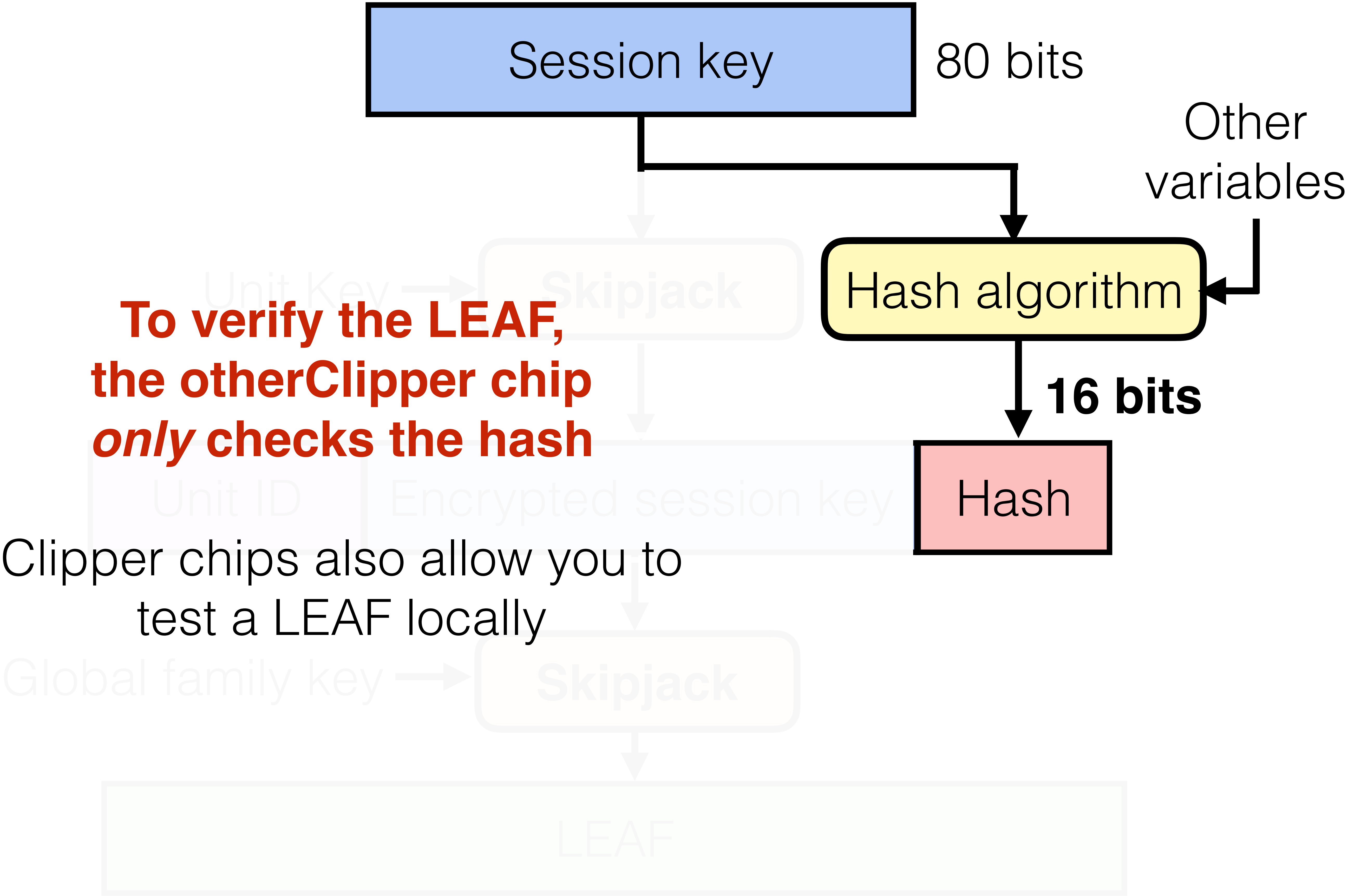
LEAF message structure



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)

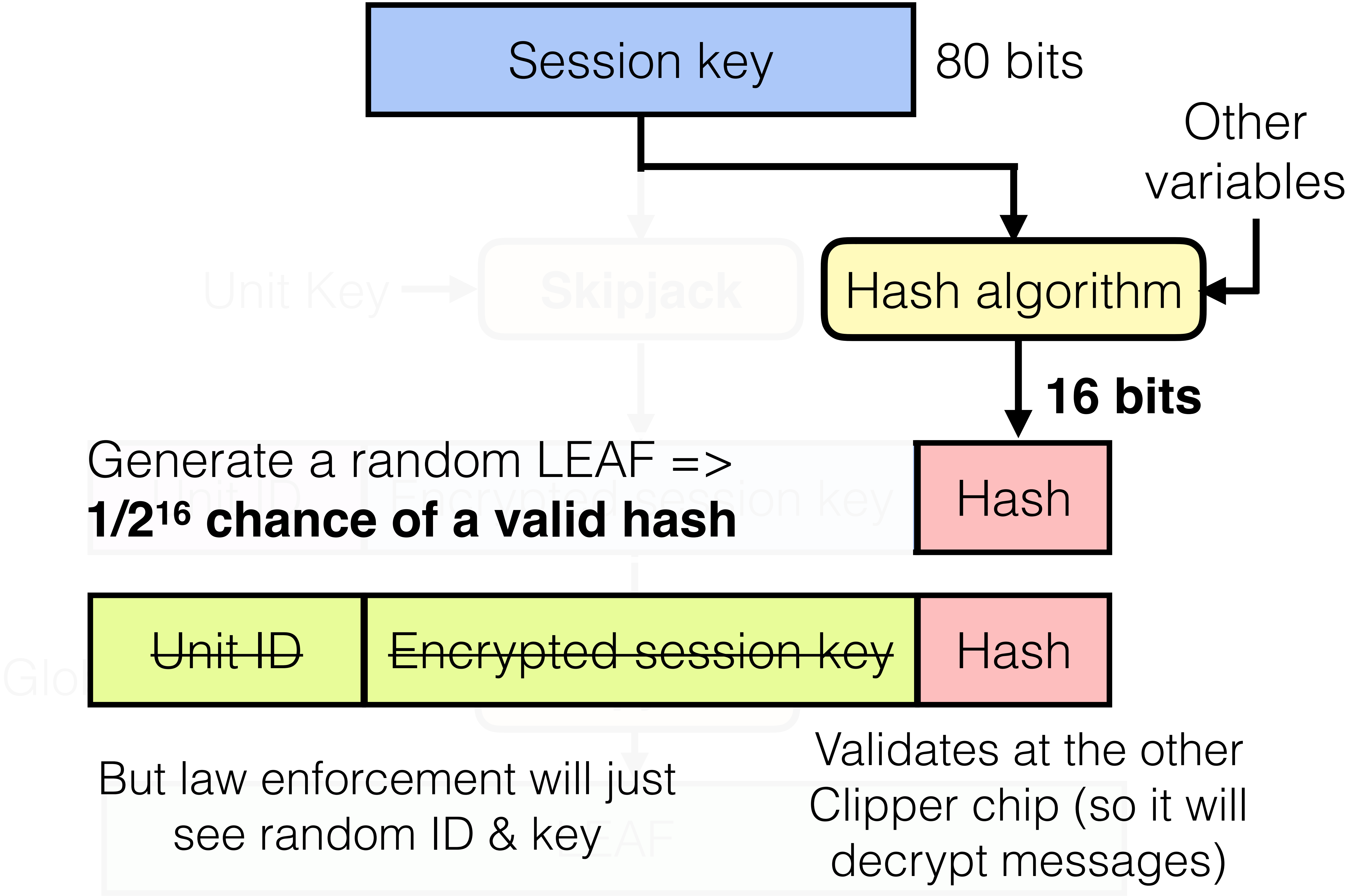
LEAF: failure



**To verify the LEAF,
the otherClipper chip
only checks the hash**

Clipper chips also allow you to test a LEAF locally

LEAF: failure



Generate a random LEAF =>
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)