Engineering Cryptographic Software

The Programming Assignments

Hien Pham and Amin Abdulrahman

January 2026

The Setup

What you will need hardware-wise:

- A laptop/computer
- An STM32Nucleo-L4R5ZI board
- A micro USB cable

What you will need software-wise:

- The VM available from: `https://nce.mpi-sp.org/index.php/s/y4ddz4cwYgxjKD7`[1]
- A recent version of VirtualBox + its expansion pack
  - → The VM can be imported like this: VirtualBox → File → Import Appliance → Select `CryptoEngineering-VM-v20260107.ova`

[1] We also have copies on a USB drive

Update the code

Please run:

`git pull --rebase`

(Optionally: `git stash` beofre and `git stash apply` after)

Assignment 0

Adding up 1000 integers

Summation

Simple task: Write a Jasmin program that sums up 1000 integers. Then make it fast.
Where to start:

- Inside the VM, find the directory of the assignment under `~/cryptoeng/assignment0-sum`.
- Check the README for detailed information.
- You will only need to modify: `sum.jazz`
- `sum_wrapper.h` hints at the interface: The first argument is a pointer to an array of 32-bit integers. The second argument is a 32-bit integer defining the number of integers to be summed up (in our case, 1000).

Assignment 1

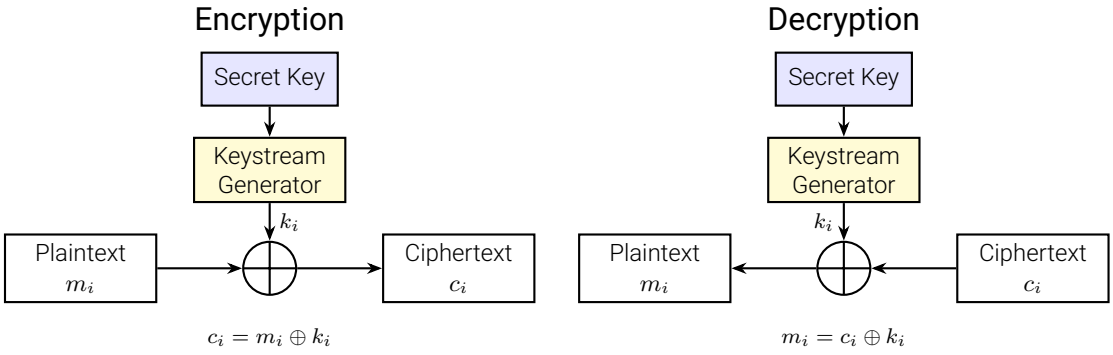
ChaCha20

Stream Cipher Recap: General

- Symmetric key cipher
- General idea: Combine plaintext with a stream of pseudorandom bytes, the *keystream*
- Loosely inspired by the one-time pad (OTP)

Stream Cipher Recap: General

- Symmetric key cipher
- General idea: Combine plaintext with a stream of pseudorandom bytes, the *keystream*
- Loosely inspired by the one-time pad (OTP)



Stream Cipher Recap: Security & Comparison

For a stream cipher to be secure, it should offer the following:

- Have a “random”-looking keystream
- Bias-free keystream
- Large-period keystream
- Impossibility to recover the key from the ciphertext or keystream

Stream Cipher Recap: Security & Comparison

For a stream cipher to be secure, it should offer the following:

- Have a “random”-looking keystream
- Bias-free keystream
- Large-period keystream
- Impossibility to recover the key from the ciphertext or keystream

In comparison to block ciphers (e.g., AES):

- Operating on small chunks of data (e.g., in a streaming context) will not make the output a full-sized block
- More resistant to noise on transmission channel
- Stream ciphers are often faster and more easily implementable in hardware

ChaCha20

- Stream cipher proposed by Bernstein in 2008
- Alternative to AES on platforms without hardware acceleration
- ARX design: **A**dd, **R**otate, **X**OR
- “Easily” implementable in constant-time
- Used in TLS, SSH, and other modern protocols
- Details: `https://datatracker.ietf.org/doc/html/rfc8439`

ChaCha20

- Stream cipher proposed by Bernstein in 2008
- Alternative to AES on platforms without hardware acceleration
- ARX design: **A**dd, **R**otate, **X**OR
- “Easily” implementable in constant-time
- Used in TLS, SSH, and other modern protocols
- Details: `https://datatracker.ietf.org/doc/html/rfc8439`

Parameters:

- 256-bit key (32 bytes)
- 96-bit nonce (12 bytes)
- 32-bit block counter (4 bytes)
- Generates 64-byte keystream blocks

ChaCha20

- Has a state of 64 bytes, 4×4 matrix of 32-bit words
- With c = constant, k = key, b = blockcount, and n = nonce:

cccccccc 0	cccccccc 1	cccccccc 2	cccccccc 3
kkkkkkkk 4	kkkkkkkk 5	kkkkkkkk 6	kkkkkkkk 7
kkkkkkkk 8	kkkkkkkk 9	kkkkkkkk 10	kkkkkkkk 11
bbbbbbbb 12	nnnnnnnn 13	nnnnnnnn 14	nnnnnnnn 15

Constants: `0x61707865`, `0x3320646e`, `0x79622d32`, `0x6b206574`

ChaCha20

- Elementary operation: *Quarterround (QR)*
  - Four add-xor-rotate operations each:
    1. `a += b; d ^= a; d <<<= 16;`
    2. `c += d; b ^= c; b <<<= 12;`
    3. `a += b; d ^= a; d <<<= 8;`
    4. `c += d; b ^= c; b <<<= 7;`
  - Note: Addition $\mod 2^{32}$
- Per block: 20 rounds = 10 $\times$ (column round + diagonal round)
- Column round: QR(0, 4, 8, 12), QR(1, 5, 9, 13), QR(2, 6, 10, 14), QR(3, 7, 11, 15)
- Diagonal round: QR(0, 5, 10, 15), QR(1, 6, 11, 12), QR(2, 7, 8, 13), QR(3, 4, 9, 14)

ChaCha20: Summary

1. Initialize the state (as previously shown)
2. Compute 20 rounds
3. Add ($\mod 2^{32}$) initial state to newly obtained state
4. Obtain 64 bytes of keystream
5. XOR keystream onto plaintext

Ultimate goal:

```
export fn crypto_stream_chacha20_ietf(
  reg u32 ct_ptr, reg ptr u8[12] nonce_ptr,
  reg ptr u8[32] sk_ptr, reg u32 ct_len)
```

Implementing ChaCha20 on Cortex-M4

- Strategy for optimizing on the M4
  - Write `quarterround` function in Jasmin
  - (Merge 4 `quarterround` functions into a full round)
  - Implement loop over 20 rounds in Jasmin
  - Implement loop over message length in Jasmin
  - Optimize inner loop over 20 rounds:
    - Keep data in registers as much as possible (reduce loads/stores)
    - Eliminate `ROR` instructions

Useful features of the M4

- 16 state words won't fit into registers, you need the stack
  - `-auto-spill(-all)`
  - Use `() = #spill(X);` and `() = #unspill(X);`

Useful features of the M4

- 16 state words won't fit into registers, you need the stack
  - `-auto-spill(-all)`
  - Use `() = #spill(X);` and `() = #unspill(X);`
- Second input of arithmetic instructions goes through barrel shifter
- Can shift/rotate one input **for free**
- Examples:
  - `a = b ^ (c << 2)`: left-shift `c` by 2, xor to `b`, store result in `a`
  - `c = a + (b >>r 5)`: right-rotate `b` by 5, add to `a`, store result in `c`
  - Note: Ordering is important! Can only shift latter argument

Assignment 2

ECDH on Curve25519

Before we start

Please run:

```
git stash
git pull --rebase
git stash apply
```

Overview of this exercise

- Implement **constant-time** conditional move

Overview of this exercise

- Implement **constant-time** conditional move
- Implement scalar multiplication on Ed25519 using **double-and-add**

Overview of this exercise

- Implement **constant-time** conditional move
- Implement scalar multiplication on Ed25519 using **double-and-add**
- Implement scalar multiplication on Ed25519 using **double-and-add-always**

Overview of this exercise

- Implement **constant-time** conditional move
- Implement scalar multiplication on Ed25519 using **double-and-add**
- Implement scalar multiplication on Ed25519 using **double-and-add-always**
- Verify correctness against the Python reference implementation

Overview of this exercise

- Implement **constant-time** conditional move
- Implement scalar multiplication on Ed25519 using **double-and-add**
- Implement scalar multiplication on Ed25519 using **double-and-add-always**
- Verify correctness against the Python reference implementation
- Verify if the implementations are constant-time (CT) using Jasmin's CT checker

Structure of Assignment 2

The scalar multiplication is $k \cdot Q$, where:

- k is the *scalar*, i.e., an integer

Structure of Assignment 2

The scalar multiplication is $k \cdot Q$, where:

- k is the *scalar*, i.e., an integer
- Q is a *point* on **the twisted Edwards curve** $-x^2 + y^2 = 1 + dx^2y^2$
  - $d = -121665/121666$ over the field $\mathbb{F}_p$ with $p = 2^{255} - 19$, i.e.,
    $d = 37095705934669439343138083508754565189542113879843219016388785533085940283555$

Structure of Assignment 2

The scalar multiplication is $k \cdot Q$, where:

- k is the *scalar*, i.e., an integer
- Q is a *point* on **the twisted Edwards curve** $-x^2 + y^2 = 1 + dx^2y^2$
  - $d = -121665/121666$ over the field $\mathbb{F}_p$ with $p = 2^{255} - 19$, i.e., $d = 37095705934669439343138083508754565189542113879843219016388785533085940283555$
  - Its coordinates are in $\mathbb{F}_p$

Structure of Assignment 2

The scalar multiplication depends on two things: field arithmetic and group arithmetic.

- Field arithmetic over $\mathbb{F}_p$:
  - Modular addition, modular subtraction, modular multiplication, inversion
  - Provided in `src/fe25519.jazz`

Structure of Assignment 2

The scalar multiplication depends on two things: field arithmetic and group arithmetic.

- Field arithmetic over $\mathbb{F}_p$:
  - Modular addition, modular subtraction, modular multiplication, inversion
  - Provided in `src/fe25519.jazz`
- Group arithmetic:
  - Point addition, point doubling
  - Provided in `src/ge25519.jazz`

Structure of Assignment 2

The scalar multiplication depends on two things: field arithmetic and group arithmetic.

- Field arithmetic over $\mathbb{F}_p$:
  - Modular addition, modular subtraction, modular multiplication, inversion
  - Provided in `src/fe25519.jazz`
- Group arithmetic:
  - Point addition, point doubling
  - Provided in `src/ge25519.jazz`
- Scalar multiplication is provided in `src/smult.jazz` and `src/smult_ct.jazz`

Structure of Assignment 2

Available test targets:

- `make TARGET=fe25519 test-board`
- `make TARGET=ge25519 test-board`
- `make test-board`
- `make CT=no test-board`
- `make TARGET=all test-board`
- `make TARGET=fe25519_rd32 test-board`

Constant-time conditional move

The **non-constant-time** version in `src/fe25519.jazz`:

```
fn fe25519_cmov(reg ptr u32[N] pr px,
                reg u32 b)
                -> reg ptr u32[N] {
  if (b == 1) {
    pr = #copy(px);
  }

  pr = pr;
  return pr;
}
```

To make it **constant-time**:

- `pr = b*px + (1 - b)*pr`
- Expand `b` to all-one/all-zero mask
- Use AND as multiplication
- Use XOR as addition

To check if your implementation is correct, run: `make TARGET=fe25519 test-board`

Jasmin's Constant-time Checker

Annotate your function's arguments with `public` or `secret` types:

- `public`: data may be allowed to leak
- `secret`: must not leak
- Example: `#[ct = "secret * public -> secret"]`
- Reference: https://jasmin-lang.readthedocs.io/en/stable/tools/ct.html#type-system

Jasmin's Constant-time Checker

Annotate your function's arguments with `public` or `secret` types:

- `public`: data may be allowed to leak
- `secret`: must not leak
- Example: `#[ct = "secret * public -> secret"]`
- Reference: `https://jasmin-lang.readthedocs.io/en/stable/tools/ct.html#type-system`

Run Jasmin's CT checker on `fe25519_cmov` with:

```
jasmin-ct --arch arm-m4 src/fe25519.jazz --slice=fe25519_cmov
```

- `--arch`: target architecture
- `--slice`: function to check CT property

Scalar Multiplication on Ed25519

Double-and-add:

$R \leftarrow Q$
for $i \leftarrow n-2$ downto 0 do
    $R \leftarrow 2Q$
    if $(k)_2[i] = 1$ then
        $R \leftarrow R + Q$
    end if
end for
return R

Double-and-add-always:

$R \leftarrow Q$
for $i \leftarrow n-2$ downto 0 do
    $R \leftarrow 2R$
    if $(k)_2[i] = 1$ then
        $R \leftarrow R + Q$
    else
        $R \leftarrow R + \mathcal{O}$
    end if
end for
return R

Scalar Multiplication on Ed25519

In the scalar multiplication:

- Addition and doubling are *point addition and doubling* (in `ge25519.jazz`)

Scalar Multiplication on Ed25519

In the scalar multiplication:

- Addition and doubling are *point addition and doubling* (in `ge25519.jazz`)
- `-auto-spill` is used

Scalar Multiplication on Ed25519

In the scalar multiplication:

- Addition and doubling are *point addition and doubling* (in `ge25519.jazz`)
- `-auto-spill` is used
- To avoid register allocation errors, you may need to use `#[spill]` when introducing a new register.

Scalar Multiplication on Ed25519

In the scalar multiplication:

- Addition and doubling are *point addition and doubling* (in `ge25519.jazz`)
- `-auto-spill` is used
- To avoid register allocation errors, you may need to use `#[spill]` when introducing a new register.

Once you are done, do:

- `make CT=no test-board` to test the double-and-add implementation
- `make test-board` to test the double-and-add-always implementation
- Annotate and run Jasmin's CT checker on the function `crypto_scalarmult`

Some Jasmin notations that may help you for this exercise:

- `#copy(X)`: copy the content of an array to another array
- `#[spill] reg u32 X`: tell the Jasmin compiler to automatically spill register `X` onto the stack
- `!X`: bit negation of `X` (i.e., `!101 = 010`)