esse

Advanced Security for Systems Engineering – VO 09: Applied Cryptography

Lukas Brandstetter
Clemens Hlauschek
Christian Schanes

INSO – Industrial Software
Institute of Information Systems Engineering | Faculty of Informatics | TU Wien

Agenda

Threat Model

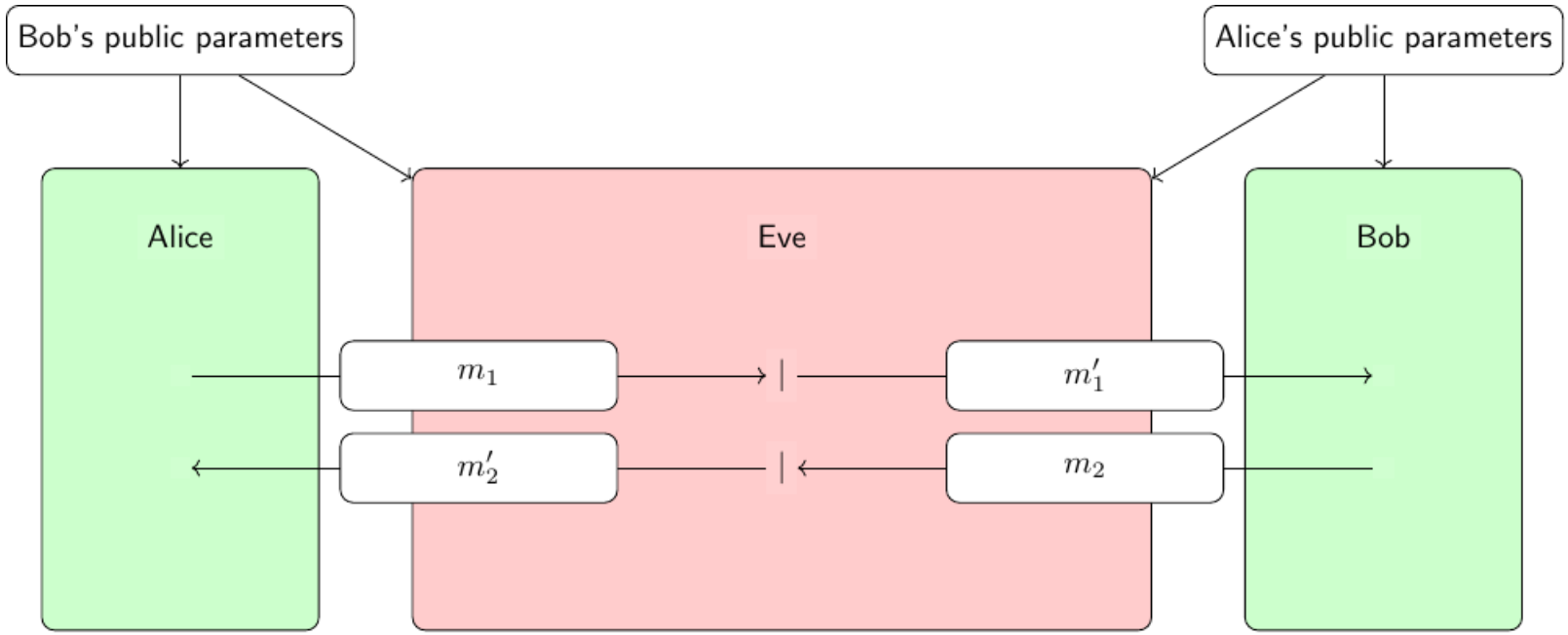
Encryption Schemes

Out-of-Model Attacks

- Simple Timing Attack
- Padding Oracle Attack

Threat Model

Threat Model



Passive: $m_i = m_i'$

Adversaries Capabilities

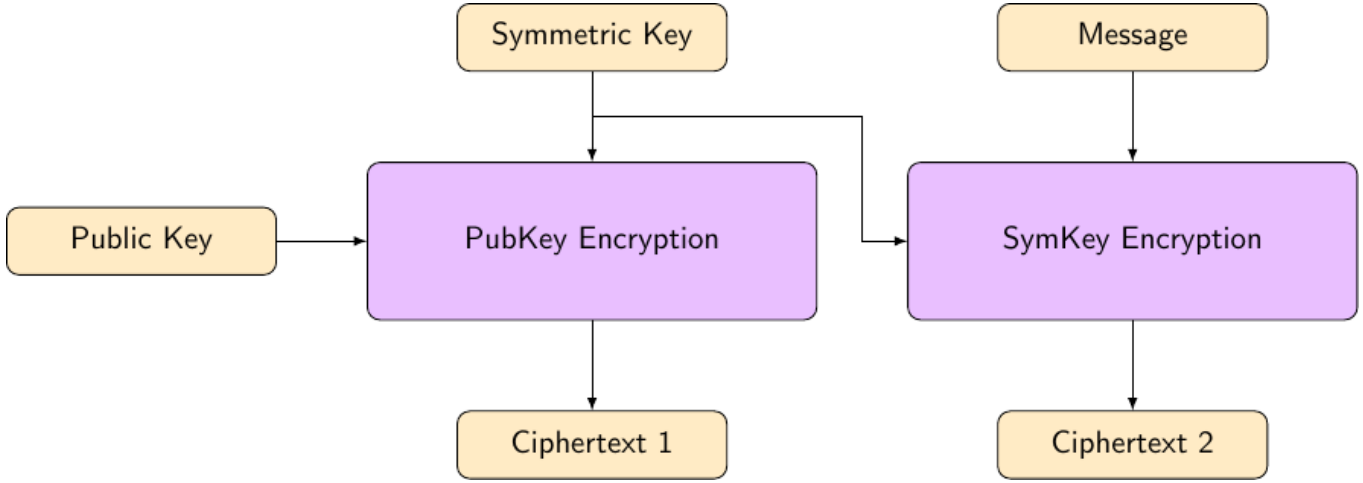
- Depend on exact model
  - Passive: eavesdropping
  - Active: tampering with, blocking, delaying, reordering messages
  - Advanced active: sometimes even corrupting peers

- Mostly Probabilistic Polynomial Time (PPT) adversary

Encryption Schemes

Encryption Algorithms: Keywords

- Symmetric, Secret-key: $m = D(k, E(k, m))$
  - 3DES, AES, (X)Salsa20, ChaCha
  - Fast, but Key Distribution problem
- Asymmetric, Public-key: $m = D(sk, E(pk, m))$
  - RSA, ElGamal, Elliptic Curves
- Hybrid:



Encryption Schemes: From Oneway Function/PRP to Secure Cryptographic Scheme

- Oneway function (with trapdoor)/pseudorandom permutation (PRP)
- Hardness assumptions
- Threat model and goals (IND-CCA, IND-CPA)
- Secure cryptographic scheme with reduction to hardness assumption

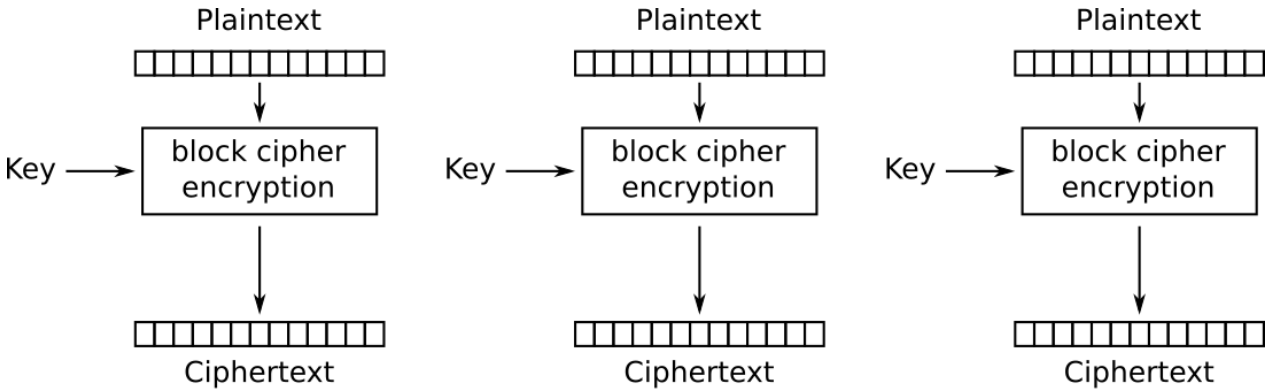
RSA Cryptosystem

- **Assumption:**
  - Hardness related to Integer Factorization problem
- **Basic Primitive:**
  - $N = p \cdot q$ with $p, q \in \mathbb{P}$
  - Operations are computed $\mod N$
  - $sk : d \quad pk : e$ with $e \cdot d = 1 \mod \phi(N)$
  - $E : m^e$
  - $D : m^d$
- **Secure Scheme:**
  - Never use plain (textbook) RSA, use OAEP or at least PKCSv1.5

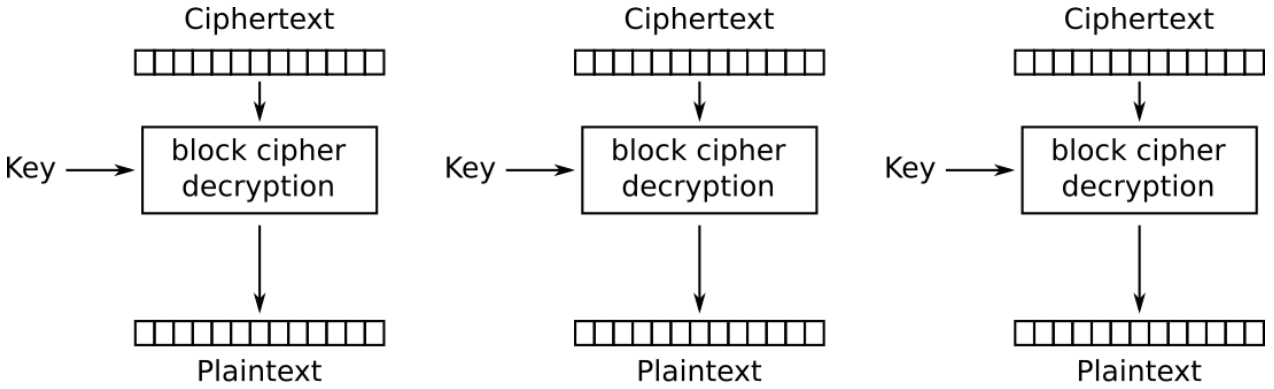
ElGamal Cryptosystem

- **Assumption:**
  - Hardness of Discrete Logarithm
- **Basic Primitive:**
  - $p \in \mathbb{P}$, g is generator of $\mathbb{Z}_p$
  - Operations are computed $\mod P$
  - $sk : x \quad pk : g^x$ with x uniform random sampled in $\mathbb{Z}_p$
  - $E : (c_0 = g^y, c_1 = pk^y \cdot m)$ with y uniform sampled in $\mathbb{Z}_p$
  - $D : \frac{c_1}{(c_0)^x}$
- **Secure Scheme:**
  - Never use plain, use, e.g., Cramer-Shoup

ECB

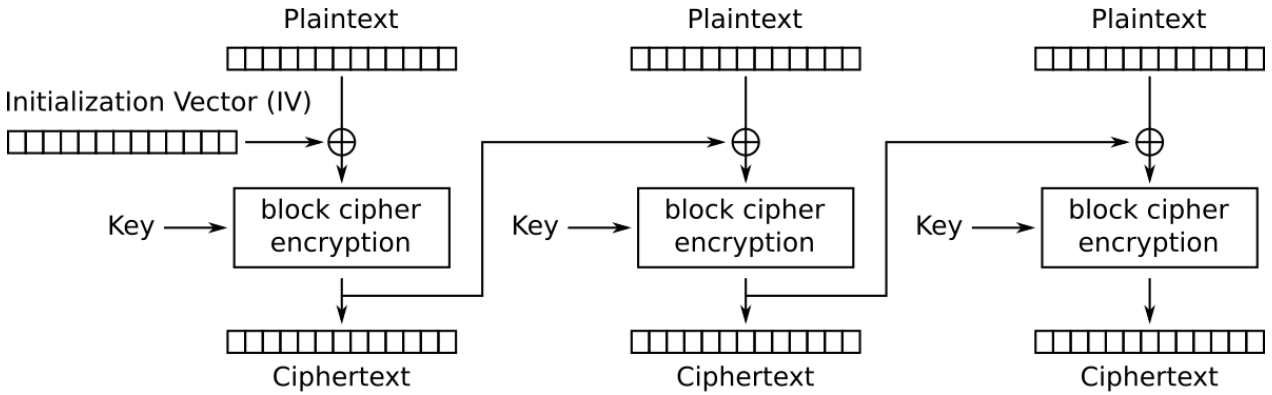


Electronic Codebook (ECB) mode encryption

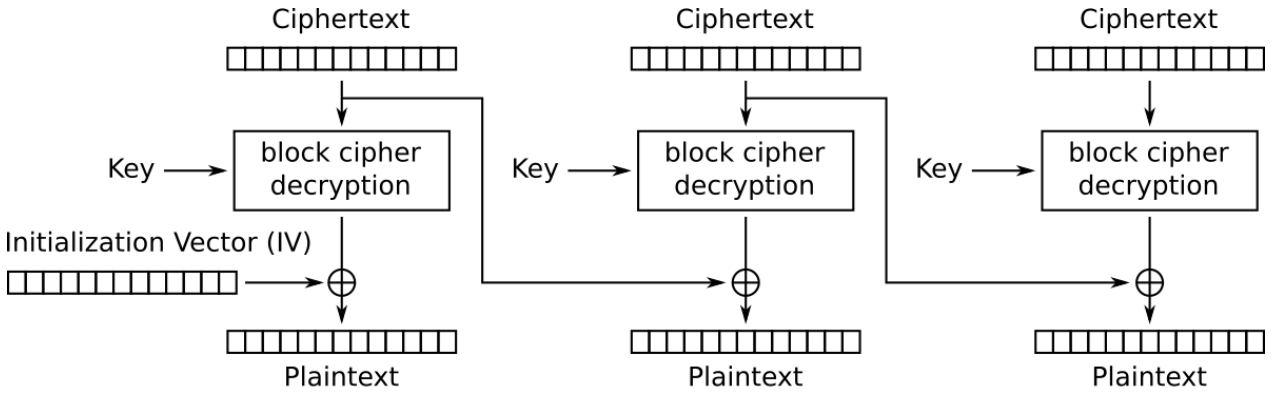


Electronic Codebook (ECB) mode decryption

CBC



Cipher Block Chaining (CBC) mode encryption



Cipher Block Chaining (CBC) mode decryption

Out-of-Model Attacks

Attacks and Implementation Errors

- Provable security against all (IND-CCA/IND-CPA) adversaries
- Implementers must take care to ensure security against different out-of-model attacks, e.g., Sidechannel Attacks (SCA)

Simple Timing Attack

We discuss an example of a cryptographic side channel attack (SCA)

- Variations of this attack in many different implementations / cryptographic primitives
- We use Message Authentication Code (MAC) verification for illustration purpose

MAC Verification

What is the problem?

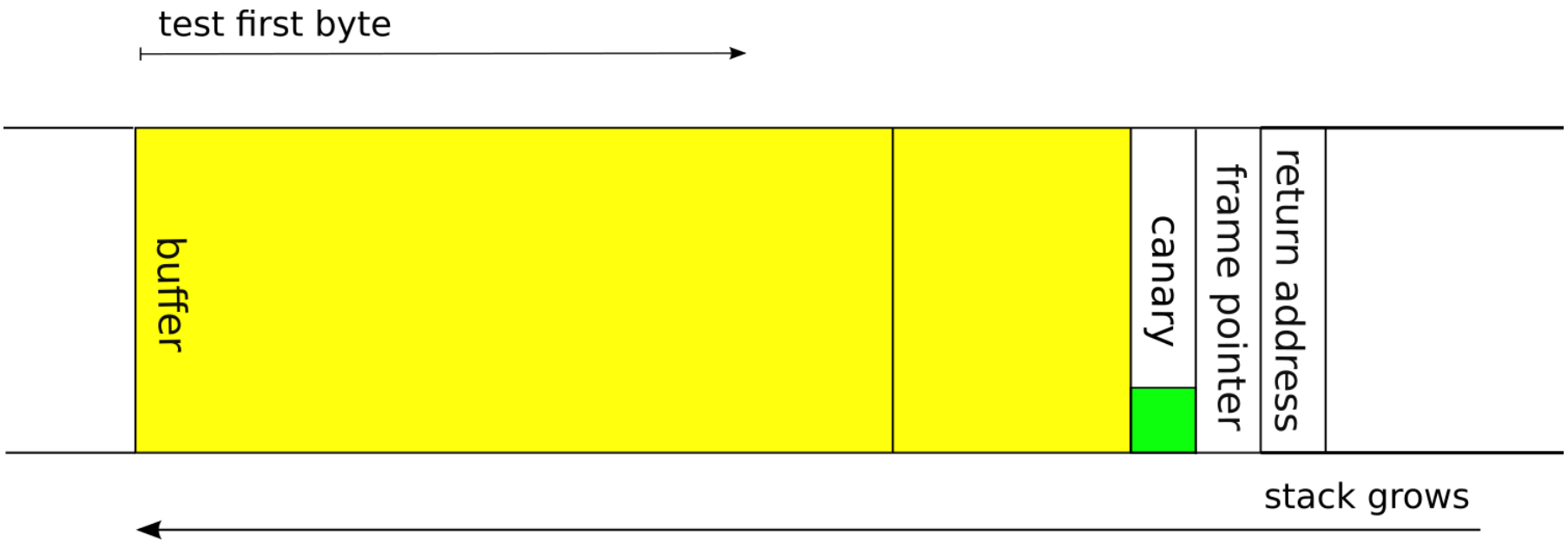
```
1 int verify(char * key, char * msg, char * atag) {
2
3     char atag_computed[32];
4
5     atag_computed = HMAC_SHA2(key, msg);
6
7     return memcmp(atag_computed, atag, 32);
8
9 }
```

MAC Verification

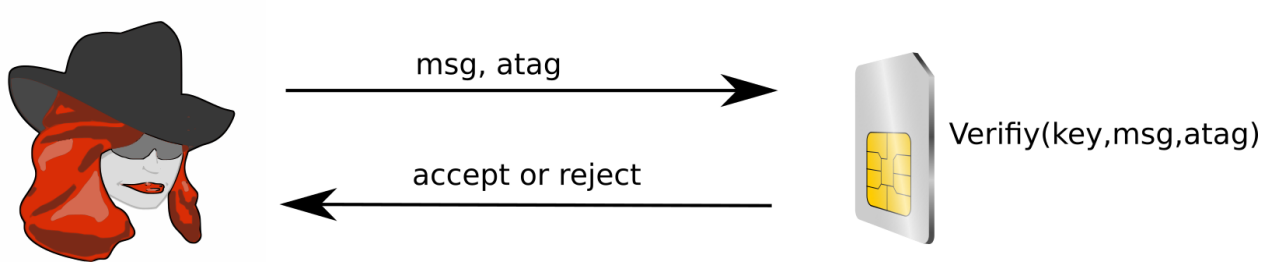
Problem: `memcmp` implements a byte-by-byte comparison.

As soon as the first inequality is found, `while`-loop in `memcmp` stops.

Remember how we brute-forced the stack canary?



MAC Verification: Timing Attack



Timing Attack: compute the valid authentication tag for a target message

1. Send queries $msg, atag$ with all possible first bytes for $atag$. When the first byte is correct, response time is slightly longer.
2. Fix the first byte, repeat to find all other bytes.

MAC Verification: Timing Attack

Same problem with compare functions in many other languages: Java, Python, etc.

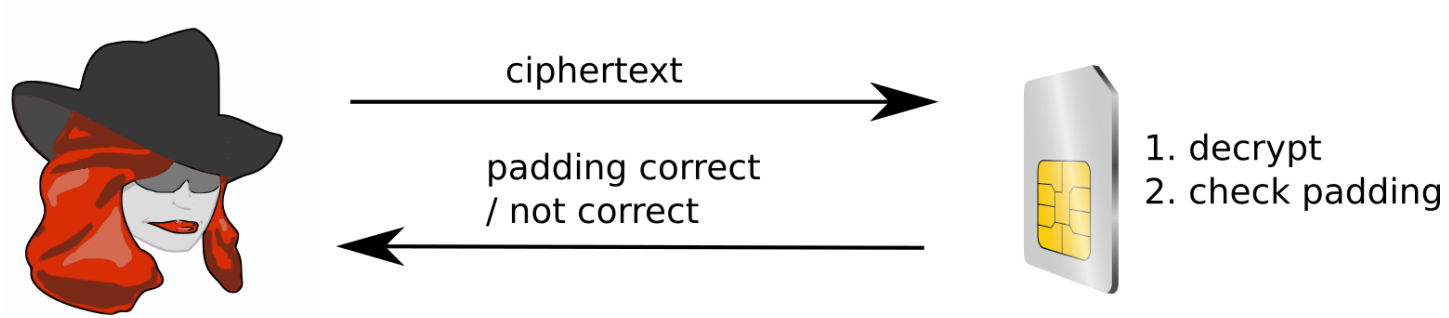
Avoid vulnerability by either

- implementing constant-time comparison. (Beware of compile-time optimizations)
- comparing MACs of the MACs instead. Attacker cannot guess values being compared.

  `MAC(key, atag)==MAC(key, atag_computed))`

Same attack sometimes applicable for RSA Signature verification, ...

Padding Oracle Attack



Device answers whether padding is correct or not:

- Error Message
- Timing side channel

We can use this Padding Oracle to decrypt any ciphertext without knowledge of the secret key.

Padding Oracle Attack

Common Padding schemes:

- PKCS#7

```
01
02 02
03 03 03
04 04 04 04
05 05 05 05 05
...
```

- Zero Byte Padding
- ...

Padding Oracle Attack

Recap: Property of bitwise XOR

- Bitwise addition $(\mathrm{mod}\ 2)$
- Cancel component by XORing twice:

$$x \oplus y \oplus x = y$$

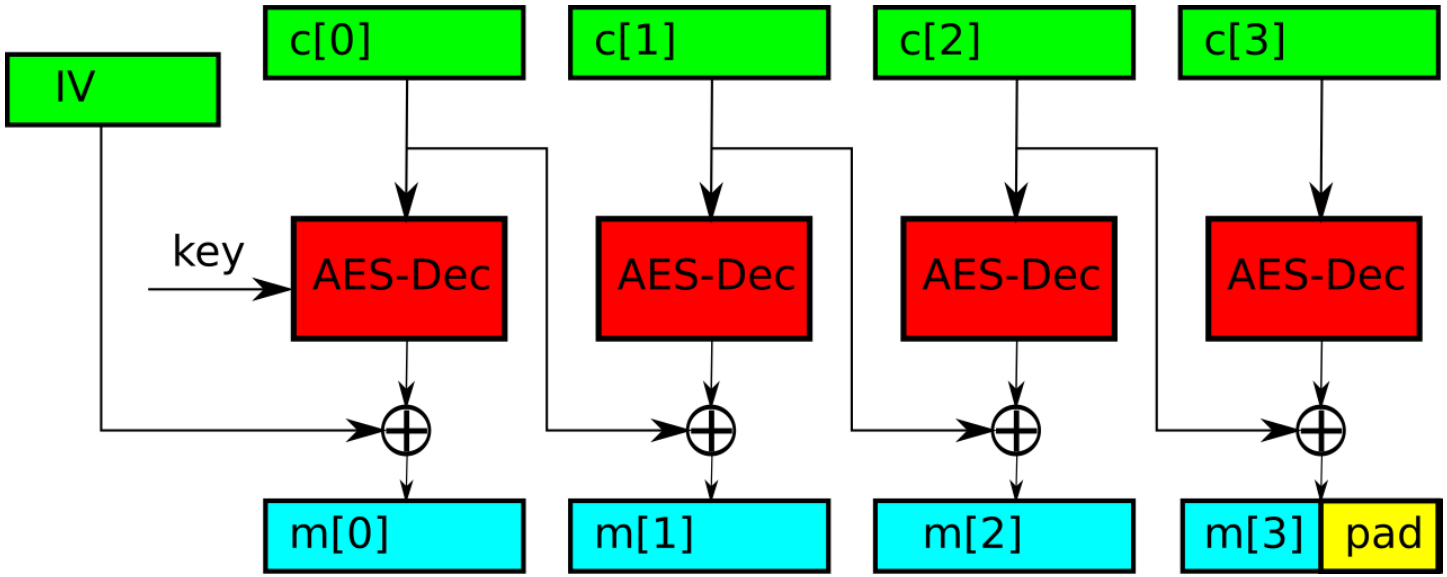
$$x = 011011$$

$$y = 101001$$

$$x \oplus y = 110010$$

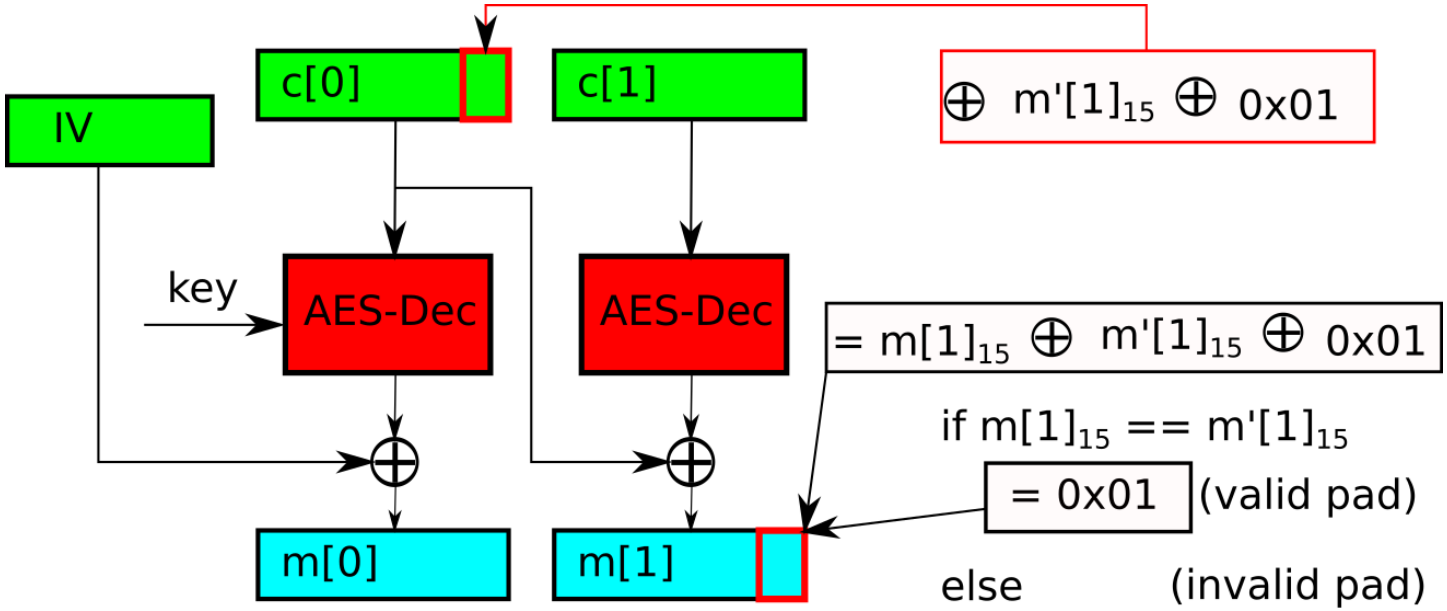
Padding Oracle Attack

We have ciphertext $c = (c[0], c[1], c[2], c[3])$ and want to decrypt $m[1]$.

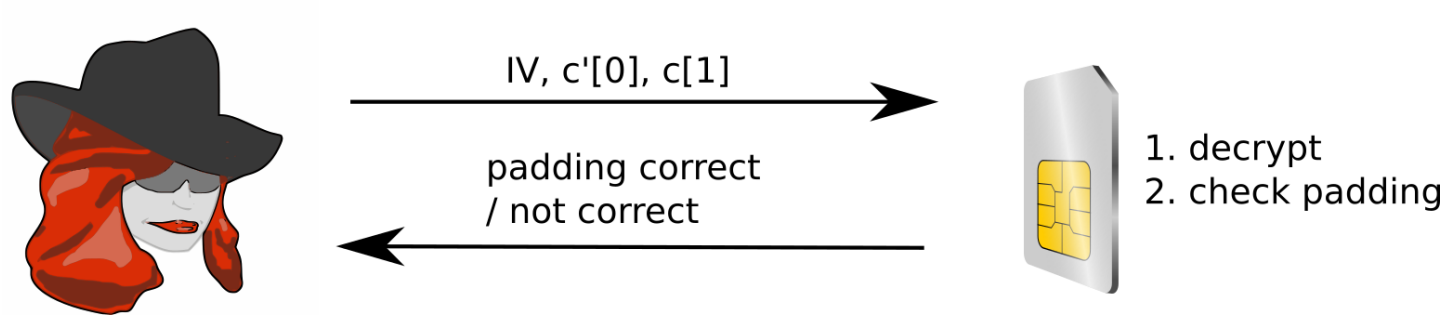


Padding Oracle Attack

Fist, make a guess $m'[1]_{15}$ for the last byte of $m[1]$



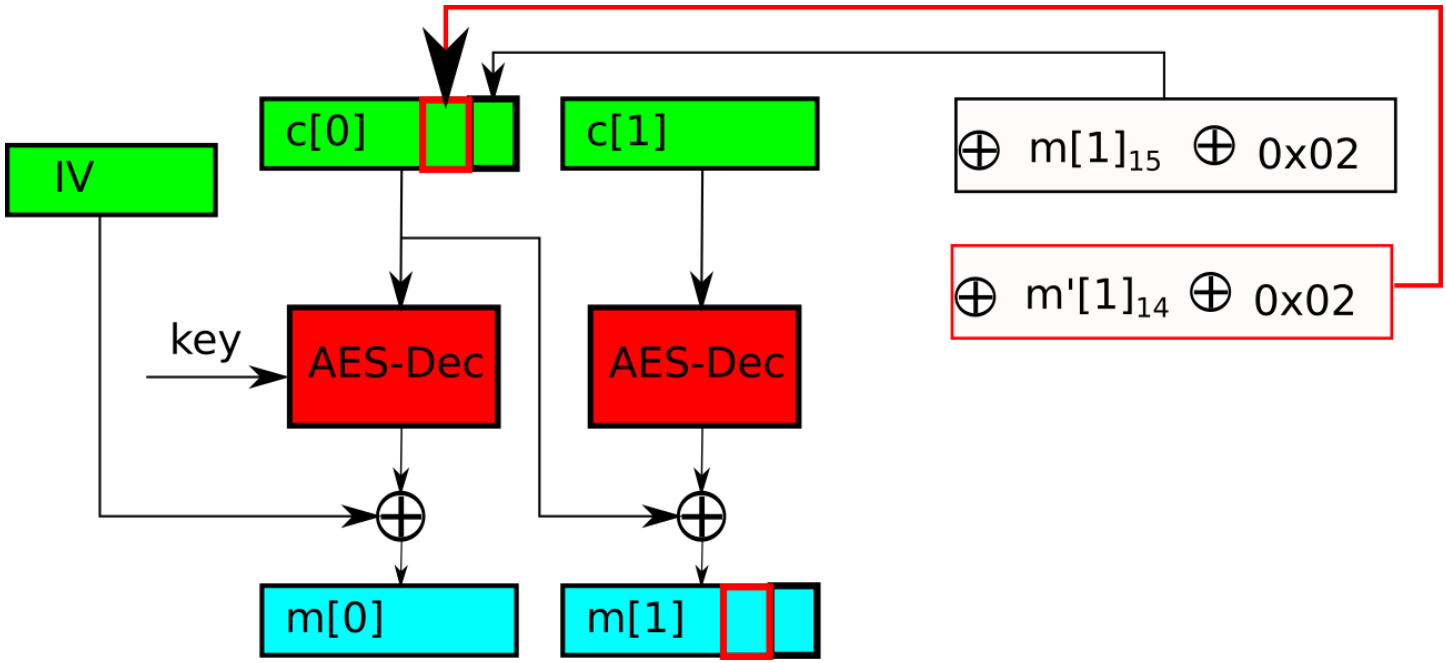
Padding Oracle Attack



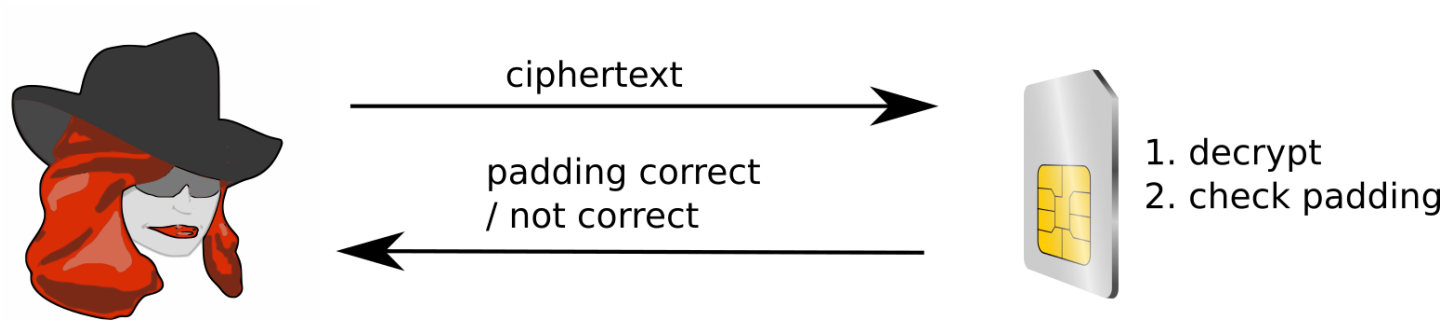
1. Calculate $c'[0] := c[0] \oplus m'[1]_{15} \oplus 0x01$ with guess $m'[1]_{15}$ for last plaintext byte $m[1]_{15}$.
2. Send $(IV, c'[0], c[1])$ to device
3. If oracle response was "invalid padding", iterate with another guess, but if oracle says "valid padding", our guess $m'[1]_{15}$ was correct.

Padding Oracle Attack

Next, fix the last byte $m[1]_{15}$ and make a guess for $m[1]_{14}$.



Padding Oracle Attack



Proceed until the whole block $m[1]$ is recovered. This way, any ciphertext block can be decrypted.

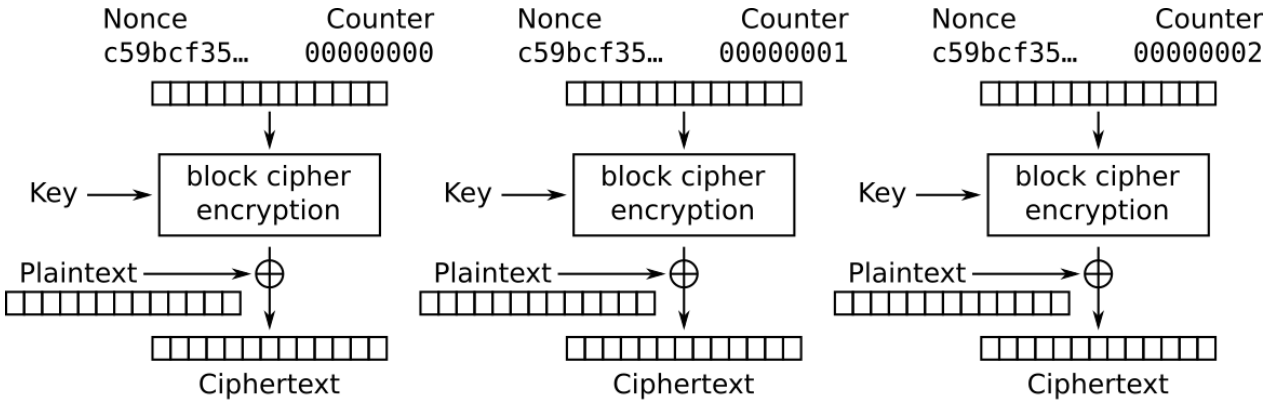
Padding Oracle Attack

The Attack was introduced in 2002 [Vaudenay'02].

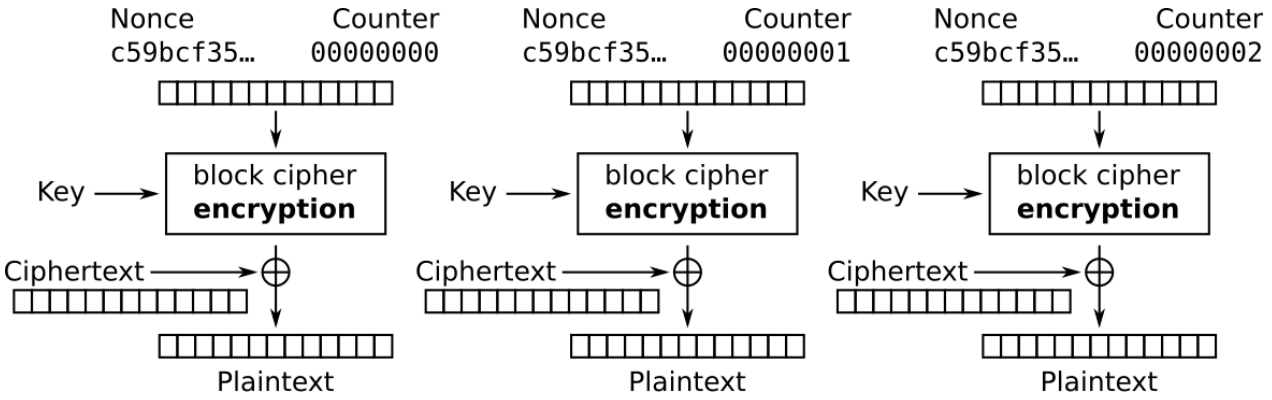
In the following years many applications were attacked

- Smartcards, Hardware Security Module (HSM), ...
- TLS was vulnerable too
- Many attacks work in a similiar fashion

CTR



Counter (CTR) mode encryption



Counter (CTR) mode decryption

Signatures and Message Authentication Codes

- Symmetric: $V(k, S(k, m), m) = 1$
  - HMAC, CBC-MAC
- Asymmetric: $V(pk, S(sk, m), m) = 1$
  - DSA, ECDSA
- $S(\cdot)$ is only efficient with k or sk
- $V(\cdot)$ is efficient with k or pk

- Ensure that message is from sender and not tampered with
- Signatures need verifiability for third parties, MACs require only direct peers

Fixing the Above Problems

- Authenticate the Ciphertext with MAC
- Block Modes that do exactly this exist
- GCM is the most popular (AEAD)
  - CTR + authentication
- Better: GCM-SIV (nonce misue resistance)
- If you don't care about patents: OCB
- Whatever you do, you want Authenticated Encryption

Recommended Crypto Libraries

- There are libraries for most tasks
- Use the right primitive
- NaCl and libsodium are good examples
  - High Level API
  - Authenticated Encryption
  - Hybrid Encryption
  - Hard to use wrong
  - Bindings in many languages
  - `nacl.cr.yp.to`, `libsodium.org`

NaCl and Libsodium Secret-key

```
crypto_secretbox_keygen(key);
randombytes_buf(nonce, nonce_len);
crypto_secretbox_easy(ct, pt, pt_len, nonce, key);
crypto_secretbox_open_easy(pt, ct, ct_len, nonce, key);
```

- Encryption: XSalsa20 stream cipher
- Authentication: Poly1305 MAC
- Never reuse nonce

NaCl and Libsodium Public-key

```
crypto_box_keypair(apk, ask);
crypto_box_keypair(bpk, bsk);
randombytes_buf(nonce, nonce_len);
crypto_box_easy(ct, pt, pt_len, nonce, bpk, ask);
crypto_box_open_easy(pt, ct, ct_len, nonce, apk, bsk);
```

- Key exchange: X25519
- Encryption: XSalsa20 stream cipher
- Authentication: Poly1305 MAC
- Never reuse nonce

Literature/Links

- Jonathan Katz, Yehuda Lindell: Introduction to Modern Cryptography, CRC Press, 2014
- Vaudenay: Security Flaws Induced by CBC Padding. Applications to SSL, IPSEC, WTLS. EUROCRYPT'02
- Boeck, et al: Return Of Bleichenbacher's Oracle Threat (ROBOT), Usenix Sec'18
- NaCl Library, `nacl.cr.yp.to`
- Libsodium Library, `libsodium.org`
- [Boneh] Dan Boneh (Stanford): Online Cryptography Class. `http://crypto-class.org`

Thank's for your attention!

https://security.inso.tuwien.ac.at/