Title page

In this tutorial we will give an overview of Jasmin, a programming language designed to write high-assurance high-speed cryptography.

We are Miguel and Santiago from MPI-SP.

The tutorial will be pretty hands-on, so let's begin right away with an example.

[HINT: NEXT FRAME]

memeq

Jasmin is a low-level programming language with C-like syntax.

This is a simple function that checks if two memory regions `p` and `q` coincide in the first `n` quadwords.

Jasmin comes with tools to verify that code is

- correct (relative to a specification);
- safe (e.g., it doesn't divide by zero); and
- secure (e.g., against side-channel attacks).

memeq

It has assignments, whiles, ifs, and functions.

But is is lower-level than C: it really is structured assembly.

Each Jasmin instruction must correspond to an assembly instruction.

[HINT: NEXT FRAME]

memeq 2

Indeed, we can use assembly instructions directly in the Jasmin source file.

[HINT: NEXT FRAME]

Here I'm using the `INC` x86-64 instruction to increment `i`, this is the dashed red arrow.

memeq 2

There are two reasons we need a language at a lower-level than C: efficiency and security.

Regarding efficiency, we can be much more precise and detailed with our optimizations.

Regarding security, we can see many low-level problems at source-level, without the compiler getting in the way.

memeq 2

Well, but then why bother with Jasmin when we have assembly? Same reasons, different emphasis: security and efficiency.

A structured language with a clearly defined semantics avoids tons of the problems of trusting large assembly codebases.

It is also important for efficiency: higher-assurances on our code allows more aggressive optimizations.

memeq 2

Jasmin gives structure to assembly programs with functions, conditionals and loops without compromising efficiency or security.

[HINT: NEXT FRAME]

Their compilation is standard and predictable:

- functions (in dotted green) compile to a label and a return;
- loops (in continuous red) compile to a check and a backward jump; and
- conditionals (in dashed blue) compile to a check and a jump to the else branch.

More online

Now let's write some Jasmin.

[HINT: Have people open the files and setup Docker.]

[HINT: Leave the Formosa slide while people work.]

gimli

In `jazz_gimli.h` you can see the functions we will implement today.

In `gimli.jazz` you have the function declarations with empty bodies and some hints.

These functions use pointers, so let me quickly present the syntax.
[HINT: Show `otp_fixed` from `otp.jazz`.]

gimli: sboxes

Operators are basically the same as in C, for instance here addition is plus and exclusive or is caret.

[HINT: After we're done with sboxes.]

The next functions use control flow and arrays, so let us check the syntax for them.

otp

[HINT: Show `otp.jazz`]

This function takes three pointers, XORs the message and the key and writes the result to the ciphertext.

Array accesses have the same syntax as in C.

For loops in Jasmin are always unrolled, this means that they need to have constant ranges.

This function will have $16 \times 3 = 48$ instructions.

EasyCrypt

One of the reasons to use Jasmin is that we can formally prove things about programs in the EasyCrypt proof assistant.

[HINT: Show terminal with `jasminc otp.jazz -ec otp`.]

Here I extract a representation of the otp function which I can paste in an EC file.

[HINT: Show EasyCrypt file.]

The proof assistant lets us write statements like "if the key comes from a uniform distribution, the output is uniformly distributed."

gimli

[HINT: Back to gimli.]

safety

Show how to avoid common pitfalls.

side channels

Show the CT checker and the SCT checker.

gimli: AVX

Jasmin supports vectorized (AVX) instructions.