Technical assessment of *herumi* libraries

The *mcl* and *bls* libraries

Ref.	20-07-732-REP
Version	1.2
Date	November 13th, 2020
Made for	The Ethereum Foundation
Conducted by	Quarkslab

Quarkslab SAS
13 rue Saint Ambroise
75011 Paris, France

Contents

1. Project Information

Document History			
Version	**Date**	**Details**	**Authors**
1.2	2020-11-27	Final version	Laurent Grémy & Christian Heitman
1.1	2020-11-13	Draft version (close to the final version)	Laurent Grémy & Christian Heitman
1.0	2020-10-23	Draft version	Laurent Grémy & Christian Heitman

Quarkslab		
Contact	**Position**	**E-mail address**
Frédéric Raynal	Quarkslab CEO	`fraynal@quarkslab.com`
Matthieu Duez	Service Manager	`mduez@quarkslab.com`
Laurent Grémy	R&D Engineer	`lgremy@quarkslab.com`
Christian Heitman	R&D Engineer	`cheitman@quarkslab.com`

The Ethereum Foundation		
Contact	**Position**	**E-mail address**
Kirk Baird	Software Engineer at Sigma Prime for the development of lighthouse and Ethereum 2.0 client.	`kirk@sigmaprime.io`
Carl Beekhuizen	Ethereum 2.0 Researcher at the Ethereum Foundation	`carl.beekhuizen@ethereum.org`
Mamy Ratsimbazafy	Blockchain scalability and Ethereum Researcher at Status	`mamy@status.im`
Danny Ryan	Ethereum 2.0 Researcher at the Ethereum Foundation	`danny@ethereum.org`

2. Executive Summary

2.1 Context

The Ethereum Foundation is currently launching the 2.0 version of the Ethereum blockchain, see the Ethereum 2.0 website. One of the modifications with this new iteration of the blockchain is the usage of a proof of stake algorithm instead of a proof of work algorithm in order to reach the consensus to append a new block to the blockchain. This new consensus algorithm is based at some point on votes and commitments which must be signed. To require less storage capacities, these different signatures may at some point be aggregated. In order to aggregate signatures, the choice of the Ethereum 2.0 designers was to use BLS signatures [BLSsig] [BLSsigRFC].

BLS signatures are defined over a pairing, which can be viewed at a high-level point as a map e from two groups $(\mathbb{G}_1, +)$ and $(\mathbb{G}_2, +)$ of prime order r to a group $(\mathbb{G}_T, \cdot)$ of order r. Upon the family of pairing-friendly curves that are proposed in cryptography to match efficient computations and a reasonable security level, a BLS curve [BLScurve] denoted by *BLS12_381* in [Pairing] was chosen. To avoid the confusion between the two BLS acronyms, even if sometimes the context will help distinguish the two, we will use the term **BLS12_381** to speak about the standardized BLS curve chosen to be the pairing-friendly curve used by the Ethereum 2.0 blockchain, and **BLSsig** when referring to the signature algorithm or a cryptographic primitive related to this algorithm.

Since signature plays an important role in the consensus process, it is important to review the implementation of the primitives needed by Ethereum 2.0 and built upon BLSsig. This report will focus on implementations provided in the herumi project.

2.2 Methodology

The audit has been divided into three main stages, which were:

- **Stage 1**: focused on the architecture of the code, the interaction between the *bls* and the *mcl* libraries, the different backends.
- **Stage 2**: focused on identifying the different relevant parts in the specifications, especially the RFCs.
- **Stage 3**: focused on the code itself, its design, its adherence with the specifications.

2.3 Chronology

The audit was performed by two security engineers for a total of 68 man-days between the 16th of September and the 13th of November. Some details about the chronology are provided below:

- July 3rd, 2020: quote sent in reply to the RFP.
- September 16th, 2020: beginning of the audit and call with the Ethereum Foundation, setup of the communication channel gathering the different interlocutors.
- October 2nd, 2020: one issue and one pull request submitted to the `mcl` project.
- October 23th, 2020: intermediate draft (version 1.0) sent to the Ethereum Foundation.
- November 4th, 2020: two issues submitted to the `bls` project.
- November 13th, 2020: second draft (version 1.1) sent to the Ethereum Foundation.

- November 27th, 2020: final version (version 1.2) sent to the Ethereum Foundation.

2.3.1 Communication channel

The Ethereum Foundation established a Telegram channel gathering the author of the `herumi` librairies Shigeo Mitsunari, the Ethereum Foundation contacts and the two Quarkslab auditors.

2.3.2 Deliverables

The deliverables are of two types:

- three issues and one pull request submitted to both the *bls* and *mcl* projects: Issue 85, Pull request 86, Issue 66 and Issue 67;
- the present report.

2.4 Report synthesis

2.4.1 Synthesis

Among all the *herumi* repositories listed in https://github.com/herumi which may be used to implement an Ethereum 2.0 client, two projects are of main interest:

- https://github.com/herumi/mcl: portable and fast pairing-based cryptography library (BSD-3-Clause license).
- https://github.com/herumi/bls: implementation of BLS threshold signature, which supports the BLS Signatures specified at Ethereum 2.0 Phase 0 (BSD-3-Clause license).

They are indeed the building blocks of the Go and Rust bindings for example. The Go and Rust bindings were also reviewed to better understand the use of some functions of the core projects.

2.4.2 Findings and recommendations

Some of our findings were submitted as GitHub issues or pull requests. We provide here a summary of the four most significant findings and recommendations, the exhaustive ones are available in Section 6.

A first recommendation is to document both libraries in detail (as well as to comment the code base and explain the design and technical decisions made), see Section 6.1. Indeed for now, there is little to no documentation of the projects. This will not only help significantly the end user at the time of building and using the library, but also will help in the maintainability and further development of both libraries (as well as future audits).

The three other recommendations are about the *mcl* library. They are focused on the back ends of the *mcl* library, especially their interface in Section 6.2, their interaction in Section 6.3 and their configuration in Section 6.4.

As explained in *Findings and Recommendations*, these do not represent an immediate risk in terms of security. However, they **do** degrade the overall reliability and correctness of the libraries and should be addressed promptly, before integrating them into a production-like context. Due to them, it is possible subtle and important bugs go unnoticed.

To conclude, we believe that tackling these aspects, specially providing a clear, well-define and common interface to the back ends, will substantially improve the quality of the library and make it more suitable for code maintainability as well as potential future review.

3. Code overview

3.1 Audited versions

We list below the audited versions of the two projects listed in Section 2.4.1:

- *mcl* library: commit *dc65ed2ccc413d6236b30b627e4e34050f04a5fe* (version 1.23);
- *bls* library: commit *e4663751b56f3e55f035bb4b1444d5273c5ae36e*.

For information, the number of lines of C++ code[1] is about:

- 50731 for the *mcl* library;
- 4466 for the *bls* library.

The large majority (50722 lines for *mcl* and 4465 lines for *bls*) of this code corresponds to Shigeo Mitsunari, the maintainer of the libraries.

3.2 The *bls* library

3.2.1 Overview

The *bls* library depends heavily on the *mcl* library. Most of the functions it provides are either a direct call to a function of the *mcl* library or a short combination of them; only a few have a more complex implementation. Therefore, most of our remarks will be done in Section 3.3. Here, we will only comment on the serialization/deserialization process and some general observations.

3.2.2 Serialization/Deserialization

Generally speaking, serialization/deserialization algorithms are problematic functions since malformed, corrupted or even manipulated data can lead to unhandled errors.

As previously mentioned, most of the serialization functions resort directly to the *mcl* serialization functions (which will be reviewed in their corresponding section). However, there are some, such as the uncompressed variations, that are implemented here. The reason of this approach remains unknown but they still rely on the *mcl* library.

According to [BLSsigRFC], the serialization/deserialization process should follow the format specified by ZCash[2].

When analyzing the implementation of the uncompressed variations we noted some issues related to the specification, mostly on the deserialization ones.

Let us analyze the `blsPublicKeyDeserealizeUncompress` function[3], presented below.

[1] Found by using `find . -type f \( -name "*.c" -o -name "*.cpp" -o -name "*.h" -o -name "*.hpp" \) -exec wc -l {} \; | awk '{s+=$1} END {print s}'`.

[2] A more detailed version can be found in Appendix C of [BLSsigRFC].

[3] The body of the function is contained within an `#ifdef` block, which according to the `README` file is enabled for the *ETH2.0* spec. The same remarks apply to the `blsSignatureDeserializeUncompressed` function.

Listing 3.1: `blsPublicKeyDeserealizeUncompress` function (`src/bls_c_impl.hpp#L660-L683`).

```
660 mclSize blsPublicKeyDeserializeUncompressed(blsPublicKey *pub, const void *buf, mclSize bufSize)
661 {
662 #ifdef BLS_ETH
663     if (g_curveType != MCL_BLS12_381) return 0;
664     const mclSize retSize = serializedPublicKeySize * 2;
665     if (bufSize < retSize) return 0;
666     const uint8_t *src = (const uint8_t*)buf;
667     G1& x = *cast(&pub->v);
668     if (isZeroFormat(src, retSize)) {
669         x.clear();
670     } else {
671         if (x.x.deserialize(src, serializedPublicKeySize) == 0) return 0;
672         if (x.y.deserialize(src + serializedPublicKeySize, serializedPublicKeySize) == 0) return 0;
673         x.z = 1;
674     }
675     if (!x.isValid()) return 0;
676     return retSize;
677 #else
678     (void)pub;
679     (void)buf;
680     (void)bufSize;
681     return 0;
682 #endif
683 }
```

The first issue concerns the zero element. According to the specification, the three most significant bits of a *G1* or *G2* encoding are used to provide information about the underlying element. The second most significant bit indicates that the point is at infinity. If this bit is set, the remaining bits of the group element's encoding should be set to zero. This is checked with the function `isZeroFormat` (line 668). The issue here is that a malformed element with the second most significant bit set to one and not all of the remaining bits set to zero will fail this check. In this case, instead of returning `INVALID` as the specification states, it is processed as a non-zero element[4][5]. It is worth noting that after the deserialization there is a validation over the resulting element: `x.isValid` (line 615). As far as we can tell, the function does not consider the mentioned case. Therefore, the deserialization function may be loading an invalid element.

Another issue that we noted is that the function does not check whether the first and third most significant bits are set or not (neither mask them before calling the `deserialize` function at lines 671 and 672). These bits are always set to zero when dealing with a valid (uncompressed) serialized input. This suggests that the function is not prepared to deal with malformed serialized elements.

Some extra remarks that may impact the correct behaviours of the functions:

- The function also assumes that the elements to deserialize are normalized, since it sets the `z` component of the underlying element to `1` (line 673). In case of a malformed element, this approach may lead to an error (assuming it is not detected by the `isValid` function).

[4] Refer to *Item 4* of the Point deserialization procedure section.
[5] This does not happen in the compressed variant of the function as we will see later.

- Both the `serialize` and `deserialize` methods are called using the default value (`IoSerialize`) for its third parameter (`ioMode`). This may lead to errors in the future as it may be changed inadvertently in the *mcl* library (specially, given the fact that it is not clear why it was chosen as the default one).

Important: We recommend adapting these functions to follow closely the specification as well as contemplate the case of malformed inputs.

3.2.3 General observations

mcl library API usage

We noticed that many of the functions in *bls* do not use the API provided by *mcl*. Instead, they are reimplemented in *bls* calling internal functions of *mcl*.

For example, let us compare the `blsSecretKeySetLittleEndian` (Listing 3.2) and `mclBnFr_setLittleEndian` (Listing 3.3) functions:

Listing 3.2: `blsSecretKeySetLittleEndian` function (`src/bls_c_impl.hpp#L185-L189`).

```
int blsSecretKeySetLittleEndian(blsSecretKey *sec, const void *buf, mclSize bufSize)
{
    cast(&sec->v)->setArrayMask((const char *)buf, bufSize);
    return 0;
}
```

Listing 3.3: `mclBnFr_setLittleEndian` function (`mcl/include/mcl/impl/bn_c_impl.hpp#L160-L164`).

```
int mclBnFr_setLittleEndian(mclBnFr *x, const void *buf, mclSize bufSize)
{
    cast(x)->setArrayMask((const char *)buf, bufSize);
    return 0;
}
```

Considering `&sec->v` is referring to `mclBnFr` (Listing 3.4), we can see that both functions are the same in essence.

Listing 3.4: `blsSecretKey struct` (`include/bls/bls.h#L56-L58`).

```c
typedef struct {
    mclBnFr v;
} blsSecretKey;
```

The same pattern repeats in many of the functions of *bls*. The reason behind this approach is not clear. However, it has disadvantages as any change in *mcl* API is not directly reflected on the *bls* side.

Pointer dereference

There is no `null` pointer check in the API provided by the library. Therefore, it assumes that all received pointers are valid ones. Let us consider the `blsPublicKeySerialize` function (Listing 3.5):

Listing 3.5: `blsPublicKeySerialize` function (`src/bls_c_impl.hpp#L364-L367`).

```cpp
mclSize blsPublicKeySerialize(void *buf, mclSize maxBufSize, const blsPublicKey *pub)
{
    return cast(&pub->v)->serialize(buf, maxBufSize);
}
```

The field `v` of the pointer to the public key `pub` is accessed without first checking if it is valid. This pattern repeats across the entire library.

Note: The same pattern repeats in the API of the *mcl* library.

3.3 The *mcl* library

3.3.1 Overview

This library is the cornerstone of *bls*. It is a quite complex library with multiple build modes.

This library allows the user to select among three different back ends that implement the basic operations upon which more complex ones are built. These are: a) *GMP*, b) *LLVM*, and c) *XBYAK*.

The efforts of this code review were focused on the classes, structures and functions used by the *bls* library. More precisely, the classes `FpT` and `EcT` were the starting points in the process of understanding the code base.

In the following sections we will describe how each back end works and implements the functionality needed by *bls*. Along the way, we will make remarks about the code and provide recommendations to improve potential issues.

It is worth noting that there is little to no documentation available. The code base has very few comments as well. Therefore, a lot of time and effort was put into understanding the design and inner workings of the library.

3.3.2 Back ends: Internals

Overview

The back ends mainly provide the operations of `FpT`[6]. As mentioned, there are three different ones, each trying to improve the performance of the previous.

To better understand how each back end is implemented we first need to describe the `FpT` class.

The `FpT` class provides multiple operations such as `add`, `sub`, `neg`, `mul`, `sqr`, etc. The implementation of these operations is separated from the class. They are encapsulated within the `Op` structure. The `FpT` class has a static field `op_` of type `struct Op` and the methods corresponding to the operations are mere wrappers that invoke the proper method of `Op`. For example, the following code snippet represents the `add` method:

Listing 3.6: `FpT::add` method (`include/mcl/fp.hpp#L496`).

```
static inline void add(FpT& z, const FpT& x, const FpT& y) { op_.fp_add(z.v_, x.v_, y.
↪v_, op_.p); }
```

As mentioned, the method calls the `fp_add` function from the `Op` structure.

The `Op` structure contains pointers to the basic operations of `FpT` (as well as of `Fp2T` and `FpDblT`). The implementation of these operations depends on the selected back end. When the structure is initialized, all the function pointers are set to the corresponding value.

In the following sections we will describe each back end in more detail.

The *GMP* back end

The `MCL_USE_GMP` flag enables the *GMP* back end. The functions are implemented in the `src/low_func.hpp` file. They all follow a similar pattern, presented below:

Listing 3.7: Code structure of the Fp operations.

```
template<size_t N, class Tag = Gtag>
struct OperationName {
    static inline void func(/* Parameters. */)
    {
        /* Implementation. */
    }
    static const FuncPtr f;
};

template<size_t N, class Tag>
const FuncPtr OperationName<N, Tag>::f = OperationName<N, Tag>::func;
```

An operation is called using the `OperationName<N, Tag>::f` pointer. This particular design allows to override the default[7] implementation of the operation (such is the case of the *LLVM* back end, as we will see in the next section).

Below is the implementation of the addition operation, to exemplify the aforementioned code structure:

[6] They provide operations for `Fp2T` and `FpDblT` as well. However, we will focus on the ones from `FpT` as the same design applies to all the classes.

[7] We will refer again to the default implementation, later in this section, when we talk about the `MCL_USE_VINT` flag.

Listing 3.8: Implementation of the addition operation (`src/low_func.hpp#L424-L455`).

```
// z[N] <- (x[N] + y[N]) % p[N]
template<size_t N, bool isFullBit, class Tag = Gtag>
struct Add {
    static inline void func(Unit *z, const Unit *x, const Unit *y, const Unit *p)
    {
        if (isFullBit) {
            if (AddPre<N, Tag>::f(z, x, y)) {
                SubPre<N, Tag>::f(z, z, p);
                return;
            }
            Unit tmp[N];
            if (SubPre<N, Tag>::f(tmp, z, p) == 0) {
                copyC<N>(z, tmp);
            }
        } else {
            AddPre<N, Tag>::f(z, x, y);
            Unit a = z[N - 1];
            Unit b = p[N - 1];
            if (a < b) return;
            if (a > b) {
                SubPre<N, Tag>::f(z, z, p);
                return;
            }
            /* the top of z and p are same */
            SubIfPossible<N, Tag>::f(z, p);
        }
    }
    static const void4u f;
};

template<size_t N, bool isFullBit, class Tag>
const void4u Add<N, isFullBit, Tag>::f = Add<N, isFullBit, Tag>::func;
```

As shown in Listing 3.8, the function depends on many others (such as `AddPre` and `SubPre`) that follow the same structure.

The last step missing in the description of the back end is the initialization of the `Op` structure, place where the `OperationName<N, Tag>::f` pointer is assigned to the corresponding field of the structure.

The initialization is done in the `Op::init` function, located at `src/fp.cpp#L382`. This function is complex given its many conditional compilation flags. However, the function `setOp` (and `setOp2` as well) assigns the fields of `Op` with the correct pointers. Continuing with the example of the addition operation[8], we have:

[8] The assignment of the `add` operation depends on the value of the `isFullBit`. For the sake of brevity, we only show the `true` case.

Listing 3.9: Assignment of the fp_add member of Op (src/fp.cpp#L252).

```
    op.fp_add = Add<N, true, Tag>::f;
```

The assignment of the rest of the operations of FpT follow the same logic (that is, declaration, definition and assignment). We will talk about this in more detail in *Initialization of the Op structure*.

It is worth noting that there is a preprocessor flag (MCL_USE_VINT) present in the implementation of the operation (src/low_func.hpp) that allows the use of an alternative implementation of the *GMP* library (just the required functions), called *VINT*. Listing 3.10 shows this:

Listing 3.10: Example of the use of the MCL_USE_VINT flag (src/low_func.hpp#L59-L73).

```
// (carry, z[N]) <- x[N] + y[N]
template<size_t N, class Tag = Gtag>
struct AddPre {
    static inline Unit func(Unit *z, const Unit *x, const Unit *y)
    {
#ifdef MCL_USE_VINT
        return mcl::vint::addN(z, x, y, N);
#else
        return mpn_add_n((mp_limb_t*)z, (const mp_limb_t*)x, (const mp_limb_t*)y, N);
#endif
    }
    static const u3u f;
};
template<size_t N, class Tag>
const u3u AddPre<N, Tag>::f = AddPre<N, Tag>::func;
```

Therefore, when no back end is selected, all operations are implemented using the vint module (that is, what we referred earlier as the default implementation).

The *LLVM* back end

The implementation of the *LLVM* back end is more complex than the previous one.

In this case, the operations of FpT are generated from scratch in *LLVM IR* (and subsequently compiled to assembly code). The file src/llvm_gen.hpp provides many classes and structures to simplify the process, such as Generator, Operand, Function, etc. The Generator structure, for example, allows to generate code for many *LLVM* instructions such as: add, sub, and, etc.

Note: One important thing to note is that the Generator component generates *LLVM* code entirely by its own without using any of the tools provided by *LLVM*. It is not clear the reason behind this approach, possibly to avoid the dependency on *LLVM* libraries. However, implementing all the infrastructure to generate *LLVM IR* code has some drawbacks. *LLVM* libraries are widely used and therefore very well tested and mature. In addition, relying on them may provide all the latest features and bug fixes as soon as they are released.

The file src/gen.cpp contains the implementation of the operations of FpT using the mentioned components as building blocks. For instance, Listing 3.11 is the implementation of the addition operation:

Listing 3.11: Implementation of the addition operation (`src/gen.cpp#L483-L528`).

```
483 void gen_mcl_fp_add(bool isFullBit = true)
484 {
485     resetGlobalIdx();
486     Operand pz(IntPtr, unit);
487     Operand px(IntPtr, unit);
488     Operand py(IntPtr, unit);
489     Operand pp(IntPtr, unit);
490     std::string name = "mcl_fp_add";
491     if (!isFullBit) {
492         name += "NF";
493     }
494     name += cybozu::itoa(N) + "L" + suf;
495     mcl_fp_addM[N] = Function(name, Void, pz, px, py, pp);
496     verifyAndSetPrivate(mcl_fp_addM[N]);
497     beginFunc(mcl_fp_addM[N]);
498     Operand x = loadN(px, N);
499     Operand y = loadN(py, N);
500     if (isFullBit) {
501         x = zext(x, bit + unit);
502         y = zext(y, bit + unit);
503         Operand t0 = add(x, y);
504         Operand t1 = trunc(t0, bit);
505         storeN(t1, pz);
506         Operand p = loadN(pp, N);
507         p = zext(p, bit + unit);
508         Operand vc = sub(t0, p);
509         Operand c = lshr(vc, bit);
510         c = trunc(c, 1);
511     Label carry("carry");
512     Label nocarry("nocarry");
513         br(c, carry, nocarry);
514     putLabel(nocarry);
515         storeN(trunc(vc, bit), pz);
516         ret(Void);
517     putLabel(carry);
518     } else {
519         x = add(x, y);
520         Operand p = loadN(pp, N);
521         y = sub(x, p);
522         Operand c = trunc(lshr(y, bit - 1), 1);
523         x = select(c, x, y);
524         storeN(x, pz);
525     }
526     ret(Void);
527     endFunc();
528 }
```

In this example, we can see how the prototype of the function representing the `Add` operation is created at line `495`. Then, the body is implemented between lines `497` (`beginFunc`) and `527` (`endFunc`).

Making sure that the resulting operation is implemented correctly is difficult since, in this particular case, it even involves branches. Therefore, testing is crucial to spot any error.

There is no direct memory manipulation on behalf of the function that implements the operation. Therefore, the risk of potential issues resulting from an error is low. However, the operation itself does manipulate memory arrays and, given the nature of the implementation, it is difficult to detect and assess any issue.

LLVM IR code is emitted each time a function representing an *LLVM* instruction is called.

The `src/gen.cpp` file is an executable by itself. When ran, it outputs the *LLVM IR* representation of the functions, referred as `base64.ll` in the `CMakeList.txt` file.

The `base64.ll` file is then compiled to finally obtain the functions in native code. This step is not mandatory, as the assembly code of these functions is included in the project (`src/asm` folder). However, this is the process by how they were obtained[9].

As mentioned in the previous section, the functions involved in the initialization of the `Op` structure follow a pattern as shown in Listing 3.12 (`Add<...>::f` holds the pointer to the function itself).

Listing 3.12: Code structure of the `FpT` operations.

```
template<size_t N, bool isFullBit, class Tag>
const void4u Add<N, isFullBit, Tag>::f = Add<N, isFullBit, Tag>::func;
```

When the *LLVM* back end is enabled (that is, the `MCL_USE_LLVM` flag is set), the `f` pointer of each operation is replaced with the pointer to the corresponding implementation. Also, the `low_func_llvm.hpp` header is included (through conditional compilation, `src/fp.cpp#L14-L17`). The latter file is where the overriding of the functions take place. In the case of the addition operation[10], this is how it is achieved (Listing 3.13):

Listing 3.13: Override of the `Add<n, true, tag>::f` pointer with the address of the LLVM implementation (`src/low_func_llvm.hpp#L34`).

```
#define MCL_DEF_LLVM_FUNC2(n, tag, suf)                          \
...                                                              \
template<>const void4u Add<n, true, tag>::f = &mcl_fp_add ## n ## suf;  \
...
```

The initialization step of the `Op` structure is the same as the in the case of the *GMP* back end.

The *XBYAK* back end

In this back end, the operations of `FpT` are generated dynamically with the help of the *Xbyak* library. The `FpGenerator` structure is responsible for this. It extends the `CodeGenerator` class provided by *Xbyak*, which has the functionality to generate native code in an easy way.

The `FpGenerator` class implements the operations in an assembly-like fashion. Continuing with the addition operation example, below we show how it is implemented using *Xbyak*:

[9] This process seems to differ depending on the build system used. When the *CMake* one is used, the assembly generation is skipped (an object file is directly generated from the `base64.ll` file). In contrast, when the *Makefile* is used they can be generated as described in the readme.md file of the project.

[10] Multiple versions of the same function are generated, built using different parameters such as `isFullBit`. For this reason, a suffix is added, as seen in the example, which identifies the version of a given function.

Listing 3.14: Implementation of the addition operation (`src/fp_generator.hpp#L649-L696`).

```
649     void3u gen_fp_add()
650     {
651         align(16);
652         void3u func = getCurr<void3u>();
653         if (pn_ <= 4) {
654             gen_fp_add_le4();
655             return func;
656         }
657         if (pn_ == 6) {
658             gen_fp_add6();
659             return func;
660         }
661         StackFrame sf(this, 3, 0, pn_ * 8);
662         const Reg64& pz = sf.p[0];
663         const Reg64& px = sf.p[1];
664         const Reg64& py = sf.p[2];
665         const Xbyak::CodeGenerator:LabelType jmpMode = pn_ < 5 ? T_AUTO : T_NEAR;
666 
667         inLocalLabel();
668         gen_raw_add(pz, px, py, rax, pn_);
669         mov(px, pL_); // destroy px
670         if (isFullBit_) {
671             jc(".over", jmpMode);
672         }
673 #ifdef MCL_USE_JMP
674         for (int i = 0; i < pn_; i++) {
675             mov(py, ptr [pz + (pn_ - 1 - i) * 8]); // destroy py
676             cmp(py, ptr [px + (pn_ - 1 - i) * 8]);
677             jc(".exit", jmpMode);
678             jnz(".over", jmpMode);
679         }
680         L(".over");
681             gen_raw_sub(pz, pz, px, rax, pn_);
682         L(".exit");
683 #else
684         gen_raw_sub(rsp, pz, px, rax, pn_);
685         jc(".exit", jmpMode);
686         gen_mov(pz, rsp, rax, pn_);
687         if (isFullBit_) {
688             jmp(".exit", jmpMode);
689             L(".over");
690             gen_raw_sub(pz, pz, px, rax, pn_);
691         }
692         L(".exit");
693 #endif
694         outLocalLabel();
695         return func;
696     }
```

The first thing to notice is that the code resembles its *LLVM* counterpart (Listing 3.11). First it "reads" the inputs from the stack frame (lines `662` to `664`) and then it generates the code of the operation (lines `667` to `694`)[11].

[11] It is worth noting that the implementation depends on the `MCL_USE_JMP` preprocessor definition, which is

Some of the remarks made in the *LLVM* back end about the difficulty to assess the implementation apply to this one as well. However, in this case there is no easy way to access the generated code. Another aspect to consider is that there is manipulation of dynamic memory, as the code is emitted dynamically (discussed later).

Listing 3.15 shows `gen_raw_add`, one of the auxiliary functions of `gen_fp_add`, where we can see better how much the code resembles plain assembly instructions. It is worth mentioning that in order to achieve this "similarity" a lot of work has to be done by the *Xbyak* library.

Listing 3.15: Auxiliary function of `gen_fp_add` (`src/fp_generator.hpp#L381-L394`)

```
/*
    pz[] = px[] + py[]
*/
void gen_raw_add(const RegExp& pz, const RegExp& px, const RegExp& py, const Reg64& t,
↪ int n)
{
    mov(t, ptr [px]);
    add(t, ptr [py]);
    mov(ptr [pz], t);
    for (int i = 1; i < n; i++) {
        mov(t, ptr [px + i * 8]);
        adc(t, ptr [py + i * 8]);
        mov(ptr [pz + i * 8], t);
    }
}
```

Each time a function representing an assembly instruction is executed the corresponding opcodes are emitted to the memory region that will hold the operations.

When the `MCL_USE_XBYAK` flag is defined, the `Op` structure includes the `FpGenerator` (through conditional compilation) among its members, as seen in Listing 3.16:

only used here and there is no documentation describing how or when to use it.

Listing 3.16: Conditional compilation of the FpGenerator in the Op structure (include/mcl/op.hpp#L203-L206):

```
#ifdef MCL_USE_XBYAK
    FpGenerator *fg;
    mcl::Array<Unit> invTbl;
#endif
```

The Op::init function calls indirectly the FpGenerator::init method, which dynamically generates all the operations and sets the corresponding pointers in Op.

One last aspect to mention about this back end is that it only supports the case when the Unit[12] size is 8[13]. It is worth noting that the entirety of the FpGenerator code depends on the value of the preprocessor define MCL_SIZEOF_UNIT for the case it is equal to 8 (src/fp_generator.hpp#L14).

Initialization of the Op structure

In the previous sections we described how each back end implements the operations of FpT. In this section we will describe in more detail how the Op structure is initialized.

As we have already mentioned, the Op structure holds pointers to the many operations available, which are used as building blocks to implement more complex ones.

The function Op::init is responsible for the binding of each operation to its implementation (that is, setting the pointer of an operation to the right function). To help with this task, there are various auxiliary functions, most notably: setOp, setOp2 and initForMont.

Listing 3.17 is the implementation of the setOp function:

Listing 3.17: setOp function (src/fp.cpp#L287-L316).

```
template<size_t N>
void setOp(Op& op, Mode mode)
{
    // generic setup
    op.fp_isZero = isZeroC<N>;
    op.fp_clear = clearC<N>;
    op.fp_copy = copyC<N>;
    if (op.isMont) {
        op.fp_invOp = fp_invMontOpC;
    } else {
        op.fp_invOp = fp_invOpC;
    }
    setOp2<N, Gtag, true, false>(op);
#ifdef MCL_USE_LLVM
    if (mode != fp::FP_GMP && mode != fp::FP_GMP_MONT) {
#if MCL_LLVM_BMI2 == 1
        const bool gmpIsFasterThanLLVM = false;//(N == 8 && MCL_SIZEOF_UNIT == 8);
        Xbyak::util::Cpu cpu;
        if (cpu.has(Xbyak::util::Cpu::tBMI2)) {
            setOp2<N, LBMI2tag, (N * UnitBitSize <= 384), gmpIsFasterThanLLVM>(op);
```

(continues on next page)

[12] The Unit type is defined in include/mcl/gmp_util.hpp#L56-L60, and can be of size 4 or 8.
[13] This can be clearly seen in the function of Listing 3.15, since the scale parameter of the index is fixed at 8.

(continued from previous page)

```
        } else
#endif
        {
            setOp2<N, Ltag, (N * UnitBitSize <= 384), false>(op);
        }
    }
#else
    (void)mode;
#endif
}
```

Here we can see how some operations are set (`fp_isZero`, `fp_clear`, etc.) and how others are set in `setOp2` based on some extra conditions. Below (Listing 3.18) we can see the code of the `setOp2` function:

Listing 3.18: `setOp2` function (`src/fp.cpp#L246-L285`).

```
template<size_t N, class Tag, bool enableFpDbl, bool gmpIsFasterThanLLVM>
void setOp2(Op& op)
{
    op.fp_shr1 = Shr1<N, Tag>::f;
    op.fp_neg = Neg<N, Tag>::f;
    if (op.isFullBit) {
        op.fp_add = Add<N, true, Tag>::f;
        op.fp_sub = Sub<N, true, Tag>::f;
    } else {
        op.fp_add = Add<N, false, Tag>::f;
        op.fp_sub = Sub<N, false, Tag>::f;
    }
    if (op.isMont) {
        if (op.isFullBit) {
            op.fp_mul = Mont<N, true, Tag>::f;
            op.fp_sqr = SqrMont<N, true, Tag>::f;
        } else {
            op.fp_mul = Mont<N, false, Tag>::f;
            op.fp_sqr = SqrMont<N, false, Tag>::f;
        }
        op.fpDbl_mod = MontRed<N, Tag>::f;
    } else {
        op.fp_mul = Mul<N, Tag>::f;
        op.fp_sqr = Sqr<N, Tag>::f;
        op.fpDbl_mod = Dbl_Mod<N, Tag>::f;
    }
    op.fp_mulUnit = MulUnit<N, Tag>::f;
    if (!gmpIsFasterThanLLVM) {
        op.fpDbl_mulPre = MulPre<N, Tag>::f;
        op.fpDbl_sqrPre = SqrPre<N, Tag>::f;
    }
    op.fp_mulUnitPre = MulUnitPre<N, Tag>::f;
    op.fpN1_mod = N1_Mod<N, Tag>::f;
    op.fpDbl_add = DblAdd<N, Tag>::f;
    op.fpDbl_sub = DblSub<N, Tag>::f;
    op.fp_addPre = AddPre<N, Tag>::f;
    op.fp_subPre = SubPre<N, Tag>::f;
    op.fp2_mulNF = Fp2MulNF<N, Tag>::f;
```

(continues on next page)

(continued from previous page)

```
284     SetFpDbl<N, enableFpDbl>::exec(op);
285 }
```

As we can see, the initialization of `Op` depends on many conditions that make it difficult to follow (compile-time flags, in the case of Listing 3.17; and run-time flags, in the case of Listing 3.18).

More initialization is done in `Op::init` after calling the aforementioned functions. Listing 3.19 exemplifies this:

Listing 3.19: Extract from `Op::init` function (`src/fp.cpp#L509-L525`).

```
509 #ifdef MCL_USE_LLVM
510     if (primeMode == PM_NIST_P192) {
511         fp_mul = &mcl_fp_mulNIST_P192L;
512         fp_sqr = &mcl_fp_sqr_NIST_P192L;
513         fpDbl_mod = &mcl_fpDbl_mod_NIST_P192L;
514     }
515     if (primeMode == PM_NIST_P521) {
516         fpDbl_mod = &mcl_fpDbl_mod_NIST_P521L;
517     }
518 #endif
519 #if defined(MCL_USE_VINT) && MCL_SIZEOF_UNIT == 8
520     if (primeMode == PM_SECP256K1) {
521         fp_mul = &mcl::vint::mcl_fp_mul_SECP256K1;
522         fp_sqr = &mcl::vint::mcl_fp_sqr_SECP256K1;
523         fpDbl_mod = &mcl::vint::mcl_fpDbl_mod_SECP256K1;
524     }
525 #endif
```

These code blocks depend on compile-time flags which make them conditional. As it can be seen, there are some operations that are initialized at different places, under very special conditions (Listing 3.17 and Listing 3.18). For example, `fp_mul` is initialized in Listing 3.18 at lines 260, 263 and 268 (depending on some conditions). Later, set again in Listing 3.19 at lines 511 and 521[14], depending on other conditions. Most probably these conditions have to do with optimizations. Considering all these possibilities, it is difficult to determine when the implementation of one operation will be used over the others.

Lastly, the `initForMont` function is called at the very end of the `init` function. We can see it in Listing 3.20:

Listing 3.20: `initForMont` function (`src/fp.cpp#L349-L377`).

```
static bool initForMont(Op& op, const Unit *p, Mode mode)
{
    const size_t N = op.N;
    bool b;
    {
        mpz_class t = 1, R;
        gmp::getArray(&b, op.one, N, t);
```

(continues on next page)

[14] And, in case the *XBYAK* back end is selected, it is set again (after the call of `setOp2`) by the `FpGenerator` (`src/fp_generator.hpp#L331`).

(continued from previous page)

```
        if (!b) return false;
        R = (t << (N * UnitBitSize)) % op.mp;
        t = (R * R) % op.mp;
        gmp::getArray(&b, op.R2, N, t);
        if (!b) return false;
        t = (t * R) % op.mp;
        gmp::getArray(&b, op.R3, N, t);
        if (!b) return false;
    }
    op.rp = getMontgomeryCoeff(p[0]);
    if (mode != FP_XBYAK) return true;
#ifdef MCL_USE_XBYAK
    if (op.fg == 0) op.fg = Op::createFpGenerator();
    bool useXbyak = op.fg->init(op);

    if (useXbyak && op.isMont && N <= 4) {
        op.fp_invOp = &invOpForMontC;
        initInvTbl(op);
    }
#endif
    return true;
}
```

In case the *XBYAK* back end is used, the implementation of some operations are JITed (as explained in *The XBYAK back end*) and some pointers of `Op` are set (in this case, overridden as some of them were set prior to this function call).

Another aspect to consider in the initialization of the `Op` structure is the *Mode* parameter. It specifies which back end will be used. Some features are enabled or disabled based on it (and for some cases the Montgomery representation is enabled). The possible modes are the followings: `FP_AUTO`, `FP_GMP`, `FP_GMP_MONT`, `FP_LLVM`, `FP_LLVM_MONT`, and `FP_XBYAK`. The mode is generally set to `FP_AUTO` throughout the code (set as the default value in each function that takes it as a parameter; for example, in the `FpT::init` method). When set to `FP_AUTO` everything seems to work as expected as the mode is switched to the corresponding one (that is, `FP_GMP`, `FP_LLVM` or `FP_XBYAK`) "automatically". This is achieved through the conditionally compiled code blocks that depend on the `MCL_USE_{GMP,LLVM,XBYAK}` flags. The following excerpt (Listing 3.21) shows how it is done for the `XBYAK` case:

Listing 3.21: Excerpt from `Op::init` (`src/fp.cpp#L406-L416`).

```
#ifdef MCL_USE_XBYAK
    if (mode == FP_AUTO) mode = FP_XBYAK;
    if (mode == FP_XBYAK && bitSize > 384) {
        mode = FP_AUTO;
    }
    if (!isEnableJIT()) {
        mode = FP_AUTO;
    }
#else
    if (mode == FP_XBYAK) mode = FP_AUTO;
#endif
```

Let us suppose the mode is set to `FP_GMP` (either because the default parameter was changed or the mode was mistakenly set) and the compilation flag is set to `MCL_USE_XBYAK`. Under this scenario the error would go unnoticed and the initialization of `Op` would be invalid. The same happens for the `LLVM` mode.

Important: We recommend re-engineering this part of the code to avoid these kind of issues.

Initialization of the *XBYAK* back end

In this section, we will explain in more detail some of the internals of `FpGenerator`.

Listing 3.22 shows the initialization function of the generator:

Listing 3.22: `FpGenerator::init` method (`src/fp_generator.hpp#L245-L256`).

```
    bool init(Op& op)
    {
        if (!cpu_.has(Xbyak::util::Cpu::tAVX)) return false;
        reset(); // reset jit code for reuse
        setProtectModeRW(); // read/write memory
        init_inner(op);
        // ToDo : recover op if false
        if (Xbyak::GetError()) return false;
//      printf("code size=%d\n", (int)getSize());
        setProtectModeRE(); // set read/exec memory
        return true;
    }
```

The `init_inner` method (`src/fp_generator#L260-L369`) is responsible for initializing the functions of `Op`. More precisely, all JITed code is emitted within this function. As we can see in the code snippet, previous to its call, the method `setProtectModeRW` is invoked, leaving the memory region in a read/write state. Then, after emitting the code, the memory region is left as read/execute (`setProtectModeRE` method) as necessary from a security standpoint.

We found that both `setProtectModeRW` and `setProtectModeRE` functions throw exceptions by default whenever an error occurs. As a matter of fact, these functions receive a parameter to indicate this, which is `true` by default. Exceptions are thrown with the help of the `XBYAK_THROW_RET` preprocessor macro, which depends on the `XBYAK_NO_EXCEPTION` flag (`src/`

xbyak.h#L267-L310). In case this flag is not set (default state), the XBYAK_THROW_RET effectively throws a C++ exception. In the opposite case, it sets a static variable with the error code and returns it as well. In the latter case, any issue generated during the invocation of setProtectModeRE would go unnoticed since the return value is not checked anywhere within the init function (src/fp_generator.cpp#L251 and src/fp_generator.cpp#L256, respectively). It is worth noting that there is an error check on line 254, after the init_inner function is executed. Moreover, in case of error, there is no recovery action on behalf of the calling function (as seen in the *To Do* item in the code). This could lead to an inconsistent state of Op, since some of its operations might not have been initialized.

Important: We recommend checking the return value of both setProtectModeRW and setProtectModeRE to prevent silent errors in case of compiling the library with different compilation flags. We also recommend checking the return value of the init calling function and take the necessary actions in case of error.

There is one more thing to notice in the init method. The generator only works if the CPU has AVX support. In case it doesn't, no exception or error message is presented to the user and the generator simply falls back to the default implementation of the operations.

Important: We recommend to explicitly warn the user in this case.

The invocation of the FpGenerator depends on the MCL_USE_XBYAK flag. It is called from within the initForMont function (Listing 3.20), which in turn is called from Op::init. The exact relationship between initFromMont and the *XBYAK* back end is not clear since there are modes that are explicitly based on the Montgomery representation (such as FP_GMP_MONT and FP_LLVM_MONT).

Important: We strongly suggest re-engineering the commented functions, specially Op::init. This would help to better define the limits between the many back ends that currently seem to be quite coupled.

The FpGenerator extends Xbyak::CodeGenerator. It is initialized with a fixed amount of space for allocating the JITed code, specifically: 4096 * 9 bytes. This number seems to be sufficient for the current requirements, although it is not clear how it was calculated.

The CodeGenerator is prepared to dynamically grow the memory region containing the code. However, in this particular setup the memory limit is fixed, and as explained earlier in case it is reached either an exception or an error is raised with the CodeGenerator class. We found that the initialization of this class is complex, as we can see below:

Listing 3.23: CodeArray class constructor (src/xbyak.h#L999-L1012).

```
explicit CodeArray(size_t maxSize, void *userPtr = 0, Allocator *allocator = 0)
    : type_(userPtr == AutoGrow ? AUTO_GROW : (userPtr == 0 || userPtr ==␣
↪DontSetProtectRWE) ? ALLOC_BUF : USER_BUF)
    , alloc_(allocator ? allocator : (Allocator*)&defaultAllocator_)
    , maxSize_(maxSize)
    , top_(type_ == USER_BUF ? reinterpret_cast<uint8_t*>(userPtr) : alloc_->
↪alloc((std::max<size_t>)(maxSize, 1)))
```

(continues on next page)

(continued from previous page)

```
    , size_(0)
    , isCalledCalcJmpAddress_(false)
{
    if (maxSize_ > 0 && top_ == 0) XBYAK_THROW(ERR_CANT_ALLOC)
    if ((type_ == ALLOC_BUF && userPtr != DontSetProtectRWE && useProtect()) && !
↪setProtectMode(PROTECT_RWE, false)) {
        alloc_->free(top_);
        XBYAK_THROW(ERR_CANT_PROTECT)
    }
}
```

After following the parameters from the `FpGenerator`, we can determine that the type of allocation used is `ALLOC_BUF` and that the allocator used is the `defaultAllocator_`. The latter depends on the `XBYAK_USE_MMAP_ALLOCATOR` flag. The parameter `usrPtr` seems to have a dual purpose, being used both to specify the type of memory allocation (where the possible values are `0`, `AutoGrow` and `DontSetProtectRWE`, shown in Listing 3.24) and as a user-defined memory pointer to store the JITed code when none of the mentioned values are used. In the specific case of `FpGenerator`, its value is set to `Xbyak::DontSetProtectRWE`.

Listing 3.24: Possible values of `usrPtr` (`src/xbyak.h#L925-L927`).

```
// 2nd parameter for constructor of CodeArray(maxSize, userPtr, alloc)
void *const AutoGrow = (void*)1; //-V566
void *const DontSetProtectRWE = (void*)2; //-V566
```

This kind of parameter management can introduce bugs. However, this escapes the scope of the current library since it is contained within *Xbyak*.

3.3.3 Back ends: Conclusion

One of the main drawbacks of the current design is that it makes difficult to assess with ease which implementation of each operation is being used at any given compilation or parametrization of the library. Although most probably this approach was taken to allow easy prototyping, testing and benchmarking of different implementations, it has turned into a disadvantage from the security and correctness perspective. Due to the complex nature of the code, subtle bugs could be hard to spot, weakening the overall reliability of the library.

The way of initializing the `Op` structure, that is making each available operation to point to its corresponding implementation, is complex and non-uniform throughout the back ends. Moreover, it is spread across several functions. This problem is worsened by the multiple compile-time flags that enable conditional compilation of multiple pieces of code.

Besides the lack of uniformity of the back ends interface, there is an interaction between them that makes even harder to follow the control flow. For example, the back ends are designed in such a way that it would seem they don't need to be fully implemented to work, since if one back end does not implement one operation the default one is used. As stated above, probably this allowed for testing and measuring the performance of different implementations at an early development stage, however, it may become a source of potential issues.

As this library lays the foundation of the whole *bls* library it is very important to be able to easily identify which implementation is being used.

Another important aspect to consider is the parametrization of the library[15]. There are many preprocessor defines that modify functions and/or conditionally compile blocks of code. Some of them are defined when building the library and the others take a default value. The purpose of many of them is not entirely clear (nor documented). The libraries *Xbyak* and *Cybozulib* also count with their own defines (in the latter case there is no documentation at all).

Our recommendations will focus mainly on improving the overall design and general programming practices:

- Addition of a clear and precise interface for back ends. Ideally, a back end should have an initialization and finalization routine, as well as a well-defined list of operations to implement.
- The selection of each back end should be mutually exclusive (in the build system as well as in the code). In the case a fallback/default mechanism is supported make it explicit.
- Better definition of all parameters of the library, their possible values and the default ones.

3.3.4 Serialization/Deserialization

The serialization/deserialization is implemented using the `Serializable struct`, which holds the general logic of the process. It requires that any class that derives from it implements a `load` and a `save` method. These methods are responsible for handling the specific aspects of the serialization/deserialization related to the class implementing them.

There are multiple serialization modes available, represented by the `IoMode` enumeration (`include/mcl/mcl.op.hpp#L89-L106`). We focused on `IoSerialize`, which is the only one relevant for the purposes of this audit. We considered the case when the `isETHserialization` is `true` as well, for the same reason.

G1 and *G2* are implemented using `EcT`, `FpT` and `Fp2T`. Therefore, we reviewed only the `load` and `save` of those classes.

As we commented in Section 3.2.2, the serialization/deserialization is generally a problematic process from a security standpoint since it involves parsing untrusted data and dynamic memory manipulation. Fortunately, in this case the potential issues posed by those factors are reduced since the process is fairly simple and there is no direct dynamic memory manipulation involved.

Serialization

Let us start by reviewing the `save` method from `EcT` class, which follows the format described in the documents mentioned in section *Serialization/Deserialization*. This function calls the `save` method of the underlying element (either `FpT` or `Fp2T`) and sets the three most significant bits accordingly after serializing the corresponding element. As far as we can tell, the function only deviates from the described format in the `isMSBserialize` function (`include/mcl/ec.hpp#L998`). Depending on the return value of this function, the serialization adds an extra byte when storing the element. It is worth mentioning that under the current configuration of the library, this does not seem to occur. However, it should be reviewed to make sure it does not occur under any configuration.

The `save` method of `Fp2T` is quite straightforward as it only has to store two elements of type `FpT`, one after the other. To this end, it relies on the `save` method of the latter. It is worth mentioning that this function also depends on the `IoSerialize` mode and the `isETHserialization` flag in order to work properly.

[15] We will comment more on this on the *Compilation flags* and the *Preprocessor defines* sections.

The `save` method of `FpT` follows the specification, storing the element in big-endian form correctly.

Deserialization

In the case of `EcT`, the deserialization does not follow the analogous process of the serialization, explained in the previous section, since the elements `FpT` and `Fp2T` are processed in the `load` method of `EcT` (instead of calling the `load` method of `FpT` and `Fp2T`). The reason of this approach is not clear.

The `EcT::load` method relies on the `setArray` one, implemented by both `FpT` and `Fp2T`. The latter, from a high-level perspective, loads an element from a buffer similarly to what the `load` method would do. However, the implementation differs as `setArray` does some extra processing (calls to the `copyAndMask` and `toMont` functions). The function seems to follow correctly the specification and, subsequently, loads the underlying elements as expected.

Conclusions

As a final remark, we recommend unifying the different approaches of the serialization process. On one hand, the `save` and `load` methods of `EcT`. On the other, the compressed and uncompressed variations. This design may generate subtle discrepancies for some unforeseen corner cases.

Regarding the compressed and uncompressed variations, it is worth noting that despite vast differences in the implementation, both produced the same results when put to test.

3.3.5 General observations

The following remarks concern about general aspects.

Cybozulib and *Xbyak* libraries integration

The *mcl* library relies on two other libraries: Cybozulib and Xbyak. Both of them have its own repository: cybozulib and xbyak, respectively. However, these libraries are not included as external dependencies but have been copied inside the code base.

Important: We highly recommend keeping them in their original repositories and include them as external libraries.

Cybozulib seems to be a multi-purpose library, used extensively in *mcl*. It is worth mentioning there is no documentation available regarding this library. Some parameters (such as `CYBOZU_DONT_USE_STRING` and `CYBOZU_DONT_USE_EXCEPTION`) that alter the behavior of certain functions should be better documented.

Xbyak is a C++ header-only library that provides functionality to emit x86(IA32), x64(AMD64, x86-64) assembly code dynamically, and is the core of the *XBYAK* back end. This library is very complex in nature as well as extensive. In this case, there is documentation available that describes the basic usage of the library and the parameters that alter some of the functioning of the code.

Although the library has more than 10 years of development, it should be noted that it deals with a very difficult topic such as dynamic code generation. Any error or bug in this library can impact significantly on the corresponding back end.

Build system

There are two build system scripts (for Unix-like systems): one based on `make` (`Makefile`) and the other on `cmake` (`CMakeLists.txt`). The `readme.md` file presents both. However, given the many compilation flags available, it is not clear that both compile the library in the exact same way. There are other differences regarding the compilation of the *LLVM* back end as well (described in section Section 3.3.2).

On the other hand, we found that other projects using the *bls* library such as bls-eth-go-binary do not use any of the build system provided but instead compile the library on their own.

Important: We recommend unifying the build system for clarity and consistency. The use of the compiled version should be enforced in projects depending on this library as well.

Compilation flags

There are multiple compilation flags. The most important ones are: `MCL_USE_GMP`, `MCL_USE_LLVM` and `MCL_USE_XBYAK`. Each one represents a different back end. In principle, these seem to be used in a mutually exclusive way. However, taking a look at `CMakeLists.txt` we can see that the default values are[16]:

- `MCL_USE_GMP=ON`
- `MCL_USE_LLVM=OFF`
- `MCL_USE_XBYAK=ON`

Note: The remarks made here and in the rest of the chapter about the compilation flags were based on the `CMakeLists.txt` file.

Handling compilation flags in this mode can lead to incorrect builds of the library by setting flags that should be mutually exclusive.

Another flag that may cause confusion is `MCL_USE_ASM`. This flag is used to generate the assembly code for the LLVM back end. However, in the `CMakeLists.txt` file, it is `ON` by default, but the `MCL_USE_LLVM` is `OFF`. This leads to think that it is needed for the *GMP* and *XBYAK* back ends when, in principle, it is not.

Important: We recommend improving these aspects to avoid the mentioned issues.

[16] This was covered in more detail in the *Back ends: Internals* section.

Preprocessor defines

There are many preprocessor defines spread throughout the code base. Many of them are used for conditionally compile multiple pieces of code. These make following and understanding the normal execution flow of the entire library quite challenging. In many cases, the reasons for using these defines are not clear, as well as the conditions that enables or disables them. We came across several `#if 1` blocks, presumably, put there to test the performance of some portions of code.

As commented in previous sections, this programming practices can introduce bugs under certain configurations of the library that can be hard to spot. We recommend reviewing these and refactoring them to increase the reliability and readability of the code.

Testing

One concerning aspect is how (or whether) the library is tested under all possible configurations. The complex nature of the implemented operations and the multiple ways to do it demand special emphasis on testing and the infrastructure around it. Some tests require the user to manually select the back end mode (refer to section Section 3.3.2 for a detailed description), thus preventing the possibility of automating the process.

Automatically testing the library for all available back ends is highly recommended[17].

Note that, even if all the tests are non-deterministic because they use the output of a PRNG seeded by non-constant values, the tests we reviewed output the results of the PRNG, which allows, in case of a failure, to reproduce the code knowing only the value that has raised the error.

[17] Similar comments apply to the *bls* library.

4. Adeherence with the specifications

In this section, we will concentrate our efforts on the implementation of some core operations of the different RFCs, as described in [BLSsigRFC] and [HashToCurve].

4.1 The *bls* library

The *bls* library intends to provide an API to perform the cryptographic primitives of BLSsig needed to sign a message, verify the signature and the aggregated versions of these primitives.

4.1.1 Bird's eye view on pairings for Ethereum 2.0

Let $(\mathbb{G}_1, +)$, $(\mathbb{G}_2, +)$ and $(\mathbb{G}_T, \cdot)$ be three different groups of prime order r. Let G_1, G_2 and g_t be respectively a generator of each of the three groups. A pairing on $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ is a bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$. The different groups are instantiated thanks to the BLS12_381 specification [Pairing].

We will briefly summarize how to sign a message m. Let s_k be an element in $\mathbb{Z}/r\mathbb{Z}$, which will be the secret key; the associated public key is computed as $p_k = s_k G_1$. Let H be a function mapping a bit string to an element of $\mathbb{G}_2$. The signature of a message m is computed as $s = s_k H(m)$. To verify a signature, it is needed to verify if $e(G_1, s)$ is equal to $e(p_k, H(m))$. Note that we can define also public key in $\mathbb{G}_2$ and map the message in $\mathbb{G}_1$. The choice in Ethereum 2.0 is the one we described, where a public key belongs to $\mathbb{G}_1$ and the hash of a message belongs to $\mathbb{G}_2$, also called **minimal-pubkey-size** according to [BLSsigRFC].

To aggregate signatures, let $s_{k,0}$ and $s_{k,1}$ be two secret keys, $p_{k,0}$ and $p_{k,1}$ respectively the two associated public keys and m a message to be signed. Let $s_0 = s_{k,0}H(m)$ and $s_1 = s_{k,1}H(m)$ be two signatures of m by respectively the two secret keys. Let s be the aggregated signature of s_0 and s_1, that is $s = s_0 s_1$. To check if s is composed of both s_0 and s_1, it suffices to check that $e(G_1, s)$ is equal to $e(p_{k,0} + p_{k,1}, H(m))$.

This way to aggregate signatures is however subject to a **rogue-key attack**, which is an attack that builds a specific public key in order to annihilate the contribution of other legitimate public keys. In our previous example, if $p_{k,1} = -p_{k,0} + xG_1$, where x is a scalar in $\mathbb{Z}/r\mathbb{Z}$, then the previous computation will be led to the equality of both parts, even if the contribution of $p_{k,0}$ will be hidden by $p_{k,1}$. However, it is mostly impossible with such a public key to find the associated secret key. A way to mitigate the rogue-key attack is to provide a proof that the expected possessor of the secret key associated to a public key has indeed this knowledge. This is the proof of possession scheme described in §3.3 of [BLSsigRFC].

In Ethereum 2.0, the ciphersuite is `BLS_SIG_BLS12381G2_XMD:SHA-256_SSWU_RO_POP_`, described in [BLSsigRFC]. This means that:

- the pairing friendly curve is BLS12_381;
- the signature scheme variant is the minimal-pubkey-size one;
- the *hash_to_point*, or *hash_to_curve*, implementation will follow §8.8.2 of [HashToCurve];
- the signature scheme will use the proof of possession scheme to avoid the rogue-key attack.

Note however that the rogue-key mechanism in Ethereum 2.0 [Eth2.0Ph0] is different than the one of [BLSsigRFC].

4.1.2 Usage of the library for the Ethereum 2.0 purpose

First, the *bls* library must be compiled using the `BLS_ETH=1` compilation definition, since basically, not defining `BLS_ETH` will allow to use the library in the minimal signature size instead of the minimal pubkey size needed by Ethereum 2.0. We follow the documentation in the `README.md`, which states that

```
#define BLS_ETH
#include <mcl/bn384_256.h>
#include <bls/bls.h>
```

Note: The file `mcl/bn384_256.h` does not exist in the `mcl` project, it is likely the file `mcl/bn_c384_256.h`.

To use the `bls` library, the documentation states that the following preamble is needed

```
int err = blsInit(MCL_BLS12_381, MCLBN_COMPILED_TIME_VAR);
if (err != 0) { ... }

blsSetETHmode(BLS_ETH_MODE_LATEST);
```

Note: In the `bls-eth-go-binary` project, the `Init` function which gathers all these initialization functions takes as input a curve identifier, but only the one of `MCL_BLS12_381` may be used. In the `bls-eth-rust` however, the `init_library` function does not take an input and forces to use the identifier of BLS12_381, which is the only one necessary for this specific library.

The `blsSetETHmode` function does not return an error if the curve identifier is the one of BLS12_381 and if the mode is `BLS_ETH_MODE_DRAFT_07`, which corresponds to `BLS_ETH_MODE_LATEST`. Unlike some identifiers, e.g., the curve identifier in `mcl/include/mcl/curve_type.h`, the `BLS_ETH_MODE` are defined thanks to `#define` and not `enum` type.

Note: The status of the function `blsSetETHmode` seems not really clear for us. This function suggests that it would affect something for an Ethereum 2.0 behaviour, but the function modifies a variable `g_irtfHashAndMap`, which is used in the function `toG`, a function in which this variable is used only if `BLS_ETH` is not defined. According to the *bls* `README.md`[1], this function has little to no impact, since the `toG` function is used in a function that seems not advised to work with in the context of Ethereum 2.0. This issue was reported as Issue 67

The `blsInit` function can be summarized to this portion of code, when the variable `curve` is equal to `MCL_BLS12_381`, `BLS_ETH` is defined and `__wasm__` is disabled.

```
int blsInit(int curve = MCL_BLS12_381, int compiledTimeVar)
{
   const mcl::CurveParam* cp = mcl::getCurveParam(curve);
   if (cp == 0) return -1;
   bool b;
   initPairing(&b, *cp);
```

(continues on next page)

[1] https://github.com/herumi/bls/tree/e4663751b56f3e55f035bb4b1444d5273c5ae36e#functions-corresponding-to-eth20-spec-names

(continued from previous page)

```
    if (!b) return -1;
    g_curveType = curve;

    mclBn_setETHserialization(1);
    g_P.setStr(&b, "...", 10);
    mclBn_setMapToMode(MCL_MAP_TO_MODE_HASH_TO_CURVE_07);
    if (!b) return -101;
    return 0;
}
```

The function `getCurveParam` returns, if the identifier of the curve matches the allowed ones (e.g., the `MCL_BLS12_381` one), some parameters of the curve, which are, using the notation of [Pairing]:

- a string representing the value, written in hexadecimal, of the parameter t, where t is equal to $-2^{63} - 2^{62} - 2^{60} - 2^{57} - 2^{48} - 2^{16}$ for BLS12_381;
- the coefficient b of the equation of the curve $y^2 = x^3 + b$, where b is equal to 4 for BLS12_381;
- an information `xi_a` about the b' coefficient, for which $b' = b(\texttt{xi\_a} + u)$ if the curve is of M-type, $b' = \texttt{xi\_a} - u$, where `xi_a` is equal to 1;
- a boolean which indicates if the curve is of M-type or not, i.e., D-type, which is true for BLS12_381;
- the identifier of the curve for the *mcl* project.

The `initPairing` pairing function computes from the curve parameters listed above the needed elements, e.g., r and p, which initialize, among others, the `Fr` and `Fp` objects. Some of the underlying functions return a boolean to report problem, which is correctly checked all along the chain.

The next functions finish the initialization. One sets the internal variable `isETHserialization_` of the `FpT` class to true, the other sets the generator `g_P` of the group $\mathbb{G}_1$ to the value in base 10 of §4.2.1 of [Pairing] and the last function indicates that the `hash_to_curve` function must be used with a mode close to implement the RFC [HashToCurve], see Section 4.2.1.

Note: The `mclBn_setETHserialization` takes an integer as a parameter, which for now must be to equal to 1 to set the variable as true, and any other values to be false. It may be better to change this integer to a boolean, if the value-range remains only true or false.

4.1.3 General usage of the *bls* project

In this section, we will follow the sections of [BLSsigRFC], and link to the appropriate functions in the *bls* project.

Note: For now on, the section about the test vectors in [BLSsigRFC] is empty, the tests implemented in the *bls* library are therefore in a best-effort mode, but corner cases may remain.

To build the tests in the Ethereum 2.0 context, a modification may need to be done to enable them. We propose here a way. After installing the *mcl* project as a submodule `git submodule update --init --recursive`, modify the `build.sh` file at the root of the *bls* project by adding

-DBLS_ETH=1 in the ..._build() functions in order to enable the *BLS_ETH=1* mode, then run the build.sh script. The resulting binaries to run the tests are in the build/bin directory.

KeyGen

First, the RFC begins by describing a way to generate a secret key. The RFC states however that any function that produces a "statistically close to uniformly random in the range" $[1, r)$.

Note: The underlying functions in the RFC are all implemented in the *mcl* project, since some HMAC-SHA2 functions may be used through the *cybozu* project and the I2OSP and OS2IP functions are already defined to perform the hash_to_curve [HashToCurve].

The procedure in the *bls* project is less documented. Two functions may be used to generate a private key

```
/*
	set secretKey if system has /dev/urandom or CryptGenRandom
	return 0 if success else -1
*/
BLS_DLL_API int blsSecretKeySetByCSPRNG(blsSecretKey *sec);
/*
	set user-defined random function for setByCSPRNG
	@param self [in] user-defined pointer
	@param readFunc [in] user-defined function,
	which writes random bufSize bytes to buf and returns bufSize if success else␣
↪returns 0
	@note if self == 0 and readFunc == 0 then set default random function
	@note not threadsafe
*/
BLS_DLL_API void blsSetRandFunc(void *self, unsigned int (*readFunc)(void *self, void␣
↪*buf, unsigned int bufSize));
```

By default, the blsSecretKeySetByCSPRNG requests either /dev/urandom or CryptGenRandom.

Note: The CryptGenRandom is considered as deprecated and "Microsoft may remove this API in future releases." We recommend to switch to BCryptGenRandom as soon as possible.

Note also that the behaviour of /dev/random and /dev/urandom is more and more closer[2].

A post treatment of the random string is done using the setArrayMask function, which uses the copyAndMask function. A simplification of the algorithm seems to be close to the following, where b_0 is the bit size of a number N, which is the upper bound on the generated number.

1. Let x be a random number.
2. Let $x \leftarrow x \bmod 2^{b_0}$.
3. If $x \geq N$, let $x \leftarrow x \bmod 2^{b_0 - 1}$.

Note: Such an algorithm does not provide a distribution "statistically close to uniformly random" in $[1, N)$. We highly recommend to document the design choice, especially if it intends

[2] https://lore.kernel.org/lkml/20200131204924.GA455123@mit.edu/

to provide secret key that may be used in the wild. This function can be used for testing purpose, but should be marked as it if this is the only goal of the function. Note also that [EIP-2333] and [BLSsigRFC] described the same way to generate a secret key from an input key material. An explanation about the design choices may be found in §5.1 of [HashToCurve].

The second function allows to set a custom PRNG. This custom PRNG must return the size of the output on success, since it is the condition for the `read` function not to report an error.

Note: Using another PRNG will not change the bias resulting from the `copyAndMask` function. Note that the return code of, e.g., `RAND_bytes` of OpenSSL or `mbedtls_ctr_drbg_random` of mbedTLS must be wrapped to match the requirements of the function.

Management of the secret keys

Management of the secret keys is the sole responsibility of the user. There is no function to wipe the memory from the secrets, and especially from the secret key. The users need to implement such function by themselves.

SkToPk

The second function is the computation of a public key from a secret key. The corresponding implementation in the *bls* project is `blsGetPublicKey`, however with some differences. The most important one is that the function returns a `blsPubliKey`, which is no more than a point in the subgroup of order r of $\mathbb{G}_1$. To be closer to the RFC at this point, the code will likely be

```
void blsGetPublicKey(void *buf, mclSize maxBufSize, const blsSecretKey *sec, bool␣
↪compressed) {
  blsPublicKey pub; blsGetPublicKey(&pub, sec);
  if (compressed)
    blsPublicKeySerialize(buf, maxBufSize, &pub);
  else
    blsPublicKeySerializeUncompressed(buf, maxBufSize, &pub);
}
```

Note: For more information about the serialization / deserialization variants, please refer to Section 3.2.2.

The second less important remark is that the underlying scalar of the `blsSecretKey` is not checked to be in the correct range, i.e., $[1, r)$.

Note also that the multiplication algorithm is the classical one, not a one that may be constant time, see Section 4.2.2. This may lead to have a `PopProve` implementation which is not constant time, since the API proposed by [BLSsigRFC] in §3.3.2 recomputes the public key from the secret key. We however do not see a situation where the `SkToPk` computation must absolutely be done in a constant-time way for the Ethereum 2.0 case. Indeed, the computation of the public key may be done only once offline and stored to be provided when needed. However, if such a computation is not possible, or a more complicated protocol where the public key needs to be computed on the fly, the implementation must become constant-time.

`KeyValidate`

This process allows to verify if a public key:

1. is a valid point on the curve,
2. is not the identity element,
3. is in the correct subgroup.

The input of the function is a serialized public key, and the deserialization algorithm verifies the first requirement. The second operation needs to define previously the point at infinity and then compare this point with the deserialized public key.

Note: We do not find a function in the *mcl* project which verifies if an element is the identity element or not, but we may miss such a function.

The last operation is implemented in the `blsPublicKeyIsValidOrder` function.

`CoreSign`

There are two functions that may be used to sign: `blsSign` and `blsSignHash`. In the Ethereum 2.0 context, both functions have the same behaviour, since the `toG` function, which gives a return code, returns always `true` if `BLS_ETH` is defined. The function uses the `mulCT` function, which is supposed to provide a constant-time multiplication, see Section 4.2.2. As in the `SkToPk` function, the signature is not serialized at the end of the function, to correspond to the RFC, either the `blsSignatureSerializeUncompressed` or `blsSignatureSerialize` functions must be used.

`CoreVerify`

As in the signature process, two functions may be used, `blsVerify` and `blsVerifyHash`, and are equivalent when compiling with `BLS_ETH=1`. These two functions differ with the RFC since both the public key and the signature are supposed to be deserialized and valid. This process skips then the first part of the RFC:

```
1. R = signature_to_point(signature)
2. If R is INVALID, return INVALID
3. If signature_subgroup_check(R) is INVALID, return INVALID
4. If KeyValidate(PK) is INVALID, return INVALID
5. xP = pubkey_to_point(PK)
```

Then, an implementation that takes signature and public key from the wild, as it may be the case for an Ethereum 2.0 client, must implement these instructions before using `blsVerify` or `blsVerifyHash`. This corresponds basically to

- for 1., use `blsSignatureDeserialize` or `blsSignatureDeserializeUncompressed`.
- for 3., use `blsignatureIsValidOrder`.
- for 4., use `KeyValidate`, as described in Section 4.1.3.

Note: There will be various ways to avoid recomputation in this function, as caching the results of `KeyValidate`, see §2.5 of [BLSsigRFC], or swapping 5. and 4. in order to not deserialize two times the public key, and then modifying a bit the input of `KeyValidate` to accept a deserialized public key.

`Aggregate`

The `Aggregate` function proposes a method to aggregate n `blsSignature` into one. As in the previous implementations of the *bls* project, the signatures are considered to be previously deserialized and valid. Note that the condition that n is non-zero is checked in the `blsAggregateSignature` function, but there is no return code associated to that, even if the precondition in the RFC allows to get the `INVALID` return code. The resulting signature is neither serialized at the end of the aggregation process.

`CoreAggregateVerify`

In the *bls* project, there are three different functions to check an aggregated signature:

- `int blsFastAggregateVerify(const blsSignature *sig, const blsPublicKey *pubVec, mclSize n, const void *msg, mclSize msgSize);`
- `int blsAggregateVerifyNoCheck(const blsSignature *sig, const blsPublicKey *pubVec, const void *msgVec, mclSize msgSize, mclSize n);`
- `int blsVerifyAggregatedHashes(const blsSignature *aggSig, const blsPublicKey *pubVec, const void *hVec, size_t sizeofHash, mclSize n)`.

Only `blsAggregateVerifyNoCheck` and `blsVerifyAggregatedHashes` match the API of `CoreAggregateVerify`. Compared to the RFC, the public keys and the signature are supposed to be deserialized and checked to be valid, as it is the case in all the API of the *bls* project. Then, the return code of this implementation, which is actually a boolean, will concern:

- the precondition call, which is that if `n` is zero, the result is `INVALID`, i.e., 0 in our case;
- the condition 11 in the RFC, which is `VALID`, i.e., 1 in our case, or `INVALID`, i.e., 0 in our case, about the verification of the signature.

The `msgVec` (resp. `hVec`) variable is supposed to contain all the `n` messages of the same size `msgSize` (resp. `sizeofHash`) and concatenated each other.

Inside the `BLS_ETH` definition of `blsAggregateVerifyNoCheck`, there are two possible pieces of code, activated by a `#if 1`. The activated part of code is commented as `1.1 times faster` than the other one. Let us however take a look at the non-activated case for now, which is the closest possible to the RFC.

First, recall that a pairing computation can be decomposed into two main operations, see Chap 3 of [GuiPai]:

- the Miller loop (ML);
- the final exponentiation (FE).

However, if the computation involves a lot of pairing operations, it is possible to "aggregate" the intermediate results of the ML computations without performing the FE operation, and perform only the FE when the pairing needs to be fully computed. This is why, in the code, the call to `millerLoop` is used, and not the one to `pairing`, and before the check condition, a call to `finalExp`.

Compared to the RFC, the check condition is $C_1/C_2 = C_1 \cdot C_2^{-1} = C1 \cdot \texttt{pairing}(R, -P)$.

Having in mind this implementation, we can detail now the activated part of the implementation, which is according to the comment faster than this previous implementation. This function works by computing the ML into chunks of 16 ML batch computations thanks to the `millerLoopVec` implementation.

The `blsVerifyAggregatedHashes` function has the same behaviour as the `blsAggregateVerifyNoCheck` one and the same optimization process to speed up the computation (there is however no way to select a less optimized implementation).

Note: As it, a call to this function does not prevent against a rogue-key attack. This is however not the purpose of the function.

Since the `bls` project is designed for a proof of possession scheme, then we switch to the §3.3 of [BLSsigRFC]. The case of `blsFastAggregateVerify` will be discussed in Section 4.1.3, since it may be used securely specifically in this context.

Note: Since all the core operations of the RFC are implemented or close to respect the RFC, implementing the basic scheme or the message augmentation scheme can be done in a relative small amount of time.

`PopProve` and `PopVerify`

The implementation of `blsGetPop` uses previous functions of the API of the project, especially in this case the `blsSign` function and the serialization one. Let us remember the `PopProve` and `CoreSign` procedures.

```
PopProve(SK) procedure              | CoreSign(SK, message) procedure
  1. PK = SkToPk(SK)                |
  2. Q = hash_pubkey_to_point(PK)   |   1. Q = hash_to_point(message)
  3. R = SK * Q                     |   2. R = SK * Q
  4. proof = point_to_signature(R)  |   3. signature = point_to_signature(R)
  5. return proof                   |   4. return signature
```

In the `blsGetPop` implementation, the `PopProve` procedure can be summarized as

```
blsPopProve(SK) procedure
  1. PK = SkToPk(SK)
  2. CoreSign(SK, PK)
```

However, if the `hash_pubkey_to_point` and `hash_to_point` are implementations of the `hash_to_curve` procedure, their difference comes from the DST (Domain Specific Tag): the DST for the `hash_pubkey_to_point` is `BLS_POP_BLS12381G2_XMD:SHA-256_SSWU_RO_POP_` and the one for `hash_to_point` is `BLS_SIG_BLS12381G2_XMD:SHA-256_SSWU_RO_POP_`, the difference is in the `BLS_POP...` versus `BLS_SIG...`. The same thing needs to be applied to the `blsVerifyPop` function, which uses the `blsVerify` implementation and then the wrong DST.

Note: We report this issue in Issue 66.

FastAggregateVerify

This last procedure has the same purpose as the `CoreVerify` function, but instead of aggregating different signatures for different messages, this function assumes that all the signatures were done for a same message. When the aggregation of the public key is finished, this aggregated public key is used in the `blsVerify`, which is in line with the RFC.

4.2 The *mcl* library

The *mcl* library provides the low level functions used in the *bls* project. We reviewed mostly the implementation of `hash_to_curve` as specified in [HashToCurve] and the multiplication of a point on an elliptic curve by a scalar.

4.2.1 The `hash_to_curve` function

An implementation of the `hash_to_curve` algorithm, described in [HashToCurve] is provided through the `mclBnG1_hashAndMapTo` and `mclBnG2_hashAndMapTo` functions defined in `include/bn.h` according to the documentation of the API. These functions are mostly wrappers around respectively the `hashAndMapToG1` and `hashAndMapToG2` functions[3]. Let us describe the mechanism of these functions, by taking a look at the `mclBnG2_hashAndMapTo` function, the `mclBnG1_hashAndMapTo` function differs a few in the organization of the underlying functions but not about the global mechanism.

Listing 4.1: `mclBnG2_hashAndMapTo` method (`include/mcl/impl/bn_c_impl.hpp#L460-464`).

```
int mclBnG2_hashAndMapTo(mclBnG2 *x, const void *buf, mclSize bufSize)
{
	hashAndMapToG2(*cast(x), buf, bufSize);
	return 0;
}
```

This function is however not as generic as described in [HashToCurve], then some parameters or functionality are enforced during the computation of the mapping, we will then described some of them, linking to the relevant part in the RFC or external documentations if needed.

Listing 4.2: `hashAndMapToG2` method (`include/mcl/bn.hpp#L2056-2071`).

```
inline void hashAndMapToG2(G2& P, const void *buf, size_t bufSize)
{
	int mode = getMapToMode();
	if (mode == MCL_MAP_TO_MODE_WB19 || mode >= MCL_MAP_TO_MODE_HASH_TO_CURVE_06) {
		BN::param.mapTo.mapTo_WB19_.msgToG2(P, buf, bufSize);
		return;
	}
	Fp2 t;
	t.a.setHashOf(buf, bufSize);
	t.b.clear();
	bool b;
```

(continues on next page)

[3] Note however that in the *bls* project, there is no call to both of the functions of the API and directly a call to the underlying functions.

(continued from previous page)

```
    mapToG2(&b, P, t);
    // It will not happen that the hashed value is equal to special value
    assert(b);
    (void)b;
}
```

Note: In many parts of the code, the pattern `assert(X); (void)X;` is used, probably by habit of the programmer. It may be good to pack this two instructions into one macro.

The first line of the function gets the mode which must be used by the function. The mode seems to represent which version of the standard [HashToCurve] is targeted, plus some extra mode, probably coming from old Ethereum 2.0 specifications before an alignment on the RFC. The modes are selected thanks to `mclBn_setMapToMode`, which in the end stores the mode into the variable `mapToMode_` of a `MapTo` object. The different modes are the following

Listing 4.3: Different `MCL_MAP_TO_MODE` (`include/mcl/curve_type.h#L39-53`).

```
/*
    remark : if irtf-cfrg-hash-to-curve is completely fixed, then
    MCL_MAP_TO_MODE_WB19, MCL_MAP_TO_MODE_HASH_TO_CURVE_0? will be removed and
    only MCL_MAP_TO_MODE_HASH_TO_CURVE will be available.
*/
enum {
    MCL_MAP_TO_MODE_ORIGINAL, // see MapTo::calcBN
    MCL_MAP_TO_MODE_TRY_AND_INC, // try-and-incremental-x
    MCL_MAP_TO_MODE_ETH2, // (deprecated) old eth2.0 spec
    MCL_MAP_TO_MODE_WB19, // (deprecated) used in new eth2.0 spec
    MCL_MAP_TO_MODE_HASH_TO_CURVE_05 = MCL_MAP_TO_MODE_WB19, // (deprecated) draft-
↪irtf-cfrg-hash-to-curve-05
    MCL_MAP_TO_MODE_HASH_TO_CURVE_06, // (deprecated) draft-irtf-cfrg-hash-to-curve-06
    MCL_MAP_TO_MODE_HASH_TO_CURVE_07, // draft-irtf-cfrg-hash-to-curve-07
    MCL_MAP_TO_MODE_HASH_TO_CURVE = MCL_MAP_TO_MODE_HASH_TO_CURVE_07 // the latset␣
↪version
};
```

Among all these modes, only `MCL_MAP_TO_MODE_TRY_AND_INC`, `MCL_MAP_TO_MODE_ORIGINAL` and `MCL_MAP_TO_MODE_HASH_TO_CURVE_07` (and then, at this state of the code, `MCL_MAP_TO_MODE_HASH_TO_CURVE`) will make the `mclBn_setMapToMode` function setting the mode to the chosen one, and then return `0` instead of `-1`, to indicate an error.

Note: The return code of the function is not checked in the file `src/bls_c_impl.hpp` of the *bls* project, which may cause an error in futur changes.

In the file `ffi/cs/mcl/mcl.cs` of the *mcl* project, the input value of `mclBn_setMapToMode` is fixed to `int MCL_MAP_TO_MODE_HASH_TO_CURVE = 5`, which corresponds to `MCL_MAP_TO_MODE_HASH_TO_CURVE_07`. The return value is not checked, which may cause error if the function is modified. In `ffi/go/mcl/mcl.go`, the error is checked.

Since the variable `mapToMode_` in the *bls* project is set to `MCL_MAP_TO_MODE_HASH_TO_CURVE_07`, we will follow this path in the code, we then jump to the `msgToG2` function .

Listing 4.4: `msgToG2` method (`include/mcl/mapto_wb19.hpp#L532-537`).

```cpp
void msgToG2(G2& out, const void *msg, size_t msgSize) const
{
    const char *dst = "BLS_SIG_BLS12381G2_XMD:SHA-256_SSWU_RO_POP_";
    const size_t dstSize = strlen(dst);
    msgToG2(out, msg, msgSize, dst, dstSize);
}
```

In this call of the overloaded function `msgToG2`, the Domain Specific Tag (DST) is set to `BLS_SIG_BLS12381G2_XMD:SHA-256_SSWU_RO_POP_`, which is the one proposed in §4.2.3 of [BLSsigRFC]. This DST name is not chosen randomly, it contains useful information. The first one is about `POP`, which means that the proof of possession scheme to avoid the rogue-key attack is used in the context of the BLS signatures `BLS_SIG`. The choice of the curve is `BLS12381`, which is BLS12_381, and the hash is mapped in `G2` $\mathbb{G}_2$, the minimal pubkey size variant. The `XMD` string is used to define that the variants of the subfunction `expand_message`: here, the variant described in §5.4.1 of [HashToCurve] will be used, with as hash function SHA-256. The `RO` string stands for random oracle encoding and the `SSWU` string the simplified Shallue-van de Woestijne-Ulas algorithm to map an element of a field to an elliptic curve specified on this field.

Note: In the specialization of `msgToG1`, the DST is set to `BLS_SIG_BLS12381G2_XMD:SHA-256_SSWU_RO_POP_` instead of `BLS_SIG_BLS12381G1_XMD:SHA-256_SSWU_RO_POP_`, the difference lies in the `G1` versus `G2` part. This issue was reported in Issue 85 and corrected by commit 0d9af2d2032960919fc6e656262e3a318922b249. This issue affected the *minimal signature size*, which is not the one chosen for Ethereum 2.0.

Note: The `mclBnG2_hashAndMapTo` is then specific to fit in the needs of specific applications, in which Ethereum 2.0 fits. The usage for the *bls* project with proof of possession for the variants *minimal signature size* and *minimal pubkey size* are also respected. It may be however not convenient for other applications, which may need to specify a proper DST.

With all these parameters in mind, it is not surprising to see in the code two functions that implement the `hash_to_curve` function described in §3 of [HashToCurve].

Listing 4.5: `msgToG2` method (`include/mcl/mapto_wb19.hpp#L526-531`).

```
void msgToG2(G2& out, const void *msg, size_t msgSize, const void *dst, size_t␣
↪dstSize) const
{
    Fp2 t[2];
    hashToFp2(t, msg, msgSize, dst, dstSize);
    Fp2ToG2(out, t[0], &t[1]);
}
```

At this point, we will not continue to dig into all the functions and will only describe some recommendations about the underlying functions. Note that for now on the DST requirements are not checked, which are that the length of the DST must be nonzero and at most 255 bytes, see respectively §3.1 and §5.4.1 of [HashToCurve].

These checks may be done in the `expand_message_xmd` function, implemented in

```
void expand_message_xmd(uint8_t out[], size_t outSize, const void *msg, size_t␣
↪msgSize, const void *dst, size_t dstSize)
```

However, the test to verify the second property on the DST is in an `assert` check, which is removed when the compilation uses the flag `-DNDEBUG`, which is by default how both the *mcl* and *bls* projects are compiled.

Another test seems to be too restricted, since the function must abort if $\lceil \text{outSize}/32 \rceil > 255 \Rightarrow \text{outSize} > 8161$, which is a condition on `n` for the first part of the condition or about `outSize` for the second part. Such a test exists, in an `assert` too, but it limits `outSize` to 256 bytes, instead of the 8160 allowed.

```
assert((outSize % mdSize) == 0 && 0 < outSize && outSize <= 256);
```

This `assert` gives us also a restriction on `outSize`, since it must be a multiple of 32. It is not a requirement of [HashToCurve], since in case of an output size different than a multiple of 32 bytes, a substring function will be called to keep only the necessary bytes. All these remarks do not involve a problem for the Ethereum 2.0 usage on this function, but it involves that the *mcl* project may not be used as a building block for a general purpose.

Note: We think that the `assert` politic may be reviewed, in order to have different behavior depending on the debug purpose, e.g., the `assert(0)` in the code, and verification of the input.

The second part of the `msgToG2` function uses the `sswuG2` function, which in a comment refers to another implementation in https://github.com/algorand/bls_sigs_ref, most precisely in `python-impl/opt_swu_g2.py`. This implementation is trivially not constant time. Note that a constant time implementation is proposed in §G.2.3 of [HashToCurve]. Having such a non-constant time implementation for Ethereum 2.0 is not necessarily.

Note: In PR 86, we added the test vectors of [HashToCurve] which corresponds to the cases that concern Ethereum 2.0, i.e., the one in §J.10.2 and §K.1. This PR was merged into master and refined in commit 2758b01440744a5e52371a6ab55ae05b26ed5955. In [HashToCurve], the output length for each test is always a multiple of 32, which is the only length that would be accepted by the implementation.

4.2.2 The `mulCT` functions

In BLSsig, a way to protect the secret key is to implement a constant-time multiplication algorithm when the secret key is used. It allows to mitigate some side-channel attacks, especially the timing ones. In the API of the *mcl* project, two functions provide an implementation of a point (i.e., a `mclBnG2` in the case of a signature for our case) on an elliptic curve by a scalar (i.e., a `mclBnFr`)

Listing 4.6: Multiplication methods (`include/mcl/bn.h#L420-424`).

```
MCLBN_DLL_API void mclBnG2_mul(mclBnG2 *z, const mclBnG2 *x, const mclBnFr *y);
/*
    constant time mul
*/
MCLBN_DLL_API void mclBnG2_mulCT(mclBnG2 *z, const mclBnG2 *x, const mclBnFr *y);
```

We however do not more dig into these functions. A private communication in end October with the author of the libraries informs us that the current `mulCT` function is not constant-time and that such a constant-time implementation is planned to be pushed in the next few months.

5. Conclusion

5.1 Regarding the code review

The code review covered both the *bls* and the *mcl* libraries, the latter being the cornerstone of former.

The majority of the functions provided by *bls* are either a direct call to a function of *mcl* or a short combination of them. Only a few have a more complex implementation. Therefore, most of the time was spent on *mcl*.

The *mcl* library is quite complex. It implements advanced cryptographic primitives, including multiple algorithmic optimizations. The library allows the user to select among three different back ends that implement the basic operations upon which more complex ones are built, namely: *GMP*, *LLVM*, and *XBYAK*.

The review process was focused in understanding the design of the *mcl* library as well as the functionality used by *bls*. It is worth mentioning that there is little to no documentation available. The code base has very few comments as well. These factors slowed down the review process significantly and made it rather difficult. A lot of time and effort was needed to understand the design and inner workings of the library.

We attempted to describe how each back end works and implements the functionality needed by *bls*, in order to determine their weak points. We found that the main drawback of the *mcl* library concerns its design. This approach was taken, most probably, to simplify the process of prototyping, testing and benchmarking different implementations and optimizations. However, it turns into a great disadvantage from a security and correctness perspective.

The lack of uniformity and clear definition regarding the interface of the back ends, and the existing interaction between them, makes the control flow of the entire library really hard to follow and understand.

The back ends are designed in such a way that it would seem they don't need to be fully implemented to work. That is, if one back end does not implement one operation the default one is used. As stated above, probably this was done to easily test and measure the performance of different implementations at an early development stage, however, it becomes a source of potential issues.

The current design makes it quite difficult to assess with ease which implementation of each operation is being used at any given compilation or parametrization of the library. Due to the complex nature of the code subtle bugs could be hard to spot and, even more importantly, go unnoticed, weakening the overall reliability of the library. This problem is worsen by the multiple preprocessor defines spread throughout the entire code base, and which are not always clear in intention or value; and compile-time flags that enable conditional compilation of multiple pieces of code.

Both the *LLVM* and the *XBYAK* back ends are very complex pieces of software, specially the latter. They both generate native code but in very different ways. The first, generating LLVM IR code that is compiled into assembly and subsequently into machine code. The LLVM IR code generator is written from scratch, without resorting to any of the readily available *LLVM* library to this end. The second, JITting x86/64 code using a called *Xbyak*, developed by the same author. This library is very complex in nature and may deserve an audit on its own as well. Therefore, making sure that both back ends implement the operations according to their specification and in a robust way turns into a real challenge.

We strongly recommend addressing the remarks made throughout the report before going forward with the integration of the library into a production-like context. As discussed, the main aspect to consider and improve is the design. However, there are other aspects that are important and should be addressed as well, such as the parametrization of the library. In this case, it should be clearly stated which parameters are used, their default value and their range where applicable. This does not only affect the final user at the moment to choose how to build the library but also affects the maintainability of the code base. There are many parameters within the code whose purpose are not always clear, nor the full implications in how the operations function. In this sense, we consider that the library is prone to misconfiguration.

5.2 Security concerns

The context of this audit is the use of the *bls* and *mcl* projects for Ethereum 2.0 purposes. In this context, a difference between the implementations of the specifications may lead to an incorrect consensus in the blockchain. In this purpose, we reviewed some of the key important points about how inputs are processed, how the computations are performed and how the outputs are formatted. This includes:

- the serialization and deserialization processes;
- the signature scheme implementation;
- the way to map a bit string to an element which may be used by the signature scheme.

All these points are covered by draft of RFCs [HashToCurve], [Pairing] and [BLSsigRFC]. Even if these drafts are still ongoing, they become more and more stable. Even if some sections may evolve, this will probably affect Ethereum 2.0 at most marginally.

Serialization and deserialization processes allow to communicate with the external world, which implies that some data may be malformed, intentionally or not. In this area, input data cannot be considered as trusted data, and then it is needed that deserialization rejects all malformed data. In the opposite, serialization needs to be always correct in order to be processed and not rejected because of an incorrect implementation. In this line, the deserialization procedure is not in a clear adherence with the RFC, which may lead to use incorrect data by other functions. Even if these functions may be able to detect these nonsense data, it is important to strongly adhere with the RFC.

The *bls* and *mcl* projects are implemented in a way that the projects may be used in more generic projects using pairing-based cryptography than only for the Ethereum 2.0 purpose. All of our findings about the core operations in the *bls* and *mcl* projects were not problematic for an Ethereum 2.0 purpose. Note however that most of the functions in the *bls* project assume that checking the preconditions of the RFC were performed by the user, as the serialization / deserialization process.

Another more important concern is about the management of the secret data, and especially the secret keys. For now on, the management of the secret key in memory is the sole responsibility of the user, and there is no way to wipe secrets directly from the *bls* API. Using the secret key in core BLS signature implementation is for now not constant time, as the main author wrote to us. As the comment on the `mclBnG2_mulCT` states that the implementation is constant time, we think that the documentation must be updated to warn about.

The few documentation available is also a security concern, in the sense that it is difficult for a user to infer if a function has the expected behaviour, and often relies only on the fact that the name of the function and its arguments seems to match something expected. The design choice of the function, e.g., the `#if 1` in the `blsAggregateVerifyNoCheck` function is also questionable, since it looks close to a research investigation to provide some speed-up instead of something ready for production. It may also be more difficult to maintain the base code by a larger community.

All in one, we think that the *bls* and *mcl* projects are not mature enough to match the security expectations of a sensitive project, as the Ethereum 2.0 one.

6. Findings and Recommendations

In this section we present a summary of the most concerning findings discussed throughout the report. The majority of them are related to design issues of the library. Although they do not pose a strict and immediate security risk, we consider they should be addressed promptly to improve the overall reliability, robustness and, therefore, security of the library.

Important: It is important to keep in mind that, although the issues described below are not classified as critical, they **do** represent a potential risk to the library. The current design choices, the complex nature of the implementation of the back ends and the multiple parametrization of the library make it prone to subtle bugs that might be present in the code base.

In that regard, we **strongly** suggest to improve some of the aspects mentioned below.

6.1 Issue: Lack of documentation

Description: Except the *bls* and *mcl* APIs and the READMEs, there is little to no documentation on both projects.

Recommendation: Having an exhaustive documentation will help to better understand the technical choices that were done in the libraries, will help the users to use the libraries in the best way, and may help to federate a community to maintain and enhance the code for its different usages. This will also help future audits of the libraries.

References: -

Related issues: -

6.2 Issue: Inconsistent back end interface (*mcl* library)

Description: Each back end implements the operations needed in a very different way. Under the current design it is not possible to clearly see the boundaries of each back end specially when the implemented operations are used. More importantly, the interfaces of the back ends differ from each other greatly. This makes the initialization function of the `Op` struct (`init`) particularly complex (the problem is increased given the multiple options, back end variations and conditionally compiled blocks of code). The details of how the functions implemented in the selected back end are difficult to see with the clarity expected for this kind of library.

Recommendation: We recommend providing a uniform and well-defined interface across all back ends considering initialization and finalization routines as well. This would greatly improve the clarity and readability of the code base. It would allow the users of the library to know exactly which operations are implemented by each back end and how and where they are used.

References: *The GMP back end*, *The LLVM back end*, *The XBYAK back end* and *Initialization of the Op structure*.

Related issues: *Issue: Ambiguous interaction between back ends (mcl library)*, *Issue: Potential inconsistency between compile-time and run-time parameters in Op structure initialization (mcl library)*.

6.3 Issue: Ambiguous interaction between back ends (*mcl* library)

Description: The current relationship, and subsequent interactions, between the different back ends is not clear. The current design seems to allow for "incomplete" implementations of each one (that is, it doesn't seem necessary for a particular back end to implement all the required operations). When this happens, the "default" implementation seems to be used. The latter is difficult to determine given the complexity of the `Op::init` function (for the reasons commented in issues *Issue: Inconsistent back end interface (mcl library)* and *Issue: Potential inconsistency between compile-time and run-time parameters in Op structure initialization (mcl library)*) and the possible parametrization of the library.

Recommendation: In case a fallback/default mechanism between the different back ends is allowed, make it explicit. This can be contemplated in the interface of each back end (as proposed in issue *Issue: Inconsistent back end interface (mcl library)*).

References: *The GMP back end*, *The LLVM back end*, *The XBYAK back end*, *Initialization of the Op structure*, *Compilation flags* and *Preprocessor defines*.

Related issues: *Issue: Ambiguous interaction between back ends (mcl library)*, *Issue: Ambiguous library parametrization (mcl library)*.

6.4 Issue: Ambiguous library parametrization (*mcl* library)

Description: The *mcl* library provides multiple ways for parametrization. The most important parameters are the ones that specify which back end to use. However, there are many more. Some of them are used to specify the unit size, the maximum bit size for `FpT`, whether or not the VINT module uses a fixed buffer and so on. Others are related to the configuration on the *Cybozulib* and *Xbyak* libraries. Only a few are documented.

On the other hand, some of them which, in principle, seem to be mutually exclusive can be set simultaneously leading to potentially incorrect builds.

Recommendation: Document all configuration flags/parameters customizable by the user, detailing possible values and the default one. In case there are parameters that should be used mutually exclusive, enforce it in the code base and the build system scripts as well. Customizable parameters from libraries should also be included in the documentation.

References: *Compilation flags*, *Preprocessor defines*, *Cybozulib and Xbyak libraries integration*, and *Build system* (we also discussed them in general throughout the entire *Back ends: Internals* section).

Related issues: *Issue: Lack of documentation*, *Issue: Multiple and potentially inconsistent build systems (mcl library)*.

6.5 Issue: RFCs API is not completely followed

Description: In many to all the functions of the *bls* and *mcl* APIs, the preconditions state by especially the RFCs are not checked, or only partially, and sometimes in `assert`, which are removed with the `-DNDEBUG` compilation flags. Comments in the `.h` defining the API may contain some information, but not all of them and is not always up-to-date. It lets the sole responsibility of the user to implement the checks, in addition to deserialize the inputs and serialize the outputs.

Recommendation: As it is, the *bls* API, which will be the component targeted by many cryptographic applications, can not be used in a security perspective without implementing an extra API layer around the *bls* API. If it must be complicated to have a general purpose API for all the applications combining both security and efficiency, implementing this extra layer for the function in §2 of [BLSsigRFC] may however be used by many applications and may serve as example to implement more complicated primitives.

References: *The hash_to_curve function* and *General usage of the bls project*.

Related issues: *Issue: Lack of documentation*.

6.6 Issue: Management of the secrets

Description: The *bls* API does not contain a function, for example `blsSecretKeyClear`, which will wipe the secrets in memory. It then lets the users the responsibility to wipe the memory by themselves. The function that will be used the most with the secret key, the `mulCT` function, is not yet constant time, as said by the main author of the library, who plans to implement such a constant-time multiplication in a few months. Such an implementation will mitigate side-channels attacks, especially the timing ones.

Recommendation: The implementation of a zeroization of the secret key will be really valuable for a lot of usage, which needs to be done with care. Having a reference implementation in the *bls* project will allow to avoid some custom zeroization which may not be really secure. About `mulCT`, the comment indicates that the function is constant-time, which doesn't seem to be the case actually.

References: *Management of the secret keys* and *The mulCT functions*.

Related issues: -

6.7 Issue: Pointer dereference issues

Description: Both libraries do not check for the validity of the pointers received. All the checks are left to the user of the libraries. Any misuse on behalf of the user can lead to invalid memory accesses in any of the libraries.

Recommendation: Add checks in all the functions publicly exposed by the library.

References: *Pointer dereference*.

Related issues: -

6.8 Issue: Potential inconsistency between compile-time and run-time parameters in *Op* structure initialization (*mcl* library)

Description: The `Op::init` function, responsible for initializing the `Op` structure, includes a `mode` parameter. It is used to select which back end to use. Its possible values are: `FP_AUTO`, `FP_GMP`, `FP_GMP_MONT`, `FP_LLVM`, `FP_LLVM_MONT`, and `FP_XBYAK`. Under current conditions this parameter is set to `FP_AUTO` (as it is the default value used by the functions calling `Op::init`). Within the `Op:init` function, this parameter is updated according to the back end selected at compile time (using `MCL_USE_GMP`, `MCL_USE_LLVM` or `MCL_USE_XBYAK`) to match it. However, this update only works in cases where the value of the mode is set to `FP_AUTO`. The code in charge of the update is compiled conditionally (depending on the flags listed previously). This parameter is used during the initialization process to enable/disable different features/optimizations.

To summarize, the `mode` parameter is directly related to the `MCL_USE_{GMP,LLVM,XBYAK}` flag. The current approach is prone to misconfiguration, and it could lead to potential issues. If the aforementioned case arises (that is, the default value is changed), a "de-synchronization" between the `mode` parameter and the selected back end will occur. This error will go unnoticed and the library will not behave as expected by the user.

Recommendation: We recommend re-engineering this part of the code to make it robust to the described scenario. Particularly, the initialization of the `Op` should not depend on possibly conflicting parameters. It is also recommended to review the entire function to improve its readability as there might be other related issues, given the complexity of the initialization process.

References: *Initialization of the Op structure*, and *Compilation flags*.

Related issues: *Issue: Inconsistent back end interface (mcl library)*, *Issue: Ambiguous interaction between back ends (mcl library)*, *Issue: Ambiguous library parametrization (mcl library)*.

6.9 Issue: Tests do not contemplate the main possible parameters in an automatic way (*mcl* library)

Description: There are tests available to run, however, some require to manually select the mode which make them unsuitable for execute them automatically on each commit. Also, the tests consider the library is built under only one back end at a time.

Recommendation: We recommend to modify the tests to run all modes in an automatic way, which would allow contemplation of the testing of all the different back ends at the same time.

References: *Testing*.

Related issues: -

6.10 Issue: Potential issues with the `FpGenerator` (*mcl* library)

Description: The `FpGenerator` is responsible for JITing the implementation of the operations, using the *Xbyak* library to that end. The initialization method of this class does not check the return values of certain functions, which indicate if the calls were successful or not. Under certain conditions these functions generate an exception, which ultimately stops the execution of the program (there is a compilation flag to control this behavior). However, when exceptions are disabled the error goes unnoticed since, as mentioned, the return values are not check.

The `FpGenerator` only initializes the operations when the CPU has support for AVX technology. Otherwise, it resorts to the "default" back end. However, the user is not properly warn about this (since the library can be compiled for the *XBYAK* back end anyways.

Another point to consider is the buffer where the JITed code is written to has a fixed size. Its value is not properly documented (though it seems to be enough for the current configuration).

Recommendation: We recommend adding the necessary checks so the library can properly work under the described conditions, and behave as the user of the library would expect.

References: *Initialization of the XBYAK back end* and *Initialization of the Op structure*.

Related issues: -

6.11 Issue: Libraries not maintained for a general purpose

Description: The *bls* and *mcl* libraries may be used for signature scheme algorithms in the minimal-signature-size or minimal-pubkey-size variants, as well as using the proof of possession schemes. However, the DST enforced in some part of the code for the hash function may not serve general purposes. This issue do however not concern the Ethereum 2.0 purpose.

Recommendation: The library seems to put a lot of effort to be compatible with the Ethereum 2.0 specifications, as indicated in Issue #66. This choice is understandable because of the release of the Ethereum 2.0 blockchain in a few weeks. However, documenting that other functionalities are not yet fully supported will help users to use the libraries for the supported purposes.

References: *PopProve and PopVerify* and *The hash_to_curve function*.

Related issues: *Issue: Lack of documentation*.

6.12 Issue: Multiple and potentially inconsistent build systems (*mcl* library)

Description: There are two build system scripts (for Unix-like systems). One corresponds to `make` and the other to `cmake`. Although, the `readme.md` file in the project presents both, it is not clear they build the library in the same exact way. This problem is aggravated by the many flags and parameters that can be use to build the library.

Recommendation: We recommend unifying the build system, for clarity and consistency. The use of the compiled version should be enforced in projects depending on this library as well.

References: *Build system*.

Related issues: *Issue: Ambiguous library parametrization (mcl library)*.

6.13 Issue: Inconsistent use of *mcl* API on behalf of the *bls* library

Description: The *mcl* library provides an API to access the functionalities it provides. However, the *bls* library does not use them and calls what can be consider "internal" functions of *mcl*. Most of the time, the functions implemented in *bls* are a copy of the functions provided by *mcl*. This can lead to errors in case the *mcl* library changes.

Recommendation: We recommend to use the public interface provided by the *mcl* library whenever possible (or document the reasons of why is done in case the opposite approach has to be taken).

References: *mcl library API usage*.

Related issues: -

7. Bibliography

[BLScurve] P. Barreto, B. Lynn and M. Scott, Constructing elliptic curves with prescribed embedding degrees, SCN 2002, LNCS, vol. 2576, pp 257–267, Springer 2003. https://eprint.iacr.org/2002/088.pdf

[BLSsig] D. Boneh, B. Lynn and H. Shacham, Short Signatures from the Weil Pairing, Journal of Cryptology. 17 (4), pp 297–319, 2004

[BLSsigRFC] D. Boneh, S. Gorbunov, R. Wahby, H. Wee and Z. Zhang, BLS Signatures, work in progress, Internet-Draft, version 4, September 2020. https://tools.ietf.org/html/draft-irtf-cfrg-bls-signature-04

[EIP-2333] C. Beekhuizen, BLS12-381 Key Generation, EIP-2333, September 2019. https://eips.ethereum.org/EIPS/eip-2333

[Eth2.0Ph0] Ethereum 2.0 Phase 0 -- The Beacon Chain, 15 September 2020. https://github.com/ethereum/eth2.0-specs/blob/dev/specs/phase0/beacon-chain.md

[GuiPai] Guide to Pairing-Based Cryptography, Nadia El Mrabet and Marc Joye, Chapman & Hall/CRC, 2016

[HashToCurve] A. Faz-Hernandez, S. Scott, N. Sullivan, R. Wahby and C. Wood, Hashing to Elliptic Curves, work in progress, Internet-Draft, version 10, June 2020. https://tools.ietf.org/html/draft-irtf-cfrg-hash-to-curve-10

[Pairing] Y. Sakemi, T. Kobayashi, T. Saito and R. Wahby, Pairing-Friendly Curves, work in progress, Internet-Draft, version 8, September 2020. https://tools.ietf.org/html/draft-irtf-cfrg-pairing-friendly-curves-08.html