This work is distributed under the CC BY-SA license, unless stated otherwise.

Hardware Security

1. Cryptographic implementations (tutorial)

Ján Jančár

Summer 2026

Objectives

- Identify critical targets for future attacks
- Move beyond abstract mathematical definitions
- See how algorithms are structured in code
- Create a mental map of implementations

Plan

2 parts

1. Hands-on implementation
2. Examining open-source crypto libraries

Jupyter Notebooks & JupyterLab

Setup

1. **Create a virtual environment**
   - Linux/macOS (Bash):
     ```
     python3 -m venv venv && source venv/bin/activate
     ```
   - Windows:
     ```
     python -m venv venv
     .\venv\Scripts\activate
     ```
2. **Download tutorial contents**
   - From PANDA
3. **Install dependencies**
   - `pip install -r requirements.txt`
4. **Launch the lab**
   - `jupytext-config set-default-viewer`
   - `jupyter lab`
5. **Open the notebook**
   - Right click -> Open with ... -> Notebook

or just use

myBinder

Jupyter Notebooks & JupyterLab

Quickstart

- Handy tutorial
- Run cells with `Shift + Enter` or ▶
- Save your work occasionally with `Ctrl + S`
- Tab-completion (simple auto-complete)
- `object?` -> object help
- `object.method?` -> method help

Part 1: Hands-on Implementation

Jupyter

- **Task**: Implement the missing pieces
- **Targets**: AES, RSA, ECC
- Focus on the **key-dependent** parts
  - Always my first question: Where does the key go?

Task 1: AES

S-Box and key addition

- **Task**: Complete the `sub_bytes` and `add_round_key` functions.
- AES Standard
- **Questions**
  - What are the operations performed with the key?
  - Any ideas on side-channel leakage?

Task 1: AES

S-Box and key addition (solution)

```
SBOX = [0x63, 0x7C, 0x77, ..., 0x16]

def sub_bytes(state):
    return [SBOX[b] for b in state]

def add_round_key(state, round_key):
    return [s ^ r for s, r in zip(state, round_key)]
```

```
# Initial Round
state = add_round_key(state, round_keys[0])

# Rounds 1 to 9
for i in range(1, 10):
    state = sub_bytes(state)
    ...
```

plaintext

key

Task 1: AES

S-Box and key addition (takeaways)

- S-Box is a lookup table: A non-linear operation
  - Yes, it is a composition of multiplicative inverse in $GF(2^8)$ and a certain affine transform, but that is not (generally) how it is implemented.
- In the first round it is applied to the XOR of the plaintext and the key
- **More resources**:
  - A nice explanation of how the T-table AES implementation looks like
  - A Stick Figure Guide to the Advanced Encryption Standard (AES)
  - An illustrated guide to AES
  - Cryptohack course - Symmetric cryptography

Task 2: RSA

The modular exponentiation loop

- **Task**: Implement the “Square and Multiply” algorithm for modular exponentiation ($m = c^d \pmod{n}$).
- Iterative approach (no recursion necessary)
- Exponentiation by squaring
- **Questions**
  - What are the operations performed with the key?
  - Any ideas on side-channel leakage?

Task 2: RSA

The modular exponentiation loop (solution)

```
def square_and_multiply(base, exponent, modulus):
    res = Mod(1, modulus)
    bits = bin(exponent)[2:]

    for bit in bits:
        res = res**2
        if bit == "1":
            res = res * base

    return int(res)
```

Try this algorithm in your head with small numbers: $3^{1101010} \pmod 7$ <--

What power does the `res` variable hold during the iterations of the loop?

The Mod class does the modular reduction by itself. We scan the bits of the exponent left-to-right (msb to lsb) and do square-and-multiply.

Task 2: RSA

The modular exponentiation loop (takeaways)

- The key is scanned bit-by-bit (left-to-right or right-to-left), there can be *branching* based on these bit values
- All operations are done *modulo* the composite number n (the public key)
- You need to reduce $(\mathrm{mod}\ n)$ after every operation. If you do not, the resulting number would have $2^{2059} \approx 10^{620}$ bits, which is roughly 10^{540} times the number of atoms in the universe.
- **More resources**:
  - Modular arithmetic
  - Cryptohack course - Asymmetric cryptography

Task 3: ECC

The scalar multiplication loop

- **Task**: Implement the “Double and Add” algorithm for scalar multiplication ($Q = [k]P$).
- Iterative approach (no recursion necessary)
- Double and add
- **Questions**
  - What are the operations performed with the key?
  - Any ideas on side-channel leakage?

Task 3: ECC

The scalar multiplication loop (solution)

Initialize to P (instead of $\mathcal{O}$).

Skip one bit (the msb).

```
def scalarmult(point, scalar, curve):
    res = point
    bits = bin(int(scalar))[3:]

    for bit in bits:
        res = curve.dbl(res)
        if bit == "1":
            res = curve.add(res, point)

    return res
```

The `curve` class does the addition formulas. We scan the bits of the exponent left-to-right (msb to lsb) and do double-and-add.

Task 3: ECC

The scalar multiplication loop (takeaways)

- The key is scanned bit-by-bit (left-to-right or right-to-left), there can be *branching* based on these bit values
- All operations are done with *points on the elliptic curve* using *addition formulas*
- Handling the point-at-infinity ($\mathcal{O}$) is tricky, some formulas can not do it. We thus skip it.
- **More resources**:
  - Cryptohack course - Elliptic curves

Part 2: Examining open-source crypto libraries

- Real-world libraries are complex!
- Optimizations, platform-specific code, and bulk.
- Hardware-focused implementations are different.
- **Goal**: Find the “targets” we just implemented in production code.

Task 4: AES in Mbed TLS

- **Library**: Mbed TLS (Embedded focus)
- **Task**: Locate the AES implementation and answer questions about it.
- **Hint**: It is in a submodule.
- **Questions**
  - Is it using an SBox table?
  - Any other tables?
  - Where/how does it do the key schedule?
  - Does it contain multiple implementations? If so, which ones?

Task 4: AES in Mbed TLS

Solution

- Is it using an SBox table?
  - Yes, here, but only used in key schedule and the last encryption round.
- Any other tables?
  - Yes, otherwise uses T-table approach to AES.
- Where/how does it do the key schedule?
  - Here, using the SBox table.
- Does it contain multiple implementations? If so, which ones?
  - Yes, AES-NI and AESCE.

Task 5: RSA in libgcrypt

- **Library**: libgcrypt (Behind GnuPG)
- **Task**: Locate the RSA implementation and answer questions about it.
- **Questions**
  - What are the operations performed with the key?
  - Does it mention any side-channel attack countermeasures?
    - If so which?
    - Can you find out more about them?

Task 5: RSA in libgcrypt

Solution

- What are the operations performed with the key?
  - A square-and-multiply or blinding and then square-and-multiply in RSA-CRT.
- Does it mention any side-channel attack countermeasures?
  - Yes, blinding, as a defence against the acoustic attack mentioned during the lecture (info here).
  - Instead of $m \equiv c^d \pmod{n}$ do:

```
1  r ←$ Z*_n                  //sample random value
2  y ≡ c r^e  (mod n)         //blind
3  x ≡ y^d  (mod n)           //modexp
4  m ≡ x r^-1  (mod n)        //unblind
```

  - This works because $m \equiv xr^{-1} \equiv y^d r^{-1} \equiv (cr^e)^d r^{-1} \equiv (c^d r) r^{-1} \equiv c^d \pmod{n}$

Because $e \equiv d^{-1} \pmod{\varphi(n)}$

Task 6: ECC in Micro-ECC

- **Library**: Micro-ECC (Tiny, single-file)
- **Task**: Locate the scalar multiplication implementation and answer questions about it.
- **Questions**
  - What are the operations performed with the key?
  - Does it mention any side-channel attack countermeasures?
    - If so which?
    - Can you find out more about them?

Task 6: ECC in Micro-ECC

Solution

- What are the operations performed with the key?
  - A certain ladder-based scalar multiplication algorithm is used.
- Does it mention any side-channel attack countermeasures?
  - It fixes the bit-length of the scalar.
  - It blinds the inversion of the nonce in ECDSA.
  - It randomizes the projective Z coordinate of the input point.

Takeaways

- You do the takeaways now