Manual of faz_ecc_avx2

July, 2017

Abstract

faz_ecc_avx2 is an optimized mathematical software library for computing Edwards Digital Signature Algorithm (EdDSA) and the Diffie-Hellman functions X25519 and X448. This library was specially accelerated with the Intel Advanced Vector eXtensions version 2 (AVX2).

This library belongs to the research project titled:

"High Performance Implementation of the Elliptic Curve
Cryptography using Vector Instructions"

authored by:

- Armando Faz Hernández (armfazh@ic.unicamp.br)
- Julio López (jlopez@ic.unicamp.br)
- Ricardo Dahab (rdahab@ic.unicamp.br)

This research work is part of a paper submission to the ACM Journal of Mathematical Software. Software will be publicly available after paper acceptance in the Collected Algorithms from ACM (CALGO) website and in the ACM Digital Library. During reviewing process, source code is available only to reviewers and journal editors through the ACM submission website.

Contents

1 Installation

1.1 Installation Prerequisites

The following instructions are suitable to work in a Linux environment. Compilation steps may be different under Windows, MacOS, or Android operating systems.

Compilation. The tools required before to start compilation are the following:

- A file extractor, such as `tar`.
- A C compiler which supports the Intel Intrinsics and AVX2 instruction set. Additionally, compiler must be able to generate code for Haswell, Broadwell or Skylake micro-architectures.

  We recommend the use of one of the following compilers:

  - GNU C compiler[1] version 4.8.5 or later.
  - Intel C Compiler[2] version 16 or later.
  - Clang[3] compiler version 3.8 or later.
- `make`. Tool for automated compilation.
- `cmake`.[4] Compilation tool-kit version 2.8.4 or later.

Documentation. For source code documentation, you will need the following tools:

- `doxygen`[5] tool.
- LaTeX program.
- Graphviz. For generating diagrams.

See Section 5.4 for more details about generation of documentation.

1.2 Third-Party Supporting Code

Our library is self contained. It does not require any other software beyond than the tool-chain for compilation and the standard libraries.

1.3 Source Code Portability

Our code has been compiled with three different compilers: the GNU C Compiler, the Intel C Compiler, and the Clang compiler. In all the cases, code was compiled with the following compilation flags:

```
-O3 -Wall -Wextra -pedantic -std=c99 -mavx2 -march=native -mtune=native
```

Using these flags code is verified to follow the standard of C99 (ISO/IEC 9899:1999); thus, our code does not report any errors and passes all testing programs. See Section 2, for more details about testing driver.

[1] gcc: `http://gcc.gnu.org`

[2] icc: `https://software.intel.com/en-us/c-compilers`

[3] clang: `https://clang.llvm.org/`

[4] cmake: `https://cmake.org/`

[5] doxygen: `http://www.stack.nl/~dimitri/doxygen/`

1.4 Machine Dependent Code

Our code is machine dependent since the main purpose is to accelerate the calculation of cryptographic algorithms using a special vector unit called AVX2, which stands for Advanced Vector eXtensions version 2. Thus, our library runs only on AVX2-ready processors.

The Haswell, Broadwell, Skylake, and Kaby Lake are code-names of Intel micro-architectures that support AVX2, and consequently, they are also able to execute our library. AMD's Ryzen processor also supports AVX2, however, we have not tested our code in this processor.

To verify if your machine supports AVX2 instructions, there are several ways to test it:

1. If your are in a Linux environment, execute:

```
$ cat /proc/cpuinfo
```

This command will report in the *flags* field the features that processor supports. It is mandatory that your processor has the features listed in the following table.

Micro-arch.	Flag	Feature Description
Haswell, Broadwell, Skylake, Kaby Lake.	**sse2** **ssse3** **sse4_1** **sse4_2**	SIMD Streaming Extensions.
	avx	Advanced Vector eXtensions.
	avx2	Advanced Vector eXtensions version 2.
	bmi1 **bmi2**	Bit Manipulation Instructions version 1, and 2. The `MULX` integer multiplier belongs to this feature.
	cmov	Conditional Move Instruction.
Skylake, Kaby Lake.	**adx**	`ADOX`/`ADCX` new instructions for addition with carry and addition with overflow.

2. A second way to verify the support for AVX2 instructions consists on running a companion program that verifies these flags. Follow the instructions of compilation (in Section 1.5), and then, run cpu_id program.

```
$ cd apps/samples
$ gcc -o cpu_id cpu_id.c
$ ./cpu_id
```

This program will report whether processor has or not the required features to run our library. For example: this is the output produced by our Haswell computer:

```
=== Environment Information ====
Program compiled with: 4.8.5
bit_CMOV    : [Yes]
bit_SSE     : [Yes]
bit_SSE2    : [Yes]
bit_SSE3    : [Yes]
```

```
bit_SSSE3   : [Yes]
bit_SSE4_1  : [Yes]
bit_SSE4_2  : [Yes]
bit_AVX     : [Yes]
bit_AVX2    : [Yes]
bit_BMI     : [Yes]
bit_BMI2    : [Yes]
bit_ADX     : [No]
Machine supports our library: Yes
```

and the following is the output produced by our Skylake processor.

```
=== Environment Information ====
Program compiled with: 4.8.5
bit_CMOV    : [Yes]
bit_SSE     : [Yes]
bit_SSE2    : [Yes]
bit_SSE3    : [Yes]
bit_SSSE3   : [Yes]
bit_SSE4_1  : [Yes]
bit_SSE4_2  : [Yes]
bit_AVX     : [Yes]
bit_AVX2    : [Yes]
bit_BMI     : [Yes]
bit_BMI2    : [Yes]
bit_ADX     : [Yes]
Machine supports our library: Yes
```

1.5 Compilation and Installation Process

Follow the steps for compiling and installing our library.

1. Once you have `faz_ecc_avx2.tar.gz` file, extract it in an empty folder.

   ```
    $ tar xfz faz_ecc_avx2.tar.gz
   ```

2. This command will create a new folder containing a directory structure like this:

   ```
   eddsa_avx2
   ├── apps
   │   ├── bench
   │   ├── tests
   │   └── samples
   ├── doc
   ├── include
   ├── src
   ├── CMakeLists.txt
   └── README.txt
   ```

3. After that, create an empty folder for building the library.

   ```
   $ cd eddsa_avx2
   $ mkdir build
   $ cd build
   ```

4. Then execute the cmake command specifying the parent folder (or the path where the CMakeLists.txt file can be found). cmake will scan the compiler and headers required for compilation.

   ```
   $ cmake ..
   ```

5. If everything is fine, cmake will create a Makefile file containing instructions for compilation. Then run make tool:

   ```
   $ make all
   ```

6. After compilation, you can (optionally) install library in your system by typing:

   ```
   $ make install
   ```

   Beware that installation on operating system may require administrative permissions.

1.6 Customized Compilation

1.6.1 Setting a Compiler

If you want to switch to a determined compiler, you must indicate to cmake the path of the new compiler. For that end, you must set the environment variable CC as follows:

```
$ export CC=clang
$ cmake ..
```

Then, cmake will verify this compiler and will produce an output similar to this:

```
-- The C compiler identification is Clang 3.9.1
-- Check for working C compiler: /usr/bin/clang
-- Check for working C compiler: /usr/bin/clang -- works
...
```

After that, compile library by running make command.

```
$ make all
```

1.6.2 Setting Installation Directory

By default, cmake will install library in /usr/local folder. However, if you want to install library in other path, you must specify this to cmake as follows:

```
$ cmake -DCMAKE_INSTALL_PREFIX:PATH=/opt ..
```

This command will tell cmake to install library in /opt folder. Then, run:

```
$ make install
```

After that, the `/opt` directory will have the following files installed in your system:

```
/opt
├── include
│   ├── faz_ecdh_avx2.h
│   ├── faz_eddsa_avx2.h
│   └── faz_fp_avx2.h
└── lib
    └── libfaz_crypto_avx2.a
```

2 Testing Library

faz_ecc_avx2 contains a set of unit tests called by a driver program for verifying the functionality of the library. To run those tests, perform the following instructions:

```
$ cd eddsa_avx2
$ mkdir build
$ cd build
$ cmake ..
$ make all
$ ./bin/tests
```

Program will run and produce an output similar to this:

```
=== Testing Arith  ===
===   p=2^255-19   ===
===== 1-way AVX2 =====
Test mul/sqr: 50000 OK
Test mul/inv: 50000 OK
===== 1-way x64  =====
Test mul/sqr: 50000 OK
Test mul/inv: 50000 OK
===== 2-way AVX2 =====
Test mul/sqr: 50000 OK
...
```

Once program ends, it will report whether or not passed all the tests.

2.1 Our Testing Environment

We use three different computers for testing our library.

Haswell This is a desktop computer that has a Core-i7 4770 running at 3.4 GHz, this processor belongs to the Haswell micro-architecture family of Intel, and also known as the 4-th family of Core processors.

Yoga This is a laptop computer that has a Core-i3 4012Y running at 1.5 GHz, this processor belongs to the Haswell micro-architecture family of Intel, and also known as the 4-th family of Core processors.

Skylake This is a desktop computer that has a Core-i7 6700K running at 4.0 GHz, this processor belongs to the Skylake micro-architecture family of Intel, and also known as the 6-th family of Core processors.

All systems run Fedora 24 operating system.

3 Benchmarking Library

3.1 Previous Considerations

You must beware about the following issues when benchmarking library. Since we report timings in clock cycles, then it is recommended to turn off the following processor technologies to get accurate measurements.

- Intel Hyper-Threading. This technology makes that a physical processing core be shared by two or more logical processor units. Then, when measuring a code with Intel Hyper-Threading activated, the clock cycle counter could measure the time spend by other programs running in the same physical core.
- Intel Speed-Step. This technology makes that a processor varies the operating clock frequency. Then, when processor has a low workload, Intel Speed-Step reduces frequency with the aim to save energy. This can affect benchmark measurements.
- Intel Turbo Boost. This is the complement of the Intel Speed-Step, whenever a processor has a CPU-bound workload, then Intel Turbo Boost increases clock frequency beyond the limits of the nominal frequency, affecting also the clock cycle counter. Measuring timings with this technology activated can result on non-reproducible timings.

Additionally, we also recommend to perform benchmarking using the a minimal operating system. This mode can be activated by running the Level 3 of Linux operating system.

```
$ telinit 3
```

This command will enter in the level 3 of the system, which disables graphical user interface. To recover from Level 3 you can execute

```
$ telinit 5
```

to return to the default mode.

3.2 Benchmark Report

Our library has a companion program to perform the benchmark of the library. Compile this program using:

```
$ cd eddsa_avx2
$ mkdir build
$ cd build
$ cmake ..
$ make all
$ ./bin/bench
```

When you run the benchmark program, program will produce an output like the following:

```
=== Benchmarking Arith ===
======  p=2^255-19  ======
======  1-way AVX2  ======
addition       =      7 cc
subtraction    =      9 cc
```

```
multiplication =     56 cc
squaring       =     69 cc
coef_reduction =     22 cc
inversion      =  17140 cc
square_root    =  17593 cc
...
```

Benchmark program takes a couple of minutes to finish (this mainly depends on the clock frequency of the benchmarking machine). The `bench` program reports the number of clock cycles taken for computing each operation used on the library. We measure the following primitives:

- Prime field arithmetic for $\mathbb{F}_q$, where $q = 2^{255} - 19$ and $q = 2^{448} - 2^{224} - 1$. Using both the single, 2-way, and 4-way operations.
- Elliptic curve arithmetic on twisted Edwards curves. Point additions and doublings.
- Miscellaneous. w-NAF recoding, signed digit regular recoding, and hash function evaluations.
- Diffie-Hellman computation. Time for computing key generation and shared secret using `X25519` and `X448`.
- EdDSA operations. Time for computing key generation, signature generation, and signature verification using `Ed25519` and `Ed448`.

For protocol operations, program reports timings measured in μ-seconds (10^{-6} seconds), the ratio of operations per second, and the clock cycles taken for completing an operation.

```
===== Benchmarking DH ====
======  X25519 AVX2 ======
ecdh->keygen   :  12 μs,  81694.1 oper/s,  41522 cycles/op
ecdh->shared   :  37 μs,  26859.3 oper/s, 126293 cycles/op
== Benchmarking EdDSA ====
====== Ed25519 AVX2 ======
eddsa->keygen  :  12 μs,  81365.1 oper/s,  41690 cycles/op
eddsa->sign    :  13 μs,  72506.0 oper/s,  46784 cycles/op
eddsa->verify  :  46 μs,  21541.9 oper/s, 157467 cycles/op
...
```

4 Usage and Driver Examples

For this section, we assume that library was already compiled and installed in /opt folder.

4.1 Hello world.

This is a simple program to verify the correct linkage of the library. This program can be found in eddsa_avx2/apps/samples/hello.c path. Now, we list the hello.c file:

```c
#include<faz_eddsa_avx2.h>
#include<stdio.h>

int main(void)
{
    printf("Size of an Ed25519 key: %d bytes.\n”,
           ED25519_KEY_SIZE_BYTES_PARAM);
    return 0;
}
```

To compile this program with the GNU C Compiler (gcc) execute the next steps:

```
$ cd apps/samples
$ gcc -o hello hello.c -I/opt/include -L/opt/lib -lfaz_crypto_avx2
$ ./hello
```

If compilation was successful, this will be the output after running the hello program.

```
Size of an Ed25519 key: 32 bytes.
```

4.2 Prime Field Arithmetic

The program eddsa_avx2/apps/samples/field_arith.c examplifies how to use the Fp global object.

```c
#include<faz_fp_avx2.h>
#include<stdio.h>

void identity(const struct _struct_Fp_1way * fp)
{
   /* Initialize variables */
   argElement_1w a      = fp->init(); argElement_1w b      = fp->init();
   argElement_1w add_ab = fp->init(); argElement_1w sub_ab = fp->init();
   argElement_1w left   = fp->init(); argElement_1w right  = fp->init();
   argElement_1w prime  = fp->get_modulus();

   printf("Testing the identity: (a+b)*(a-b) = (a^2-b^2) \n”);
   printf("Modulus: ”); fp->print(prime);

   /* Two random numbers */
   fp->rand(a); printf("a: ”); fp->print(a);
   fp->rand(b); printf("b: ”); fp->print(b);

   /* Left-hand side */
   fp->add(add_ab,a,b);          fp->cred(add_ab);
   fp->sub(sub_ab,a,b);          fp->cred(sub_ab);
   fp->mul(left,add_ab,sub_ab);  fp->cred(left);
   printf("left: ”);             fp->print(left);

   /* Right-hand side */
   fp->sqr(a);                   fp->cred(a);
   fp->sqr(b);                   fp->cred(b);
   fp->sub(right,a,b);           fp->cred(right);
   printf("right: ");            fp->print(right);

   /* Equality testing */
   if( fp->cmp(left,right) == 0 )
       printf("Test passed.\n\n");
   else
       printf("Test failed.\n\n");

   /* Clearing variables */
   fp->clear(a);         fp->clear(b);
   fp->clear(add_ab);    fp->clear(sub_ab);
   fp->clear(left);      fp->clear(right);
   fp->clear(prime);
}

int main(void)
{
    identity(&Fp.fp25519._1way_x64);
    identity(&Fp.fp25519._1way);
    identity(&Fp.fp448._1way);
    return 0;
}
```

To compile this program with the Clang compiler (`clang`) execute the next steps:

```
$ cd apps/samples
$ clang -o field_arith field_arith.c -I/opt/include \
  -L/opt/lib -lfaz_crypto_avx2
$ ./field_arith
```

If compilation was successful, this will be an output similar to the one produced by `field_arith` program.

```
Testing the identity: (a+b)*(a-b) = (a^2-b^2)
Prime: 0x7fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffed
a: 0x267a08ce69569234f496b41c13d93342fae9df3145e04a6674bf1a5622c2c76f
b: 0x7155c0268e59e6b4fe135448fb1b49eba62f92f022f7a2c6ab03025616165dcd
left: 0x3715b9125633b48de8e5c097d3f1d536a7615c180ec60b4c2899a6f36ab73384
right: 0x3715b9125633b48de8e5c097d3f1d536a7615c180ec60b4c2899a6f36ab73384
Test passed.
...
```

4.3 Elliptic Curve Diffie-Hellman

Now, we show an example of the calculation of a shared secret between Alice and Bob using the X25519 Diffie-Hellman function. File: `eddsa_avx2/apps/samples/ecdh.c`.

```
#include<faz_ecdh_avx2.h>
#include<faz_fp_avx2.h>  /* For print_bytes and random_bytes */
#include<stdio.h>

```

```
int main(void)
{
  size_t keysize = ECDH25519_KEY_SIZE_BYTES;
  ECDH_X25519_KEY alice_private,alice_session,alice_shared;
  ECDH_X25519_KEY bob_private,bob_session,bob_shared;

  printf("=== X25519 DH Example ===\n");
  /* Alice session key generation */
  random_bytes(alice_private,keysize);
  ECDHX.X25519.keygen(alice_session,alice_private);
  printf("Alice private key:\n");    print_bytes(alice_private,keysize);
  printf("Alice session key:\n");    print_bytes(alice_session,keysize);

  /* Bob session key generation */
  random_bytes(bob_private,keysize);
  ECDHX.X25519.keygen(bob_session,bob_private);
  printf("Bob private key:\n");      print_bytes(bob_private,keysize);
  printf("Bob session key:\n");      print_bytes(bob_session,keysize);

  /* Shared secret generation */
  ECDHX.X25519.shared(alice_shared,bob_session,alice_private);
  ECDHX.X25519.shared(bob_shared,alice_session,bob_private);
  printf("Alice shared secret:\n");  print_bytes(alice_shared,keysize);
  printf("Bob shared secret:\n");    print_bytes(bob_shared,keysize);

  return 0;
}
```

To compile this program with the Clang compiler (clang) execute the next steps:

```
$ clang -o ecdh ecdh.c -I/opt/include \
  -L/opt/lib -lfaz_crypto_avx2
$ ./ecdh
```

If compilation was successful, and after running the ecdh program, program will print an output similar to this:

```
=== X25519 Example ===
Alice private key:
0x5a8ba93dbc00cf2d7ff4e8421a6ff0c507189a50152bcfe11ace38c8cd31e761
Alice session key:
0x6283ceb1ae91013681cc1e0cdc2d2582e557a9b43f5ac3d821e7bd6c4d6e0f07
Bob private key:
0x28eb924f067e1d52c427eb04ba11e499022fff63ee76284d7de61cc188b6fc73
Bob session key:
0x45c13739af0b0e62c7e05e307d835b98a7846f1c092642f5e7cabff4a9e6eb6a
Alice shared secret:
0x2801ce8dd194a6bd93a90ef0590ea47bf41b4b22f4986464a61c07a7114261b9
Bob shared secret:
0x2801ce8dd194a6bd93a90ef0590ea47bf41b4b22f4986464a61c07a7114261b9
```

4.4 EdDSA signatures

The following script is an example about the use of Ed25519 signature scheme. You can found this file in `eddsa_avx2/apps/samples/eddsa.c`.

```
#include<faz_eddsa_avx2.h>
#include<faz_fp_avx2.h>   /* For print_bytes and random_bytes */
#include<stdio.h>
#include<string.h>

int main(void)
{
  size_t size_key = ED25519_KEY_SIZE_BYTES_PARAM;
  size_t size_sig = ED25519_SIG_SIZE_BYTES_PARAM;
  Ed25519_PublicKey pub_key;
  Ed25519_PrivateKey pri_key;
  Ed25519_Signature signature;
  int response = 0;

  uint8_t * message = (uint8_t *)"Keep Calm and Carry On”;
  const size_t size_msg = strlen((const char *)message);

  printf("=== Ed25519 Example ===\n”);
  /* Key generation */
  random_bytes(pri_key,size_key);
  EdDSA.Ed25519.keygen(pub_key,pri_key);
  printf("Alice’s Private Key:\n”); print_bytes(pri_key,size_key);
  printf("Alice’s Public Key:\n”);  print_bytes(pub_key,size_key);

  printf("Message in ascii: \n%s\n”,message);
  printf("Message in hex: \n”); print_bytes(message,size_msg);

  /* Signature generation */
  EdDSA.Ed25519.sign(signature,message,size_msg,pub_key,pri_key);
  printf("Ed25519 Signature:\n”);  print_bytes(signature,size_sig);

  /* Signature verification */
  response = EdDSA.Ed25519.verify(message,size_msg,pub_key,signature);
  if( response == EDDSA_VERIFICATION_OK )
    printf("Valid signature.\n”);
  else
    printf("Signature invalid.\n”);

  return 0;
}
```

To compile this program with the GNU C Compiler (`gcc`) execute the next steps:

```
$ gcc -o eddsa eddsa.c -I/opt/include \
  -L/opt/lib -lfaz_crypto_avx2
$ ./eddsa
```

If compilation was successful, the following is a similar output to the one produced by `eddsa` program.

```
=== Ed25519 Example ===
Alice’s Private Key:
0x643cd6bc8245d0d04f83cfc189daaa6d08ebc18b7ded4a6ad51b8a118ff9073c
Alice’s Public Key:
```

```
0xbbfd3c9726ec2d53bb3a36d2d9c315be0e3d943ac71d811df5bbbd9d29839fb3
Message in ascii:
Keep Calm and Carry On
Message in hex:
0x6e4f20797272614320646e61206d6c6143207065654b
Ed25519 Signature:
0x0fe79adaba74849b6728c9600d0ae212cc60a4deb9f016d84cfb3634235cd2a
a2c569caaeaf2d3811f731aec3701a846137d2bdac3714c0fa941b8ad6b991a3a
Valid signature.
```

5 Source Code Structure

Our library is split in three main modules or software layers: the prime field arithmetic, the elliptic curve Diffie-Hellman functions, and the Edwards Digital Signature functions.

5.1 Prime Field Arithmetic

This software layer provides the functions to compute arithmetic operations in the fields: $\mathbb{F}_q$, where $q = \{2^{255} - 19, 2^{448} - 2^{224} - 1\}$, which are the primes used in the implementation of the elliptic curve cryptographic protocols detailed below. All the functions contained in this layer can be accessed by including the header file:

```
#include<faz_fp_avx2.h>
```

5.1.1 Data Types

Single field elements, i.e. $A \in \mathbb{F}_q$, are represented by the following data types:

- `Element_1w_Fp25519`. A single element of $\mathbb{F}_q$, for $q = 2^{255} - 19$.
- `Element_1w_Fp448`. A single element of $\mathbb{F}_q$, for $q = 2^{448} - 2^{224} - 1$.
- `argElement_1w`. A pointer to a variable of type `Element_1w_Fp25519` or `Element_1w_Fp448`.

For the parallel implementation of prime field arithmetic, we define two-way and four-way variables as follows:

- `Element_2w_Fp25519`. Two elements $A, B \in \mathbb{F}_q$, for $q = 2^{255} - 19$. Following the notation of research paper, they are packed as $\langle A, B \rangle$.
- `Element_4w_Fp25519`. Four elements $A, B, C, D \in \mathbb{F}_q$, for $q = 2^{255} - 19$. Following the notation of research paper, they are packed as $\langle A, B, C, D \rangle$.
- `Element_2w_Fp448`. Two elements $A, B \in \mathbb{F}_q$, for $q = 2^{448} - 2^{224} - 1$. Following the notation of research paper, they are packed as $\langle A, B \rangle$.
- `Element_4w_Fp448`. Four elements $A, B, C, D \in \mathbb{F}_q$, for $q = 2^{448} - 2^{224} - 1$. Following the notation of research paper, they are packed as $\langle A, B, C, D \rangle$.

5.1.2 Operations

For each field and data types, we define the following operations:

add Computes a square: $c = a + b$.

sub Computes a square: $c = a - b$.

mul Computes a square: $c = a \times b$.

sqr Computes a square: $c = a^2$.

inv Computes a modular inverse: $c = \frac{1}{a}$.

sqrt Computes a square-root: $c = \sqrt{a}$.

cred Coefficient reduction of a variable.

cmp Compare for equality two variables.

rand Set a random number in a variable.

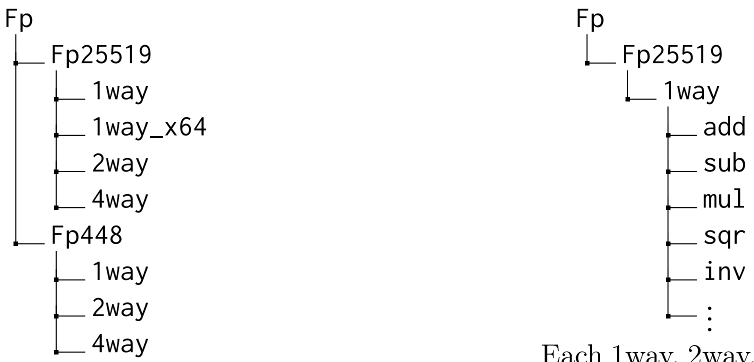
print Print a variable in hexadecimal base.

init Allocate dynamic memory for a variable.

clear	Deallocate dynamic memory of a variable.	get_modulus	Returns the prime q.

5.1.3 Usage of Prime Field Arithmetic

All of these combination of operations and data types are condensed into a single global variable called `Fp`. This variable is a data structure containing function pointers to operations defined for each prime field, as depicted in Figure 1.



Structure of global variable `Fp`.

Each 1way, 2way, and 4way fields have pointers to the operations listed in previous section.

Figure 1: Global variable `Fp` containing pointer to functions.

Example 5.1. This is an example of the addition of $C = A +_2 B$; which denotes the following operation:

$$\begin{aligned} C_0 &= A_0 + B_0 \\ C_1 &= A_1 + B_1 \end{aligned}$$

where $A_0, A_1, B_0, B_1 \in \mathbb{F}_q$, and these elements are packed as $A = \langle A_0, A_1 \rangle$, and $B = \langle B_0, B_1 \rangle$. In this version, variables are statically allocated.

```c
#include<faz_fp_avx2.h>

int main(void)
{
   Element_2w_Fp25519 A,B,C;
   Fp.fp25519._2way.add(C,A,B);
   return 0;
}
```

5.1.4 Misc Functions

This layer defines some functions for manipulating string of bytes.

- print_bytes. Print a string of bytes of length n in hexadecimal base.
- random_bytes. Fill a string of bytes of length n with random data. **Warning!** This function **must be** rewritten to include a stronger pseudo-random number generator. Our implementation accesses to the Linux file `/dev/urandom` to generate random bytes.

- compare_bytes. Compare for equality two strings of bytes of length n. Outputs 0 whenever both strings match; otherwise, returns a non-zero value.

5.2 Elliptic Curve Diffie-Hellman

This section describes the second software layer of this library. To use the functions of this layer include the following header file:

```
#include<faz_ecdh_avx2.h>
```

5.2.1 Constants

- `ECDH25519_KEY_SIZE_BYTES`. This is the size (expressed in bytes) of the keys used to perform the Diffie-Hellman protocol using `X25519`. This value is set to 32 bytes.

- `ECDH448_KEY_SIZE_BYTES`. This is the size (expressed in bytes) of the keys used to perform the Diffie-Hellman protocol using `X448`. This value is set to 56 bytes.

5.2.2 Data Types

- `ECDH_X25519_KEY`. This is the data type used to store a `X25519` key.

- `ECDH_X448_KEY`. This is the data type used to store a `X448` key.

- `argECDHX_Key`. Pointer to an ECDH key.

5.2.3 Operations

The Diffie-Hellman protocol defines only two operations:

- `keygen`. Alice uses this function giving as an input her private key; then, `keygen` function will generate a session key for Alice.

- `shared`. Alice invokes this function giving as an input the session key of Bob and her private key; then, `shared` function produces a shared secret for Alice.

5.2.4 Usage of ECDH Functions

To access to the ECDH functions, we provide the global variable called `ECDHX` that contains pointers to the implementation of the operations listed before. Figure 2 shows the structure of the `ECDHX` variable.

Example 5.2. This is an example of the calculation of a shared secret for Alice using function `X448`.

```
#include<faz_ecdh_avx2.h>
#include<faz_fp_avx2.h> /* For print_bytes */
#include<stdio.h>

void alice_shared(
   argECDHX alice_shared,
   argECDHX bob_session,
   argECDHX alice_private
```

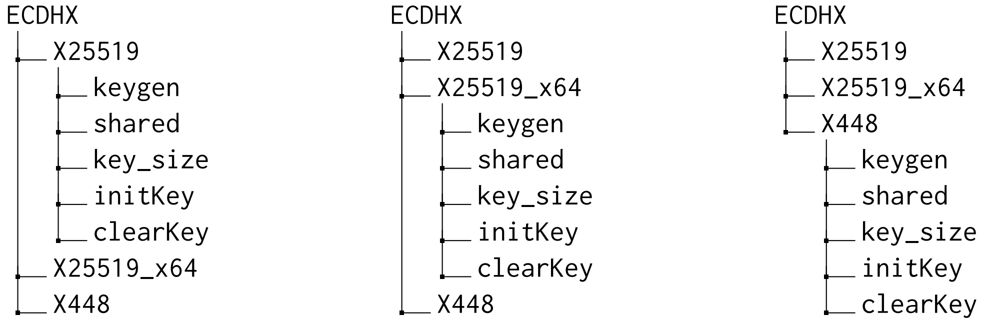


Figure 2: Global variable ECDHX containing pointer to functions X25519 and X448.

```
)
{
  size_t keysize = ECDH448_KEY_SIZE_BYTES;
  ECDHX.X448.shared(alice_shared,bob_session,alice_private);
  printf("Alice shared secret: \n");
  print_bytes(alice_shared,keysize);
}
```

5.3 EdDSA

This section describes the third software layer of the library; such layer corresponds to the signature generation using the EdDSA algorithm. To have access to the definitions provided by this layer include the header file:

```
#include<faz_eddsa_avx2.h>
```

5.3.1 Constants

These are some constants defined for EdDSA algorithm.

- ED25519_KEY_SIZE_BYTES_PARAM. This is the size of a public or private key on the Ed25519 signature scheme. This value is set to 32 bytes.
- ED25519_SIG_SIZE_BYTES_PARAM. This is the size of an Ed25519 signature. This value is set to 64 bytes.
- ED448_KEY_SIZE_BYTES_PARAM. This is the size of a public or private key on the Ed448 signature scheme. This value is set to 57 bytes.
- ED448_SIG_SIZE_BYTES_PARAM. This is the size of an Ed448 signature. This value is set to 114 bytes.
- EDDSA_NOCONTEXT. This flag is used as a parameter to indicate to signature the use of no-context, i.e. the use of a zero length context associated to a message.

5.3.2 Data Types

These are the data types provided by this software layer:

- Ed25519_PrivateKey. Ed25519 private key variable
- Ed25519_PublicKey. Ed25519 public key variable.
- Ed25519_Signature. Ed25519 signature variable.
- Ed448_PrivateKey. Ed448 private key variable.
- Ed448_PublicKey. Ed448 public key variable.
- Ed448_Signature. Ed448 signature variable.
- argEdDSA_PrivateKey. Pointer to a private key.
- argEdDSA_PublicKey. Pointer to a public key.
- argEdDSA_Signature. Pointer to a signature.

5.3.3 Operations

These are the functions to compute digital signatures.

- `keygen` . Creates a public key from a private key.
- `sign` . Computes the signature of a message.
- `verify` . Verifies whether the tuple (message, context, signature, public key) are valid.
- `key_size` . Returns the size of keys.
- `signature_size` . Returns the size of signatures.
- `initKey` . Allocates dynamic memory for storing a key.
- `initSignature` . Allocates dynamic memory for storing a signature.
- `clearKey` . Deallocates dynamic memory of a key.
- `clearSignature` . Deallocates dynamic memory of a signature.

The following are some error codes that the previous operations could return. These flags are enumerated in the `EDDSA_FLAGS` variable.

- `EDDSA_KEYGEN_OK`. Key generation was computed successfully.
- `EDDSA_SIGNATURE_OK`. Signature generation was computed successfully.
- `EDDSA_VERIFICATION_OK`. Signature verification indicates that signature is valid for that message.
- `EDDSA_INVALID_SIGNATURE`. Signature verification indicates that signature is invalid for that message.
- `EDDSA_ERROR_PUBLICKEY`. Signature verification indicates that public key is not in the appropriate format.
- `EDDSA_ERROR_CONTEXT`. Signature verification indicates that context was not provided in the appropriate format.
- `EDDSA_ERROR_PHFLAG`. Indicates that phflag is different from 0 or 1.

5.3.4 Usage of EdDSA

To access to the EdDSA functions, our library provides a global variable called EdDSA, which is a structure containing pointers to functions defined above. The Figure 3 shows the structure of this variable.

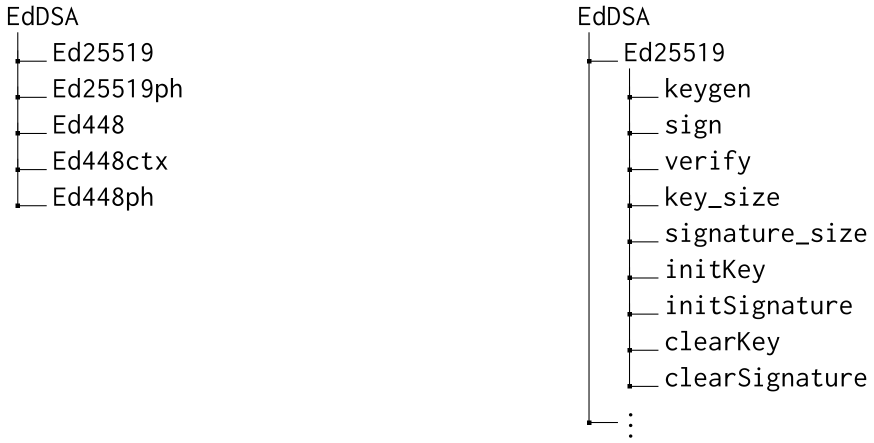


Figure 3: Global variable EdDSA containing pointer to functions Ed25519 and Ed448.

Example 5.3. This is an example of the calculation of a signature of the message “Keep Calm and Carry On” using the Ed25519 signature scheme.

```c
#include<faz_eddsa_avx2.h>
#include<faz_fp_avx2.h> /* For printing keys */
#include<stdio.h>       /* For printf */
#include<string.h>      /* For strlen */

void signature(argECDHX alice_public, argECDHX alice_private)
{
  size_t size_sig = ED25519_SIG_SIZE_BYTES_PARAM;
  Ed25519_Signature signature;

  uint8_t * message = (uint8_t *)"Keep Calm and Carry On";
  const size_t size_msg = strlen((const char *)message);

  EdDSA.Ed25519.sign(signature,
                     message, size_msg,
                     alice_public, alice_private);

  printf("Signature: \n");
  print_bytes(signature,size_sig);
}
```

5.4 Compiling Source Code Documentation

Inside the folder eddsa_avx2/doc, there is a configuration file for running doxygen toolkit; which is used to generate source code documentation.

```
$ cd eddsa_avx2/doc
$ doxygen gen_doc.doxyfile
```

Once command ends, a new folder will be created eddsa_avx2/doc/html, this is a HTML machine-readable documentation of the source code, and can be visualized on an HTML browser. For example on Firefox:

```
$ cd eddsa_avx2/doc
$ firefox html/index.html
```

6 Terms and Conditions

This library is released under the: GNU LESSER GENERAL PUBLIC LICENSE Version 3, 29 June 2007

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below.

0. Additional Definitions.

   As used herein, "this License" refers to version 3 of the GNU Lesser General Public License, and the "GNU GPL" refers to version 3 of the GNU General Public License.

   "The Library" refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

   An "Application" is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

   A "Combined Work" is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the "Linked Version".

   The "Minimal Corresponding Source" for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

   The "Corresponding Application Code" for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work.

1. Exception to Section 3 of the GNU GPL.

   You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL.

2. Conveying Modified Versions.

   If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

   (a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or

   (b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.

The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

(a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License.

(b) Accompany the object code with a copy of the GNU GPL and this license document.

4. Combined Works.

   You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

   (a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License.

   (b) Accompany the Combined Work with a copy of the GNU GPL and this license document.

   (c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document.

   (d) Do one of the following:

      0) Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.

      1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version.

   (e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries.

   You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

   (a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License.

(b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License.

   The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy's public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.