

# Are Crypto APIs Good Friends of Developers?

Pascal Junod



area41, Zürich (Switzerland), June 10th, 2016

# Outline

- 1 Motivations
- 2 Security Models
- 3 Primitives
- 4 Security Parameters
- 5 Cryptographic Parameters
- 6 API Simplicity
- 7 Conclusion

# Motivations

## Theorem

*Assuming no alien presence, at least 99.9999999857% of the users of crypto libraries on planet Earth are not djb.*

## Proof.

There is a single djb on planet Earth. A lower bound on the world population is  $7 \times 10^9$ . □

- ... yet, many people use crypto libraries on a daily basis.
- Most of the time, the final result **does not bring the expected cryptographic strength.**

# Not Convinced?

## **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

M. Egele et al. *An Empirical Study of Cryptographic Misuse in Android Applications*, ACM-CCS 2012.



# Not Convinced?

## 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 toward improving overall cryptographic security in Android applications.

# Not Convinced?

**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.

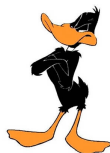
**Rule 4:** Do not use constant salts for PBE. [2, 5]

**Rule 5:** Do not use fewer than 1,000 iterations for PBE. [2, 5]

**Rule 6:** Do not use static seeds to seed `SecureRandom(·)`.

M. Egele et al. *An Empirical Study of Cryptographic Misuse in Android Applications*, ACM-CCS 2012.

# This Talk is not about...



... blaming **arrogant cryptographers** that do not know or do not care about programming and software development issues.

# This Talk is not about...



... trashing **ignorant** crypto libraries **developers** that do not know everything about cryptography.

# This Talk is not about...



... trolling 99.9995% of the **idiot users** of crypto libraries that do horrible mistake most of the time.

# This Talk is not about...

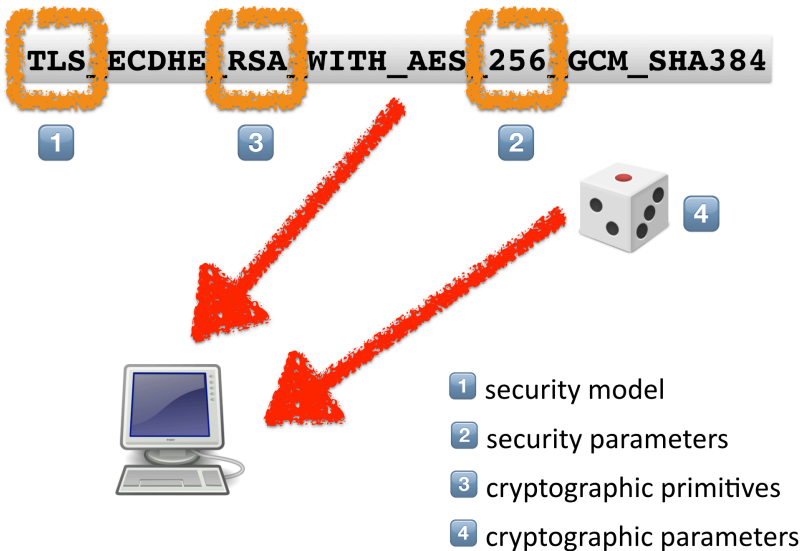


... the **other** security aspects of crypto libraries, namely attacks related to grey- and white-box adversaries: side-channel attacks, SW reverse engineering, etc.

# Goals of this Talk

- Exposing a real-world problem that has consequences every day in practice: the **lack of security-related usability** in most cryptographic libraries.
- Showing a sample of current **good** and **bad** practices.
- Propose **usability requirements** on cryptographic libraries.

# Outline





# Outline

- 1 Motivations
- 2 Security Models**
- 3 Primitives
- 4 Security Parameters
- 5 Cryptographic Parameters
- 6 API Simplicity
- 7 Conclusion

# Symmetric Encryption Security Models

- Just in the symmetric-key encryption setting, many theoretical security models exist:
  - **IND-CPA**: indistinguishability under chosen-plaintext attacks
  - **IND-CCA**: indistinguishability under chosen-ciphertext attacks
  - **INT-PTXT**: integrity of plaintexts
  - **INT-CTXT**: integrity of ciphertexts
  - **NM-CPA**: non-malleability under chosen-plaintext attacks
  - **NM-CCA**: non-malleability under chosen-ciphertext attacks
  - ...

# Security Models

**Definition 2.2 (Non-Malleability of a Symmetric Encryption Scheme [6])** Let  $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$  be a symmetric encryption scheme. Let  $b \in \{0, 1\}$  and  $k \in \mathbb{N}$ . Let  $A_{\text{cpa}} = (A_{\text{cpa}_1}, A_{\text{cpa}_2})$  be an adversary that has access to one oracle and let  $A_{\text{cca}} = (A_{\text{cca}_1}, A_{\text{cca}_2})$  be an adversary that has access to two oracles. Now, we consider the following experiments:

<b>Experiment <math>\text{Exp}_{\mathcal{SE}, A_{\text{cpa}}}^{\text{nm-cpa-}b}(k)</math></b> $K \xleftarrow{R} \mathcal{K}(k)$ $(\vec{c}, s) \leftarrow A_{\text{cpa}_1}^{\mathcal{E}_K(\mathcal{LR}(\cdot, b))}(k)$ $\vec{p} \leftarrow \mathcal{D}_K(\vec{c})$ $x \leftarrow A_{\text{cpa}_2}(\vec{p}, \vec{c}, s)$ <b>Return</b> $x$	<b>Experiment <math>\text{Exp}_{\mathcal{SE}, A_{\text{cca}}}^{\text{nm-cca-}b}(k)</math></b> $K \xleftarrow{R} \mathcal{K}(k)$ $(\vec{c}, s) \leftarrow A_{\text{cca}_1}^{\mathcal{E}_K(\mathcal{LR}(\cdot, b)), \mathcal{D}_K(\cdot)}(k)$ $\vec{p} \leftarrow \mathcal{D}_K(\vec{c})$ $x \leftarrow A_{\text{cca}_2}(\vec{p}, \vec{c}, s)$ <b>Return</b> $x$
---	---

**Definition 2.1 (Indistinguishability of a Symmetric Encryption Scheme [2])** Let  $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$  be a symmetric encryption scheme. Let  $b \in \{0, 1\}$  and  $k \in \mathbb{N}$ . Let  $A_{\text{cpa}}$  be an adversary that has access to one oracle and let  $A_{\text{cca}}$  be an adversary that has access to two oracles. Now, we consider the following experiments:

<b>Experiment <math>\text{Exp}_{\mathcal{SE}, A_{\text{cpa}}}^{\text{ind-cpa-}b}(k)</math></b> $K \xleftarrow{R} \mathcal{K}(k)$ $x \leftarrow A_{\text{cpa}}^{\mathcal{E}_K(\mathcal{LR}(\cdot, b))}(k)$ <b>Return</b> $x$	<b>Experiment <math>\text{Exp}_{\mathcal{SE}, A_{\text{cca}}}^{\text{ind-cca-}b}(k)</math></b> $K \xleftarrow{R} \mathcal{K}(k)$ $x \leftarrow A_{\text{cca}}^{\mathcal{E}_K(\mathcal{LR}(\cdot, b)), \mathcal{D}_K(\cdot)}(k)$ <b>Return</b> $x$
--	--

**Definition 2.3 (Integrity of an Authenticated Encryption Scheme)** Let  $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$  be a symmetric encryption scheme. Let  $k \in \mathbb{N}$ , and let  $A_{\text{ptxt}}$  and  $A_{\text{ctxt}}$  be adversaries each of which has access to two oracles. Consider the following experiments:

<b>Experiment <math>\text{Exp}_{\mathcal{SE}, A_{\text{ptxt}}}^{\text{int-ptxt}}(k)</math></b> $K \xleftarrow{R} \mathcal{K}(k)$ If $A_{\text{ptxt}}^{\mathcal{E}_K(\cdot), \mathcal{D}_K(\cdot)}(k)$ makes a query $C$ to the oracle $\mathcal{D}_K^*(\cdot)$ such that – $\mathcal{D}_K^*(C)$ returns 1, and – $M \stackrel{\text{def}}{=} \mathcal{D}_K(C)$ was never a query to $\mathcal{E}_K(\cdot)$ <b>then return 1 else return 0.</b>	<b>Experiment <math>\text{Exp}_{\mathcal{SE}, A_{\text{ctxt}}}^{\text{int-ctxt}}(k)</math></b> $K \xleftarrow{R} \mathcal{K}(k)$ If $A_{\text{ctxt}}^{\mathcal{E}_K(\cdot), \mathcal{D}_K(\cdot)}(k)$ makes a query $C$ to the oracle $\mathcal{D}_K^*(\cdot)$ such that – $\mathcal{D}_K^*(C)$ returns 1, and – $C$ was never a response of $\mathcal{E}_K(\cdot)$ <b>then return 1 else return 0.</b>
---	--

Source: M. Bellare and C. Namprepmpre, *Authenticated Encryption: Relations among Notions and Analysis of the Generic*

# Security Models

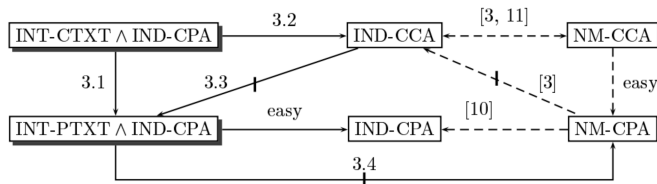
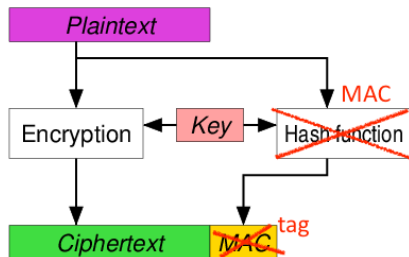


Figure 1: **Relations among notions of symmetric encryption:** An arrow denotes an implication while a barred arrow denotes a separation. The full arrows are relations proved in this paper, annotated with the number of the corresponding Proposition or Theorem, while dotted arrows are reminders of existing relations, annotated with citations to the papers establishing them.

---

Source: M. Bellare and C. Namprempre, *Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm*, Asiacrypt 2000.

# Generic Composition: Encrypt-and-MAC



Source: Wikipedia

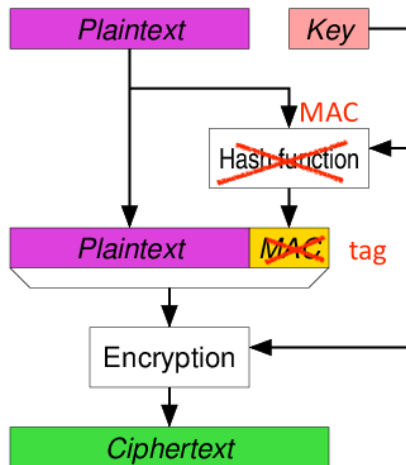
# Security of Generic Composition: EaM

Security		Weak MAC		Strong MAC	
		Result	Reason	Result	Reason
Privacy	IND-CPA	Insecure	Proposition 4.1	Insecure	Proposition 4.1
	IND-CCA	Insecure	IND-CPA insecure and IND-CCA $\rightarrow$ IND-CPA	Insecure	IND-CPA insecure and IND-CCA $\rightarrow$ IND-CPA
	NM-CPA	Insecure	IND-CPA insecure and NM-CPA $\rightarrow$ IND-CPA	Insecure	IND-CPA insecure and NM-CPA $\rightarrow$ IND-CPA
Integrity	INT-PTXT	Secure	Theorem 4.3	Secure	Theorems 4.3 and 2.5
	INT-CTXT	Insecure	Proposition 4.4	Insecure	Proposition 4.4

Figure 4: Summary of results for the *Encrypt-and-MAC* composition method.

Source: M. Bellare and C. Nampreppe, *Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm*, *Asiacrypt* 2000.

# Generic Composition: MAC-then-Encrypt



Source: Wikipedia

# Security of Generic Composition: MtE

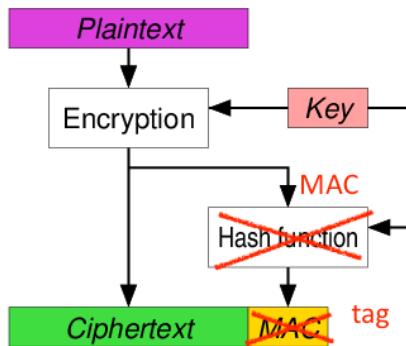
Security		Weak MAC		Strong MAC	
		Result	Reason	Result	Reason
Privacy	IND-CPA	Secure	Theorem 4.5	Secure	Theorem 4.5
	IND-CCA	Insecure	NM-CPA insecure and NM-CPA $\rightarrow$ IND-CCA	Insecure	NM-CPA insecure and NM-CPA $\rightarrow$ IND-CCA
	NM-CPA	Insecure	Proposition 4.6	Insecure	Proposition 4.6
Integrity	INT-PTXT	Secure	Theorem 4.5	Secure	Theorems 4.5 and 2.5
	INT-CTXT	Insecure	IND-CPA secure and NM-CPA insecure and INT-CTXT $\wedge$ IND-CPA $\rightarrow$ NM-CPA	Insecure	IND-CPA secure and NM-CPA insecure and INT-CTXT $\wedge$ IND-CPA $\rightarrow$ NM-CPA

Figure 5: Summary of results for the *MAC-then-encrypt* composition method

Source: M. Bellare and C. Namprepre, *Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm*, *Asiacrypt* 2000.



# Generic Composition: Encrypt-then-MAC



Source: Wikipedia

# Security of Generic Composition: EtM

Security		Weak MAC		Strong MAC	
		Result	Reason	Result	Reason
Privacy	IND-CPA	Secure	Theorem 4.7	Secure	Theorem 4.9
	IND-CCA	Insecure	NM-CPA insecure and NM-CPA $\rightarrow$ IND-CCA	Secure	Theorem 4.9
	NM-CPA	Insecure	Proposition 4.6	Secure	IND-CCA secure and IND-CCA $\rightarrow$ NM-CPA
Integrity	INT-PTXT	Secure	Theorem 4.7	Secure	INT-CTXT secure and INT-CTXT $\rightarrow$ INT-PTXT
	INT-CTXT	Insecure	IND-CPA secure and NM-CPA insecure and INT-CTXT $\wedge$ IND-CPA $\rightarrow$ NM-CPA	Secure	Theorem 4.9

Figure 6: Summary of results for the *encrypt-then-MAC* composition method

Source: M. Bellare and C. Nampreppe, *Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm*, *Asiacrypt* 2000.

# Generic Composition: Summary

- In practice, facing a **passive adversary** only is extremely **infrequent**.
- The only viable symmetric encryption model is **authenticated encryption**.
- There is only two reasonable (i.e., fool-safe) options:
  - use a well-equipped and secure authenticated encryption mode, like e.g. AES-GCM;
  - use an encrypt-then-authenticate scheme, like e.g. AES-CTR + HMAC-SHA256 + HKDF.

# Generic Composition: What could Go Wrong?


- AES in ECB mode;
- AES in CBC mode with constant IV;
- AES in CBC mode with random IV;
- AES in CBC mode with random IV, HMAC-SHA256 on the ciphertext, both using the same key;
- etc.


# Back to Software

- Many SW crypto libraries allow the following options to the developer:
  - secure ciphers without mode of operations;
  - secure ciphers with insecure modes of operations, like AES-ECB, AES-CTR or AES-CBC;
  - secure ciphers with secure modes of operations, but with insecure parameters, like AES-GCM with repeating nonces;
  - using the same key for encryption and authentication when not using authenticated encryption modes;
  - etc.

# Do we Really Need Them?

## Ciphers and cipher modes

- Authenticated cipher modes EAX, OCB, GCM, SIV, CCM, and ChaCha20Poly1305
- Unauthenticated cipher modes CTR, CBC, XTS, CFB, OFB, and ECB 

algorithm type	name
authenticated encryption schemes	<a href="#">GCM</a> , <a href="#">CCM</a> , <a href="#">EAX</a>
<a href="#">high speed</a> stream ciphers	<a href="#">ChaCha8</a> , <a href="#">ChaCha12</a> , <a href="#">ChaCha20</a> , <a href="#">Panama</a> , <a href="#">Sosemanuk</a> , <a href="#">Salsa20</a> , <a href="#">XSalsa20</a>
AES and AES candidates	<a href="#">AES</a> (Rijndael), <a href="#">RC6</a> , <a href="#">MARS</a> , <a href="#">Twofish</a> , <a href="#">Serpent</a> , <a href="#">CAST-256</a>
other <a href="#">block ciphers</a>	<a href="#">IDEA</a> , <a href="#">Triple-DES</a> (DES-EDE2 and DES-EDE3), <a href="#">Camellia</a> , <a href="#">SEED</a> , RC5, Blowfish, TEA, XTEA, Skipjack, SHACAL-2
 <a href="#">block cipher modes of operation</a>	ECB, CBC, CBC ciphertext stealing (CTS), CFB, OFB, counter mode (CTR)
message authentication codes	<a href="#">VMAC</a> , <a href="#">HMAC</a> , <a href="#">GMAC (GCM)</a> , <a href="#">CMAC</a> , CBC-MAC, DMAC, Two-Track-MAC

# Dream Crypto Library Requirement #1

**Offer only high-level APIs  
implementing authenticated  
encryption or sealing.**

## Dream Crypto Library Requirement #2

**Force the developer to build the library with**

- DUSE\_LOW\_LEVEL\_API for having access to other modes; warn at compilation time and in debug mode for every use of an unsecure, low-level API.**



## Example of Good Practice: libsodium

```
#define MESSAGE (const unsigned char *) "test"
#define MESSAGE_LEN 4

unsigned char key[crypto_auth_KEYBYTES];
unsigned char mac[crypto_auth_BYTES];

randombytes_buf(key, sizeof key);
crypto_auth(mac, MESSAGE, MESSAGE_LEN, key);

if (crypto_auth_verify(mac, MESSAGE, MESSAGE_LEN, key) != 0) {
    /* message forged! */
}
```

# Example of Good Practice: libsodium

```
#define MESSAGE ((const unsigned char *) "test")
#define MESSAGE_LEN 4
#define CIPHERTEXT_LEN (crypto_secretbox_MACBYTES + MESSAGE_LEN)

unsigned char nonce[crypto_secretbox_NONCEBYTES];
unsigned char key[crypto_secretbox_KEYBYTES];
unsigned char ciphertext[CIPHERTEXT_LEN];

randombytes_buf(nonce, sizeof nonce);
randombytes_buf(key, sizeof key);
crypto_secretbox_easy(ciphertext, MESSAGE, MESSAGE_LEN, nonce, key);

unsigned char decrypted[MESSAGE_LEN];
if (crypto_secretbox_open_easy(decrypted, ciphertext, CIPHERTEXT_LEN, nonce, key) != 0) {
    /* message forged! */
}
```

# Outline

- 1 Motivations
- 2 Security Models
- 3 Primitives**
- 4 Security Parameters
- 5 Cryptographic Parameters
- 6 API Simplicity
- 7 Conclusion

# The Dark Side of Crypto Diversity

## Ciphers and cipher modes

- Authenticated cipher modes EAX, OCB, GCM, SIV, CCM, and ChaCha20Poly1305
- Unauthenticated cipher modes CTR, CBC, XTS, CFB, OFB, and ECB
- AES (including constant time SSSE3 and AES-NI versions)
- AES candidates Serpent, Twofish, MARS, CAST-256, RC6
- Stream ciphers Salsa20/XSalsa20, ChaCha20, and RC4
- DES, 3DES and DESX
- National/telecom block ciphers SEED, KASUMI, MISTY1, GOST 28147
- Other block ciphers including Threefish-512, Blowfish, CAST-128, IDEA, Noekeon, TEA, XTEA, RC2, RC5, SAFER-SK
- Large block cipher construction Lion

## Hash functions and MACs

- SHA-1, SHA-224, SHA-256, SHA-384, and SHA-512
- SHA-3 winner Keccak-1600
- SHA-3 candidate Skein-512
- Authentication codes HMAC, CMAC, Poly1305, SipHash
- RIPEMD-160, RIPEMD-128, Tiger, Whirlpool
- Hash function combinators (Parallel and Comb4P)
- National standard hashes HAS-160 and GOST 34.11
- Non-cryptographic checksums Adler32, CRC24, CRC32
- Obsolete algorithms MD5, MD4, MD2, CBC-MAC, X9.19 DES-MAC

# The Dark Side of Crypto Diversity

- Many crypto libraries offer the option to use:
  - non-cryptographic algorithms (e.g., CRC32, ADLER, Mersenne Twister)
  - insecure ciphers (e.g., DES, RC4, MD5, SHA1, insecure variants of CBC-MAC)
  - insecure protocols (e.g., SSL 2.0, SSL 3.0)
  - insecure ciphersuites (e.g.,  
    `TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5`);
  - etc.

## Dream Crypto Library Requirement #3

**Force the developer to build the library with `-DUSE_WEAK_CIPHERS` for having access to weak or obsolete ciphers; warn at compilation time and in debug mode for every use of an unsecure, low-level API.**

# Self-Defending Implementations

Example of Triple-DES: a self-defending implementation should check that  $!(\text{key}[0..7] == \text{key}[8..15] == \text{key}[16..23])$ .

```
/*  
 * TripleDES Key Schedule  
 */  
void TripleDES::key_schedule(const byte key[], size_t length)  
{  
    des_key_schedule(&round_key[0], key);  
    des_key_schedule(&round_key[32], key + 8);  
  
    if(length == 24)  
        des_key_schedule(&round_key[64], key + 16);  
    else  
        copy_mem(&round_key[64], &round_key[0], 32);  
}  
  
}
```

# Self-Defending Implementations

## Other example: PyCrypto counter mode API

```
def new(nbits, prefix=b(""), suffix=b(""), initial_value=1, overflow=0, little_endian=False,
        allow_wraparound=False, disable_shortcut=_deprecated):
    """Create a stateful counter block function suitable for CTR encryption modes.
```

Each call to the function returns the next counter block.  
Each counter block is made up by three parts::

```
    prefix || counter value || postfix
```

The counter value is incremented by 1 at each call.



# Self-Defending Implementations

:Parameters:

nbits : integer

Length of the desired counter, in bits. It must be a multiple of 8.

prefix : byte string

The constant prefix of the counter block. By default, no prefix is used.

suffix : byte string

The constant postfix of the counter block. By default, no suffix is used.

initial\_value : integer

The initial value of the counter. Default value is 1.

overflow : integer

This value is currently ignored.

little\_endian : boolean

If *\*True\**, the counter number will be encoded in little endian format.

If *\*False\** (default), in big endian format.

allow\_wraparound : boolean

If *\*True\**, the counter will automatically restart from zero after reaching the maximum value (`2**nbits-1`).

If *\*False\** (default), the object will raise an *\*OverflowError\**.

disable\_shortcut : deprecated

This option is a no-op for backward compatibility. It will be removed in a future version. Don't use it.



# Self-Defending Implementations



**cryptopathe** opened this issue on 14 Nov 2014 · 1 comment



**cryptopathe** commented on 14 Nov 2014

Dear pyCrypto developers,

I can't figure out any useful situation where the `allow_wraparound` flag defined in `lib/Crypto/Util/Counter.py` would be useful in practice. Instead, allowing the counter to wrap in the CTR mode can open a serious security issue (one can XOR the portions of the ciphertext generated with the same counter values, and one immediately gets the XOR of the corresponding parts of the plaintexts, which is a serious problem for non-random plaintexts).

If you split the CTR initial value into a nonce and a random initial counter value, and that you get exceptions because of the counter wrapping around, then it is because your counter has probably a too small bit width for your application. With a proper counter size, this situation should only happen with a probability that is negligible in practice.

I have quickly searched on Github examples where `allow_wraparound` would be set to `True`, and found none.

In summary, this option, introduced in 2.1.0alpha2, can IMHO only help developers to write bad crypto code, and it should be removed.

# Self-Defending Implementations



dlitz commented on 10 Jul

Owner

I think you're probably right.

When I designed `Counter` 7 years ago, I was probably worried about a split nonce being too small and I imagined that people might deal with this by just using a full 128-bit random nonce and relying on their sparse distribution to avoid block collisions. Another reason it might have been useful was to help avoid a timing side-channel in code where authentication happens after decryption.

At this point, though, I don't know of any protocol that actually does either of those things, and the world has moved on to better things like GCM, SIV, and chacha20poly1305. Meanwhile, I *have* seen confusion about the `allow_wraparound` option, and it's also one of those weird PyCrypto-specific options that doesn't exist in other libraries, so it'd probably be wise to just drop it.

Thanks!

## Dream Crypto Library Requirement #4

**Offer only capabilities that are useful for developers and that do not allow him/her to trigger security-related mistakes.**

# Outline

- 1 Motivations
- 2 Security Models
- 3 Primitives
- 4 Security Parameters**
- 5 Cryptographic Parameters
- 6 API Simplicity
- 7 Conclusion

# Self-Defending Implementations ?

- Ciphers designed to have a flexible key length (Blowfish, RC5, etc.): please emit a warning, or even better, refuse to key-schedule in standard conditions when the key length is smaller than 80 bits!
- For instance, `libgcrypt` implementation of Blowfish key-schedule does a self-test, looks for weak keys, but allows 32-bit keys...

# Self-Defending Implementations ?

- Most libraries don't tell anything when you generate an RSA key with a too short key length.
- In 2016, we know that 1024 bits is a strict minimum only valid for data with a very short life, and that we should use at least 1536-bit keys, better 2048-bit ones.

# RSA Case: openssl

```
nowhere:apps pjunod$ ./openssl version
```

```
OpenSSL 1.0.2d 9 Jul 2015
```

```
nowhere:apps pjunod$ ./openssl genrsa
```

```
Generating RSA private key, 2048 bit long modulus
```

```
.....  
.....+++..+++
```

```
e is 65537 (0x10001)
```

```
-----BEGIN RSA PRIVATE KEY-----
```



## RSA Case: openssl

```
nowhere:apps pjunod$ openssl version
```

```
OpenSSL 0.9.8zg 14 July 2015
```

```
nowhere:apps pjunod$ openssl genrsa
```

```
Generating RSA private key, 512 bit long modulus
```

```
.....+++++
```

```
.....+++++
```

```
e is 65537 (0x10001)
```

```
-----BEGIN RSA PRIVATE KEY-----
```

## RSA Case: openssl

```
nowhere:apps pjunod$ ./openssl genrsa 128
```

Generating RSA private key, **128** bit long modulus

```
..+++++
```

```
..+++++
```

```
e is 65537 (0x10001)
```

```
-----BEGIN RSA PRIVATE KEY-----
```

## RSA Case: openssl

```
nowhere:apps pjunod$ ./openssl genrsa 64
```

Generating RSA private key, **64** bit long modulus

```
.+++++
```

```
.+++++
```

```
e is 65537 (0x10001)
```

```
-----BEGIN RSA PRIVATE KEY-----
```

## RSA Case: openssl

```
nowhere:apps pjunod$ ./openssl genrsa 32
```

Generating RSA private key, **32** bit long modulus

```
.+++++
```

```
.+++++
```

```
e is 65537 (0x10001)
```

```
-----BEGIN RSA PRIVATE KEY-----
```

# RSA Case: openssl

```
for (;;) {  
    /*  
     * When generating ridiculously small keys, we can get stuck  
     * continually regenerating the same prime values. Check for this and  
     * bail if it happens 3 times.  
     */  
    unsigned int degenerate = 0;  
    do {  
        if (!BN_generate_prime_ex(rsa->q, bitsq, 0, NULL, NULL, cb))  
            goto err;  
    } while ((BN_cmp(rsa->p, rsa->q) == 0) && (++denerate < 3));  
    if (denerate == 3) {  
        ok = 0;                /* we set our own err */  
        RSAerr(RSA_F_RSA_BUILTIN_KEYGEN, RSA_R_KEY_SIZE_TOO_SMALL);  
        goto err;  
    }  
}
```

# RSA Case: libressl



[LibreSSL 2.4.0](#) released May 31st, 2016

LibreSSL is a version of the TLS/crypto stack forked from [OpenSSL](#) in 2014, with goals of modernizing the codebase, improving security, and applying best practice development processes.

```
eduroam-3-153:openssl pjunod$ ./openssl genrsa 32
```

Generating RSA private key, **32** bit long modulus

```
.+++++
```

```
.+++++
```

```
e is 65537 (0x10001)
```

```
-----BEGIN RSA PRIVATE KEY-----
```

# RSA Case: BoringSSL

---

## BoringSSL

BoringSSL is a fork of OpenSSL that is designed to meet Google's needs.

```
somewhere:tool pjunod$ ./bssl genrsa -bits 32
-----BEGIN RSA PRIVATE KEY-----
MCsCAQACBQC7lZ3NAgMBAAECBFg1hoECAwDdoQIDANitAgIEAQICQOECAmWN
-----END RSA PRIVATE KEY-----
somewhere:tool pjunod$ █
```

## RSA Case: libgcrypt

```
static gpg_err_code_t
generate_std (RSA_secret_key *sk, unsigned int nbits, unsigned long use_e,
             int transient_key)
{
    gcry_mpi_t p, q; /* the two primes */
    gcry_mpi_t d;    /* the private key */
    gcry_mpi_t u;
    gcry_mpi_t t1, t2;
    gcry_mpi_t n;    /* the public key */
    gcry_mpi_t e;    /* the exponent */
    gcry_mpi_t phi;  /* helper: (p-1)(q-1) */
    gcry_mpi_t g;
    gcry_mpi_t f;
    gcry_random_level_t random_level;

    if (fips_mode ())
    {
        if (nbits < 1024)
            return GPG_ERR_INV_VALUE;
        if (transient_key)
            return GPG_ERR_INV_VALUE;
    }
}
```



# RSA Case: Botan in 2015

```
#include <botan/keypair.h>
#include <botan/internal/assert.h>

namespace Botan {

/*
 * Create a RSA private key
 */
RSA_PrivateKey::RSA_PrivateKey(RandomNumberGenerator& rng,
                                size_t bits, size_t exp)
{
    if(bits < 512)
        throw Invalid_Argument(algo_name() + ": Can't make a key that is only " +
                                to_string(bits) + " bits long");
    if(exp < 3 || exp % 2 == 0)
        throw Invalid_Argument(algo_name() + ": Invalid encryption exponent");
}
```

# RSA Case: Botan in 2016

```
namespace Botan {  
  
    /*  
    * Create a RSA private key  
    */  
    RSA_PrivateKey::RSA_PrivateKey(RandomNumberGenerator& rng,  
                                    size_t bits, size_t exp)  
    {  
        if(bits < 1024)  
            throw Invalid_Argument(algo_name() + ": Can't make a key that is only " +  
                                    std::to_string(bits) + " bits long");  
        if(exp < 3 || exp % 2 == 0)  
            throw Invalid_Argument(algo_name() + ": Invalid encryption exponent");  
    }  
}
```

## RSA Case: Keyczar

```
// static
RSAPrivateKey* RSAPrivateKey::GenerateKey(int size) {
    if (!KeyType::IsValidCipherSize(KeyType::RSA_PRIV, size))
        return NULL;

    static const int kAESSizes[] = {128, 192, 256, 0};
#ifdef COMPAT_KEYCZAR_06B
    static const int kHMACSHA1Sizes[] = {256, 0};
    static const int kRSASizes[] = {512, 768, 1024, 2048, 3072, 4096, 0};
#else
    static const int kHMACSizes[] = {160, 224, 256, 384, 512, 0};
    static const int kRSASizes[] = {1024, 2048, 3072, 4096, 0};
#endif
    static const int kDSASizes[] = {1024, 2048, 3072, 0};
    static const int kECDSASizes[] = {192, 224, 256, 384, 0};
```

# Myriad of Other Examples

- Small Diffie-Hellman groups
- Weak elliptic curves
- Weak ciphersuites (EXPORT, involving DES, ...)
- etc.

## Dream Crypto Library Requirement #5

- **Offer only security parameters and configuration options that are meaningful in terms of security.**
- **Offer safe security parameters by default.**
- **If really required, weaker/obsolete variants could be made available, but such that it requires an effort from the developer and emitting warnings at compilation time.**

# Outline

- 1 Motivations
- 2 Security Models
- 3 Primitives
- 4 Security Parameters
- 5 Cryptographic Parameters**
- 6 API Simplicity
- 7 Conclusion

# Cryptographic Parameters

- Without (cryptographically secure) randomness, no cryptographic security!
- Most cryptographic libraries do a rather good job in providing a cryptographically secure PRNG (CPRNG) to the developer.
- However, most cryptographic libraries put the **responsibility to use the CPRNG** on the developer shoulders.
- Two bad things can happen:
  - Forget to use random values;
  - Use a non-cryptographic PRNG.

# Cryptographic Parameters: Nonces

If you use the code in the [authenticated encryption](#) example, be sure each message gets a unique IV. The `DeriveKeyAndIV` produces predictable IVs for demonstration purposes, but it violates semantic security because two messages under the same key will produce the same ciphertext.

If you choose to generate a random IV and append it to the message, be sure to authenticate the `{IV,Ciphertext}` pair.



# Cryptographic Parameters: User Responsibility

```
#define MESSAGE ((const unsigned char *) "test")
#define MESSAGE_LEN 4
#define CIPHERTEXT_LEN (crypto_secretbox_MACBYTES + MESSAGE_LEN)

unsigned char nonce[crypto_secretbox_NONCEBYTES];
unsigned char key[crypto_secretbox_KEYBYTES];
unsigned char ciphertext[CIPHERTEXT_LEN];

randombytes_buf(nonce, sizeof nonce);
randombytes_buf(key, sizeof key);
crypto_secretbox_easy(ciphertext, MESSAGE, MESSAGE_LEN, nonce, key);

unsigned char decrypted[MESSAGE_LEN];
if (crypto_secretbox_open_easy(decrypted, ciphertext, CIPHERTEXT_LEN, nonce, key) != 0) {
    /* message forged! */
}
```

**Generating the  
nonce is the  
user's  
responsibility.**

The nonce doesn't have to be confidential, but it should never ever be reused with the same key. The easiest way to generate a nonce is to use `randombytes_buf()`.

# Cryptographic Parameters: Nonces Reuse

## GCM-SIV: Full Nonce Misuse-Resistant Authenticated Encryption at Under One C/B

Shay Gueron<sup>1</sup> and Yehuda Lindell<sup>2</sup>

## Trivial Nonce-Misusing Attack on Pure OMD

Tomer Ashur and Bart Mennink

## Online Authenticated-Encryption and its Nonce-Reuse Misuse-Resistance

Viet Tung Hoang<sup>1,2</sup> Reza Reyhanitabar<sup>3</sup> Phillip Rogaway<sup>4</sup> Damian Vizár<sup>3</sup>

<sup>1</sup> Dept. of Computer Science, Georgetown University, USA

<sup>2</sup> Dept. of Computer Science, University of Maryland, College Park, USA

<sup>3</sup> EPFL, Lausanne, Switzerland

<sup>4</sup> Dept. of Computer Science, University of California, Davis, USA

# Cryptographic Parameters: EdDSA

Furthermore, it is well known that ECDSA's session keys are much less tolerant than the long-term key of slight deviations from randomness, even if the session keys are not revealed or reused. For example, Nguyen and Shparlinski in [61] presented an algorithm using lattice methods to compute the long-term ECDSA key from the knowledge of as few as 3 bits of  $r$  for hundreds of signatures, whether this knowledge is gained from side-channel attacks or from non-uniformity of the distribution from which  $r$  is taken.

EdDSA avoids these issues by generating  $r = H(h_b, \dots, h_{2b-1}, M)$ , so that different messages will lead to different, hard-to-predict values of  $r$ . No per-message randomness is consumed. This idea of generating random signatures in a secretly deterministic way, in particular obtaining pseudorandomness by hashing a long-term secret key together with the input message, was proposed by Barwood in [9]; independently by Wigley in [79]; a few months later in a patent application [57] by Naccache, M'Raihi, and Levy-dit-Vehel; later by M'Raihi, Naccache, Pointcheval, and Vaudenay in [55]; and much later by Katz and Wang in [47]. The patent application was abandoned in 2003.

# Cryptographic Parameters: API Responsibility

Signing a string is as follows. Note that a PRNG is required because the Digital Signature Standard specifies a per-message random value.

```
ECDSA<ECP, SHA1>::PrivateKey privateKey;
privateKey.Load(...);
ECDSA<ECP, SHA1>::Signer signer( privateKey );

AutoSeededRandomPool prng;
string message = "Yoda said, Do or do not. There is no try.";
string signature;

StringSource s( message, true /*pump all*/,
    new SignerFilter( prng,
        signer,
        new StringSink( signature )
    ) // SignerFilter
); // StringSource
```

```
sig = ECDSA_do_sign(digest, 20, eckey);
if (sig == NULL)
{
    /* error */
}
```

**Generating the nonce is the API's responsibility.**

```
DSA_PrivateKey* dsakey = dynamic_cast<DSA_PrivateKey*>(key.get());

if(!dsakey)
{
    std::cout << "The loaded key is not a DSA key!" << std::endl;
    return 1;
}

PK_Signer signer(*dsakey, "EMSA1(SHA-1)");

DataSource_Stream in(message);
byte buf[4096] = { 0 };
while(size_t got = in.read(buf, sizeof(buf)))
    signer.update(buf, got);

sigfile << base64_encode(signer.signature(rng)) << std::endl;
```

## Dream Crypto Library Requirement #6

**In encryption mode, the IVs/nonces should, whenever possible, be generated in a transparent way and returned by the API, as it is done for public-key crypto.**

# Outline

- 1 Motivations
- 2 Security Models
- 3 Primitives
- 4 Security Parameters
- 5 Cryptographic Parameters
- 6 API Simplicity**
- 7 Conclusion

# API Simplicity

```
int encrypt(unsigned char *plaintext, int plaintext_len, unsigned char *aad,
            int aad_len, unsigned char *key, unsigned char *iv,
            unsigned char *ciphertext, unsigned char *tag)
{
    EVP_CIPHER_CTX *ctx;

    int len;

    int ciphertext_len;

    /* Create and initialise the context */
    if(!(ctx = EVP_CIPHER_CTX_new())) handleErrors();

    /* Initialise the encryption operation. */
    if(1 != EVP_EncryptInit_ex(ctx, EVP_aes_256_gcm(), NULL, NULL, NULL))
        handleErrors();

    /* Set IV length if default 12 bytes (96 bits) is not appropriate */
    if(1 != EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GCM_SET_IVLEN, 16, NULL))
        handleErrors();

    /* Initialise key and IV */
    if(1 != EVP_EncryptInit_ex(ctx, NULL, NULL, key, iv)) handleErrors();

    /* Provide any AAD data. This can be called zero or more times as
     * required
     */
    if(1 != EVP_EncryptUpdate(ctx, NULL, &len, aad, aad_len))
        handleErrors();

    /* Provide the message to be encrypted, and obtain the encrypted output.
     * EVP_EncryptUpdate can be called multiple times if necessary
     */
    if(1 != EVP_EncryptUpdate(ctx, ciphertext, &len, plaintext, plaintext_len))
        handleErrors();
    ciphertext_len = len;

    /* Finalise the encryption. Normally ciphertext bytes may be written at
     * this stage, but this does not occur in GCM mode
     */
    if(1 != EVP_EncryptFinal_ex(ctx, ciphertext + len, &len)) handleErrors();
    ciphertext_len += len;

    /* Get the tag */
    if(1 != EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GCM_GET_TAG, 16, tag))
        handleErrors();

    /* Clean up */
    EVP_CIPHER_CTX_free(ctx);

    return ciphertext_len;
}
```

# API Simplicity: Error Handling

## RETURN VALUES

---

ECDSA\_size() returns the maximum length signature or 0 on error.

ECDSA\_sign\_setup() and ECDSA\_sign() return 1 if successful or 0 on error.

ECDSA\_verify() and ECDSA\_do\_verify() return 1 for a valid signature, 0 for an invalid signature and -1 on error. The error codes can be obtained by [ERR\\_get\\_error\(3\)](#).

```
if (!ECDSA_sign (...)) { handle_error (); }
```

```
if (ECDSA_verify (...)) { /* good signature */...}
```



# API Simplicity: Error Handling

```
#define MESSAGE (const unsigned char *) "test"
#define MESSAGE_LEN 4

unsigned char pk[crypto_sign_PUBLICKEYBYTES];
unsigned char sk[crypto_sign_SECRETKEYBYTES];
crypto_sign_keypair(pk, sk);

unsigned char signed_message[crypto_sign_BYTES + MESSAGE_LEN];
unsigned long long signed_message_len;

crypto_sign(signed_message, &signed_message_len,
            MESSAGE, MESSAGE_LEN, sk);

unsigned char unsigned_message[MESSAGE_LEN];
unsigned long long unsigned_message_len;
if (crypto_sign_open(unsigned_message, &unsigned_message_len,
                    signed_message, signed_message_len, pk) != 0) {
    /* Incorrect signature! */
}
```

## API Simplicity: Error Handling

```
nowhere:rsa pjunod$ grep ERR_REASON  
rsa_err.c | wc -l
```

68

## Dream Crypto Library Requirement #7

- **In debug mode, a crypto library could possibly return detailed error messages.**
- **In production mode, a crypto library should return only a boolean flag: OK/NOK.**
- **API and error handling should be as simple as possible.**

# Outline

- 1 Motivations
- 2 Security Models
- 3 Primitives
- 4 Security Parameters
- 5 Cryptographic Parameters
- 6 API Simplicity
- 7 Conclusion**

# Random Thoughts I

- Crypto libraries must stop assuming that their crypto users are omniscient cryptographers. 100% of them are not (and btw, omniscient cryptographers do not exist).
- In a crypto library, a non-secure functionality is a superfluous feature that must be killed.
- Crypto libraries must stop put security-related responsibilities on developers' shoulders.
- Crypto libraries must categorically refuse to generate too small keys or cryptographic groups.
- Crypto libraires must refuse with all possible energy to use obsolete/weak cryptographic configurations.
- Whenever possible, crypto libraries must generate themselves the required IV/nonces.
- Crypto libraries must be extremely simple to use.

## Random Thoughts II

**The perfect cryptographic library does not exist yet, even if a few of them converge in the good direction!**

# Thank You!

# Any Question?