Exercises

Exercises marked with one or more stars (⋆) are more difficult than the others (the more stars, higher the difficulty) and may be skipped.

Throughout the exercises, it is advised to often check that the programs that you write are accepted by the `jasminc` compiler, i.e., that they can be successfully translated into assembly.

Part 0: getting started

This exercise is meant to ensure that the Jasmin tool box is properly installed and to practice the basics of its command-line interface.

Look at the file `src/nbaesenc.jazz`.

Run the compiler on this file to produce assembly code (make sure the `AES` library, available in the `src/aeslib` directory, can be found by the compiler).

Assemble the resulting assembly into a binary file and call the encryption scheme from an other programming language: to this end, use one of the example programs given in the `bindings/` directory. Instructions are available in the `Makefile` in that directory.

Note that the programs generated by the Jasmin compiler target x86_64 processors featuring `AVX2` and `AES-NI` instruction-set extensions. If you don't have such a processor, you won't be able to run these programs. This is not a severe limitation. To double-check the capabilities of your processor, in the `bindings/` directory, run `make test_aes && ./test_aes`.

Part 1: constant-time programming

Subtraction

The goal is to implement a constant-time function `subtract(reg u64 x y) -> reg u64` that computes max(0, x − y), i.e., the difference of its arguments if this difference is positive and zero otherwise. A template is given in file `constantTime/subtract.jazz`.

1. Implement a first version using an `if`.
2. Is this implementation constant-time? Use `-checkCT` to confirm your answer.
3. Write a branchless implementation.
4. Confirm that this second version is indeed secure.

Optionally, run the functions to test their behaviors.

Homework:

5. ⋆ The `SUB` instruction computes the sign of the difference in a flag: can you exploit this property to optimize your program?

Fast exponentiation

This exercise studies the well-known “fast exponentiation” algorithm. To compute the e-th power of x, this algorithm iterates on all bits of the binary representation of the exponent as follows:

```
r ← 1
repeat
    if e is odd
        r ← r × x
    x ← x²
    e ← e ÷ 2
until e = 0
return r
```

We will (first) implement this algorithm for 64-bit machine words, so overflows may occur and results might be surprising. A template of the function `exp(reg u64 x e) -> reg u64` is given in the file `constantTime/exp.jazz`.

0. ⋆ Optionally, implement this algorithm in Jasmin.
1. Look at the first implementation (`exp_v0`) of this algorithm in the file `correction/exp.jazz`. Is this function constant-time? What are the issues?
2. Fix the implementation to make it secure.

Homework:

3. ⋆ Instruction `SHR` sets the `zf` flag when its result is zero: it can be used as a criterion to exit the `while` loop. Would it be secure? If we assume (or enforce) that the size of the exponent is public, the value of this bit can be “declassified” to make the program well-typed.
4. ⋆ Implement a *modular* version, i.e., a function `mod_exp(reg u64 x e m) -> reg u64` that performs computations modulo `m`. Under which pre-condition are there no overflow? Is a single final modular reduction enough or should the multiplication also be made modular?
5. ⋆⋆ Make a version that operates on 128-bit values held in arrays of two 64-bit values, i.e., write a function `exp_u128(reg u64[2] x, reg u64 e) -> reg u64[2]`. This requires to first implement the 128-bit multiplication.

Conditional copy

In this exercise, the task is to copy the contents of an array into an other array when a condition holds; otherwise the destination array is to be filled with zeros. All of this in a constant-time manner. The array lengths are public, but the condition is secret; the array contents are also secret. A template is given in file `constantTime/ccopy.jazz`.

1. Write a first (non-constant-time) version using `for` loop(s) and an `if`. What's better: nesting the `if` inside the loop or the opposite? Caveat: moving data in memory may require going through an intermediate register.
2. Use `while` loops instead of `for` loops. Does it help in making the program constant-time?
3. Make the program constant-time using arithmetic only (no conditional move). Hint: the `SETcc` instruction allows to move a boolean flag into a `u8` register. Caveat: there is no 8-bit multiplication available in Jasmin.
4. Use conditional moves instead. Try with both `while` and `for` loops. Does it make a difference?

Part 2: encryption scheme

Two-blocks encryption

In this exercise, the aim is to implement an encryption scheme for 256-bit messages. The basic idea is to split the message into two blocks and use the 128-bit encryption scheme designed earlier on each of the blocks. Care must be taken to ensure that the nonce is not used twice with the same key.

There are two functions to fill in the file `exercises/encryption/nbaesenc2.jazz`. A few helpers function are already provided.

Both functions to implement should have the same functionality, but one of them is more efficient: some intermediate values are common to the encryption of both blocks and can be computed once.

Details are given in the source file: read it carefully.

⋆⋆⋆ Counter mode

Generalize the encryption scheme to apply to messages of arbitrary size. (Only consider messages that are properly padded to a length that is a multiple of the block size.) To this end, the nonce can be used as a counter that gets incremented after each block. Read more about this mode of operation: https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#Counter_(CTR).

Some useful instructions

- `?{zf}, x = #DEC(x);` decrement x, zf is true if the result is 0
- `?{zf, cf}, x = #SHR(x, 1);` compute x » 1, zf is true if the result is 0, cf is true is the initial value of x mod 2 = 1
- `?{sf}, d = #SUB(x, y);` compute x - y and set the result in d, sf is true is the result is negative.
- `c = #SETcc(cond);` set the value of cond (a boolean) into c an u8

- `x = (16u)c;` cast an expresion into a u16, if b is an u8 it is a zero extention.
- `?{}, i = #set0();` initialize the variable i (of type u64) to 0
- `?{}, i = #set0_16();` initialize the variable i (of type u16) to 0
- `cond = #BT_16(s_cond, 0);` extra the value of the bit 0 of s_cond and set it into cond.
- `(int)i` cast the word i (u8, u16, u32, u64) into an int