libENI Documentation

Release docs

Sep 27, 2018

Contents

Official C++ implementation of libENI, which is part of the Lity project.

CHAPTER 1

Contents

1.1 Getting Started

1.1.1 Download the Prebuilt libENI

See libENI releases for the latest release.

- *libeni-dev*: for ENI operation developers
- *libeni*: for general ENI users.

Releases

Version	libeni-dev	libeni
v1.3.3	Ubuntu 16.04, CentOS 7	Ubuntu 16.04, CentOS 7
v1.2.x	Ubuntu 16.04, CentOS 7	Ubuntu 16.04, CentOS 7
v1.2.0	Ubuntu 16.04, CentOS 7	Ubuntu 16.04, CentOS 7

Prerequisites

libeni-dev	libeni
• Boost >= 1.58 • OpenSSL >= 1.0.2	• OpenSSL >= 1.0.2

See Prerequisites for platform specific prerequisites guide.

Install

```
tar zxvf libeni.tgz --strip-components 1 -C ${LIBENI_PATH}
```

Validate the Shared Libraries

```
cd ${LIBENI_PATH}/lib
sha512sum -c *.so.sha512
```

You should get a list of OKs if all libraries are good.

```
eni_caesar_cipher.so: OK
eni_crypto.so: OK
eni_reverse.so: OK
eni_scrypt.so: OK
```

Test Manually

See Testing Prebuilt ENI Operations for how to test the prebuilt shared libraries of ENI operations.

1.1.2 Build From Source

Prerequisites

- Boost >= 1.58
- CMake >= 3.1
- OpenSSL >= 1.0.2
- SkyPat >= 3.1.1 (see SkyPat releases)

Download Source Code

```
git clone https://github.com/CyberMiles/libeni.git ${LIBENI_PATH}
```

Build with CMake

```
cd ${LIBENI_PATH}
mkdir build
cd build
cmake ..
make
```

Run Tests

In your build directory, run `ctest`. The result looks like the below.

```
Test project ${LIBENI_PATH}/build
    Start 1: crypto_unittests
1/7 Test #1: crypto_unittests .................   Passed    0.02 sec
    Start 2: t0000-smoke
2/7 Test #2: t0000-smoke ......................   Passed    0.01 sec
    Start 3: t0001-testlib
3/7 Test #3: t0001-testlib ....................   Passed    0.03 sec
    Start 4: t0002-examples-eni-reverse
4/7 Test #4: t0002-examples-eni-reverse .......   Passed    0.02 sec
    Start 5: t0003-examples-eni-caesar
5/7 Test #5: t0003-examples-eni-caesar ........   Passed    0.02 sec
    Start 6: t0004-tools-eni-crypto
6/7 Test #6: t0004-tools-eni-crypto ...........   Passed    0.07 sec
    Start 7: unittests
7/7 Test #7: unittests ........................   Passed    0.01 sec

100% tests passed, 0 tests failed out of 7

Label Time Summary:
auto          =   0.15 sec*proc (5 tests)
regression    =   0.15 sec*proc (5 tests)
unittest      =   0.03 sec*proc (2 tests)

Total Test time (real) =   0.19 sec
```

See *Testing/Temporary/LastTest.log* for the detailed output of all tests.

1.2 Developer Guide

In this tutorial, we will guide you through how to create new ENI operations with libENI in C++.

1.2.1 Prerequisites

In order to build your ENI operations, you need to install *libeni-dev* first.

See *Getting Started* for more information.

Implement an ENI Operation

Here, we use *examples/eni/reverse* as an example. In this example, we will create an ENI operation called *reverse* that takes a string, and returns the reversed string.

The below code piece shows how developers use this ENI operation when writing a contract in Solidity.

```
string memory reversed;
reversed = eni("reverse", "The string to be reversed.");
```

1.2.2 Subclass `EniBase`

In order to implement an ENI operation, you need to `#include <eni.h>`, create a subclass of `eni::EniBase`, and implement the following functions.

0. A constructor that takes a string as its parameter. Remember to pass the string to the constructor of the superclass, `eni::EniBase`, which will convert the raw string into a `json::Array` containing the arguments for your ENI operation.
1. A destructor.
2. Three pure virtual functions, which should be implement privately.
    - `parse` to parse the arguments.
    - `gas` to calculate gas consumption from the arguments.
    - `run` to execute your ENI operation with the arguments.

```
#include <eni.h>
class Reverse : public eni::EniBase {
public:
  Reverse(const std::string& pArgStr)
    : eni::EniBase(pArgStr) { ... }

  ~Reverse() { ... }

private:
  bool parse(const json::Array& pArgs) override { ... }

  eni::Gas gas() const override { ... }

  bool run(json::Array& pRetVal) override { ... }
};
```

The `parse` function takes a `json::Array` containing the arguments given to your ENI operation. To ensure the other two functions `gas` and `run` process the arguments in the same way, please validate, preprocess, and store the arguments into member variables in the `parse` function.

The `parse` function should return `true` when all arguments are good, and return `false` otherwise. (i.e. when the given arguments are not correct, e.g., lacking arguments, or wrong type).

In this example, the `json::Array` constructed by `eni::EniBase` contains only the argument string for ENI operation *reverse*.

```
["The string to be reversed."]
```

Here we just take the first argument and convert it to a string.

```
class Reverse : public eni::EniBase {
  ...
private:
  bool parse(const json::Array& pArgs) override {
    m_Str = pArgs[0].toString();
    return true;
  }

  std::string m_Str;
};
```

Check the documentation to see more detail about how arguments are converted into a `json::Array`.

Before your ENI operation is run, you need to estimate how much gas it will cost. Override the pure virtual function `gas`, and return your estimated gas cost.

In this example, we use the string length as its gas consumption.

```
class Reverse : public eni::EniBase {
  ...
private:
  eni::Gas gas() const override {
    return m_Str.length();
  }
};
```

Return `0` when error occurs (e.g., gas is incalculable).

Override the pure virtual function `run`, and push the result of your ENI operation back into the `json::Array`.

```
class Reverse : public eni::EniBase {
  ...
private:
  bool run(json::Array& pRetVal) override {
    std::string ret(m_Str.rbegin(), m_Str.rend());
    pRetVal.emplace_back(ret);
    return true;
  }
};
```

Return `true` only when your ENI operation is successfully executed.

1.2.3 Export the ENI Operation with C Interface

Your ENI operation will be called via its C interface, so be sure to export the C interface with `ENI_C_INTERFACE(OP, CLASS)`, where *OP* is your ENI operation name (i.e., *reverse* in this example), and *CLASS* is the name of implemented class (i.e., `Reverse` in this example).

```
ENI_C_INTERFACE(reverse, Reverse)
```

Build the ENI Operations Into a Shared Library

Please add these flags `-std=c++11 -fPIC` when compiling your ENI operation into a shared library. See GCC Option Summary for explanation to these flags.

Specify the path to libENI headers with `-I${LIBENI_PATH}/include`.

You might also want to link to libENI by specifying the path `-L${LIBENI_PATH}/lib`, and the library name `-leni`.

Here is an example Makefile for *examples/eni/reverse*. Please be aware that the flags and commands might differ if you're using different compilers.

```
CPPFLAGS=-I${LIBENI_PATH}/include
CXXFLAGS=-std=c++11 -fPIC
LDFLAGS=-L${LIBENI_PATH}/lib
LDADD=-leni

all:
	g++ ${CPPFLAGS} ${CXXFLAGS} ${LDFLAGS} -shared -oeni_reverse.so eni_reverse.cpp
↪${LDADD}
```

Test Your ENI Operations

1.2.4 Test From `EniBase` Interface

Your ENI operations will only be accessed from the two public member functions of `eni::EniBase`.

- `Gas getGas()` should return the gas cost of your ENI operation.
- `char* start()` should run your ENI operation and return the results in JSON format.

You may test your subclass through these two public functions.

```
eni::EniBase* functor = new Reverse("[\"Hello World\"]");
ASSERT_NE(functor, nullptr);
EXPECT_EQ(functor->getGas(), 12);
char* ret = functor->start();
EXPECT_EQ(::strcmp(ret, "[\"!dlroW olleH\"]"), 0);
free(ret);
delete functor;
```

1.2.5 Test From Shared Library Interface

See *the documentation* for how to test the shared libraries of your ENI operations.

Testing ENI Operations

Setup Environment

Make sure libENI can be found in your environment. See *Getting Started* for how to install libENI.

You might want to try the following settings if libENI is installed but not found in your environment.

```
PATH=${PATH}:${LIBENI_PATH}/bin
LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:${LIBENI_PATH}/lib
```

Tools for Testing

Simulate ENI Operations

We provide two CLI tools *eni_gas* and *eni_run*, which simulate how the ENI operations are called in a shared library from EVM.

Check the usage of the tools by running it without any arguments.

```
NAME
  eni_gas - the estimated gas cost when running the ENI operation
  eni_run - run the ENI operation and print the result in JSON format

SYNOPSIS
  eni_gas <LIBPATH> <OP> <JSON>
  eni_run <LIBPATH> <OP> <JSON>

DESCRIPTION
```

(continues on next page)

(continued from previous page)

```
  LIBPATH - path to the shared library
  OP      - name of the ENI operation
  JSON    - parameters passed to the operation in JSON format
```

Test Prebuilt Shared Libraries

Here we demonstrate how to test the ENI operation built from our examples and tools with the CLI tools *eni_gas* and *eni_run*.

Test "eni_reverse.so" (*examples/eni/reverse*)

```
eni_gas ${LIBENI_PATH}/lib/eni_reverse.so reverse '["Hello World!"]'
eni_run ${LIBENI_PATH}/lib/eni_reverse.so reverse '["Hello World!"]'
```

The first call *eni_gas* will output *12*, and the second one *eni_run* will output *["!dlroW olleH"]*.

Test "eni_caesar_cipher.so" (*examples/eni/caesar_cipher*)

```
eni_gas ${LIBENI_PATH}/lib/eni_caesar_cipher.so caesar_encrypt '["Hello World!",
↪"HELLOGCC"]'
eni_run ${LIBENI_PATH}/lib/eni_caesar_cipher.so caesar_encrypt '["Hello World!",
↪"HELLOGCC"]'
eni_gas ${LIBENI_PATH}/lib/eni_caesar_cipher.so caesar_decrypt '["Olssv Dvysk!",
↪"HELLOGCC"]'
eni_run ${LIBENI_PATH}/lib/eni_caesar_cipher.so caesar_decrypt '["Olssv Dvysk!",
↪"HELLOGCC"]'
```

Output for the above lines.

```
12
["Olssv Dvysk!"]
12
["Hello World!"]
```

Test "eni_crypto.so" (*tools/eni_crypto*)

Get the JSON files for testing from the directory test/t0004.

```
eni_gas ${LIBENI_PATH}/lib/eni_crypto.so rsa_encrypt $(cat pub_encrypt.json)
eni_run ${LIBENI_PATH}/lib/eni_crypto.so rsa_encrypt $(cat pub_encrypt.json)
eni_gas ${LIBENI_PATH}/lib/eni_crypto.so rsa_decrypt $(cat priv_decrypt.json)
eni_run ${LIBENI_PATH}/lib/eni_crypto.so rsa_decrypt $(cat priv_decrypt.json)
```

Output for the above lines.

```
12
["An encrypted hex-string that is 512 characters in length."]
256
["Hello World!"]
```

Consensus Test

The tool *consensus.py* aims to ensure that an ENI operation always returns the same result when given the same input. This is crucial for ENI to work on blockchain. Otherwise it will break the consensus between nodes.

Prerequisites

Besides the prerequisites of libENI, *Python 3* is also required to run *consensus.py*.

Usage

Check the usage of *consensus.py* by running it with the argument `-h`.

```
usage: consensus.py [-h] TEST_LIST

libENI Consensus Test

positional arguments:
  TEST_LIST   JSON description file for list of tests

optional arguments:
  -h, --help  show this help message and exit
```

Format of JSON Description File

The format of the input JSON description file looks like this.

```
{
    "path": [
        ["op", "type:arg", ...],
        ...
    ]
}
```

Argument	Description
path	**Absolute** path to the shared library.
op	Name of the ENI operation.
type:arg	Format of parameters passed to the ENI operation. String: a random string of the given length. • *type*: `s` • *arg*: length of string File: a string containing the file content. • *type*: `f` • *arg*: path to the file Only specific string-type shown above are supported for now.

Here is an example JSON file in libENI's own tests.

```
{
    "/home/libeni/build/examples/eni/eni_reverse.so": [
        ["reverse", "s:32"]
    ],
    "/home/libeni/build/examples/eni/eni_caesar_cipher.so": [
        ["caesar_encrypt", "s:32", "s:16"],
        ["caesar_decrypt", "s:32", "s:16"]
    ],
    "/home/libeni/build/tools/eni_crypto/eni_crypto.so": [
        ["rsa_encrypt", "f:data/pub.pem",  "s:17"],
        ["rsa_decrypt", "f:data/priv.pem", "f:data/rsa"]
    ]
}
```

Run Consensus Tests

The *consensus.py* will run each ENI operation specified in the JSON file several times (3 by default), and check if it returns the same output for the same input.

```
consensus.py tests.json
```

The output of the consensus tests will look like this (for the above example).

```
Case #0: /home/libeni/build/tools/eni_crypto/eni_crypto.so rsa_encrypt
PARAMS: ["-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEApr/atzUuBArbsWHkn8tU
mq00tCV4CcLfVUVg+zr41sixYIb54rd5EFWoQ1xecYMZIbnoTl2vu9awHBZF18Dk
NlG8pjw1Vw5EjaWrCDn734lcKGhsYe20H7138XgznwhJNuAeNy2OOwbEIkSl4jf8
Ro+7nOuMK5yeHXAPZEMCnEipd/7gGk0aWP/E1XoqkZJnBUYN5N6mOgtV3jv62w+X
lNlozUySI0mBmjgyymAhPm4qx5Zcf/Wmg42vbIFRrBl5PgWKGsY0L7xcPRDPAPRt
ndUPr+CLLk5KjyHI8a2WiYrJvjUGTQNyqPM5MmLLfHMkjkbE6DshWbMZona+/5ji
3wIDAQAB
-----END PUBLIC KEY-----","4POB4HDNKUXEGP2U3"]
STDOUT: [
↪"0e8cb1615bf5802a7d4e4e6e09a802bc8db648b8cbff70a3321e11275491dd54cc2d1b3e130cecb8fac37cb047b137ea65
↪"]

Case #1: /home/libeni/build/tools/eni_crypto/eni_crypto.so rsa_decrypt
PARAMS: ["-----BEGIN RSA PRIVATE KEY-----
MIIEowIBAAKCAQEApr/atzUuBArbsWHkn8tUmq00tCV4CcLfVUVg+zr41sixYIb5
4rd5EFWoQ1xecYMZIbnoTl2vu9awHBZF18DkNlG8pjw1Vw5EjaWrCDn734lcKGhs
Ye20H7138XgznwhJNuAeNy2OOwbEIkSl4jf8Ro+7nOuMK5yeHXAPZEMCnEipd/7g
Gk0aWP/E1XoqkZJnBUYN5N6mOgtV3jv62w+XlNlozUySI0mBmjgyymAhPm4qx5Zc
f/Wmg42vbIFRrBl5PgWKGsY0L7xcPRDPAPRtndUPr+CLLk5KjyHI8a2WiYrJvjUG
TQNyqPM5MmLLfHMkjkbE6DshWbMZona+/5ji3wIDAQABAoIBAAjwNdAmSJ4s2tPq
VHAAXTuhVzbk30deq8wNWQJ+icIxpdhvw8tUXGf0v31E4UciaOF27q3stbPS8UPA
KeRD0bfbr8oVZiKRgDk7jSx2tzqnSUpdNpoVPNeKt3g5IkM/FXWck+IPThV56l+P
4Hh82cgKglsKAUyBK7SWQiz0rpoj8MWlkG0TblsMVLnOTAO0N3p3NiHxv1eUJrHK
wyI42Mkb+nUm1jKSUAg9JuOQJUWcKzlgs8Z4+gvvOukO1tTs4EBMZdn2wYC0+BSE
qB0Sx496fuIZ0YPExwF21h2bansEuG2kN5OnW80vnUT724bGvGv3yffyK3fZhE2M
WdwDJtkCgYEA3Vw/O8cRxSv6gU4bWH6YE24XQz/pRvOsLLcQeXrxbxvm1ZsD65ou
tpvA0/eF3c5KRAhoqgRGPDV7eHvRdo9v6Ih4mwp6wR9bEGU3beHCIjZPb5nCCGtk
TCNiVt+MIXKBHXt9lKBjTnmbCvRt+chRz8yFwRpdu49GawOX6NY8YasCgYEAwNfh
TbTRawCqC1WY1weFIn9y9NN1reFVvYZhb6stdfBUVAG6vjB0dGG0bZCJUH/+h65/
AuvahIugw5AA+H8iTTeB2KpgCc2FmiUviohug39GMz6oabkzZH9KAZjCf5/zMhm3
```

(continues on next page)

(continued from previous page)

```
IvtVDMDXBJah7SFYsxM1sBfklPAHFlAe7zP/950CgYBM60IZzonRPv/0MKT18j97
+PRibPHtsrywaQhzfhIpLsPek9gf5Vq4H5U40rkUoxtRWq6r7YJOZ7M44aWekicr
4Ugvb8vKEdA9+T3yk9E2vDKMMBypek/G2UDRuSpjcPuGuCOiIr1/RmhmvRr+AerT
z1jnCfdqNlYc14nQ4ajnswKBgDtlAj6lt25mePketwFbjpTOfkCLtI4Gfhrufaog
JdNSXxa0paiYUfXadfDc+H3zvhhaFUJ4FAiI3M3+112yAoWX2AU8jHHYfBK660aW
uLsFg0CbRtGxOfP1BH0zaIxYXlYK943trQdNiawfHOZlQ+V7wChpY3y/5N7pdG2m
LWs9AoGBAMEgKXwA2ubWrxe622PHXwgUx9oja3LwmuT3oQZDtwxfs4lw3xzIgGps
WVvgNL2aceE/qkI032ysKTIbM3JvKa7AzrGKDi8XbyE98QSKM9qyFmdrTG7UIbSo
DNen8V4qgCV/z34+6uxWMR7AozgQmzrKogmxhZpIYdyqO4F35cMb
-----END RSA PRIVATE KEY-----",
↪"65b4474b010b1992cfa93a57238be244248dd22060b2fe7f65791b9aecbd1086ff05a1e47977766646a7c2aac3550e2cea
↪"]
STDOUT: ["Hello World!"]

Case #2: /home/libeni/build/examples/eni/eni_caesar_cipher.so caesar_encrypt
PARAMS: ["Y54WGC2FXDQBXQHXYS7O0XTSLNYC1P7L","JJCENMR2BJ89MEE0"]
STDOUT: ["H54FPL2OGMZKGZQGHB7X0GCBUWHL1Y7U"]

Case #3: /home/libeni/build/examples/eni/eni_caesar_cipher.so caesar_decrypt
PARAMS: ["0QQGCGXT6ZWR2C57AR2T9IRRL94LY80J","FURNHPTQVQYRYZZS"]
STDOUT: ["0LLBXBSO6URM2X57VM2O9DMMG94GT80E"]

Case #4: /home/libeni/build/examples/eni/eni_reverse.so reverse
PARAMS: ["OQ837FGSXH558HEV2H3AZOAI9FSGDUZP"]
STDOUT: ["PZUDGSF9IAOZA3H2VEH855HXSGF738QO"]
```

1.3 Documentation

1.3.1 Types

ENI Types

These types are provided to be coherent with primitive types of Lity (Solidity).

ENI Integers

All integer types in ENI is implemented using boost::multiprecision::number. Some of them are aliases for types predefined in boost::multiprecision::cpp_int.

Integer Type	Size (bits)	Signed	Note
`eni::s256`	256		Alias for `boost::multiprecision::int256_t`.
`eni::Int`	256		Alias for `eni::s256`.
`eni::u256`	256		Alias for `boost::multiprecision::uint256_t`.
`eni::u160`	160		Size of an Ethereum address.
`eni::u128`	128		Alias for `boost::multiprecision::uint128_t`.
`eni::u64`	64		
`eni::UInt`	256		Alias for `eni::u256`.

Operations on ENI Integers

See the documentation for boost::multiprecision::number for supported operations.

Suggested Use of ENI Integers

This section does not exist yet. *(´-ι_-`)*

ENI Boolean

`eni::Bool` is an alias for C++ `bool`.

ENI Address

`eni::Address` is an alias for `eni::u160` (20 bytes, size of an Ethereum address).

Convert ENI Types to C++ String

`eni::to_string` uses boost::lexical_cast internally to convert ENI types to `std::string`.

All ENI integers, `eni::Bool`, `eni::Address` are supported.

```
std::string to_string(eni::TypeName);
```

Usage

```
eni::Int int32max(2147483647);
std::string s = eni::to_string(int32max); // "2147483647"

eni::Bool bTrue(true);
std::string t = eni::to_string(bTrue);    // "true"
```

Abstract Data Types

See `eni::Vector` and `eni::StringMap`.

This section does not exist yet. *(´-ι_-`)*

JSON Types

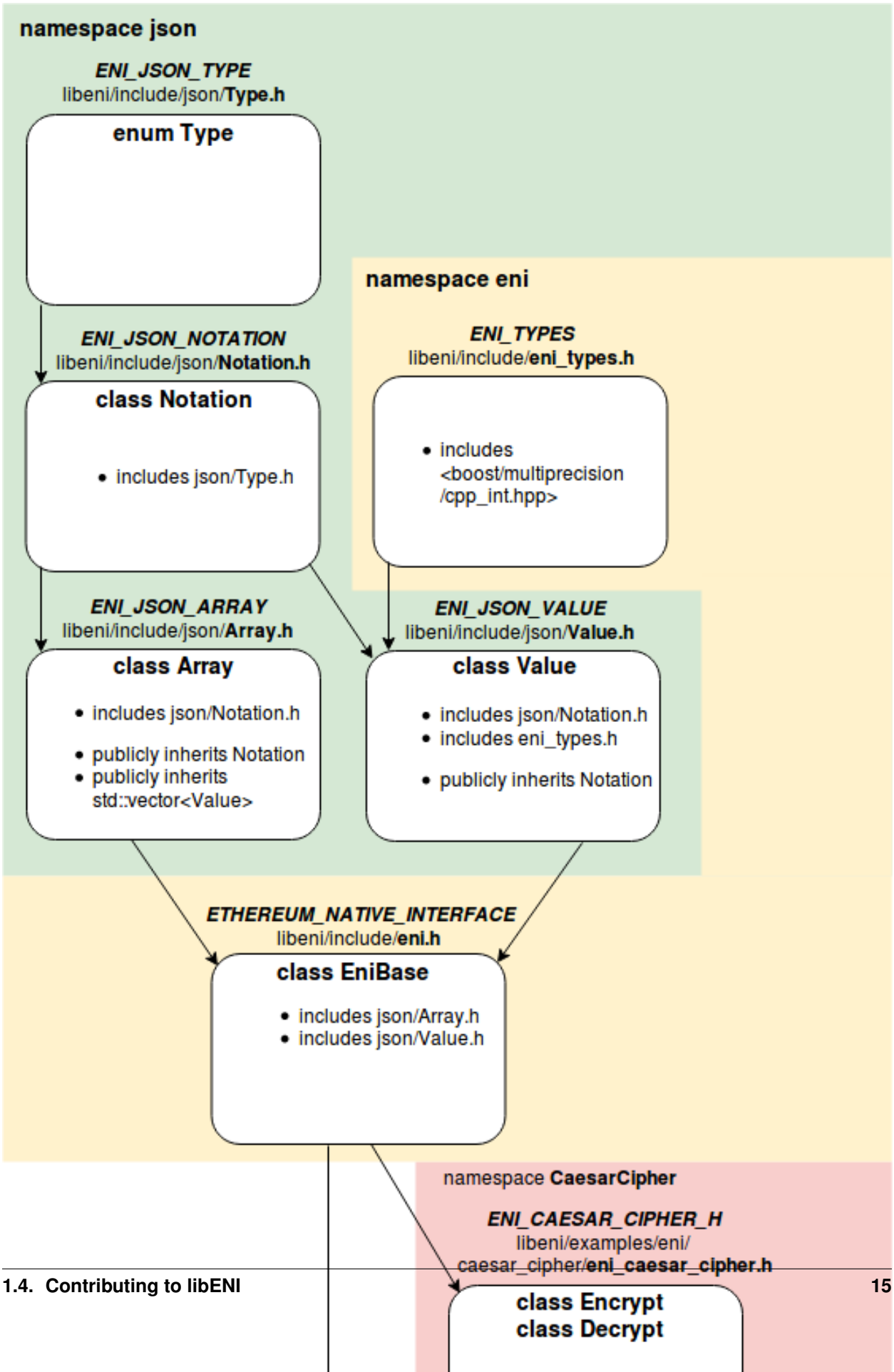
This section does not exist yet. *(´-ι_-`)*

1.4 Contributing to libENI

See *Getting Started* for how to build libENI.

1.4.1 Overview of libENI Code

The following image illustrates the libENI code hierarchy and the use of namespaces.


namespace json
ENI_JSON_TYPE
libeni/include/json/Type.h
enum Type
namespace eni
ENI_JSON_NOTATION
libeni/include/json/Notation.h
class Notation
includes json/Type.h
ENI_TYPES
libeni/include/eni_types.h
includes <boost/multiprecision /cpp_int.hpp>
ENI_JSON_ARRAY
libeni/include/json/Array.h
class Array
includes json/Notation.h
publicly inherits Notation
publicly inherits std::vector<Value>
ENI_JSON_VALUE
libeni/include/json/Value.h
class Value
includes json/Notation.h
includes eni_types.h
publicly inherits Notation
ETHEREUM_NATIVE_INTERFACE
libeni/include/eni.h
class EniBase
includes json/Array.h
includes json/Value.h
namespace CaesarCipher
ENI_CAESAR_CIPHER_H
libeni/examples/eni/ caesar_cipher/eni_caesar_cipher.h
class Encrypt
class Decrypt

Directory Structure

Path	Description
docs/	Documentations.
examples/	Examples of how to use libENI.
include/	Header files for libENI.
lib/	Implementations for libENI.
test/	All tests for libENI and its examples.
tools/	Tools and modules for libENI.

1.4.2 Report an Issue

Please provide the following information as much as possible.

- The version (commit-ish) your using.
- Your platform, environment setup, etc.
- Steps to reproduce the issue.
- Your expected result of the issue.
- Current result of the issue.

1.4.3 Create a Pull Request

- Fork from the *master* branch.
- Avoid to create merge commits when you update from *master* by using `git rebase` or `git pull --rebase` (instead of `git merge`).
- Add test cases for your pull request if you're proposing new features or major bug fixes.
- Build and test locally before submit your pull request. See *Getting Started* for how to test libENI.

Please try to follow the existing coding style of libENI (although it is neither well-styled nor documented at this moment), which is basically based on LLVM coding standards.