FTMGS: Fair Traceable Multi-Group Signatures

User Manual (DRAFT)

Version 0.2

Contents

1 Introduction

The FTMGS library implements the *Fair Traceable Multi-Group Signatures* scheme as defined in [1].

Fair Traceable Multi-Group Signatures (FTMGS, pronounced: fat-mugs), is a primitive that supports anonymity with extended concerns that rise in realistic scenarios. It can be regarded as a primitive that has the flavor of anonymous signatures with various revocations but with a refined notion of access control (via multiple groups) and thus supporting anonymous activities in a fashion similar to anonymous credential systems. The main issues that make this primitive suitable to various trust relationships are:

- The group manager (GM) creates and manages a group with the help of some Fairness Authorities (FA), which are only involved when special circumstances arise.
- Users (U) join a group, and become members (M), if the group manager (GM) allows them to do so.
- Members (M) issue group signatures (on behalf of the whole group) which can be verified with the *group public key* (GPK). These group signatures provide the guarantee the their issuers are indeed members of the group, but they are anonymous and unlinkable, in the sense that there is no *direct* way to identify which member issued a given signature, nor it is even possible to link different signatures as having been issued by the same member of the group.
- Belonging to a group usually implies that members fulfill the set of privileges required by the group manager (GM) to join the group.

  In authorization and access control scenarios, issuing a group signature is a suitable way to prove, in an anonymous and unlinkable way, that the issuer fulfills a set of privileges required by the GM to join the group.

  Sometimes, in authorization scenarios, a user must simultaneously prove that is a member of several groups at tehsame time, and thus she fulfills the privileges to belong to all these groups, therefore she must issue group signatures for all of them, and at the same time she must prove that all these signatures were issued by the same real anonymous user, that is, they have not been issued by a collusion of different users belonging to different groups.
- It includes multi-group features to guarantee that several signatures have been issued by the same anonymous user with no detriment of users anonymity. This allows limited local linkability most useful in many cases (linking is user controlled).
- It includes a mechanism to dissuade the group members from sharing their private membership keys. This is very useful in increasing the incentive for better access control to anonymous credentials.
- In the undesirable case of abuse of anonymity, the group manager in collaboration with the fairness authorities provide a mechanism to identify

which member of the group actually issued a given group signature, breaking in this way with the anonymity of the misbehaved user.

Note that this breaking of anonymity can only be done if all the fairness authorities agree on doing so, because there are enough circumstances that motivate such action. However, if any of the fairness authorities thinks there are not enough reasons for that, then anonymity can not be broken. In this way, the fairness authorities become the guarantee that anonymity will be only broken when there are enough reasons to do so.

- On the other way, if a member is under suspicion, it is possible to obtain, again if the fairness authorities agree, a *member tracing key* that allows to identify the signatures that were issued by such a member of the group under suspicion. Again, the fairness authorities become the guarantee that anonymity will be only broken when there are enough reasons to do so.

- This previous mechanism can also be used to revoke membership from the group for a given member.

- Note that a single fairness authority alone cannot do the opening or revealing. In this way, a users sensitive information can be guaranteed only to be disclosed when there exist enough reasons.

2 Operational Model

The group manager creates a group with the collaboration of designated fairness authorities. A user, that has been authorized by some external procedure, is able to join the group by engaging in an interactive protocol with the group manager. The external users authentication can be based on her identity (DSA signature) or even an anonymous authentication supported by this new primitive (FTMGS signature). At the end of the procedure, the group manager gets some sensitive data regarding the new member (i.e. join transcript with authentication information), and the user gets a membership private key that enables her to issue signatures on behalf of the group.

When a user wants to carry out a transaction with a server, she sometimes has to generate a proof to show she has the required privilege. This proof usually implies that she belongs to several groups. In this case, she issues suitable signatures for the involved groups, and establishes a link among them to guarantee that they have been issued by the same single anonymous user. This proof is anonymous and unlinkable.

Under critical circumstances, fairness authorities and the judge open a signature to identify a malicious user. If necessary, they may also reveal her tracing trapdoor so that tracing agents, using the trapdoor, trace all the transactions she issued.

The user owns a single master key, and this key is embedded, when joining to the group, in every membership private key of hers. Because this master key is actually the private key corresponding to her public key (e.g., her DSA public key published via the PKI), she is dissuaded from sharing her membership private keys. Moreover, this binding also guarantees that different users have different master keys.

This master key provides a common nexus among all membership private keys that belong to each user, so that she can link any two signatures of hers by proving that the signatures have been issued by membership private keys into which the same master key is embedded. This capability of linking helps our scheme to enjoy multi-group features. Note that even when the user joins the group by means of an anonymous authentication, the join procedure forces her to use the same master key, so that the relationship between her master key and public key still holds.

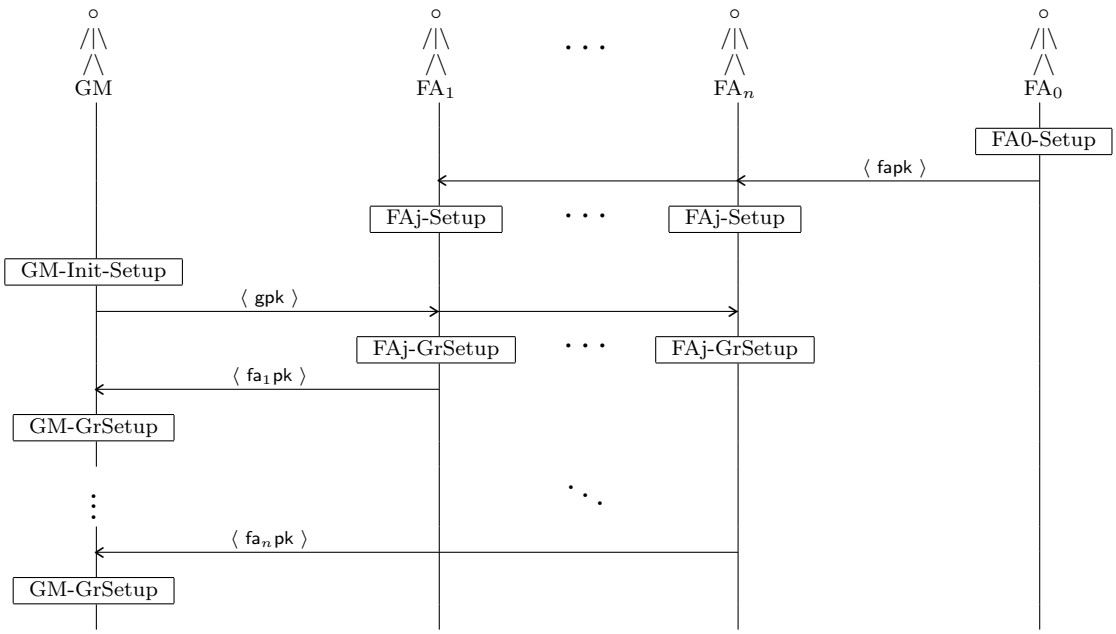
The group is created by the collaboration among the group manager and the fairness authorities. The GM is able to join new members. Fairness authorities are also involved in the setup process, in such a way that the keys related to opening and revealing are distributed among the fairness authorities. Therefore, opening a signature or revealing a member tracing key requires the agreement and participation of fariness authorities.

Opening a signature is a matter of the distributed decryption, by the fairness authorities, of part of the signature. Likewise, revealing a member tracing key is also a matter of the distributed decryption, by the fairness authorities, of the encrypted member's tracing key.

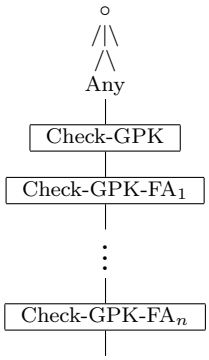
Finally, the join transcript also holds some non-repudiable proofs that allow to verify the integrity of the record, making the scheme robust against some kind of database manipulation.

3 Sequence Diagrams

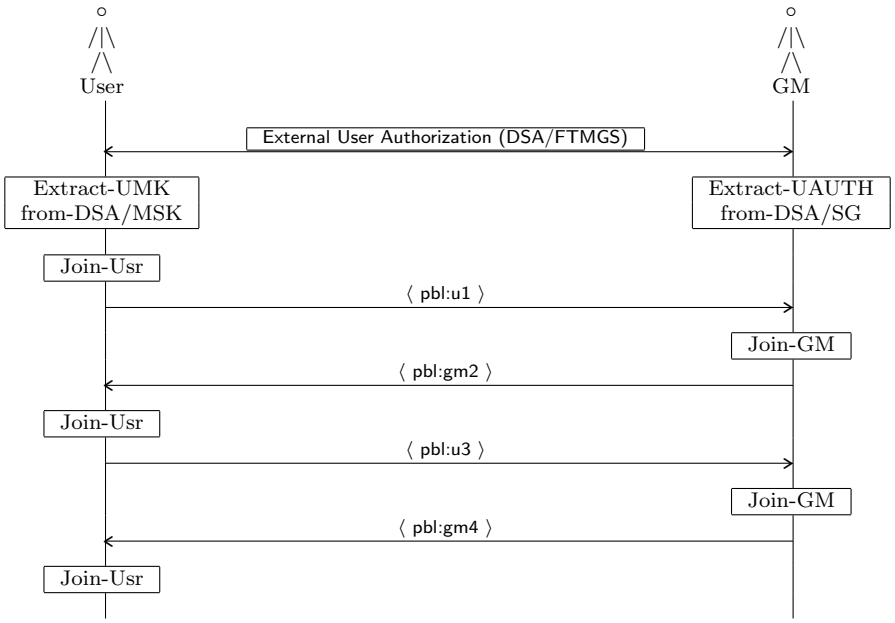
3.1 Group Setup



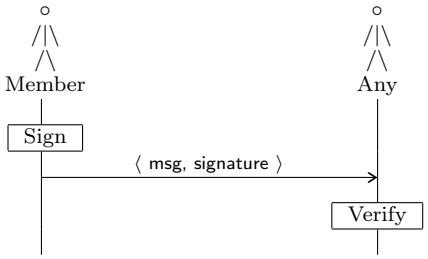
Check GPK



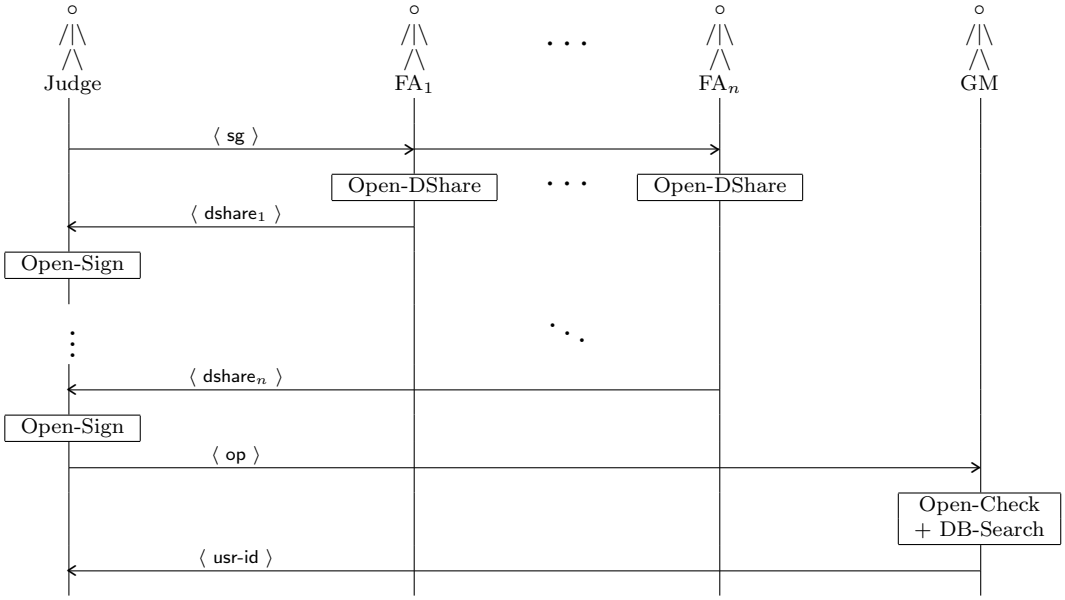
3.2 Join New Member



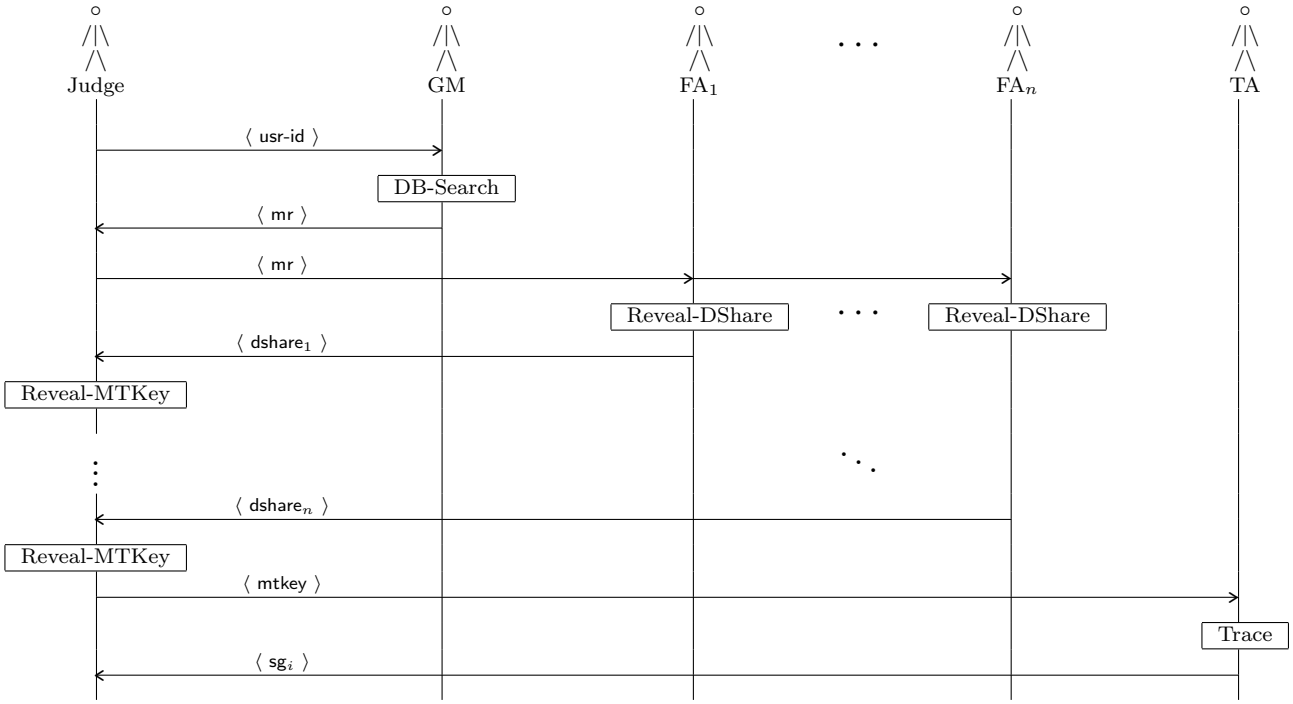
3.3 Sign / Verify



3.4 Open / Check



3.5 Reveal / Trace



4 Implementation: Features and Security Issues

- Based on: *FTMGS: Fair Traceable Multigroup Signatures* [1].

- Developed in C (ANSI-C 89) and GNU/Linux (code also for Windows)
  - Minor system dependencies (random entropy and word endianness)
  - Easy port to other platforms
- Licensed under LGPLv2.1
- It uses GMP library (LGPL) for multiple precision arithmetic
- It uses LIBTASN1 library (LGPL) for ANS1 data conversion
  - In future releases, this dependency will be removed for efficiency purposes, and ASN1 conversion will be provided internally.
- It uses SHA library code from IETF RFC-6234 (license included in the `licenses` directory)
- All functions are re-entrant
- Signatures of Knowledge follows the specified in Traceable Signatures [4].
- Non-Adaptive Drawing of Random Powers follows the specified in Traceable Signatures [4].
- Random Number generator based on *NIST-SP-800-90* document (based on SHA)
  - Entropy source at seeding is `/dev/random` for `True_Entropy`, and `/dev/urandom` for `Pseudo_Entropy` in a Linux environment, and `CryptGenRandom` in a Windows environment.
  - It implements Hash_DRBG based on SHA-256, with the following parameters: HSStrength = 256, MinEntropy = 256, SeedLen = 440.
  - Validation based on FIPS-140-2
- Sophie-Germain Prime Numbers based on [3].
- Miller-Rabin number of tests based on FIPS-186-3 recomendations
  - High-Security: (modulus factors, sophie-germain primes)
  - Low-Security: (exponents for: Group: hj ; Join: ei; FA-PrivKey: xj)

High Security

Nbits	512	1024	2048	>2048
Ntests	40	40	56	64

Low Security

Nbits	512	1024	2048	>2048
Ntests	7	5	4	4

- FTMGS Security Parameters:

Nu	1024	2048	3072
K	128	256	512

- Accepts DSA User Authentication in Join. DSA [1024, 2048, 3072].
- Precomputations enabled/disabled at compile time
- Current Development Status: Alpha

- Things To-Do:
  - Change the names of API functions and types to make them shorter.
  - Define and request an ASN.1 Object Identifier (OID) number for FTMGS
  - Prepare a battery of tests
  - Accepting RSA User Authentication in Join
  - Checking that a RSA modulus lacks of small prime factors
  - Distributed generation of RSA modulus with unknown factorization
  - Improving the automatic building and installation (GNU/Linux)
  - Developing installation packages for Debian [optional]
  - Porting the building and installation to other systems (Windows)

5 Performace

FTMGS Modular Exponentiations

		Precomp	No-Precomp	
Join		50	64	
Join	(Join+UsrAuth)	55	69	
Sign		19	19	
Vrfy		22	28	
Open		NFAS×9	NFAS×9	
Check		11	15	[1×VrfyJoinLog]
Check	(Join+UsrAuth)	14	18	
Reveal		NFAS×20+12	NFAS×24+16	[(NFAS+1)×VrfyJoinLog]
Reveal	(Join+UsrAuth)	NFAS×23+15	NFAS×27+19	
Trace		1	1	
Claim		2	2	[1×check(y==g^x)]
VrfyClaim		3	3	
Link		4	4	[2×check(y==g^x)]
VrfyLink		6	6	
VrfyJoinLog		11	15	
VrfyJoinLog	(Join+UsrAuth)	14	18	

FTMGS Timings (in seconds) [Pentium 32 bits 2GHz]

		Precomp	No-Precomp	
Join		0.33	0.42	[+ searching for prime e_i]
Join	(Join+UsrAuth)	0.35	0.44	[+ searching for prime e_i]
Sign		0.05	0.05	
Vrfy		0.05	0.08	
Open		NFAS×0.02+0.01	NFAS×0.02+0.01	
Check		0.07	0.11	[1×VrfyJoinLog]
Check	(Join+UsrAuth)	0.08	0.12	
Reveal		NFAS×0.34+0.10	NFAS×0.39+0.13	[(NFAS+1)×VrfyJoinLog]
Reveal	(Join+UsrAuth)	NFAS×0.35+0.10	NFAS×0.39+0.15	
Trace		0.01	0.01	
Claim		0.01	0.01	[1×check(y==g^x)]
VrfyClaim		0.01	0.01	
Link		0.01	0.01	[2×check(y==g^x)]
VrfyLink		0.01	0.01	
VrfyJoinLog		0.07	0.11	
VrfyJoinLog	(Join+UsrAuth)	0.08	0.12	

FTMGS Timings (in seconds) [Pentium 64 bits 3.2GHz]

		Precomp	No-Precomp	
Join		0.05	0.07	
Join	(Join+UsrAuth)	0.07	0.08	
Sign		0.01	0.01	
Vrfy		0.01	0.02	
Open		NFAS×0.0025	NFAS×0.0025	
Check		0.01	0.02	[1×VrfyJoinLog]
Check	(Join+UsrAuth)	0.01	0.02	
Reveal		NFAS×0.05	NFAS×0.05	[(NFAS+1)×VrfyJoinLog]
Reveal	(Join+UsrAuth)	NFAS×0.05	NFAS×0.06	
Trace		0.00	0.00	
Claim		0.00	0.00	[1×check(y==g^x)]
VrfyClaim		0.00	0.00	
Link		0.00	0.00	[2×check(y==g^x)]
VrfyLink		0.00	0.00	
VrfyJoinLog		0.01	0.01	
VrfyJoinLog	(Join+UsrAuth)	0.01	0.02	

6 Installing the FTMGS Library

- It depends on the following external libraries: the system math library, the GNU Multiple Precision (GMP) library, and the GNU LIBTASN1 library (this latter dependency will probably be removed in the future).

- Change working directory to 'ftmgs'

```
$ cd ftmgs
```

- To cleanup the library

```
$ make cleanup
```

- To compile the library

```
$ make
```

- To install the library (as root)

```
# make install
```

- To compile the test program

```
$ gcc -o test test.c -lftmgs -ltasn1 -lgmp -lm
```

- To run the test program

```
$ ./test
```

- To uninstall the library (as root)

```
# make uninstall
```

Please, see the README file for more updated information

7 Compiling and Linking with the FTMGS Library

Source files that make use of the facilities provided by the FTMGS library should include the header file `ftmgs.h`, where the public API is defined. This header file is installed in the system include directory by the installation process, thus the file should be included by the following directive:

```
#include <ftmgs.h>
```

The object files should be linked with the FTMGS library (`libftmgs.a` or `libftmgs.so` in unix like systems), the GNU libtasn1 and GMP libraries and the system math library:

```
$ gcc -o test test.c -lftmgs -ltasn1 -lgmp -lm
```

8 Abstract Data Types and Function API

8.1 Miscellaneous Data Types

The public API of FTMGS library is defined in the header file `ftmgs.h`, which should be included.

FTMGS Version and Revision

FTMGS version and revision numbers are defined as preprocessor macros, to allow their use from both, C programs and preprocessor conditionals. They follow the following guidelines:

- Version Major: identifies changes in public API and functionality
- Version Minor: identifies internal changes that do not affect the public behavior
- Revision: for bug fixes

They are defined as follows, where a *version major* value lower than 1 identifies alpha or beta pre-releases.

```
#define FTMGS_VERSION_MAJOR     0
#define FTMGS_VERSION_MINOR     1
#define FTMGS_REVISION          0
```

Boolean Type

The boolean type and values are defined under preprocessor conditionals to avoid clashing with other definitions. Additionally, the boolean values are checked for their right definitions.

```
#ifndef BOOL_T_DEFINED__
#define BOOL_T_DEFINED__    1
typedef char bool_t;
#endif
#ifndef TRUE
#define TRUE    1
#elif TRUE == 0
#error "Bad definition of symbol TRUE"
#endif
#ifndef FALSE
#define FALSE   0
#elif FALSE != 0
#error "Bad definition of symbol FALSE"
#endif
```

The boolean values are returned from functions to indicate either successful or erroneous execution (`TRUE` or `FALSE` respectively).

Function Return Code

The type `ftmgs_retcode_t` defines the return code for incremental (iterative) functions, which can be error (`FTMGS_ERROR`), success (`FTMGS_OK`), or still unfinished (`FTMGS_UNFINISHED`), which means that the (iterative) operation is still unfinished and it needs to be executed some more times, till either error or success is found.

```
typedef enum ftmgs_retcode_t {
    FTMGS_ERROR, FTMGS_OK, FTMGS_UNFINISHED
} ftmgs_retcode_t;
```

8.2 Abstract Data Types and Function API

The public API of FTMGS library is defined in the header file `ftmgs.h`, which should be included.

All defined data in the FTMGS library, except for enumerations, are Abstract Data Types (ADT), and therefore their internal representation is hidden (and protected). This fact has several advantages.

- The internal representation is hidden, so there is no need to expose the internal data types and internal implementation.
- As the internal representation is hidden, it diminishes the possibility (and temptation) to bypass the API and dealing with the internal data representation.
- It improves the possibility of internal modifications that do not affect to the external public API.
- It improves the binary compatibility of the library, since all data are defined as pointers to the internal representation, which can change without affecting to the external pointers.

As abstract data types, except for enumerations, any variable to deal with data must be declared as a **pointer** to the hidden data representation. Each of them will be created through a function (the **constructor**) that allocates memory space to hold the internal representation, and initializes the data to a known initial state. Moreover, when such data is not useful anymore, then it is necessary to call a function (the **destructor**) to free the allocated resources associated with the internal representation. Additionally, there is also a function (the **cloner**) that allows to clone the internal representation of the abstract data.

For each type in the library, these functions are named by the name of the type followed by the word *new*, *delete* and *clone* respectively. For example, for the type `rndctx_t`, the following code defines a variable to point to the internal representation, calls the constructors, other functions from the API, cloning and finally calls the destructors:

```
#include <ftmgs.h>
int main()
{
    unsigned x;
    rndctx_t* rctx;                              /* uninitialized variable */
    rndctx_t* random_context = rndctx_t_new();  /* constructor */
    /* ... */
    bi_random_seed(random_context, PseudoEntropy);
    /* ... */
    x = bi_random_ui(10, random_context);
    /* ... */
    rctx = rndctx_t_clone(random_context);      /* cloner     */
    /* ... */
    rndctx_t_delete(random_context);            /* destructor */
    rndctx_t_delete(rctx);                      /* destructor */
    return 0;
}
```

8.3 Random Numbers

The following values define the sources of entropy for seeding the random number generator:

- `TrueEntropy`: it seeds the random number generator with real random bits from a source of entropy that uses random noise from internal devices. If there is not enough real random bits, as required, from the source of entropy, then the operation blocks until there are enough real random bits available. This mode is **required** for secure cryptographic use.

- `PseudoEntropy`: it seeds the random number generator with random bits from a source of entropy that uses random noise from internal devices. If there is not enough real random bits, as required, from the source of entropy, then the rest of required bits are generated internally following some pseudo-random number generation. This mode is **not valid** for secure cryptographic use, but it can be used for other less secure requirement scenarios.
- `NoEntropy`: it does not use any source of entropy for seeding. It seeds the random number generator with the same fixed seed, so it is useful for debugging purposes, since, as it starts with the same seed, it repeats the same sequence of generated random numbers.

```
enum entropy_src_t {
    TrueEntropy, PseudoEntropy, NoEntropy
};
```

Abstract Data Types

- `rndctx_t`: it is used to hold the internal context of the random number generator.

  It is used extensively throughout the library, and it is required to have been seeded previously.

Construction, Copy and Destruction

The following functions declare the constructor, cloner and destructor for the aforementioned abstract data type:

```
rndctx_t* rndctx_t_new();
rndctx_t* rndctx_t_clone(const rndctx_t* o);
void rndctx_t_delete(rndctx_t* p);
```

API

- `unsigned bi_random_seed(rndctx_t* rnd_ctx, unsigned entropy_src);`

  It allows to initialize and seed the context for the random number generator. The source of entropy may be any of the aforementioned values with their explained meaning. This function must be called once before using the generator context in any other function. It returns the amount of bytes used from the entropy source.

- `unsigned bi_random_reseed(rndctx_t* rnd_ctx, unsigned entropy_src);`

  It allows to re-seed the context for the random number generator. The source of entropy may be any of the aforementioned values with their explained meaning. It returns the amount of bytes used from the entropy source.

- `void bi_random_bytes(void* buf, unsigned buflen, rndctx_t* rnd_ctx);`

  It generates `buflen` random bytes that will be stored in the memory pointed to by `buf`, by using the previously seeded random context `rnd_ctx`. The memory pointed by `buf` must have been previously allocated with enough room to hold `buflen` bytes.

- `unsigned bi_random_ui(unsigned max, rndctx_t* rnd_ctx);`

  It returns a random `unsigned` number between 0 and `max` (`max` exclusive), by using the previously seeded random context `rnd_ctx`.

8.4 Group Creation

The following values define the security parameters used when creating a group:

```
enum secpar_t {
    Nu1 = 1024, Nu2 = 2048, Nu3 = 3072
};
```

Abstract Data Types

- `ftmgs_fa_pbkey_t`: it holds the modulus $\langle \hat{n} \rangle$ and generator $\langle \hat{g} \rangle$ for the paillier encryption scheme.
- `ftmgs_faj_pbkey_share_t`: it holds the fairness authority public key share $\langle \hat{y}_j \rangle$ for the paillier encryption scheme.
- `ftmgs_faj_prkey_t`: it holds the fairness authority private key $\langle \hat{o}_j \rangle$ for the paillier encryption scheme. It allows to recover the member's tracing key.
- `ftmgs_pbkey_t`: it holds the group public key $\langle n, a, a_o, b, g, h, y, \hat{n}, \hat{g}, \hat{y} \rangle$. It allows to verify group signatures, as well as provides support for all operations dealing with the group.
- `ftmgs_prkey_t`: it holds the group manager private key, the prime factors of the group modulus $\langle p, q \rangle$. It allows to join new members to the group.
- `ftmgs_faj_gr_pbkey_share_t`: it holds the fairness authority public key share $\langle y_j, h_j \rangle$ for the el-gamal encryption scheme for a given group.
- `ftmgs_faj_gr_prkey_t`: it holds the fairness authority private key $\langle o_j \rangle$ for the el-gamal encryption scheme for a given group. It allows to open signatures.

Construction, Copy and Destruction

The following functions declare the constructors, cloners and destructors for the aforementioned abstract data types:

```
ftmgs_fa_pbkey_t* ftmgs_fa_pbkey_t_new();
ftmgs_fa_pbkey_t* ftmgs_fa_pbkey_t_clone(const ftmgs_fa_pbkey_t* o);
void ftmgs_fa_pbkey_t_delete(ftmgs_fa_pbkey_t* p);

ftmgs_faj_pbkey_share_t* ftmgs_faj_pbkey_share_t_new();
tmgs_faj_pbkey_share_t* ftmgs_faj_pbkey_share_t_clone(const ftmgs_faj_pbkey_share_t* o);
void ftmgs_faj_pbkey_share_t_delete(ftmgs_faj_pbkey_share_t* p);

ftmgs_faj_prkey_t* ftmgs_faj_prkey_t_new();
ftmgs_faj_prkey_t* ftmgs_faj_prkey_t_clone(const ftmgs_faj_prkey_t* o);
void ftmgs_faj_prkey_t_delete(ftmgs_faj_prkey_t* p);

ftmgs_pbkey_t* ftmgs_pbkey_t_new();
ftmgs_pbkey_t* ftmgs_pbkey_t_clone(const ftmgs_pbkey_t* o);
void ftmgs_pbkey_t_delete(ftmgs_pbkey_t* p);

ftmgs_prkey_t* ftmgs_prkey_t_new();
ftmgs_prkey_t* ftmgs_prkey_t_clone(const ftmgs_prkey_t* o);
void ftmgs_prkey_t_delete(ftmgs_prkey_t* p);

ftmgs_faj_gr_pbkey_share_t* ftmgs_faj_gr_pbkey_share_t_new();
ftmgs_faj_gr_pbkey_share_t* ftmgs_faj_gr_pbkey_share_t_clone(const ftmgs_faj_gr_pbkey_share_t* o);
void ftmgs_faj_gr_pbkey_share_t_delete(ftmgs_faj_gr_pbkey_share_t* p);
```

```
ftmgs_faj_gr_prkey_t* ftmgs_faj_gr_prkey_t_new();
ftmgs_faj_gr_prkey_t* ftmgs_faj_gr_prkey_t_clone(const ftmgs_faj_gr_prkey_t* o);
void ftmgs_faj_gr_prkey_t_delete(ftmgs_faj_gr_prkey_t* p);
```

API

The group setup process follows the sequence specified in the diagram in section 3.1 and 3.1.

- `void ftmgs_fa0_setup_mono(ftmgs_fa_pbkey_t* fa_pk_preimage, unsigned nu, rndctx_t* rnd_ctx);`

  It allows to create an initial public key modulus and generator for the fairness authorities, where the security parameter `nu` may have any of the aforementioned values. It is necessary that the random number generator context `rnd_ctx` had been previously seeded.

  Note: This function create the public key modulus and generator for the fairness authorities for the *paillier* encryption scheme. It is necessary to be aware that the security of the *paillier* encryption scheme relies on the unknown factorization of this public modulus, and in this case, the entity that creates this modulus is able to know such factorization, and it must be, therefore, a trusted authority with an overall power over the others. Therefore, in order to improve the security of the scheme, in next versions of this library, new functions for creating such a public modulus in a distributed and collaborative manner will be incorporated.

  ◇ In a next version of this library, a `ftmgs_fa0_setup()` function will be incorporate to implement a protocol for the distributed generation of the RSA modulus [5], to be played among the involved fairness authorities.

- `void ftmgs_faj_setup(ftmgs_faj_pbkey_share_t* faj_pk, ftmgs_faj_pbkey_share_t* faj_pk_preimage, ftmgs_faj_prkey_t* faj_sk, const ftmgs_fa_pbkey_t* fa_pk, rndctx_t* rnd_ctx);`

  It is used by each fairness authority to generate the public and private keys (`faj_pk`, and `faj_sk` respectively) for a given public modulus and generator (`fa_pk`) created by using `ftmgs_fa0_setup()`. These keys are used to deal with encryption/decryption of *member tracing keys*.

  Note that `faj_pk_preimage` may be NULL, but otherwise it will hold the preimage for the public key `faj_pk`, that is `faj_pk` $\equiv$ `faj_pk_preimage`^{2}. This preimage is used by the *Group Manager* when creating the group.

  Note that by squaring the preimage, it is assured that $\hat{y}_j \in QR(n)$.

- `void ftmgs_gm_init_setup(ftmgs_pbkey_t* gpk_preimage, ftmgs_prkey_t* gsk, unsigned nu, rndctx_t* rnd_ctx);`

  It is used by the *Group Manager* (GM) to compute the modulus, generator and related terms (a preimage of the group public key `gpk`), as well as the group private key that allows the GM to join new members to the group. The security parameter `nu` may have the aforementioned values.

- `void ftmgs_faj_group_setup(ftmgs_faj_gr_pbkey_share_t* faj_gpk, ftmgs_faj_gr_pbkey_share_t* faj_gpk_preimage, ftmgs_faj_gr_prkey_t* faj_gsk, const ftmgs_pbkey_t* gpk_preimage, rndctx_t* rnd_ctx);`

It is used by each fairness authority to generate the public and private keys (`faj_gpk`, and `faj_gsk` respectively) for a given group public key (`gpk_preimage`) created by using `ftmgs_gm_init_setup()`. These keys are used to deal with encryption/decryption for *opening* signatures.

Note that `faj_gpk_preimage` may be NULL, but otherwise it will hold the preimage for the public key `faj_gpk`, that is $\texttt{faj\_gpk} \equiv \texttt{faj\_gpk\_preimage}^2$. This preimage is used by the *Group Manager* when creating the group.

Note that by squaring the preimage, it is assured that $y_j \in QR(n)$ and $h_j \in QR(n)$.

- ```
  ftmgs_retcode_t ftmgs_gm_group_setup(ftmgs_pbkey_t* gpk,
                                       ftmgs_pbkey_t* gpk_preimage,
                                       unsigned nfas,
                                       const ftmgs_fa_pbkey_t* fa_pk_preimage,
                                       const ftmgs_faj_pbkey_share_t* faj_pk_preimage,
                                       const ftmgs_faj_gr_pbkey_share_t* faj_gpk_preimage);
  ```

  It is used by the *Group Manager* (GM) to incorporate into the group public key the public key preimages for each fairness authority that will supervise *opening* and *revealing* operations fro the group, where `nfas` specifies the total number of required fairness authorities in the process.

  This function belongs to an iterative process to incorporate the public keys for all the fairness authorities, and therefore it returns `FTMGS_UNFINISHED` while the number of incorporated fairness authority's public keys is lower that the amount of required fairness authorities (`nfas`). When the process is over, then either `FTMGS_OK` or `FTMGS_ERROR` is returned to indicate success or failure in the operation.

  Note that `gpk_preimage` may be NULL, but otherwise it will hold the preimage for the group public key `gpk`, that is $\texttt{gpk} \equiv \texttt{gpk\_preimage}^2$. This preimage is used by any entity to check that the group members have the right order. Note that by squaring the preimage, it is assured that group members belong to $QR(n)$.

- ```
  bool_t ftmgs_check_gpk(const ftmgs_pbkey_t* gpk,
                         const ftmgs_pbkey_t* gpk_preimage);
  ```

  It returns `TRUE` (`1`) if the group public key (`gpk`) is well formed with respect to the preimage (`gpk_preimage`), which means that all members have the right order and belong to $QR(n)$. It returns `FALSE` (`0`) otherwise.

  $\diamond$ In a next version of this library, this function will also check that the moduli in the group public key do not have small prime factors [2].

- ```
  ftmgs_retcode_t ftmgs_check_gpk_fa(ftmgs_pbkey_t* gpk_aux,
                                     const ftmgs_pbkey_t* gpk,
                                     const ftmgs_fa_pbkey_t* fa_pk,
                                     const ftmgs_faj_pbkey_share_t* faj_pk,
                                     const ftmgs_faj_gr_pbkey_share_t* faj_gpk);
  ```

  This function belongs to an iterative process to check that the public keys for all the fairness authorities have been incorporated to the group public key, and therefore it returns `FTMGS_UNFINISHED` while the number of checked fairness authority's public keys is lower that the amount of required fairness authorities previously incorporated to the group public key. When the process is over, then either `FTMGS_OK` or `FTMGS_ERROR` is returned to indicate success or failure in the operation.

  Note that the fairness authority's public keys are used, instead of their preimages. Also note that `gpk_aux` is used to temporarily hold the incorporation of checked public keys while the iterative process is being carried out.

- ```
  unsigned ftmgs_get_nfas_reveal(const ftmgs_pbkey_t* gpk);
  ```

  It returns the number of fairness authorities required to reveal a *member tracing key*.

- `unsigned ftmgs_get_nfas_open(const ftmgs_pbkey_t* gpk);`

  It returns the number of fairness authorities required to open a signature.

8.5 Joining New Members

Abstract Data Types

- `dss_parms_t`: it holds the DSS parameters $\langle p, q, g\rangle$, used by the user when authenticated to join the group.
- `dsa_pbkey_t`: it holds the DSA user's public key $\langle y\rangle$, used by the user when authenticated to join the group.
- `dsa_prkey_t`: it holds the DSA user's private key $\langle x\rangle$, used by the user when authenticated to join the group.
- `dlogx_t`: it holds a user's master private key $\langle x\rangle$, used by the user when authenticated to join the group. It can come from the user's DSA private key, or from a group member's private key.
- `dlog_t`: it holds a user's master public key $\langle n, g, y\rangle$, used by the user when authenticated to join the group. It can come from the user's DSA public key and DSS parameters, or from a group signature issued by the member when authenticated in joining a new group.
- `ftmgs_join_prv_t`: it holds temporal user's private data generated while the iterative process of joining to a group.
- `ftmgs_join_pbl_t`: it holds temporal public data generated while the iterative process of joining a new member to a group.
- `ftmgs_mbr_ref_t`: it holds the member's reference $\langle A_i, e_i, C_i, X_i, U_i, V_i, E_i^{\wp}, \mathcal{A}_u\rangle$ (join log) kept by the group manager for each member of the group. It holds information to allow recovery, with the collaboration of the fairness authorities, of tracing keys and opening signatures, as well as non-repudiable bindings with the user.
- `ftmgs_mbr_prkey_t`: it holds the member's private key $\langle A_i, e_i, x_i, x_i'\rangle$ that allows the member to issue anonymous and unlinkable group signatures. It holds, among other data, the user's master key.

Construction, Copy and Destruction

The following functions declare the constructors, cloners and destructors for the aforementioned abstract data types:

```
dss_parms_t* dss_parms_t_new();
dss_parms_t* dss_parms_t_clone(const dss_parms_t* o);
void dss_parms_t_delete(dss_parms_t* p);

dsa_pbkey_t* dsa_pbkey_t_new();
dsa_pbkey_t* dsa_pbkey_t_clone(const dsa_pbkey_t* o);
void dsa_pbkey_t_delete(dsa_pbkey_t* p);

dsa_prkey_t* dsa_prkey_t_new();
dsa_prkey_t* dsa_prkey_t_clone(const dsa_prkey_t* o);
void dsa_prkey_t_delete(dsa_prkey_t* p);
```

```
dlogx_t* dlogx_t_new();
dlogx_t* dlogx_t_clone(const dlogx_t* o);
void dlogx_t_delete(dlogx_t* p);

dlog_t* dlog_t_new();
dlog_t* dlog_t_clone(const dlog_t* o);
void dlog_t_delete(dlog_t* p);

ftmgs_join_prv_t* ftmgs_join_prv_t_new();
ftmgs_join_prv_t* ftmgs_join_prv_t_clone(const ftmgs_join_prv_t* o);
void ftmgs_join_prv_t_delete(ftmgs_join_prv_t* p);

ftmgs_join_pbl_t* ftmgs_join_pbl_t_new();
ftmgs_join_pbl_t* ftmgs_join_pbl_t_clone(const ftmgs_join_pbl_t* o);
void ftmgs_join_pbl_t_delete(ftmgs_join_pbl_t* p);

ftmgs_mbr_ref_t* ftmgs_mbr_ref_t_new();
ftmgs_mbr_ref_t* ftmgs_mbr_ref_t_clone(const ftmgs_mbr_ref_t* o);
void ftmgs_mbr_ref_t_delete(ftmgs_mbr_ref_t* p);

ftmgs_mbr_prkey_t* ftmgs_mbr_prkey_t_new();
ftmgs_mbr_prkey_t* ftmgs_mbr_prkey_t_clone(const ftmgs_mbr_prkey_t* o);
void ftmgs_mbr_prkey_t_delete(ftmgs_mbr_prkey_t* p);
```

API

The join process follows the sequence specified in the diagram in section 3.2.

- `void extract_umk_from_dsa(dlogx_t* x, const dsa_prkey_t* dsa_sk);`

  It is used to extract the *user's master key* from a DSA private key that have been used (a DSA signature used as authentication method) when the user was authorized to join the group. This user's master key will be used in the joining process, and it will be embedded into the user's member private key.

- ```
  void extract_uauth_from_dsa(dlog_t* uauth,
                              const dsa_pbkey_t* dsa_pk,
                              const dss_parms_t* dss_parms);
  ```

  It is used to extract the *user's authentication* from a DSA public key that have been used (a DSA signature used as authentication method) when the user was authorized to join the group. This user's authentication will be used in the joining process, and it will be embedded into the member's reference with non-repudiation purposes.

- `void extract_umk_from_msk(dlogx_t* x, const ftmgs_mbr_prkey_t* msk);`

  It is used to extract the *user's master key* from a FTMGS group member's private key that have been used (a FTMGS signature used as authentication method) when the user was authorized to join the group. This user's master key will be used in the joining process, and it will be embedded into the user's member private key.

- ```
  void extract_uauth_from_sg(dlog_t* uauth,
                             const ftmgs_sign_t* sg,
                             const ftmgs_pbkey_t* gpk);
  ```

  It is used to extract the *user's authentication* from a FTMGS signature that have been used (a FTMGS signature used as authentication method) when the user was authorized to join the group. This user's authentication will be used in the joining process, and it will be embedded into the member's reference with non-repudiation purposes.

- `void extract_dsa_from_umk(dsa_prkey_t* dsa_sk, const dlogx_t* x);`

This function is used to extract (and create) a DSA private key from a *user's master key* (which can also be extracted from a *member's private key*), in this way it discourages the user from sharing her member's private keys with other users, since in this case, they will be able to recover her DSA private key and it will allow them to impersonate the user in a world-wide manner.

◇ `void extract_umk_from_rsa(dlogx_t* x, const rsa_prkey_t* sk);`

It is used to extract the *user's master key* from a RSA private key that have been used (a RSA signature used as authentication method) when the user was authorized to join the group. This user's master key will be used in the joining process, and it will be embedded into the user's member private key.

◇ This function is not currently defined in the API. It will be included in a next version of this library.

◇
```
void extract_uauth_from_rsa(dlog_t* uauth,
                            const rsa_sign_t* sg,
                            const rsa_pbkey_t* pk);
```

It is used to extract the *user's authentication* from a RSA signature that have been used (a RSA signature used as authentication method) when the user was authorized to join the group. This user's authentication will be used in the joining process, and it will be embedded into the member's reference with non-repudiation purposes.

◇ This function is not currently defined in the API. It will be included in a next version of this library.

◇ `void extract_rsa_from_umk(rsa_prkey_t* rsa_sk, const dlogx_t* x);`

This function is used to extract (and create) a RSA private key from a *user's master key* (which can also be extracted from a *member's private key*), in this way it discourages the user from sharing her member's private keys with other users, since in this case, they will be able to recover her RSA private key and it will allow them to impersonate the user in a world-wide manner.

◇ This function is not currently defined in the API. It will be included in a next version of this library.

•
```
ftmgs_retcode_t ftmgs_join_usr(ftmgs_join_prv_t* prv,
                               ftmgs_join_pbl_t* pbl,
                               ftmgs_mbr_prkey_t* msk,
                               const ftmgs_pbkey_t* gpk,
                               const dlogx_t* umk,
                               const dlog_t* u_auth,
                               rndctx_t* rnd_ctx);
```

This function belongs to an iterative protocol, at the user's side, to join a new member to a FTMGS group, and therefore it returns `FTMGS_UNFINISHED` while the protocol is still unfinished. When the protocol is over, then either `FTMGS_OK` or `FTMGS_ERROR` is returned to indicate success or failure in the operation.

The protocol follows the sequence specified in the diagram in section 3.2. The protocol starts at the user's side, then the private and public outcome (`prv` and `pbl`) are stored for the next iteration, and the public outcome (`pbl`) is sent (usually its ASN.1 DER encoding) to the group manager, which plays its part or the protocol, then the public outcome (`pbl`) is stored for the next iteration, and sent (usually its ASN.1 DER encoding) again to the user, which plays again this protocol till the end. Note that the user's side of the protocol is the starting and ending points of the protocol.

As result, if everything was fine, the user gets her *member's private key* (`msk`) that allows her to issue FTMGS group signatures, and therefore, being authenticated as a member of the group.

If the user was authorized (and authenticated) to join the group by any external means, then the *user's master key* (`umk`) and *user's authentication* (`u_auth`)

should be extracted from the external authentication, otherwise they should be `NULL`. Note that both, user and group manager, should follow the same conventions for these cases in order to play a valid join protocol.

Note that `prv` and `pbl` are used to temporarily hold respectively the private and public data between iterations while the iterative protocol is being carried out.

- ```
  ftmgs_retcode_t ftmgs_join_gm(ftmgs_join_pbl_t* pbl,
                                ftmgs_mbr_ref_t* mr,
                                const ftmgs_pbkey_t* gpk,
                                const ftmgs_prkey_t* gsk,
                                const dlog_t* u_auth,
                                rndctx_t* rnd_ctx);
  ```

  This function belongs to an iterative protocol, at the group manager's side, to join a new member to a FTMGS group, and therefore it returns `FTMGS_UNFINISHED` while the protocol is still unfinished. When the protocol is over, then either `FTMGS_OK` or `FTMGS_ERROR` is returned to indicate success or failure in the operation.

  The protocol follows the sequence specified in the diagram in section 3.2. The protocol starts at the user's side, then the private and public outcome (`prv` and `pbl`) are stored for the next iteration, and the public outcome (`pbl`) is sent (usually its ASN.1 DER encoding) to the group manager, which plays its part or the protocol, then the public outcome (`pbl`) is stored for the next iteration, and sent (usually its ASN.1 DER encoding) again to the user, which plays again this protocol till the end. Note that the user's side of the protocol is the starting and ending points of the protocol.

  As result, if everything was fine, the group manager gets the *member's reference* (`mr`) that allows the Judge (with collaboration of GM and FAs) opening and reveal operations.

  If the user was authorized (and authenticated) to join the group by any external means, then the *user's authentication* (`u_auth`) should be extracted from the external authentication, otherwise it should be `NULL`. Note that both, user and group manager, should follow the same conventions for these cases in order to play a valid join protocol.

  Note that `pbl` is used to temporarily hold the public data between iterations while the iterative protocol is being carried out.

- ```
  bool_t ftmgs_vrfy_join_log(const ftmgs_mbr_ref_t* mr,
                             const ftmgs_pbkey_t* gpk);
  ```

  It returns `TRUE` (`1`) if the member's reference (`mr`) is consistent and well formed, which means that the non-repudiable bindings and proofs still hold, and therefore the record has not been manipulated. It returns `FALSE` (`0`) otherwise.

8.6 Signing and Verifying

The following values allow to choose the operation mode for the SHA hash engine:

```
enum sha_mode_t {
    Sha1, Sha224, Sha256, Sha384, Sha512
};
```

Abstract Data Types

- `ftmgs_sign_t`: it holds a FTMGS signature $\langle T_1, \cdots, T_7, \sigma^{\wp} \rangle$ issued by a member of the group.

Construction, Copy and Destruction

The following functions declare the constructor, cloner and destructor for the aforementioned abstract data type:

```
ftmgs_sign_t* ftmgs_sign_t_new();
ftmgs_sign_t* ftmgs_sign_t_clone(const ftmgs_sign_t* o);
void ftmgs_sign_t_delete(ftmgs_sign_t* p);
```

API

The signing process follows the sequence specified in the diagram in section 3.3.

- `bool_t ftmgs_sign_dat(unsigned which_sha,`
  `ftmgs_sign_t* sg,`
  `const void* dat, unsigned datlen,`
  `const ftmgs_pbkey_t* gpk,`
  `const ftmgs_mbr_prkey_t* msk,`
  `rndctx_t* rnd_ctx);`

  It issues a FTMGS signature (`sg`) for group public key (`gpk`) with member's private key (`msk`). The SHA engine, as selected by `which_sha` from aforementioned values, is applied to some given data bytes (`dat`) of length (`datlen`).

  It returns `TRUE` (`1`) if everything was fine, and returns `FALSE` (`0`) otherwise.

  Compatibility note: when issuing the signature, the *digest* is calculated by applying the selected SHA engine to the user's data, and then the signature it is issued over this digest as specified in the next functions:
  $digest = \mathrm{SHA}_w(\underbrace{dat}_{datlen})$

- `bool_t ftmgs_vrfy_dat(unsigned which_sha,`
  `const ftmgs_sign_t* sg,`
  `const void* dat, unsigned datlen,`
  `const ftmgs_pbkey_t* gpk);`

  It returns `TRUE` (`1`) if a FTMGS signature (`sg`) can be verified for group public key (`gpk`), and returns `FALSE` (`0`) otherwise. The SHA engine, as selected by `which_sha` from aforementioned values, is applied to some given data bytes (`dat`) of length (`datlen`).

  Compatibility note: when verifying a signature, the *digest* is calculated by applying the selected SHA engine to the user's data, and then the verification it is applied over this digest as specified in the next functions:
  $digest = \mathrm{SHA}_w(\underbrace{dat}_{datlen})$

- `bool_t ftmgs_sign_dgst(ftmgs_sign_t* sg,`
  `const void* dat_digest,`
  `unsigned dat_digestlen,`
  `const ftmgs_pbkey_t* gpk,`
  `const ftmgs_mbr_prkey_t* msk,`
  `rndctx_t* rnd_ctx);`

  It issues a FTMGS signature (`sg`) for group public key (`gpk`) with member's private key (`msk`). The signature is applied to some given data digest (`dat_digest`) of length (`dat_digestlen`) that has been generated by some hashing functions 8.11.

  It returns `TRUE` (`1`) if everything was fine, and returns `FALSE` (`0`) otherwise.

  Compatibility note: when issuing the signature, the SHA hash (truncated to security parameter k bits) is applied to the following data in the same order as specified (numbers are represented as a *big-endian* byte sequence):
  $c = \mathrm{SHA}_k(digest||B_1||\cdots||B_6||\underbrace{n||\cdots||n}_{6}||g||h||T_2^{-1}||T_5||T_7||y||T_1^{-1}||a||b||a_0||T_3||T_4||T_6)$

- `bool_t ftmgs_vrfy_dgst(const ftmgs_sign_t* sg,`
  `const void* dat_digest,`
  `unsigned dat_digestlen,`
  `const ftmgs_pbkey_t* gpk);`

  It returns `TRUE` (`1`) if a FTMGS signature (`sg`) can be verified for group public key (`gpk`) , and returns `FALSE` (`0`) otherwise. The verification is applied to some given data digest (`dat_digest`) of length (`dat_digestlen`) that has been generated by some hashing functions 8.11.

  Compatibility note: when verifying a signature, the SHA hash (truncated to security parameter k bits) is applied to the following data in the same order as specified (numbers are represented as a *big-endian* byte sequence):
  $c' = \mathrm{SHA}_k(digest||B_1||\cdots||B_6||\underbrace{n||\cdots||n}_{6}||g||h||T_2^{-1}||T_5||T_7||y||T_1^{-1}||a||b||a_0||T_3||T_4||T_6)$

8.7 Signature Opening

Abstract Data Types

- `ftmgs_opensharej_t`: it holds a decryption share $\langle \hat{\omega}_{j\sigma}, \hat{\omega}_{j\sigma}^{\wp} \rangle$ of the opening of a signature.
- `ftmgs_openacc_t`: it holds the incremental product of the opening decryption shares.
- `ftmgs_open_t`: it holds the member's reference $\langle \omega_\sigma \rangle$ result of opening a signature.

Construction, Copy and Destruction

The following functions declare the constructors, cloners and destructors for the aforementioned abstract data types:

```
ftmgs_opensharej_t* ftmgs_opensharej_t_new();
ftmgs_opensharej_t* ftmgs_opensharej_t_clone(const ftmgs_opensharej_t* o);
void ftmgs_opensharej_t_delete(ftmgs_opensharej_t* p);

ftmgs_openacc_t* ftmgs_openacc_t_new();
ftmgs_openacc_t* ftmgs_openacc_t_clone(const ftmgs_openacc_t* o);
void ftmgs_openacc_t_delete(ftmgs_openacc_t* p);

ftmgs_open_t* ftmgs_open_t_new();
ftmgs_open_t* ftmgs_open_t_clone(const ftmgs_open_t* o);
void ftmgs_open_t_delete(ftmgs_open_t* p);
```

API

The opening process follows the sequence specified in the diagram in section 3.4.

- `bool_t ftmgs_open_dshare_j(ftmgs_opensharej_t* osj,`
  `const ftmgs_sign_t* sg,`
  `const ftmgs_faj_gr_pbkey_share_t* faj_gpk,`
  `const ftmgs_faj_gr_prkey_t* faj_gsk,`
  `const ftmgs_pbkey_t* gpk,`
  `rndctx_t* rnd_ctx);`

  It is used by a fairness authority (with key-pair `faj_gsk` and `faj_gpk`) to generate a decryption share (`osj`) of the opening of a FTMGS signature (`sg`) for group public key `gpk`.

  It returns `TRUE` (`1`) if everything was fine, and returns `FALSE` (`0`) otherwise.

Compatibility note: when issuing the proof of correctness, the SHA hash (truncated to security parameter k bits) is applied to the following data in the same order as specified (numbers are represented as a *big-endian* byte sequence):
$c = \mathrm{SHA}_k(digest||B_1||B_2||n||n||g||y_j||T_2||\hat{\omega}_{j\sigma})$
where $digest = \mathrm{SHA}_k(T_1||\cdots||T_7||c||s_x||s_{x'}||s_e||s_r||s_{h'})$

- `ftmgs_retcode_t ftmgs_open_sign(ftmgs_open_t* op, ftmgs_openacc_t* osa, const ftmgs_sign_t* sg, const ftmgs_opensharej_t* osj, const ftmgs_faj_gr_pbkey_share_t* faj_gpk, const ftmgs_pbkey_t* gpk);`

  This function belongs to an iterative process to incrementally open a FTMGS signature (`sg`) for all the fairness authority's opening decryption shares (`osj`), and therefore it returns `FTMGS_UNFINISHED` while the number of added decryption shares is lower that the amount of required fairness authorities. When the process is over, then either `FTMGS_OK` or `FTMGS_ERROR` is returned to indicate success or failure in the operation.

  In case of success, `op` holds the outcome of the opening operation, and `osa` is a temporary value that must be held while the iterative process is not finished. Note that if the decryption share is not properly constructed with respect to the fairness authority's public key (`faj_gpk`) and group public key (`gpk`), then the operation will fail.

- `bool_t ftmgs_open_check(const ftmgs_open_t* op, const ftmgs_mbr_ref_t* mr, const ftmgs_pbkey_t* gpk);`

  It checks if the outcome of the opening of a signature (`op`) matches a given member's reference (`mr`), and in such case, it also checks that this member's reference is consistent (`ftmgs_vrfy_join_log()`).

  It returns `TRUE` (`1`) if both match, and returns `FALSE` (`0`) otherwise.

8.8 Member Tracing

Abstract Data Types

- `ftmgs_mtkey_sharej_t`: it holds a decryption share $\langle \hat{\tau}_{ji}, \hat{\tau}_{ji}^{\wp} \rangle$ of the revealing of a member's tracing key.
- `ftmgs_mtkey_acc_t`: it holds the incremental product of the revealing decryption shares.
- `ftmgs_mtkey_t`: it holds the member's tracing key $\langle \tau_i \rangle$ result of opening a signature.

Construction, Copy and Destruction

The following functions declare the constructors, cloners and destructors for the aforementioned abstract data types:

```
ftmgs_mtkey_sharej_t* ftmgs_mtkey_sharej_t_new();
ftmgs_mtkey_sharej_t* ftmgs_mtkey_sharej_t_clone(const ftmgs_mtkey_sharej_t* o);
void ftmgs_mtkey_sharej_t_delete(ftmgs_mtkey_sharej_t* p);

ftmgs_mtkey_acc_t* ftmgs_mtkey_acc_t_new();
ftmgs_mtkey_acc_t* ftmgs_mtkey_acc_t_clone(const ftmgs_mtkey_acc_t* o);
void ftmgs_mtkey_acc_t_delete(ftmgs_mtkey_acc_t* p);

ftmgs_mtkey_t* ftmgs_mtkey_t_new();
ftmgs_mtkey_t* ftmgs_mtkey_t_clone(const ftmgs_mtkey_t* o);
void ftmgs_mtkey_t_delete(ftmgs_mtkey_t* p);
```

API

The reveal process follows the sequence specified in the diagram in section 3.5.

- `bool_t ftmgs_reveal_dshare_j(ftmgs_mtkey_sharej_t* mtk_shj, const ftmgs_mbr_ref_t* mr, const ftmgs_faj_pbkey_share_t* faj_pk, const ftmgs_faj_prkey_t* faj_sk, const ftmgs_pbkey_t* gpk, rndctx_t* rnd_ctx);`

  It is used by a fairness authority (with key-pair `faj_sk` and `faj_pk`) to generate a decryption share (`mtk_shj`) of the revealing the member's tracing key of a member's reference (`mr`) for group public key `gpk`. It also checks that this member's reference is consistent (`ftmgs_vrfy_join_log()`).

  It returns `TRUE` (`1`) if everything was fine, and returns `FALSE` (`0`) otherwise.

  Compatibility note: when issuing the proof of correctness, the SHA hash (truncated to security parameter k bits) is applied to the following data in the same order as specified (numbers are represented as a *big-endian* byte sequence):
  $c = \mathrm{SHA}_k(digest||B_1||B_2||\hat{n}^2||\hat{g}||\hat{y}_j||U_i||\hat{\tau}_{ji})$
  where $digest = \mathrm{SHA}_k(A_i||e_i||C_i||X_i||U_i||V_i||g||y||n||c||s_{x'}||s_r||s_x)$

- `ftmgs_retcode_t ftmgs_reveal_mtkey(ftmgs_mtkey_t* mtk, ftmgs_mtkey_acc_t* mtka, const ftmgs_mtkey_sharej_t* mtk_shj, const ftmgs_mbr_ref_t* mr, const ftmgs_faj_pbkey_share_t* faj_pk, const ftmgs_pbkey_t* gpk);`

  This function belongs to an iterative process to incrementally reveal a member's tracing key for a given member's reference (`mr`) for all the fairness authority's revealing decryption shares (`mtk_shj`), and therefore it returns `FTMGS_UNFINISHED` while the number of added decryption shares is lower that the amount of required fairness authorities. When the process is over, then either `FTMGS_OK` or `FTMGS_ERROR` is returned to indicate success or failure in the operation.

  In case of success, `mtk` holds the outcome of the reveal operation, and `mtka` is a temporary value that must be held while the iterative process is not finished. Note that if the decryption share is not properly constructed with respect to the fairness authority's public key (`faj_pk`) and group public key (`gpk`), then the operation will fail.

- `bool_t ftmgs_trace(const ftmgs_sign_t* sg, const ftmgs_mtkey_t* mtk);`

  It checks if the outcome of revealing a member's tracing key (`mtk`) matches a given group signature (`sg`).

  It returns `TRUE` (`1`) if both match, and returns `FALSE` (`0`) otherwise.

8.9 Claiming Authorship

Abstract Data Types

- `ftmgs_claim_t`: it holds the proof $\langle \pi^{\wp} \rangle$ of the claiming of a signature authorship.

Construction, Copy and Destruction

The following functions declare the constructor, cloner and destructor for the aforementioned abstract data type:

```
ftmgs_claim_t* ftmgs_claim_t_new();
ftmgs_claim_t* ftmgs_claim_t_clone(const ftmgs_claim_t* o);
void ftmgs_claim_t_delete(ftmgs_claim_t* p);
```

API

- `bool_t ftmgs_claim_dat(unsigned which_sha,`
  `ftmgs_claim_t* clm,`
  `const ftmgs_sign_t* sg,`
  `const void* dat,`
  `unsigned datlen,`
  `const ftmgs_pbkey_t* gpk,`
  `const ftmgs_mbr_prkey_t* msk,`
  `rndctx_t* rnd_ctx);`

  It issues a FTMGS claim (`clm`) for group public key (`gpk`) with member's private key (`msk`) for a signature (`sg`). The SHA engine, as selected by `which_sha` from aforementioned values, is applied to some given data bytes (`dat`) of length (`datlen`).

  It returns `TRUE` (`1`) if everything was fine, and returns `FALSE` (`0`) otherwise.

  Compatibility note: when issuing the claim, the *digest* is calculated by applying the selected SHA engine to the user's data, and then the claim it is issued over this digest as specified in the next functions:
  $digest = \mathrm{SHA}_w(\underbrace{dat}_{datlen})$

- `bool_t ftmgs_vrfy_claim_dat(unsigned which_sha,`
  `const ftmgs_claim_t* clm,`
  `const ftmgs_sign_t* sg,`
  `const void* dat,`
  `unsigned datlen,`
  `const ftmgs_pbkey_t* gpk);`

  It returns `TRUE` (`1`) if a FTMGS claim (`clm`) can be verified for signature (`sg`) and group public key (`gpk`), and returns `FALSE` (`0`) otherwise. The SHA engine, as selected by `which_sha` from aforementioned values, is applied to some given data bytes (`dat`) of length (`datlen`).

  Compatibility note: when verifying a claim, the *digest* is calculated by applying the selected SHA engine to the user's data, and then the verification it is applied over this digest as specified in the next functions:
  $digest = \mathrm{SHA}_w(\underbrace{dat}_{datlen})$

- `bool_t ftmgs_claim_dgst(ftmgs_claim_t* clm,`
  `const ftmgs_sign_t* sg,`
  `const void* dat_digest,`
  `unsigned dat_digestlen,`
  `const ftmgs_pbkey_t* gpk,`
  `const ftmgs_mbr_prkey_t* msk,`
  `rndctx_t* rnd_ctx);`

  It issues a FTMGS claim (`clm`) for group public key (`gpk`) with member's private key (`msk`) for a signature (`sg`). The claim is applied to some given data digest (`dat_digest`) of length (`dat_digestlen`) that has been generated by some hashing functions 8.11.

  It returns `TRUE` (`1`) if everything was fine, and returns `FALSE` (`0`) otherwise.

  Compatibility note: when issuing the claim, the SHA hash (truncated to security parameter k bits) is applied to the following data in the same order as specified (numbers are represented as a *big-endian* byte sequence):
  $c = \mathrm{SHA}_k(dgst||B_1||n||T_7||T_6)$
  where $dgst = \mathrm{SHA}_k(digest||T_1||\cdots||T_7||c||s_x||s_{x'}||s_e||s_r||s_{h'})$

- `bool_t ftmgs_vrfy_claim_dgst(const ftmgs_claim_t* clm,`
  `const ftmgs_sign_t* sg,`
  `const void* dat_digest,`
  `unsigned dat_digestlen,`
  `const ftmgs_pbkey_t* gpk);`

It returns `TRUE` (`1`) if a FTMGS claim (`clm`) can be verified for signature (`sg`) and group public key (`gpk`), and returns `FALSE` (`0`) otherwise. The verification is applied to some given data digest (`dat_digest`) of length (`dat_digestlen`) that has been generated by some hashing functions 8.11.

Compatibility note: when verifying the claim, the SHA hash (truncated to security parameter k bits) is applied to the following data in the same order as specified (numbers are represented as a *big-endian* byte sequence):
$c = \mathrm{SHA}_k(dgst||B_1||n||T_7||T_6)$
where $dgst = \mathrm{SHA}_k(digest||T_1||\cdots||T_7||c||s_x||s_{x'}||s_e||s_r||s_{h'})$

8.10 Linking Signatures

Abstract Data Types

- `ftmgs_link_t`: it holds the proof $\langle\lambda^{\wp}\rangle$ of the linking of several FTMGS signatures.

Construction, Copy and Destruction

The following functions declare the constructor, cloner and destructor for the aforementioned abstract data type:

```
ftmgs_link_t* ftmgs_link_t_new();
ftmgs_link_t* ftmgs_link_t_clone(const ftmgs_link_t* o);
void ftmgs_link_t_delete(ftmgs_link_t* p);
```

API

- ```
  bool_t ftmgs_link_dat(unsigned which_sha,
                        ftmgs_link_t* lnk,
                        const void* dat, unsigned datlen,
                        const ftmgs_sign_t* sg0,
                        const ftmgs_pbkey_t* gpk0,
                        const ftmgs_sign_t* sg1,
                        const ftmgs_pbkey_t* gpk1,
                        const ftmgs_mbr_prkey_t* msk,
                        rndctx_t* rnd_ctx);
  ```

  It issues a FTMGS link (`lnk`) with member's private key (`msk`) for two signatures (`sg0`, `sg1`) for group public keys (`gpk0`, `gpk1`). The SHA engine, as selected by `which_sha` from aforementioned values, is applied to some given data bytes (`dat`) of length (`datlen`).

  It returns `TRUE` (`1`) if everything was fine, and returns `FALSE` (`0`) otherwise.

  Compatibility note: when issuing the link, the *digest* is calculated by applying the selected SHA engine to the user's data, and then the link it is issued over this digest as specified in the next functions:
  $digest = \mathrm{SHA}_w(\underbrace{dat}_{datlen})$

- ```
  bool_t ftmgs_vrfy_link_dat(unsigned which_sha,
                             const ftmgs_link_t* lnk,
                             const void* dat, unsigned datlen,
                             const ftmgs_sign_t* sg0,
                             const ftmgs_pbkey_t* gpk0,
                             const ftmgs_sign_t* sg1,
                             const ftmgs_pbkey_t* gpk1);
  ```

  It returns `TRUE` (`1`) if a FTMGS link (`lnk`) can be verified for signatures (`sg0`, `sg1`) and group public keys (`gpk0`, `gpk1`), and returns `FALSE` (`0`) otherwise. The

SHA engine, as selected by `which_sha` from aforementioned values, is applied to some given data bytes (`dat`) of length (`datlen`).

Compatibility note: when verifying a link, the *digest* is calculated by applying the selected SHA engine to the user's data, and then the verification it is applied over this digest as specified in the next functions:
$digest = \mathrm{SHA}_w(\underbrace{dat}_{datlen})$

- ```
  bool_t ftmgs_link_m_dat(unsigned which_sha,
                          ftmgs_link_t* lnk,
                          const void* dat,
                          unsigned datlen,
                          unsigned nsg,
                          const ftmgs_sign_t* sg[],
                          const ftmgs_pbkey_t* gpk[],
                          const ftmgs_mbr_prkey_t* msk,
                          rndctx_t* rnd_ctx);
  ```

  It is the same as previous `ftmgs_link_dat()`, but applied to an array of signatures.

- ```
  bool_t ftmgs_vrfy_link_m_dat(unsigned which_sha,
                               const ftmgs_link_t* lnk,
                               const void* dat,
                               unsigned datlen,
                               unsigned nsg,
                               const ftmgs_sign_t* sg[],
                               const ftmgs_pbkey_t* gpk[]);
  ```

  It is the same as previous `ftmgs_vrfy_link_dat()`, but applied to an array of signatures.

- ```
  bool_t ftmgs_link_dgst(ftmgs_link_t* lnk,
                         const void* dat_digest,
                         unsigned dat_digestlen,
                         const ftmgs_sign_t* sg0,
                         const ftmgs_pbkey_t* gpk0,
                         const ftmgs_sign_t* sg1,
                         const ftmgs_pbkey_t* gpk1,
                         const ftmgs_mbr_prkey_t* msk,
                         rndctx_t* rnd_ctx);
  ```

  It issues a FTMGS link (`lnk`) with member's private key (`msk`) for two signatures (`sg0`, `sg1`) for group public keys (`gpk0`, `gpk1`). The link is applied to some given data digest (`dat_digest`) of length (`dat_digestlen`) that has been generated by some hashing functions 8.11.

  It returns `TRUE` (`1`) if everything was fine, and returns `FALSE` (`0`) otherwise.

  Compatibility note: when issuing the link, the SHA hash (truncated to security parameter k bits) is applied to the following data in the same order as specified (numbers are represented as a *big-endian* byte sequence):
  $c = \mathrm{SHA}_k(dgst||B_1||B_2||n_1||n_2||T_{17}||T_{16}||T_{27}||T_{26})$
  where $dgst = \mathrm{SHA}_k(digest||T_{11}||\cdots||T_{17}||c_1||s_{1x}||s_{1x'}||s_{1e}||s_{1r}||s_{1h'}||T_{21}||\cdots||T_{27}||c_2||s_{2x}||s_{2x'}||s_{2e}||s_{2r}||s_{2h'})$

- ```
  bool_t ftmgs_vrfy_link_dgst(const ftmgs_link_t* lnk,
                              const void* dat_digest,
                              unsigned dat_digestlen,
                              const ftmgs_sign_t* sg0,
                              const ftmgs_pbkey_t* gpk0,
                              const ftmgs_sign_t* sg1,
                              const ftmgs_pbkey_t* gpk1);
  ```

  It returns `TRUE` (`1`) if a FTMGS link (`lnk`) can be verified for signatures (`sg0`, `sg1`) and group public keys (`gpk0`, `gpk1`), and returns `FALSE` (`0`) otherwise.

The verification is applied to some given data digest (`dat_digest`) of length (`dat_digestlen`) that has been generated by some hashing functions 8.11.

Compatibility note: when verifying the link, the SHA hash (truncated to security parameter k bits) is applied to the following data in the same order as specified (numbers are represented as a *big-endian* byte sequence):
$c = \mathrm{SHA}_k(dgst||B_1||B_2||n_1||n_2||T_{17}||T_{16}||T_{27}||T_{26})$
where $dgst = \mathrm{SHA}_k(digest||T_{11}||\cdots||T_{17}||c_1||s_{1x}||s_{1x'}||s_{1e}||s_{1r}||s_{1h'}||T_{21}||\cdots||T_{27}||c_2||s_{2x}||s_{2x'}||s_{2e}||s_{2r}||s_{2h'})$

- ```
  bool_t ftmgs_link_m_dgst(ftmgs_link_t* lnk,
                           const void* dat_digest,
                           unsigned dat_digestlen,
                           unsigned nsg,
                           const ftmgs_sign_t* sg[],
                           const ftmgs_pbkey_t* gpk[],
                           const ftmgs_mbr_prkey_t* msk,
                           rndctx_t* rnd_ctx);
  ```

  It is the same as previous `ftmgs_link_m_dgst()`, but applied to an array of signatures.

- ```
  bool_t ftmgs_vrfy_link_m_dgst(const ftmgs_link_t* lnk,
                                const void* dat_digest,
                                unsigned dat_digestlen,
                                unsigned nsg,
                                const ftmgs_sign_t* sg[],
                                const ftmgs_pbkey_t* gpk[]);
  ```

  It is the same as previous `ftmgs_vrfy_link_m_dgst()`, but applied to an array of signatures.

8.11 Hashing

The hash API may be used to create the digest of some data that is not directly available as an array of contiguous data bytes.

The following values allow to choose the operation mode for the SHA hash engine, as well as the length (in bytes) of the hash digest:

```
enum sha_mode_t {
    Sha1, Sha224, Sha256, Sha384, Sha512
};
enum sha_size_t {
    Sha1Size = 20, Sha224Size = 28, Sha256Size = 32,
    Sha384Size = 48, Sha512Size = 64, ShaMaxSize = Sha512Size
};
```

Abstract Data Types

- `shactx_t`: it is used to hold the internal context of the SHA engine.

Construction, Copy and Destruction

The following functions declare the constructor, cloner and destructor for the aforementioned abstract data type:

```
shactx_t* shactx_t_new();
shactx_t* shactx_t_clone(const shactx_t* o);
void shactx_t_delete(shactx_t* p);
```

API

- `bool_t sha_reset(shactx_t* sha_ctx, unsigned sha_mode);`

  It resets the context for the SHA engine, as selected by `sha_mode` from the aforementioned values.

- `bool_t sha_input(shactx_t* sha_ctx, const void* dat, unsigned datlen);`

  It incrementally incorporates a sequence of `datlen` bytes stored in `dat` to the SHA context.

- `bool_t sha_result(shactx_t* sha_ctx, void* dat_digest, unsigned* digestlen);`

  It generates a *digest* of the data previously incorporated into the SHA context. This digest is stored into `dat_digest`, which should have enough allocated memory to hold the result, which is specified by `digestlen` (a maximum value requests the length specified for the previously selected SHA engine, see aforementioned values for sha sizes). As result, `digestlen` also holds the length of the generated digest.

8.12 Data Buffer

The data buffer is used to hold a buffer of bytes representing the ASN.1 DER encoding of some abstract data type.

Abstract Data Types

- `buffer_t`: it holds a buffer of bytes dynamically allocated in the heap. It holds the *data* in memory, as well as the *size* of the data currently stored in the buffer.

Construction, Copy and Destruction

The following functions declare the constructor, cloner and destructor for the aforementioned abstract data type:

```
buffer_t* buffer_t_new();
buffer_t* buffer_t_clone(const buffer_t* o);
void buffer_t_delete(buffer_t* p);
```

API

- `char* buffer_data(const buffer_t* buff);`

  It returns a pointer to the data bytes stored in the buffer.

- `unsigned buffer_size(const buffer_t* buff);`

  It returns the size of the data bytes stored in the buffer.

8.13 ASN.1 Conversion of Data Structures

The following values define the return codes from the ASN.1 conversion functions:

```
typedef enum asn1_retcode_t {
    ASN1_Success,
    ASN1_File_Not_Found,
    ASN1_Element_Not_Found,
    ASN1_Identifier_Not_Found,
    ASN1_Der_Error,
    ASN1_Value_Not_Found,
    ASN1_Generic_Error,
    ASN1_Value_Not_Valid,
    ASN1_Tag_Error,
    ASN1_Tag_Implicit,
    ASN1_Error_Type_Any,
    ASN1_Syntax_Error,
    ASN1_Mem_Error,
    ASN1_Mem_Alloc_Error,
    ASN1_Der_Overflow,
    ASN1_Name_Too_Long,
    ASN1_Array_Error,
    ASN1_Element_Not_Empty
} asn1_retcode_t;
```

API

The following functions allow to encode in ASN.1 DER each of the aforementioned abstract data types, storing the result in the aforementioned `buffer_t` buffer of bytes. They also allow to decode from a buffer of bytes in ASN.1 DER to any of the aforementioned abstract data types:

```
asn1_retcode_t asn1_enc_dssparms(buffer_t* buff, const dss_parms_t* p);
asn1_retcode_t asn1_dec_dssparms(dss_parms_t* p, const void* buff, int len);

asn1_retcode_t asn1_enc_dsapbkey(buffer_t* buff, const dsa_pbkey_t* p);
asn1_retcode_t asn1_dec_dsapbkey(dsa_pbkey_t* p, const void* buff, int len);

asn1_retcode_t asn1_enc_dsaprkey(buffer_t* buff, const dsa_prkey_t* p);
asn1_retcode_t asn1_dec_dsaprkey(dsa_prkey_t* p, const void* buff, int len);

asn1_retcode_t asn1_enc_fapbkey(buffer_t* buff, const ftmgs_fa_pbkey_t* p);
asn1_retcode_t asn1_dec_fapbkey(ftmgs_fa_pbkey_t* p, const void* buff, int len);

asn1_retcode_t asn1_enc_grpbkey(buffer_t* buff, const ftmgs_pbkey_t* p);
asn1_retcode_t asn1_dec_grpbkey(ftmgs_pbkey_t* p, const void* buff, int len);

asn1_retcode_t asn1_enc_grprkey(buffer_t* buff, const ftmgs_prkey_t* p);
asn1_retcode_t asn1_dec_grprkey(ftmgs_prkey_t* p, const void* buff, int len);

asn1_retcode_t asn1_enc_fapbkeysh(buffer_t* buff, const ftmgs_faj_pbkey_share_t* p);
asn1_retcode_t asn1_dec_fapbkeysh(ftmgs_faj_pbkey_share_t* p, const void* buff, int len);

asn1_retcode_t asn1_enc_faprkey(buffer_t* buff, const ftmgs_faj_prkey_t* p);
asn1_retcode_t asn1_dec_faprkey(ftmgs_faj_prkey_t* p, const void* buff, int len);

asn1_retcode_t asn1_enc_fagrpbkeysh(buffer_t* buff, const ftmgs_faj_gr_pbkey_share_t* p);
asn1_retcode_t asn1_dec_fagrpbkeysh(ftmgs_faj_gr_pbkey_share_t* p, const void* buff, int len);

asn1_retcode_t asn1_enc_fagrprkey(buffer_t* buff, const ftmgs_faj_gr_prkey_t* p);
asn1_retcode_t asn1_dec_fagrprkey(ftmgs_faj_gr_prkey_t* p, const void* buff, int len);

asn1_retcode_t asn1_enc_dlog(buffer_t* buff, const dlog_t* p);
asn1_retcode_t asn1_dec_dlog(dlog_t* p, const void* buff, int len);
```

```
asn1_retcode_t asn1_enc_dlogx(buffer_t* buff, const dlogx_t* p);
asn1_retcode_t asn1_dec_dlogx(dlogx_t* p, const void* buff, int len);

asn1_retcode_t asn1_enc_joinpbl(buffer_t* buff, const ftmgs_join_pbl_t* p);
asn1_retcode_t asn1_dec_joinpbl(ftmgs_join_pbl_t* p, const void* buff, int len);

asn1_retcode_t asn1_enc_joinprv(buffer_t* buff, const ftmgs_join_prv_t* p);
asn1_retcode_t asn1_dec_joinprv(ftmgs_join_prv_t* p, const void* buff, int len);

asn1_retcode_t asn1_enc_mbrref(buffer_t* buff, const ftmgs_mbr_ref_t* p);
asn1_retcode_t asn1_dec_mbrref(ftmgs_mbr_ref_t* p, const void* buff, int len);

asn1_retcode_t asn1_enc_mbrprkey(buffer_t* buff, const ftmgs_mbr_prkey_t* p);
asn1_retcode_t asn1_dec_mbrprkey(ftmgs_mbr_prkey_t* p, const void* buff, int len);

asn1_retcode_t asn1_enc_sign(buffer_t* buff, const ftmgs_sign_t* p);
asn1_retcode_t asn1_dec_sign(ftmgs_sign_t* p, const void* buff, int len);

asn1_retcode_t asn1_enc_openshare(buffer_t* buff, const ftmgs_opensharej_t* p);
asn1_retcode_t asn1_dec_openshare(ftmgs_opensharej_t* p, const void* buff, int len);

asn1_retcode_t asn1_enc_openacc(buffer_t* buff, const ftmgs_openacc_t* p);
asn1_retcode_t asn1_dec_openacc(ftmgs_openacc_t* p, const void* buff, int len);

asn1_retcode_t asn1_enc_open(buffer_t* buff, const ftmgs_open_t* p);
asn1_retcode_t asn1_dec_open(ftmgs_open_t* p, const void* buff, int len);

asn1_retcode_t asn1_enc_mtkeyshare(buffer_t* buff, const ftmgs_mtkey_sharej_t* p);
asn1_retcode_t asn1_dec_mtkeyshare(ftmgs_mtkey_sharej_t* p, const void* buff, int len);

asn1_retcode_t asn1_enc_mtkeyacc(buffer_t* buff, const ftmgs_mtkey_acc_t* p);
asn1_retcode_t asn1_dec_mtkeyacc(ftmgs_mtkey_acc_t* p, const void* buff, int len);

asn1_retcode_t asn1_enc_mtkey(buffer_t* buff, const ftmgs_mtkey_t* p);
asn1_retcode_t asn1_dec_mtkey(ftmgs_mtkey_t* p, const void* buff, int len);

asn1_retcode_t asn1_enc_claim(buffer_t* buff, const ftmgs_claim_t* p);
asn1_retcode_t asn1_dec_claim(ftmgs_claim_t* p, const void* buff, int len);

asn1_retcode_t asn1_enc_link(buffer_t* buff, const ftmgs_link_t* p);
asn1_retcode_t asn1_dec_link(ftmgs_link_t* p, const void* buff, int len);
```

9 ASN.1 Definition of FTMGS Data Structures

9.1 Size of ASN.1 DER Encoding of FTMGS Data Structures

Size (in bytes) of the ASN.1 DER encoding of FTMGS data structures for the following security parameters [Nu: 1024, K: 128].

Group Setup

ftmgs_fa_pbkey_t	ftmgs_pbkey_t	ftmgs_prkey_t	ftmgs_faj_pbkey_share_t	ftmgs_faj_prkey_t
412	1607	137	264	264

ftmgs_faj_gr_pbkey_share_t	ftmgs_faj_gr_prkey_t
266	68

Join New Member (without User's Authentication)

ftmgs_join_prv_t:u1	ftmgs_join_pbl_t:u1	ftmgs_join_pbl_t:gm2	ftmgs_join_prv_t:u3
352	751	54	207
ftmgs_join_pbl_t:u3	**ftmgs_join_pbl_t:gm4**	**ftmgs_mbr_ref_t**	**ftmgs_mbr_prkey_t**
1326	240	1282	336

Join New Member (with User's DSA Authentication)

dss_parms_t	dsa_pbkey_t	dsa_prkey_t	dlogx_t	dlog_t
290	132	22	24	399
ftmgs_join_prv_t:u1	**ftmgs_join_pbl_t:u1**	**ftmgs_join_pbl_t:gm2**	**ftmgs_join_prv_t:u3**	
340	750	54	207	
ftmgs_join_pbl_t:u3	**ftmgs_join_pbl_t:gm4**	**ftmgs_mbr_ref_t**	**ftmgs_mbr_prkey_t**	
1345	240	1688	324	

Sign / Verify

ftmgs_sign_t	ftmgs_sign_t (with user's auth)
1308	1326

Open / Check

ftmgs_opensharej_t	ftmgs_openacc_t	ftmgs_open_t
218	137	134

Reveal / Trace

ftmgs_mtkey_sharej_t	ftmgs_mtkey_acc_t	ftmgs_mtkey_t
542	267	201

Claim

ftmgs_claim_t	ftmgs_claim_t (with user's auth)
55	73

Link

ftmgs_link_t	ftmgs_link_t (with user's auth)
55	73

9.2 ASN.1 Definition of FTMGS Data Structures

```
Ftmgs { 1 2 3 4 }   -- DUMMY OID
DEFINITIONS IMPLICIT TAGS ::=
BEGIN

-- DSA
id-dsa-base OBJECT IDENTIFIER ::= {
    iso(1) member-body(2) us(840) x9-57(10040) x9cm(4)
}

id-dsa OBJECT IDENTIFIER ::= {
    id-dsa-base 1
}

id-dsa-with-sha1 OBJECT IDENTIFIER ::=  {
```

```
    id-dsa-base 3
}

-- RFC-3279
Dss-Parms ::= SEQUENCE  {
    p       INTEGER,
    q       INTEGER,
    g       INTEGER
}

-- RFC-3279
DSAPublicKey ::= INTEGER -- public key, Y

-- RFC-5958 pg.5
DSAPrivateKey ::= INTEGER -- private key, x

-- RFC-3279
Dss-Sig-Value ::=  SEQUENCE  {
    r       INTEGER,
    s       INTEGER
}

-- Object Identifiers
id-ftmgs-base OBJECT IDENTIFIER ::= {
    1 2 3 4   -- some arcs to be registered DUMMY OID
}

id-ftmgs OBJECT IDENTIFIER ::= {
    id-ftmgs-base 1
}

id-ftmgs-with-sha1 OBJECT IDENTIFIER ::=  {
    id-ftmgs-base 3
}

id-ftmgs-with-sha256 OBJECT IDENTIFIER ::=  {
    id-ftmgs-base 5
}

id-ftmgs-with-sha512 OBJECT IDENTIFIER ::=  {
    id-ftmgs-base 7
}

-- Unsigned Small Integer (16 bits)
UInt16 ::= INTEGER (0..65535)

-- Syspar Structure Definition
SysPar ::= SEQUENCE {
    nu  UInt16,
    k   UInt16
}

-- Fairness Authorities' Public Key
FAPbKey ::= SEQUENCE {
    sp      SysPar,
    nkeys   UInt16,
    n       INTEGER,
    g       INTEGER,
    y       INTEGER
}

-- Fairness Authorities' Group Public Key
```

```
FAGrPbKey ::= SEQUENCE {
    sp      SysPar,
    nkeys   UInt16,
    n       INTEGER,
    g       INTEGER,
    y       INTEGER
}

-- Group Public Key
GrPbKey ::= SEQUENCE {
    gmpk    FAGrPbKey,
    a0      INTEGER,
    a       INTEGER,
    b       INTEGER,
    h       INTEGER,
    fapk    FAPbKey
}

-- Group Manager's Private Key [JOIN]
GrPrKey ::= SEQUENCE {
    p   INTEGER,
    q   INTEGER
}

-- Fairness Authority's Public Key
FAPbKeyShare ::= SEQUENCE {
    yj      INTEGER
}

-- Fairness Authority's Private Key [REVEAL]
FAPrKey ::= SEQUENCE {
    xj      INTEGER
}

-- Fairness Authority's Group Public Key
FAGrPbKeyShare ::= SEQUENCE {
    yj      INTEGER,
    hj      INTEGER
}

-- Fairness Authority's Group Private Key [OPEN]
FAGrPrKey ::= SEQUENCE {
    xj      INTEGER
}

-- User's Master Public Key (from DSA or FTMGS-Signature)
DLog ::= SEQUENCE {
    n           INTEGER,
    g           INTEGER,
    y           INTEGER
}

-- User's Master Private Key (from DSA o FTMGS-Signature)
DLogX ::= SEQUENCE {
    x           INTEGER
}

-- Join Proof
JoinProof ::= SEQUENCE {
    c           INTEGER,
    sx1         INTEGER,
    sr          INTEGER,
```

```
    sx          INTEGER
}

JoinU1Prv ::= SEQUENCE {
    nadrp-xx    INTEGER,
    nadrp-rr    INTEGER,
    x1i         INTEGER
}

JoinU1Pbl ::= SEQUENCE {
    nadrp-C1    INTEGER,
    nadrp-C2    INTEGER,
    nadrp-c     INTEGER,
    nadrp-sx    INTEGER,
    nadrp-sr    INTEGER,
    ci          INTEGER
}

JoinGM2Pbl ::= SEQUENCE {
    nadrp-yy    INTEGER
}

JoinU3Prv ::= SEQUENCE {
    nadrp-x     INTEGER,
    xi          INTEGER
}

JoinU3Pbl ::= SEQUENCE {
    nadrp-C3    INTEGER,
    nadrp-c     INTEGER,
    nadrp-sx    INTEGER,
    nadrp-sz    INTEGER,
    nadrp-sr    INTEGER,
    ui          INTEGER,
    vi          INTEGER,
    eiproof     JoinProof
}

JoinGM4Pbl ::= SEQUENCE {
    ai          INTEGER,
    ei          INTEGER
}

JoinPbl ::= SEQUENCE {
    status  UInt16,
    data CHOICE {
        u1      [0] JoinU1Pbl,
        gm2     [1] JoinGM2Pbl,
        u3      [2] JoinU3Pbl,
        gm4     [3] JoinGM4Pbl,
        error   [4] NULL
    }
}

JoinPrv ::= SEQUENCE {
    status  UInt16,
    data CHOICE {
        u1      [0] JoinU1Prv,
        u3      [1] JoinU3Prv,
        error   [2] NULL
    }
}
```

```
-- Member's Reference
MbrRef ::= SEQUENCE {
    ai          INTEGER,
    ei          INTEGER,
    ci          INTEGER,
    xi          INTEGER,
    ui          INTEGER,
    vi          INTEGER,
    uauth   [0] DLog        OPTIONAL,
    eiproof     JoinProof
}

-- Member's Private Key
MbrPrKey ::= SEQUENCE {
    ai          INTEGER,
    ei          INTEGER,
    xi          INTEGER,
    x1i         INTEGER
}

-- Sign
Sign ::= SEQUENCE {
    t1          INTEGER,
    t2          INTEGER,
    t3          INTEGER,
    t4          INTEGER,
    t5          INTEGER,
    t6          INTEGER,
    t7          INTEGER,
    c           INTEGER,
    sx          INTEGER,
    sx1         INTEGER,
    se          INTEGER,
    sr          INTEGER,
    sh1         INTEGER
}

-- Open
Open ::= SEQUENCE {
    a           INTEGER
}

OpenShare ::= SEQUENCE {
    alphax      INTEGER,
    c           INTEGER,
    sx          INTEGER
}

OpenAcc ::= SEQUENCE {
    nshares     UInt16,
    a           INTEGER
}

-- Reveal&Trace
MTKey ::= SEQUENCE {
    n           INTEGER,
    x           INTEGER
}

MTKeyShare ::= SEQUENCE {
    alphax      INTEGER,
```

```
    c           INTEGER,
    sx          INTEGER
}

MTKeyAcc ::= SEQUENCE {
    nshares     UInt16,
    a           INTEGER
}

-- Claim
Claim ::= SEQUENCE {
    c           INTEGER,
    sx          INTEGER
}

-- Link
Link ::= SEQUENCE {
    c           INTEGER,
    sx          INTEGER
}

END
```

10 Usage Example

The attached file `test.c`, licensed under GNU GPLv2, is an example of using the FTMGS library public API.

References

[1] V. Benjumea, S. G. Choi, J. Lopez, and M. Yung. Fair Traceable Multi-Group Signatures. In G. Tsudik, editor, *FC'08: 12th. Intl. Conf. on Financial Cryptography and Data Security*, volume 5143 of *Lecture Notes in Computer Science*, pages 231–246. Springer-Verlag, Jan. 2008. Full Version: `http://eprint.iacr.org/2008/047`.

[2] J. Camenisch and M. Michels. Proving in zero-knowledge that a number is the product of two safe primes. In *EUROCRYPT'99*, vol. 1592 of LNCS, pp. 107–122, Springer-Verlag, 1999

[3] R. Cramer and V. Shoup. Signature Schemes Based on the Strong RSA Assumption. In *6th ACM Conf on Computer and Communication Security*, 1999 *ACM Transactions on Information and System Security*, May 9, 2000.

[4] A. Kiayias, Y. Tsiounis, and M. Yung. Traceable Signatures. In *EUROCRYPT'04*, pages 571–589, 2004. Full Version: `http://eprint.iacr.org/2004/007`.

[5] P. Fouque and J. Stern. Fully distributed threshold RSA under standard assumptions. In *ASIACRYPT*, 2001.