Implementation pitfalls

- We learned about various cryptographic primitives and the provable security approach, saw many secure constructions.
- You are almost ready to employ this knowledge in practice.
- Let us review some common mistakes one needs to be aware of and avoid when implementing cryptographic protocols.

Always remember to

- Use widely accepted and believed to be secure building blocks (e.g. AES).
- Use provably secure (under reasonable assumptions) constructions (e.g. $CBC$).
- Do not assume that the schemes provide security properties other than what is proven about them (e.g. encryption does not provide authenticity).
- Realize that the use of a provably secure scheme does not guarantee that the entire system will be secure.
- Make sure that you implement exactly the scheme that was proven secure.

Not using the right primitives

- ATM-based passive optical networks commonly use a block cipher called CHURN. It’s key size is 8 bits and it’s block size is 4 bits!

Using the constructs without security proofs

- The use of the ECB mode and the Plain RSA encryption is still very common.

Not considering the security bounds

Consider the encryption algorithm of a scheme $\text{CTRS}\$[L]

Let $E: \{0,1\}^k \times \{0,1\}^n \rightarrow \{0,1\}^n$ be a block cipher.

One can prove that for every $A$ making $q$ queries there exists $B$ s.t.

$$\text{Adv}_{\text{impl-pa}}^{\text{CTRSELU}}(A) \leq \text{Adv}_{E}^{\text{prf}}(B) + \frac{q^2}{2^{n+t}}$$

Is $\text{CTRS}\$ secure?
Random numbers

- It is usually straightforward to implement the pseudo-code descriptions in C or Java.
- However, how do you implement commands like $K \overset{\$}{\leftarrow} \{0,1\}^k$?
- The C offers a built-in random number generator, that works roughly as this

```
32-bit number

procedure rand(seed)
  state = seed;
  function rand()$
    state = (state \cdot 1103515245) + 12345$
    mod 2147483648;
    return state$
  \}$

algorithm K
  K \overset{\$}{\leftarrow} \{0,1\}^{128}$
  return K

function keygen()$
  key[0] = rand();\ key[1] = rand();$
  return key$
```

- So one can implement $K \overset{\$}{\leftarrow} \{0,1\}^k$ as follows

- But looking at how rand() works we notice that

```
key[1] = (key[0] \cdot 1103515245 + 12345) mod 2^{31}$
key[2] = (((key[0] \cdot 1103515245 + 12345) \cdot 1103515245 + 12345) mod 2^{31}$
key[3] = ((((key[0] \cdot 1103515245 + 12345) \cdot 1103515245 + 12345) \cdot 1103515245 + 12345) mod 2^{31}$
```

- This means that there are still only $2^{32}$ possibilities for the key.

- Not using the right tool

- It is tempting to believe that encryption provide some authenticity.
- The first versions of the SSH protocol, IPsec specification and the WEP protocol did not use message authentication codes, and thus were subject to certain attacks.

- Not implementing exactly the provable-secure schemes

- A slightest tweak to a provably-secure scheme can make it insecure
- Diebold voting machines encrypted the votes with CBC, but used all-zero string as an IV.
- Microsoft Word and Excel used CBC, but did not pick a new random R each time.
• The Netscape browser tried to do better:

```plaintext
procedure NetscapeRandSetup()
pid = process ID;
ppid = parent process ID;
seconds = current time of day (seconds);
microseconds = current time of day (microseconds);
x = concatenation of pid, ppid,
    seconds, microseconds;
NSeed = SHA1(x);

function NetscapeGetRand()
rv = SHA1(NSeed);
NSeed = NSeed + 1 mod 2160;
return rv;
```

• This can be used as

```plaintext
algorithm K
K \in \{0,1\}^{128}
return K
```

```plaintext
function keygen()
    NetscapeRandSetup();
tmp = NetscapeGetRand();
key = first 128-bits of tmp;
return key
```

• Despite the reasonable properties of SHA1 and the 160-bit output of the generator, an adversary can learn or guess x.

Randomness for encryption
• Designers of SSH, IPsec, SSL all assumed that the last blocks of the ciphertexts in CBC can be used as IVs for the next ciphertexts.

Combining the schemes
• Recall that it is insecure in general to apply the Encrypt-and-MAC paradigm in order to achieve both privacy and authenticity.

Key management
• All users of the WEP encryption protocol use the same symmetric key.
• The key for the secure votes encryption in Diebold machines is hardwired in the code:

```plaintext
#define DESKEY ((des_key*)"F2654hd4")
```

Reference

• Y. Kohno "Implementation pitfalls". Available at http://www.cse.ucsd.edu/~mihir/cse107/yoshi.pdf