AOCL Crypto

Software Design Document

Prem Mallappa pmallapp@amd.com

Contents

I Introduction **5**

1 Preface **6**

II System Overview **7**

2 Design Consideration **9**

3 Assumptions and Dependencies (TODO: TBD) **10**

4 General Constraints (TODO: TBD) **11**

5 Goals and Guidelines **12**

III Architectural Strategies (TODO: TBD) **13**

6 Programming Details **14**

7 Apps for testing **15**

8 System Architecture **16**
8.1 Plugins 16

9 Policies and Tactics **17**

10 Library Conventions **18**
10.1 Directory Structure 18

IV Detailed System Design **19**

11 Error Reporting **20**
11.1 Design 20
11.2 API 20
11.3 Implementation 21

12 Module Manager **22**
12.1 Design 22
12.2 APIs 23

13 Dispatcher **24**

V Detailed Subsystem Design **25**

14 Key Management (TODO: WIP) **26**
14.1 Key Types 26
14.2 Key Algorithm 27
14.3 The Key format 27
14.4 The `alc_key_info_t` structure 27

15 Digests **29**
15.1 Design 29
15.2 The *Digest* class 30
15.3 API 30

16 Ciphers **32**
16.1 Symmetric Ciphers 32
16.1.1 The `alc_cipher_ctx_t` structure 32
16.1.2 The `alc_cipher_ops_t` structure 33
16.1.3 The `alc_cipher_info_t` structure 33
16.1.4 The `alc_cipher_algo_t` type 33
16.1.5 The `alc_cipher_mode_t` type 34
16.1.6 The `alc_cipher_padding_t` type 34
16.2 AES (Advanced Encryption Standard) 34
16.3 Message Authentication Codes (MAC) (TODO: WIP) 35
16.4 AEAD Ciphers (TODO: WIP) 35
16.5 Key Derivation Functions (KDF) (TODO: WIP) 35
16.5.1 Padding 35
16.6 Random Number Generator 35
16.7 Utilities 40
16.7.1 Base-64 encoding and decoding 40

17 Random Number Generator (RNG) **41**
17.1 PRNG 41
17.2 TRNG 41

18 Message Authentication Codes (MAC) **42**

VI Plugin System design (WIP) 43

19 Plugins 44
19.1 APIs 45
19.2 Plugin for Device off-loading 45

20 Device Offloading (WIP) 47
20.1 Linux's cryptodev support (/dev/crypto) 47
20.2 Device APIs 47

VII Logging System 48

21 Performance Implications 50

22 Initializing the Logger 51

23 Using the default Logger 52

24 Creating new logger 53

25 Design 54
25.1 Message class 54
25.2 Log Level class 54
25.3 Logger Type class 54

26 Use Case 55

27 Use Case 2 56

VIII Dynamic Loading Feature 57

28 Dynamic library loading 59

29 Dynamic Feature/Class loading 60

30 Design 61
30.1 DynamicLibrary class 61
30.2 ClassLoader class 61

List of Figures

Part I

Introduction

Chapter 1

Preface

AOCL Crypto library described in this document is a part of AOCL Library that provides a portable interface to cryptographic operations. The interface is designed to be user friendly, while still providing low-level primitives. This document includes: - System Architecture - High-level functional overview - Design considerations - Detailed definition of API's provided by the library

AOCL Crypto library here in after referred to as 'the library' or crypto or cryptolib for short.

This document provides details of APIs in the following categories.

- Key Management
- Digests (One-way Hash Functions)
- Symmetric Ciphers
- Public Key Algorithms
- Message Authentication Codes (MAC)
- Key Derivation Functions (KDF)
- Random Number Generator (RNG)
- Digest Signing and Verification
- Padding

AOCL Crypto also provides compatibility layer which translates libcrypto and IPP-CP APIs to its own.

Part II

System Overview

AOCL Crypto is designed to be future compatible and extendable. AOCL-Crypto library has following components.

1. Algorithm - Describes any algorithm that deals with one cryptographic function. For example, AES CBC Encode is an algorithm, supporting one mode.
2. Module - Module is a collection of algorithms grouped together (logically). For example, AES and DES will be under module “Symmetric Key”
3. Plugin - Is a loadable module, contains one or more algorithms, which registers itself with the library to extend its functionality.
4. Compatibility layer - Compatibility layer allows any application compiled linked against other libraries to work with AOCL Crypto without modifications. AOCL Crypto provides compatibility layer for IPP-CP and libcrypto(from OpenSSL).

Dynamic dispatcher in the library optimally dispatches to best possible function for a given architecture, this decision is made once in the lifetime of the process and would not add any overhead due to decision making process. Each algorithm would exist in at least 2 forms

1. A Reference Implementation
2. An Optimized Implementation.
    - AVX SIMD
    - AVX2 SIMD
    - AESNI accelerated instructions
    - Hardware off-load processing

Any x86_64 machine that doesn’t support AVX, the reference implementation (very very slow) will be available, but we dont commit to support any machine before full implementation of AVX.

Each of them are dynamically dispatched to at runtime based on CPUID features.

Offloading Support: Accelerators are the new trend in high-end computing to take the load of computing and are becoming de-facto standard in the industry. The library supports device plugins which extends the functionality to work with a device for offloading. For ex: Hash computation SHA256, greatly useful in Cryptocurrency mining need not waste CPU time, could generate millions of hashes per second using offloading.

Chapter 2

Design Consideration

AOCL Crypto is expected to cater new as well as existing customers. Current customers may already be using other solutions like IPP-CP, OpenSSL-crypto, BoringSSL-crypto, MbedTLS etc, and may not want to recompile their entire software stack with AOCL Crypto. A solution must be provided to experiment with AOCL Crypto and to enable existing software stacks to easily migrate.

A module is a subsystem of AOCL crypto like Symmetric Cipher or a Digest. Each module has various algorithms listed under them for easier management.

A plugin is a loadable module which extends a module or adds new modules. This enables AMD to deliver new algorithms as an extension to the existing software early and safely.

All are version checked, and time to time libraries are updated and upgraded so that all versions need not be maintained.

Chapter 3

Assumptions and Dependencies (TODO: TBD)

AOCL Crypto assumes following libraries/tools available on system where it is built or running.

- CMake (3.18.4 or later)
- GCC (11.0 or later)
- Git (2.30.2 or later)
- OpenSSL (3.1 or later)
- Pandoc (+ LaTeX for generating pdf docs)

Chapter 4

General Constraints (TODO: TBD)

The library will contain all the listed algorithms eventually. At the library completeness level the priority is only for implementing one over other for a given release than choosing one over the other algorithm to include in library.

OpenSSL compatibility library needs to be co-developed along with AOCL Crypto, as the requirement for drop-in replacement is crucial for AOCL Crypto to succeed.

Chapter 5

Goals and Guidelines

AOCL Crypto aims at achieving FIPS certification. Source code is expected to be highly secure and tamper resistant. All developers are requested to adhere to coding standard and guidelines provided. Recommended Readings: - Writing Secure Code (Microsoft Press) - Secure Coding Cook-book

Part III

Architectural Strategies (TODO: TBD)

Chapter 6

Programming Details

The AOCL Crypto library provides C99 like API described in detail in API. Though the internal structures are implemented using C++ with no advanced features. This decision is taken to avoid writing primitive library functions like stacks/queues or heaps to [<0;199;17M]manage the module/algorithms. Also the C++ STL provides enough gears to achieve the needed functionality with least efforts.

AOCL Crypto makes use of AMD's CPUID identification library and RNG (random number generator) library to provide additional functionality like dynamic dispatcher. The RNG library also provides needed seeds for the algorithms in need.

Plugins feature enables useful and necessary functionality to extend the library's capability itself. By providing new algorithms and new modes to existing algorithm it allows to extend current library without need for upgrade.

Later versions of the library also supports offloading of some of the computation using various device plugins. These computations may be partial or fully supported by additional accelerators in the system/platform.

Crypto library's errors are cleverly designed to report all possible error from modules and algorithms. With support to extend the current error reporting mechanism to add new errors.

Concurrency is in the heart of the design where no locks are needed allow the functionality itself. However locks are used when adding/removing modules/plugins to/from the library.

Chapter 7

Apps for testing

- Nginx(pronounced Engine-X)
- gRPC
- QATv2

Chapter 8

System Architecture

To simplify the object access types, we introduce following notion

1. Types - Each category (module) will have many types of schemes, this needs to be highlighted using one of the `type` mechanisms.
2. Attributes - All the above mentioned components have attributes, an attribute defines properties for a given object or a context, may it be an algorithm or a module.
3. Operations - The operations that can be performed using that object or on that object. For example an cipher algorithm provides encrypt()/decrypt() kind of operations, where as an hash algorithm provides hash() or digest() kind of operation. Modules provides load()/unload()/search()/init() and other operations and so on.
4. Parameters - Parameters are passed to Operations to perform the same operation slightly differently. Some cases the distinction between attributes and parameters vanishes, as the attribute itself defines the parameter. However it is maintained throughout to provide uniform interface.

8.1 Plugins

The future of cryptography cannot be easily foreseen. New types of communication/certificate mechanisms may emerge, new types of messages may be introduced. Plugins are provide flexible way to integrate both while experimenting and deploying. Design of the plugins and its interfaces are discussed in detail in later sections of this document.

Chapter 9

Policies and Tactics

For this library, GCC is the choice of compiler with LLVM/Clang also in support, Designers and developers are made sure that no compiler-specific features are used, as it looses big on portability. On Windows VC compiler (latest version as of writing VS2019) is used.

Code will honor multiple operating systems, including Linux and Windows to start with.

Library will be provided as a static archive (libalcrypto.a on Linux and alcrypto.lib on Windows) as well as a dynamic version (libalcrypto.so on Linux and alcrypto.dll on Windows)

For build system we have opted for industry standard CMake (version >=3.18.4), and for testing ‘Gtest’ (Google Test) framework is used.

This library depends on libcpuid(A CPU Identification Library), version >= 1.0 used by the dynamic dispatcher to select appropriate function.

Documentation is maintained in ‘markdown’ format, ‘pandoc’ (version >= 2.9.2.1) command is used to generate pdfs.

Chapter 10

Library Conventions

AOCL Crypto is designed to be compliant with C99 API, hence uses all standard datatypes like `uint8_t` , `uint16_t`, however we avoid using `size_t` kind of datatypes as there is no clear mention of its size w.r.t ILP64 and LP64.

Library Defines following types - User Data types - Operation types - Attribute types

All types have prefix of `alc_` followed by type/module and end with `_t` , for example - Error type : `alc_error_t` and `alc_key_t` `alc_algorithm_t` - Operation type: `alc_cipher_ops_t` and `alc_hashing_ops_t` - Attributes: `alc_key_info_t` `alc_module_info_t`, `alc_cipher_info_t`

10.1 Directory Structure

This section details the very initial directory structure layout, though heavily subjected to change, overall structure would be comparable to following

- *docs/* : Contains various documentation both for application developers and library developers.
  - *docs/internal* : AMD's internal documentation such as design / architecture etc.
- *examples/* : sub-divided into its own directories to contain examples pertaining to a logical group of algorithms
  - *examples/symmetric/* : symmetric key algorithm examples
  - *examples/digest/* : One way hash function examples
  - etc...
- *include/* : Contains all the headers
  - *include/external* : API header, C99 based
  - *include/alcp* : Internal headers for library
- *lib/* : The library itself
  - *lib/compat* : Compatibility layers
    - *lib/compat/openssl* : OpenSSL Compatibility layer
    - *lib/compat/ippcp* : Intel IPP CP compatibility layer

Part IV

Detailed System Design

Chapter 11

Error Reporting

11.1 Design

11.2 API

This section needs to be populated from Detailed Subsystem design, and each subsystem needs to either make use of existing error codes or define new ones that may be relevant only to that subsystem.

If any subsystem requires a specific error code, such system should fill in the error code in specified `alc_error_t` with their name as one of the prefixes. For example, Key management subsystem would add `ALC_E_KEY_INVALID` instead of using existing `ALC_E_INVALID`.

TODO: This structure needs to go to proper section

The `alc_error_t` is designed to contain all the information in a single 64-bit value. For external API user, its just an opaque type defined to be a pointer.

```
typedef uint64_t alc_error_t;
```

All modules in AOCL Crypto library has an assigned ID which is internal to the library. However detailed error message can be printed using the function `alc_error_str()`.

The function `alc_error_str_internal()` will perform the same action as `alc_error_str()`. Just that it prints the filename and line number where the error function was called. This is used only internally in the library.

```
/**
* \brief        Converts AOCL Crypto errors to human readable form
* \notes        This is internal usage only, prints Filename and line number
*
* \param err    Actual Error
* \param buf    Buffer to write the Error message to
* \param size   Size of the buffer @buf
* \param file   Name of the file where error occured
* \param line   Line number in @file where error occured
*/

void
alc_error_str_internal(alc_error_t err,
                       uint8_t    *buf,
                       uint64_t    size,
                       const char *file,
                       uint64_t      line
                       )
{
    assert(buf != NULL);
    assert(size != 0);
}
```

The function `alc_error_str()` will decode a given error to message string, both the buffer and length of the buffer needs to be passed by the user to know the error.

```
/**
* \brief        Converts AOCL Crypto errors to human readable form
*/
void
alcp_error_str(alc_error_t err,
               al_u8* buf,
               size_t size)
{
    assert(buf != NULL);
    assert(size != 0);

    /* Write to Buffer */
}
```

11.3 Implementation

Internally errors are represented as `class Error`

Chapter 12

Module Manager

The AOCL Crypto library has internal module management for easy house keeping. A module is a collection of algorithms, and each algorithm will register itself with the Module Manager; each algorithm registers itself using the following APIs.

- `alcp_module_register()`
- `alcp_module_deregister()`
- `alcp_module_available()`

Some of the modules internally recognized at the time of writing are: - Digests (`ALC_MODULE_DIGEST`) - Symmetric Ciphers (`ALC_MODULE_CIPHER`) - Message Authentication Codes (MAC) (`ALC_MODULE_MAC`) - Key Derivation Functions (KDF) (`ALC_MODULE_KEY`) - Random Number Generator (RNG) (`ALC_MODULE_RNG`) - Digest Signing and Verification (`ALC_MODULE_SIGN`) - Padding (`ALC_MODULE_PAD`)

Each module supports its own operation. For example, a Symmetric key module supports - `alcp_cipher_encrypt()` - `alcp_cipher_decrypt()` - `alcp_cipher_available()`

The module also supports downward API's to register and manage algorithms. An algorithm is a unit, an indivisible entity, that allows operations that are specific to each type of module.

12.1 Design

Each module is identified by the `alc_module_info_t` structure. It describes the module type and supported operations.

The Module Manager is constructed as 'Singleton' pattern, a single instance exists per process.

```
typedef enum {
    ALC_MODULE_TYPE_INVALID = 0,

    ALC_MODULE_TYPE_DIGEST,
    ALC_MODULE_TYPE_MAC,
    ALC_MODULE_TYPE_CIPHER,
    ALC_MODULE_TYPE_KDF,
    ALC_MODULE_TYPE_RNG,
    ALC_MODULE_TYPE_PADDING,

    ACL_MODULE_TYPE_MAX,
} alc_module_type_t;
```

The `alc_module_info_t` describes the module. The simple signature is checked to see if the module belongs to aocl stack.

```
typedef struct {
    const char          *name;
    alc_signature_t      signature;
    alc_module_type_t    type;
    void                *ops;
} alc_module_info_t;
```

Each module will have its own operations structure, for example: A Symmetric Cipher algorithm will provide its own 'ops' structure as described in Symmetric Cipher Ops

12.2 APIs

The API `alcp_module_register()` tries to register the module with the module manager, the registration process returns appropriate error codes to identify the registration process's outcome. Like other parts of AOCL Crypto, use the `alcp_is_error()` API to detect success or error. For more description see ALC Error Types

```
if (alcp_is_error(err)) {

}
```

```
alc_error_t
alcp_module_register(alc_module_info_t *info);
```

Chapter 13

Dispatcher

The dynamic dispatcher will populate each kind of algorithm with best suitable implementation for the architecture(on which it is currently running). During the initialization phase of the library, it scans through available implementation and selects the best possible option.

Once the best algorithm is selected, its initialization is called, which then registers itself with the module manager. Once the registration is done, any request for a given algorithm will be returned with the already selected algorithm.

The dynamic dispatcher will allow debug mode to override the selection of the function.

If a plugin is loaded, its implementation will overwrite all the algorithms that are currently selected by the dynamic dispatcher. Hence plugins to be loaded with caution.

Since plugins are dynamic, there is no way to know/distinguish loaded plugin with existing algorithm. Also it will become difficult if plugins are distinguishable by the Application developer.

In cases when the plugin registers an algorithm that is not currently part of the library, it will be treated as an extension and applications can request for the algorithms supported by the newly loaded plugin.

Part V

Detailed Subsystem Design

Chapter 14

Key Management (TODO: WIP)

Key management is decoupled from algorithms, allowing any algorithm to use any key. However each algorithm checker will ensure that only supported keys are passed down to the actual implementation.

The Key types enumeration `alc_key_type_t` suggest what keys are in possession, and `alc_key_alg_t` determines the algorithm to be used for key derivation (if any). The `alc_key_fmt_t` suggests if the keys are encoded in some format, and needed to be converted in order to use. The `alc_key_attr_t` suggest type of key in each of `alc_key_type_t`. For ex:

14.1 Key Types

```
typedef enum {
    ALC_KEY_TYPE_UNKNOWN   = 0,

    ALC_KEY_TYPE_SYMMETRIC = 0x10,  /* Will cover all AES,DES,CHACHA20 etc */
    ALC_KEY_TYPE_PRIVATE   = 0x20,
    ALC_KEY_TYPE_PUBLIC    = 0x40,
    ALC_KEY_TYPE_DER       = 0x80,
    ALC_KEY_TYPE_PEM       = 0x100,
    ALC_KEY_TYPE_CUSTOM    = 0x200,

    ALC_KEY_TYPE_MAX,
} alc_key_type_t;
```

Key management module returns following errors,

- `ALC_KEY_ERROR_INVALID` : When an Invalid key type or pattern is sent to the API
- `ALC_KEY_ERROR_BAD_LEN` : When key length is not matching with keytype
- `ALC_KEY_ERROR_NOSUPPORT` : When key type is not supported.

14.2 Key Algorithm

```
typedef enum {
    ALC_KEY_ALG_WILDCARD,
    ALC_KEY_ALG_DERIVATION,
    ALC_KEY_ALG_AGREEMENT,
    ALC_KEY_ALG_SYMMETRIC,
    ALC_KEY_ALG_SIGN,
    ALC_KEY_ALG_AEAD,
    ALC_KEY_ALG_MAC,
    ALC_KEY_ALG_HASH,

    ALC_KEY_ALG_MAX,
} alc_key_alg_t;
```

14.3 The Key format

Key format specifies if the key represented by the buffer is encoded in some form or its just a series of bytes

```
typedef enum {
    ALC_KEY_FMT_RAW,    /* Default should be fine */
    ALC_KEY_FMT_BASE64, /* Base64 encoding*/
} alc_key_fmt_t ;
```

14.4 The `alc_key_info_t` structure

The structure `alc_key_info_t` holds the metadata for the key, it is used by other parts of the library. APIs needed to manage the key is may not directly be part of this module.

```
alc_key_algo_t
alcp_key_get_algo(alc_key_info_t *kinfo);
```

```
alc_key_type_t
alcp_key_get_type(alc_key_info_t *kinfo);
```

```
\#define ALC_KEY_LEN_DEFAULT  128
\#define BITS_TO_BYTES(x) (x >> 8)

typedef struct {
    alc_key_type_t    k_type;
    alc_key_algo_t    k_algo;
    uint32_t          k_len;    /* Key length in bits */
    uint8_t           k_key[0]; /* Key follows the rest of the structure */
} alc_key_info_t;
```

Chapter 15

Digests

15.1 Design

The datatype `alc_digest_type_t` describes the digest that is being requested or operated on.

```
typedef enum _alc_digest_type
{

    ALC_DIGEST_TYPE_MD2,
    ALC_DIGEST_TYPE_MD4,

    ALC_DIGEST_TYPE_MD5,
    ALC_DIGEST_TYPE_SHA1,
    ALC_DIGEST_TYPE_SHA2,
    ALC_DIGEST_TYPE_SHA3,
} alc_digest_type_t;
```

```
typedef enum _alc_digest_attr {
    ALC_DIGEST_ATTR_SINGLE,     /* Block */
    ALC_DIGEST_ATTR_MULTIPART,  /* Stream */
} alc_digest_attr_t;
```

```
typedef struct _alc_digest_info_t {
    alc_digest_type_t type;
    alc_digest_attr_t attr;
} alc_digest_info_t;


#ifndef __cplusplus
typedef void*  alc_digest_ctx_t;
#endif
```

15.2 The *Digest* class

This is the C++ interface to the Digests, all digest algorithms will be inherited by this class.

```
class DigestInterface {
  public:
    virtual alc_error_t update(const Uint8* pBuf, Uint64 size)   = 0;
    virtual alc_error_t finalize(const Uint8* pBuf, Uint64 size) = 0;
    virtual void        finish()                                 = 0;
    virtual alc_error_t copyHash(Uint8* pBuf, Uint64 size) const = 0;
  protected:
    //static cpuid_variant cpu;
};
```

All algorithms are expected to implement the `DigestInterface` abstract base class.

```
class Sha2 : public DigestInterface {
  public:
    Sha2(alc_sha2_mode_t mode)
            :m_mode{mode}
    {
        if (cpuid::isCpu(ALC_CPU_ZEN3)) {
            m_finit{sha256}
        }
    }
    virtual bool init(args) {m_finit(args);}
    virtual bool update(args) {f_fupdate(args);}
    virtual bool finalize(args) {m_ffinalize(args);}

    // stream digest
    virtual bool compute(args) {m_fcompute(args);}
  private:
    Sha2() {}

    alc_sha2_mode_t m_mode;
    alc_sha2_attr_t m_attr;
    alc_sha2_param_t m_param;

    std::function m_fupdate, m_ffinalize, m_fcompute;
};
```

15.3 API

Digests are computed like the other subsystem, All APIs are prefixed with `alcp_digest`, `alcp` being project prefix and `digest` being subsystem prefix. Following are the C99 APIs.

1. Query: One needs to query the library and see if a required algorithm is supported. Only upon making sure the algorithm and its parameterized customization is supported by library, application writer must proceed to next steps. `c alc_error_t alcp_digest_supported(const alc_digest_info_t *);`

2. Context size determination: The context size needs to be queried form the library, this helps the Application designer to allocate and free the memory needed for the 'Handle'. `c Uint64 alcp_digest_context_size(const alc_digest_info_t *);`

3. Request : The application needs to request the 'Handle', and supposed to send required configuration via the input of type `alc_digest_info_t`. `c alc_error_t alcp_digest_request(const alc_digest_info_t* p_digest_info, p_digest_handle);`

4. Update : Both block and stream digests are treated alike, however the update() method allows application to build on previously processed blocks.

```
alc_error_t
alcp_digest_update(const alc_digest_handle_t* p_digest_handle,
                   const Uint8*               buf,
                   Uint64                     size);
```

5. Finalize: This is the marker for last block, or end of sequence. Once this is called, its not possible to call update() again.

```
alc_error_t
alcp_digest_finalize(const alc_digest_handle_t* p_digest_handle,
                     const Uint8*               p_msg_buf,
                     Uint64                     size);
```

6. Finish : This is a cleanup phase, once finish is called the session ends, and the handle is no longer valid. Hence the digest needs to be copied by the application before this step. `c void alcp_digest_finish(const alc_digest_handle_t* p_digest_handle);`

7. Copy : The digest that is computed is held as part of internal representation, that needs to be requested by application to be copied to a buffer before calling `alcp_digest_finish()`.

```
    alc_error_t
    alcp_digest_copy(const alc_digest_handle_t* p_digest_handle,
                     Uint8*                     buf,
                     Uint64                     size);
```

Chapter 16

Ciphers

16.1 Symmetric Ciphers

Symmetric ciphers uses the same key for both encryption and decryption, The key types are described in Key Types.

The library supports Symmetric ciphers with GCM, CFB, CTR and XTS modes. Supported ciphers can be checked programatically using `alcp_cipher_available()` function.

Each Algorithm registers itself with algorithm-manager, which keeps a list of currently supported algorithm. The `alcp_cipher_available()` in turn calls the internal function `alcp_algo_available()` function to check if the provided mode / keylength is supported by the algorithm.

Crypto library uses "Factory" design pattern to create and manage the Cipher module. All ciphers are requested using `alcp_cipher_request()` API, which accepts various parameters to determine cipher and exact mode to operate.

```
alc_error_t
alcp_cipher_request(alc_cipher_info_t *cinfo,
                    alc_key_info_t    *kinfo,
                    alc_context_t     *ctx
                    );
```

In the above api, `alc_cipher_info_t` is described as in `alc_cipher_info_t`, which describes the cipher action with specific key information indicated by `alc_key_info_t` and A context for the session is described by `alc_context_t`. The Context describes everything needed for the algorithm to start and finish the operation. The key type is as described in the `alc_key_info_t`.

16.1.1 The `alc_cipher_ctx_t` structure

The Cipher's context is very specific to a given cipher algorithm. This structure or its contents are purely internal to the library, hence it will be sent as a handle with opaque type.

```
typedef struct {
    void *private;
} alc_cipher_ctx_t;
```

16.1.2 The `alc_cipher_ops_t` structure

This is a structure intended to be handled by the "Module Manager". Each cipher algorithm will present following functions to the module manager.

```
```

16.1.3 The `alc_cipher_info_t` structure

Cipher metadata is contained in the `alc_cipher_info_t`, describes the Cipher algorithm and Cipher mode along with additional padding needed.

```
typedef struct {
    alc_cipher_algo_t    c_algo;
    alc_cipher_mode_t    c_mode;
    alc_cipher_padding_t c_pad;
    alc_key_info_t       c_keyinfo;
} alc_cipher_info_t;
```

16.1.4 The `alc_cipher_algo_t` type

Any new algo needs to be added towards the end of the enumeration but before the `ALC_CIPHER_ALGO_MAX`.

```
typedef enum {
    ALC_CIPHER_ALGO_NONE = 0, /* INVALID: Catch the default case */

    ALC_CIPHER_ALGO_DES,
    ALC_CIPHER_ALGO_3DES,
    ALC_CIPHER_ALGO_BLOWFISH,
    ALC_CIPHER_ALGO_CAST_128,
    ALC_CIPHER_ALGO_IDEA,
    ALC_CIPHER_ALGO_RC2,
    ALC_CIPHER_ALGO_RC4,
    ALC_CIPHER_ALGO_RC5,
    ALC_CIPHER_ALGO_AES,

    ALC_CIPHER_ALGO_MAX
} alc_cipher_algo_t ;
```

16.1.5 The `alc_cipher_mode_t` type

Cipher modes are expressed in one of the following enumerations

```
typedef enum {
    ALC_CIPHER_MODE_NONE = 0, /* INVALID: Catch the default case */

    ALC_CIPHER_MODE_ECB,
    ALC_CIPHER_MODE_CBC,
    ALC_CIPHER_MODE_CFB,
    ALC_CIPHER_MODE_OFB,
    ALC_CIPHER_MODE_CTR,

    ALC_CIPHER_MODE_CCM,
    ALC_CIPHER_MODE_GCM,
} alc_cipher_mode_t;
```

16.1.6 The `alc_cipher_padding_t` type

```
typedef enum {
    ALC_CIPHER_PADDING_NONE = 0,
    ALC_CIPHER_PADDING_ISO7816,
    ALC_CIPHER_PADDING_PKCS7,
} alc_cipher_padding_t;
```

16.2 AES (Advanced Encryption Standard)

The library supports AES(Advanced Encryption Standard), as part of the Symmetric Cipher module.

16.2.0.1 CFB (Cipher FeedBack)

CFB Mode is cipher feedback, a stream-based mode. Encryption occurs by XOR'ing the key-stream bytes with plaintext bytes. The key-stream is generated one block at a time, and it is dependent on the previous key-stream block. CFB does this by using a buffered block, which initially was supplied as IV (Initialization Vector).

16.3 Message Authentication Codes (MAC) (TODO: WIP)

16.4 AEAD Ciphers (TODO: WIP)

16.5 Key Derivation Functions (KDF) (TODO: WIP)

16.5.1 Padding

Padding will take care of aligning the data to given length and filling the newly aligned area with provided pattern.

```
/* \fn alcrypt_padding_pad Pads the given input to the size specified
 * @param ctx AlCrypto Context
 */
alc_status_t
alcp_padding_pad(alc_context_t *ctx, alc_u8 *in, size_t size);
```

```
size_t alcp_padding_size(alc_context_t *ctx);
```

```
alc_status_t alcrypt_padding_unpad(alc_context_t *ctx);
```

16.6 Random Number Generator

The AOCL Crypto library supports both PRNG and TRNG algorithms. AMD Zen series of processors provide 'RDRAND' instruction as well as 'RDSEED', however there are speculations on its security. Also it is prone to side-channel attacks.

PRNG's usually requires a seed, and not considered cryptographically secure. The OS-level PRNG(/dev/random) are not desired as well for high-security randomness, as they are known to never produce data more than 160-bits (many have 128-bit ceiling).

However there are cryptographically secure PRNGs (or in other words CRNG) which output high-entropy data.

On Unix like modern operating systems provide blocking `/dev/random` and a non-blocking `/dev/urandom` which returns immediately, providing cryptographical randomness. In theory `/dev/random` should produce data that is statistically close to pure entropy,

Also the traditional `rand()` and `random()` standard library calls does not output high-entropy data.

RNG module will support two modes 'accurate' and 'fast', along with multiple distribution formats. The library also supports 'Descrete' and 'Continuous' distribution formats. RNG type specified - i : Integer based - s : Single Precision - d : Double Precision

Continuous Distribution formats:

Distribution	Datatype	RNG	Description
Beta	s,d		Beta distribution
Cauchy	s,d		Cauchy distribution
ChiSquare	s,d		Chi-Square distribution
Dirichlet	alpha[, size])		Dirichlet distribution.
Exponential	s,d		Exponential Distribution
Gamma	s,d		Gamma distribution
Gaussian	s,d		Normal (Gaussian) distribution
Gumbel	s,d		Gumbel (extreme value) distribution
Laplace	s,d		Laplace distribution (double exponent)
Logistic	[loc, scale, size])		logistic distribution.
Lognormal	s,d		Lognormal distribution
Pareto	a[, size])		Pareto II or Lomax distribution with specified shape.
Rayleigh	s,d		Rayleigh distribution
Uniform	s,d		Uniform continuous distribution on [a,b)
Vonmises	mu, kappa[, size])		von Mises distribution.
Weibull	s,d		Weibull distribution
Wald	mean, scale[, size])		Wald, or inverse Gaussian, distribution.
Zipf	a[, size])		Zipf distribution.

Descrete Distribution formats:

Type of Distribution	Data Types	RNG	Description
Bernoulli	i	s	Bernoulli distribution
Binomial	i	d	Binomial distribution
Geometric	i	s	Geometric distribution
Hypergeometric	i	d	Hypergeometric distribution
Multinomial	i	d	Multinomial distribution
Negbinomial	i	d	Negative binomial distribution, or Pascal distribution
Poisson_V	i	s	Poisson distribution with varying mean
Uniform_Bits	i	i	Uniformly distributed bits in 32-bit chunks
Uniform	i	d	Uniform discrete distribution on the interval [a,b)
	i	i	Uniformly distributed bits in 64-bit chunks

16.6.0.1 Design

Each RNG is represented by the `alc_rng_info_t` structure. The library provides interface to query if a RNG configuration is available using `alcp_rng_supported()`, this provides the option for the application to fall back to different algorithm/configuration when not supported.

As usual with other modules, all the RNG api's return `alc_error_t` and use of `alcp_is_error(ret)` will provide sufficient information to fallback or to abort for the application.

All available RNG algorithms will register with Module Manager with type `ALC_MODULE_TYPE_RNG`, Types of Generator are described by

An RNG generator can be requested using `alcp_rng_request()`, which accepts an `alc_rng_info_t` structure, which has following layout.

```
typedef struct {
    alc_rng_type_t          r_type;
    alc_rng_source_t        r_source;
    alc_rng_distrib_t     r_distrib;
    alc_rng_algo_flags_t  r_flags;
} alc_rng_info_t;
```

```
typedef enum {
    ALC_RNG_TYPE_INVALID = 0,
    ALC_RNG_TYPE_SIMPLE,
    ALC_RNG_TYPE_CONTINUOUS,
    ALC_RNG_TYPE_DESCRETE,

    ALC_RNG_TYPE_MAX,
} alc_rng_type_t ;
```

Random Number source can be selected using following enumeration. The request function

```
typedef enum {
    ALC_RNG_SOURCE_ALGO = 0,  /* Default: select software CRNG/PRNG */
    ALC_RNG_SOURCE_OS,        /* Use the operating system based support */
    ALC_RNG_SOURCE_DEV,       /* Device based off-loading support */

    ALC_RNG_SOURCE_MAX,
} alc_rng_source_t;
```

Random Generation algorithms and their distribution are described by enumeration `alc_rng_distribution_t`.

```
typedef enum {
    ALC_RNG_DISTRIB_UNKNOWN = 0,

    ALC_RNG_DISTRIB_BETA,
    ALC_RNG_DISTRIB_CAUCHY,
    ALC_RNG_DISTRIB_CHISQUARE,
    ALC_RNG_DISTRIB_DIRICHLET,
    ALC_RNG_DISTRIB_EXPONENTIAL,
    ALC_RNG_DISTRIB_GAMMA,
    ALC_RNG_DISTRIB_GAUSSIAN,
    ALC_RNG_DISTRIB_GUMBEL,
    ALC_RNG_DISTRIB_LAPLACE,
    ALC_RNG_DISTRIB_LOGISTIC,
    ALC_RNG_DISTRIB_LOGNORMAL,
    ALC_RNG_DISTRIB_PARETO,
    ALC_RNG_DISTRIB_RAYLEIGH,
    ALC_RNG_DISTRIB_UNIFORM,
    ALC_RNG_DISTRIB_VONMISES,
    ALC_RNG_DISTRIB_WEIBULL,
    ALC_RNG_DISTRIB_WALD,
    ALC_RNG_DISTRIB_ZIPF,
    ALC_RNG_DISTRIB_BERNOULLI,
    ALC_RNG_DISTRIB_BINOMIAL,
    ALC_RNG_DISTRIB_GEOMETRIC,
    ALC_RNG_DISTRIB_HYPERGEOMETRIC,
    ALC_RNG_DISTRIB_MULTINOMIAL,
    ALC_RNG_DISTRIB_NEGBINOMIAL,
    ALC_RNG_DISTRIB_POISSON,
    ALC_RNG_DISTRIB_UNIFORM_BITS,
    ALC_RNG_DISTRIB_UNIFORM,

    ALC_RNG_DISTRIB_MAX,
} alc_rng_distrib_t;
```

Each algorithm have some flags to further extend/restrict. This may or may not have valid information. For example `ALC_RNG_DISTRIB_POISON` could be selected in multiple format 1. Normal Poison distribution 2. With Varying mean

```
typedef enum {

} alc_rng_algo_flags_t;
```

16.6.0.2 APIs

To support the fallback for applications in cases where the expected RNG support is not available, `alcp_rng_supported()`, returns error not supported. No errors if the given RNG and its Distribution support is available.

```
alc_error_t
alcp_rng_supported(const alc_rng_info_t *tt);
```

An RNG handle can be requested using `alc_rng_request()`, the context(handle) can only be used if the check `if (!alc_is_error(ret))` passes for the call.

```
alc_error_t
alcp_rng_request(const alc_rng_info_t *tt, alc_context_t *);
```

The `alcp_rng_gen_random()` generates random numbers and fills the buffer pointed by `buf` for length specified by `size` in bytes.

```
alc_error_t
alcp_rng_gen_random(alc_context_t *tt,
                    uint8_t       *buf,  /* RNG output buffer */
                    uint64_t       size  /* output buffer size */
                    );
```

16.7 Utilities

16.7.1 Base-64 encoding and decoding

Encoding to Base-64 helps to print the long data into textual format. It uses 6-bits of input to encode into one of the following characters. First 26 letters of uppercase alphabets, and next 26 letters are using lowercase alphabets, rest of them use the digits 0-9 and ' + ',' / '.

```
static char base64_table = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
                           "abcdefghijklmnopqrstuvwxyz"
                           "0123456789+/";
```

APIs include `alcp_base64_encode()` and `alcp_base64_decode()`

```
alc_error_t
alcp_base64_encode(unsigned char *in,
                   uint64_t       in_size,
                   unsigned char *out,
                   uint64_t       out_len
                   );
```

```
alc_error_t
alcp_base64_decode(unsigned char *in,
                   uint64_t       in_len,
                   unsigned char *out,
                   uint64_t       out_len
                   );
```

Chapter 17

Random Number Generator (RNG)

17.1 PRNG

17.2 TRNG

Chapter 18

Message Authentication Codes (MAC)

Part VI

Plugin System design (WIP)

Chapter 19

Plugins

Plugins are modules (refer to Modules for more information) but externally loaded to support additional functionality. Crypto framework does not differentiate between plugins and already existing modules. The application software should make additional configuration to request new algorithm.

Each plugin uses different API's to get themselves registered. This is to differentiate between existing algorithms and dynamically loaded ones.

There are two ways to configure a plugin 1. By setting environment variable `ALCP_PLUGINS` as a comma separated value. The plugin initialization happens from the library at initialization time. 2. By using `LD_PRELOAD` in which case the active initialization happens from the plugin. OR by making the library as a link time option for application.

We take the (1) as the approach as it is more portable over (2).

Each plugin will have its own `alcp_plugin_info_t` structure

```
typedef
struct  alc_plugin_info {
    const char *name;
} alc_plugin_info_t;


typedef
struct alc_plugin_ops {

} alc_plugin_ops_t;
```

19.1 APIs

This section describes the API design for 'C', the same can be used by many other languages using their respective FFI(Foreign Function Interface).

Plugins are loaded in two ways : 1. By using the ALCP_PLUGINS environment variable, which can be parsed on Linux/Windows environment for a given application in its own context. 2. Programatically using the plugin APIs.

The environment variable ALCP_PLUGINS is a comma or semi-colon separated list of plugin names. The AOCL Crypto library will load the plugins and initialize.

Plugins are identified using their filename; for example: an file name which has prefix alcp-plugin-aead (and filename as alcp-plugin-aead.so or alcp-plugin-dev-msm.so)should use the plugin name as '"aead".

Given multiple plugins ALCP_PLUGINS can be used as follows

```
$ export ALCP_PLUGINS="aead,dev-msm"
```

Plugins can be loaded using API alcp_plugin_load()

```
alc_error_t
alcp_plugin_load(const char *name);
```

If the plugin is not available error can be checked as usual.

```
if (alc_is_error(ret)) {
    // Take action
}
```

After load it is recommended that the plugin is initialized for any necessary actions that needs to be performed. Usually the loader program will call plugin_init() after successful load. The plugin_info_t will be part of the plugin which identifies the plugin and provides a callback to initialize.

Unloading of plugin can be performed using alcp_plugin_unload, if the plugin is not available, the request is simply ignored than reporting error.

```
void
alcp_plugin_unload(const char *name);
```

19.2 Plugin for Device off-loading

Offloading support is an additional feature that needs to be supported in order to extend the APIs. All the loaded device-plugins are searched for successful initialization. They are searched in the order loaded, assuming that only one of them will ever succeed on a platform.

All the others will return error codes via the alc_error_t type; and can be checked using alcp_is_error() call.

Device plugins have slightly different API naming compared to other plugins to enable the device management.

For more information please refer to the section describing off-loading support in Device Offload-ing

Chapter 20

Device Offloading (WIP)

20.1 Linux's cryptodev support (/dev/crypto)

The Linux cryptodev supports hardware crypto devices.

20.2 Device APIs

```
alc_error_t
```

Part VII

Logging System

Logger supports - Multiple loggers, accessible by name - a static class method `getLogger(const std::string&)` - C++ style logging with overloaded << operator. - Custom formatted output useful for AOCL project, such as dumping memory location with width size as *byte*, *word*, *dword*, *qword* etc. - Ability to control Log Level - Serialization of multiple calls to logger - Thread Safe Design

Chapter 21

Performance Implications

Logger to be available only in Debug build, hence having logger that can be eliminated during build time is essential.

Chapter 22

Initializing the Logger

Default Logger is initialized at the very beginning when the library is loaded. It would be through a static function to Logger class. Since the debug logger module is slightly sophisticated than a pure macro based ‘C’ implementations.

```
#ifdef DEBUG
Logger::initialize();
#endif
```

This will create default logger like ‘DummyLogger’ where every message is ignored, and any other requested Logger with default log level.

Chapter 23

Using the default Logger

The default logger in Release mode is DummyLogger, a logger gets created, but wont be usable. The default logger in Debug mode is ConsoleLogger, which throws all the messages to the console (if it has one, via ostream).

Chapter 24

Creating new logger

During initialization a default logger is created by calling `Logger::initialize()`

Chapter 25

Design

Let us first differentiate between logger, backend, message, priority etc. A Logger class is the overall logging system, connects to a backend (could be anything, a console based logging, simple file-based, xml or json etc).

Logger gets a message (as in `Message` class), each message has a message priority or level (`LogLevel` class).

25.1 Message class

Messages can be compared to see if the priority is less/greater or equal to the other. Message class carries information such as - module, there can be many modules trying to send message to a logger. - text, actual message that is to be sent - priority, level Message's priority or `LogLevel` - timestamp, optional - thread id, optional - thread name, optional - file name, at which the message has been kept (relative to project). - line num, Line number in file

25.2 Log Level class

25.3 Logger Type class

Chapter 26

Use Case

Named logger support: A Named logger is just not the default logger, can be a module specific logger.

```
#define WARN  Logger::getDefaultLogger()->warn()

util::Logger &logger = Logger::getLogger("cipher");

WARN << "This path is not supposed to be reached\n";
```

Chapter 27

Use Case 2

A Default logger with macro.

```
#define LOG     Logger::getDefaultLogger()->log()
#define DEBUG   Logger::getDefaultLogger()->log()

LOG << "This is expected print" ;

DEBUG << "value of abcd %d\n" << bcd << "\n";
```

```
template<>
Logger::Stream& LOG(std::string& str)
{
        Logger::getDefaultLogger()->log() << str;
}
```

Part VIII

Dynamic Loading Feature

Dynamic loading of library is a necessary feature to support allow extensible library. Here the extension is for supporting new devices that may arrive in near future, but library itself doesn't have to have support for immediate use.

Chapter 28

Dynamic library loading

Libraries are built as part of the “Provider Kit”, herein referred to as PK. The PK provides features / functionalities that are not already part of the library. Also it helps reduce the size of the linked library where the specific module, device is not present or not needed.

Libraries are provided both as static library and as dynamic loadable. This section just presents the dynamic loading part.

The cases where libraries are provided as static or archived versions (like as in .a or .lib), the library just needs to be linked at the final stage of compilation.

Static libraries usually hosts a ‘constructor’ functions which gets called at the very beginning of the program execution. The extended module registers itself as part of the library.

Chapter 29

Dynamic Feature/Class loading

Once the library is loaded, one way or another.

Chapter 30

Design

Since each operating systems implements dynamic loading differently, there is a wrapper class present in __dynlib.cc_. the implementation details are present in *impl/dynlib_linux.cc* for Linux or Unix specific loading which makes use of the *libdl.so* APIs *dlopen()*, *dlclose()* and *dlsym()* etc.

Windows specific implementation should be present in *impl/dynlib_win.c*

30.1 DynamicLibrary class

`DynamicLibrary` class supports following functionality: - `load()` - loads a library - `unload()` - unload a previously loaded library - `isLoaded()` - checks if the loading was successful - `getSymbol()` - get a symbol that is part of the symbol table. - `suffix()` - gets the library suffix for a given operating system. - `setSearchPath()` - Sets the search path for loading libraries, usually its just current directory

30.2 ClassLoader class

This class implements loading a class, but for now this portion is not implemented as the final decision on whether to allow C interface or C++ interface for the