The LibTMCG Reference Manual

Version 1.3.14
9 September 2018

An Implementation of the Toolbox for Mental Card Games

Heiko Stamer <HeikoStamer@gmx.net>

This is the reference manual of LibTMCG.

Revision 20180907.

Table of Contents

1 Introduction

'LibTMCG' is a C++ library for creating secure and verifiable online card games. The library contains a sort of useful classes, algorithms, and high-level protocols to support an application programmer in writing such software. The most remarkable feature is the absence of a trusted third party (TTP), i.e., neither a central game server nor trusted hardware components are necessary. Thus, with the present library there is no need for an independent referee, because the applied protocols provide a basic level of confidentiality and verifiability by itself. Consequently, the library is well-suited for peer-to-peer (P2P) environments where no TTP is available. Of course, we cannot avoid that malicious players share information about their private cards, but the protocols ensure that the shuffle of the deck is performed randomly (presumed that at least one player is honest) and thus the cards will be distributed uniformly among the players. Further, no coalition can learn the private cards of a player against his will (except for trivial conclusions). The corresponding cryptographic problem, actually called "Mental Poker", has been studied since 1979 (Shamir, Rivest, and Adleman) by many authors. LibTMCG provides the first practical implementation of such sophisticated cryptographic protocols.

The security and the verifiability rely on advanced cryptographic techniques—the so-called zero-knowledge proofs. Using these 'building blocks' the high-level protocols minimize the effect of coalitions and preserve the confidentiality of the players' strategy, i.e., the players are not required to reveal their cards at the end of the game in order to show that they did not cheat. This important property is often required in card games like Poker, where not all cards are opened during the play and the applied individual strategy must be kept secret.

LibTMCG is *Free Software* according to the definition of the Free Software Foundation. The source code is released under the GNU *General Public License* Version 2.

1.1 Further Reading

The cryptographic background and a detailed discussion of the implementation issues are beyond the scope of this manual. The interested reader is referred to the following scientific papers:

[Sc98]: Christian Schindelhauer. *Toolbox for Mental Card Games.*
Technical Report A-98-14, University of Lübeck, 1998.
`http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.6679`

[BS03]: Adam Barnett and Nigel P. Smart. *Mental Poker Revisited.*
In K.G. Paterson (Ed.): Cryptography and Coding 2003, Lecture Notes in Computer Science 2898, pp. 370–383, 2003.
`http://dx.doi.org/10.1007/978-3-540-40974-8_29`

[Gr05]: Jens Groth. *A Verifiable Secret Shuffle of Homomorphic Encryptions.*
Cryptology ePrint Archive, Report 2005/246, 2005.
`http://eprint.iacr.org/2005/246`

[Gr10]: Jens Groth. *A Verifiable Secret Shuffle of Homomorphic Encryptions.*
Journal of Cryptology, Volume 23 Issue 4, pp. 546–579, 2010.
`http://dx.doi.org/10.1007/s00145-010-9067-9`

[HSSV09]: Sebastiaan de Hoogh, Berry Schoenmakers, Boris Skoric, and Jose Villegas. *Verifiable Rotation of Homomorphic Encryptions.*
Proceedings of Public Key Cryptography 2009, Lecture Notes in Computer Science 5443, pp. 393–410, 2009.
`http://dx.doi.org/10.1007/978-3-642-00468-1_22`

[St04]: Heiko Stamer. *Kryptographische Skatrunde.* (in German)
Offene Systeme (ISSN 1619-0114), 4:10–30, 2004.
`http://www.nongnu.org/libtmcg/OS-4-2004-openskat_rev2005.pdf`

[St05]: HEIKO STAMER. *Efficient Electronic Gambling: An Extended Implementation of the Toolbox for Mental Card Games.*
Proceedings of the Western European Workshop on Research in Cryptology (WEWoRC 2005), Lecture Notes in Informatics P-74, pp. 1–12, 2005.
`http://www.nongnu.org/libtmcg/WEWoRC2005_proc.pdf`

[FS87]: AMOS FIAT and ADI SHAMIR. *How To Prove Yourself: Practical Solutions to Identification and Signature Problems.*
Advances in Cryptology – Proceedings of CRYPTO' 86. Lecture Notes in Computer Science 263, pp. 186–194, 1987.
`http://dx.doi.org/10.1007/3-540-47721-7_12`

[BR93, BR95]: MIHIR BELLARE and PHILLIP ROGAWAY. *Random Oracles are Practical: A Paradigm for Designing Efficient Protocols.*
Proceedings of 1st ACM Conference on Computer and Communications Security, pp. 62–73, 1993.
`http://cseweb.ucsd.edu/~mihir/papers/ro.html`

[BR96]: MIHIR BELLARE and PHILLIP ROGAWAY. *The Exact Security of Digital Signatures – How to Sign with RSA and Rabin.*
Advances in Cryptology – Proceedings of EUROCRYPT' 96, Lecture Notes in Computer Science 1070, pp. 399–416, 1996.
`http://web.cs.ucdavis.edu/~rogaway/papers/exact.pdf`

[Bo01]: DAN BONEH. *Simplified OAEP for the RSA and Rabin Functions.*
Advances in Cryptology – Proceedings of CRYPTO' 01, Lecture Notes in Computer Science 2139, pp. 275–291, 2001.
`http://crypto.stanford.edu/~dabo/abstracts/saep.html`

[GMR98]: ROSARIO GENNARO, DANIELE MICCIANCIO, and TAL RABIN. *An Efficient Non-Interactive Statistical Zero-Knowledge Proof System for Quasi-Safe Prime Products.*
Proceedings of 5th ACM Conference on Computer and Communication Security, pp. 67–72, 1998.
`https://cseweb.ucsd.edu/~daniele/papers/GMR.html`

[JL00]: STANISLAW JARECKI and ANNA LYSYANSKAYA. *Adaptively Secure Threshold Cryptography: Introducing Concurrency, Removing Erasures.*
Advances in Cryptology – Proceedings of EUROCRYPT' 00, Lecture Notes in Computer Science 1807, pp. 221–242, 2000.
`http://www.iacr.org/archive/eurocrypt2000/1807/18070223-new.pdf`

[CKPS01]: CHRISTIAN CACHIN, KLAUS KURSAWE, FRANK PETZOLD, and VICTOR SHOUP. *Secure and Efficient Asynchronous Broadcast Protocols.*
Advances in Cryptology – Proceedings of CRYPTO' 01, Lecture Notes in Computer Science 2139, pp. 524–541, 2001.
`http://shoup.net/papers/ckps.pdf`

1.2 Getting Started

This manual describes the application programming interface of LibTMCG. All relevant data types, public classes and security parameters are explained. The reader should have an advanced knowledge in applied cryptography and C++ programming. Reference is made at this point to the famous *Handbook of Applied Cryptography* for a brief introduction on the first topic. For the underlying communication model and some broadcast primitives the outstanding textbook *Introduction to Reliable and Secure Distributed Programming* and the corresponding exercises are recommended.

This document follows, in style and rarely in phrasing, the *Libgcrypt Reference Manual*. Thus don't be surprised, if you recognize some obvious analogies.

1.3 Preliminaries

The most card games are played with a regular card deck, i.e., cards where the pattern on the front side (face) determines the card type (e.g. the King of Spades ♠, the Seven of Hearts ♡, the Ace of Club ♣, or the Jack of Diamonds ♢) and where the back sides (face down) of all cards are indistinguishable. Only such 'regular' card decks are supported by LibTMCG and the provided card encoding schemes.

1.3.1 Terminology

The following list defines some common terms that are subsequently used in the manual.

Player: A *player* is an active participant in an electronic card game.

Observer: An *observer* is an passive party who watches the game.

Card: The term *card* means the electronic representation of a playing card.

Card Type: The *card type* is a nonnegative integer which corresponds to the pattern on the picture side of a real playing card. We assume here that such a natural encoding always exists.

Masking: *Masking* is a process which aim is to transform the card representation such that the input card and the result cannot be linked (except for trivial conclusions). Roughly speaking, masking is the (re-)encryption of a card representation such that the original card type is preserved.

Card Secret: The *card secret* contains all random values used in a masking operation. These values must be kept secret until the card is publicly revealed. Otherwise the corresponding output of the masking transformation is linkable and other players may learn the card type.

Open Card: An *open card* is a card whose type can be easily determined by all players and usually by observers as well.

Masked Card: A *masked card* (also known as face-down card) is a card whose type is unknown to a subset of players. It can be only revealed, if all players cooperate in a common computation of the type.

Private Card: A *private card* is a card whose type is only known to its owner. As long as the owner does not corporate the type of the private card stays hidden to all other players (except for trivial conclusions).

Stack: A *stack* is a not necessarily disjoint subset of the whole card deck.

Prover and Verifier: The *prover* is a player who shows some property to another party called *verifier*. For example, he wants to show that a masking operation was performed correctly, i.e., the card type is preserved by the transformation.

1.3.2 Security

"Mental Poker" solutions cannot prevent that malicious players exchange private information, for example, by telephone or Internet chat. Cryptographic protocols can only minimize the effect of such colluding parties and should try to protect the confidentiality for honest players. But even this small protection often relies on number-theoretical assumptions which are only believed to be true, i.e., problems like factoring products of large primes or computing discrete logarithms are only believed to be hard. That means, strict mathematical proofs[1] for the hardness of these problems are not known, and it is not very likely that such proofs will ever be found. However,

[1] For instance, a "tight reduction" to a known hard problem in the sense of complexity theory.

almost all public key cryptosystems rely on such assumptions and therefore you should not care about this issue, as long as reasonable security parameters are chosen and practical quantum computers are out of range.

LibTMCG was originally designed to provide security in the "honest-but-curious" (aka "semi-honest" or passive) adversary model. That means, all participants follow the protocol instructions properly but they may gather information and share them within a coalition to obtain an advantage in the game. Thus we are basically not concerned with robustness and availability issues which are hard to solve in almost asynchronous environments like the Internet. However, the most operations are verifiable such that cheating can be detected. To obtain this verifiability, the protocols deploy so-called zero-knowledge proofs which yield no further knowledge but the validity of a statement. The soundness error of these proofs is bounded by a fixed security parameter κ. Depending on your application scenario this parameter should be chosen such that there is a reasonable tradeoff between the cheating probability (which is less or equal than $2^{-\kappa}$) and the produced computational and communication complexity. LibTMCG also uses so-called zero-knowledge proofs of knowledge due to Bellare and Goldreich (see *On defining proofs of knowledge*, Advances in Cryptology – Proceedings of CRYPTO' 92, 1992), however, for convenience we will not further distinguish between these building blocks. Finally, some of the protocols (e.g. the efficient shuffle argument by Groth) are only zero-knowledge with respect to a so-called "honest verifier" who follows all protocol instructions faithfully. Since version 1.2.0 of LibTMCG we use a two-party version of a distributed coin flipping protocol by Jarecki and Lysyanskaya [JL00] to protect against malicious verifiers in that case.

Unfortunately, in practice there is another substantial problem with the detection of cheaters. It requires that an authenticated broadcast channel has been set up, where all players have read/write access. There exist protocols (so-called "reliable broadcast" or even "atomic broadcast") for creating such a channel, however, only under the additional condition that the number of parties t who act faulty or even malicious (so-called "Byzantine adversary") is reasonable small. In a full asynchronous environment like the Internet resilience is achievable for $t < n/3$ only, where n denotes the total number of parties in the protocol. LibTMCG provides a well-known protocol due to Bracha (see *An asynchronous [(n - 1)/3]-resilient consensus protocol*, Proceedings of 3rd ACM Symposium on Principles of Distributed Computing, 1984) in a slightly optimized variant by Cachin, Kursawe, Petzold, and Shoup [CKPS01]. Please note that in most cases the application programmer must decide, where the use of a broadcast channel is neccesary and appropriate. Thus, without reliable broadcast you should take into account that not necessarily the player acting as prover is the source of evil, if a verification procedure fails. This level of uncertainty is the main reason for our still limited adversary model.

Note that it is not known, whether the used protocols retain their zero-knowledge property, if they are composed and executed in a concurrent setting. Thus the application programmer should be careful and avoid parallel protocol sessions. It is an open research project to create a protocol suite whose security can be proven in the UC-framework of Canetti (see *Universally Composable Security: A New Paradigm for Cryptographic Protocols*, Cryptology ePrint Archive: Report 2000/067) or even more elaborated UC-frameworks (see e.g. Dennis Hofheinz and Victor Shoup: *GNUC: A New Universal Composability Framework*, Cryptology ePrint Archive: Report 2011/303). Furthermore, the protocols should employ concurrent zero-knowledge proofs (see Cynthia Dwork, Moni Naor, and Amit Sahai: *Concurrent Zero-Knowledge*, Journal of the ACM 51(6):851–898, 2004).

Please also note, that in some protocols the Fiat-Shamir heuristic [FS87] is used to turn interactive special honest verifier zero-knowledge arguments resp. proofs into non-interactive versions in the random oracle model. However, there are some theoretical (see e.g. Nir Bitansky, Dana Dachman-Soled, Sanjam Garg, Abhishek Jain, Yael Tauman Kalai, Adriana Lopez-Alt, and Daniel Wichs: *Why 'Fiat-Shamir for Proofs' Lacks a Proof*, TCC 2013, LNCS 7785, 2013) and pratical (see David Bernhard, Olivier Pereira, Bogdan Warinschi: *How Not to Prove Yourself:*

Pitfalls of the Fiat-Shamir Heuristic and Applications to Helios, ASIACRYPT 2012, LNCS 7658, 2012) concerns that show the insecurity of Fiat-Shamir heuristic w.r.t. the soundness of the argument resp. proof. That means, if deterministic hash functions are used as public coin [BR93], then the random oracle assumption obviously does not hold and therefore a malicious prover can manipulate the challenges in order to cheat and thus violates the soundness property. On the other hand, the Fiat-Shamir heuristic, and in general the non-interactivness of the transformed protocols, protect against a malicious verifier. Thus it is another important measure to deal with the limitation of honest verifier zero-knowledge proofs resp. arguments of knowledge without loosing their efficiency. However, non-interactive protocols are necessarily malleable (when used without unique identifiers), and the cheating verifier can generate a convincing proof of knowledge by copying one sent by the prover in a previous iteration of the protocol. This issue must be adressed by the application programmer, for example, by using fresh randomness in each card or stack operation which should be verifiable.

LibTMCG was carefully implemented with respect to timing attacks (see Paul C. Kocher: *Cryptanalysis of Diffie-Hellman, RSA, DSS, and other cryptosystems using timing attacks*, CRYPTO '95, LNCS 963, 1995). Therefore we loose some efficiency, e.g., during modular exponentiations. However, it is strongly recommended to leave the timing attack protection turned on, unless you know exactly where it is really not needed.

> **Security Advice:** We have implemented all cryptographic primitives according to the cited research papers and to the best of our knowledge. However, we can not eliminate any possibility of contained flaws or bugs, because the implementation of such complex protocols is always an error-prone process. Moreover, the scientific results are sometimes controversial or even wrong. Thus we encourage readers with advanced cryptographic background to review given references and the source code of LibTMCG. Please report any complaint or correction proposal!

1.3.3 Communication

Most cryptographic protocols are designed for a *synchronous* communication model, i.e., there is a known upper bound on message transmission delays. That means, the time period between the point at which a protocol message is sent and the point at which the message is delivered is smaller than this bound. Additionally, often the assumption is made that the computation proceeds in synchronized rounds and that the parties are connected by a complete network of private (i.e. untappable and authenticated) point-to-point channels.

There is an importand distinction between *fully synchronous* and *partially synchronous* communication model with respect to coverage and the resulting adverserial power. However, a detailed discussion of such issues is beyond the scope of this manual. The reader is referred to the famous textbook *Introduction to Reliable and Secure Distributed Programming* for an introduction and discussion on that topic.

1.4 Preparation

LibTMCG depends on three other basic libraries. Therefore you will need the corresponding development files to build LibTMCG and your application properly. The following list gives a short exposition of the used features and specifies the required versions:

- GNU Multiple Precision Arithmetic Library (`libgmp`), Version $\geq$ 4.2.0

  The library provides a powerful framework for performing arbitrary precision arithmetic on integers. Further reasons for choosing this dependency are the license compatibility, the portability, the vital maintenance, and of course, the reasonable performance.
- GNU Crypto Library (`libgcrypt`), Version $\geq$ 1.6.0

  The library provides some basic cryptographic algorithms (e.g. SHA-256, AES256, ElGamal, DSA, RSA) and an easily accessible interface for cryptographically strong pseudo random numbers. If a version $\geq 1.7.0$ is found, then also the hash function SHA-3 will be used.

- GNU Privacy Guard Error Code Library (`libgpg-error`), Version ≥ 1.12

  This library defines common error values, e.g., returned by the GNU Crypto Library.

We suppose that the reader is familiar with these libraries because their correct installation, configuration, and usage is crucial to the security of the entire application.

1.5 Header Files and Name Spaces

The interface definitions of classes, data types, and security parameters[2] are provided by the central header file `libTMCG.hh`. You have to include this file in all of your sources, either directly or through some other included file. Thus often you will simply write:

```
#include <libTMCG.hh>
```

There are no uniform C++ name spaces for the most parts of the library. Some classes and data types have the common prefix `TMCG_*` resp. `VTMF_*` while others are composed of the author names and an abbreviation of the title from the related research paper. Further there are some function names and types that are prepended by `tmcg_*`. Those interfaces should be used with care, because later they may be removed or replaced.

1.6 Building Sources

If you want to compile a source file including the `libTMCG.hh` header, you must make sure that the compiler can find it in the directory hierarchy. This is achieved by adding the path of the corresponding directory to the compilers include file search path.

However, the path to the include file has been determined at the time the source is configured. To solve this problem, LibTMCG ships with a small helper program `libTMCG-config` that knows the path to the include file and a few other configuration options. The options that need to be added to the compiler invocation are output by the `--cflags` option to `libTMCG-config`. The following example shows how it can be used at the command line:

```
g++ -c foo.cc `libTMCG-config --cflags`
```

Adding the output of '`libTMCG-config --cflags`' to the compilers command line will ensure that the compiler can find the LibTMCG header file.

A similar problem occurs when linking your program with LibTMCG. Again, the compiler has to find the library files. Therefore the correct installation path has to be added to the library search path. To achieve this, the option `--libs` of `libTMCG-config` can be used. For convenience, this option also outputs all other stuff (e.g. required third-party libraries) that is required to link your program with LibTMCG (in particular, the '`-lTMCG`' option).

The example shows how to link `foo.o` with LibTMCG to a program called `foo`:

```
g++ -o foo foo.o `libTMCG-config --libs`
```

Of course, you can also combine both examples to a single command by calling the shell script `libTMCG-config` with both options:

```
g++ -o foo foo.c `libTMCG-config --cflags --libs`
```

[2] Some security parameters are fixed at compile time of LibTMCG. Please don't change anything unless you know exactly what you are doing! Beside the apparent security concerns you will probably break the compatibility with other applications using LibTMCG.

1.6.1 Building Sources Using GNU Automake

You can use GNU Automake to obtain automatically generated Makefiles. If you do so then you do not have to care about finding and invoking the `libTMCG-config` script at all. LibTMCG provides an Automake extension that does all the stupid work for you.

`AM_PATH_LIBTMCG` ([*minimum-version*], [*action-if-found*], [*action-if-not-found*]) [Macro]

Check whether LibTMCG (at least version *minimum-version*, if given) exists on the host system. If it is found, execute *action-if-found*, otherwise do *action-if-not-found*.

Additionally, the macro defines `LIBTMCG_CFLAGS` to the flags needed for compilation in order to find the necessary header files, and `LIBTMCG_LIBS` to the corresponding linker flags.

You can use the defined variables in your `Makefile.am` as follows:

```
AM_CPPFLAGS = $(LIBTMCG_CFLAGS)
LDADD = $(LIBTMCG_LIBS)
```

1.7 Initializing the Library

The first step is the initialization of LibTMCG. The following function must be invoked early in your program, i.e., before you make use of any other capability of LibTMCG.

`bool init_libTMCG` (`const bool` *force_secmem* `=false`, `const bool` *gmp_secmem* `=false`, `const size_t` *max_secmem* `=32768`) [Function]

The function checks whether the installed third-party libraries match their required versions. Further it initializes them and returns `true`, if everything was sound. Otherwise `false` is returned and an appropriate error message is sent to `std::cerr`.

The three optional arguments define the behaviour concerning the allocation of secure memory (i.e. memory that is not paged out to disk and that is overwritten by zeros before released) from libgcrypt. By default no secure memory is used. If `force_secmem` is true, than those parts of LibTMCG that use the GNU Crypto Library will allocate and use secure memory for private keys or other secrets. However, the most classes, algorithms, and protocols of LibTMCG does not respect this option yet, because they store their secrets with the GNU Multiple Precision Arithmetic Library. With the second option `gmp_secmem` the default memory allocator of this library is replaced to use secure memory. Unfortunately, there is no way to specify whether a big integer needs secure memory and thus all memory is allocated in this fashion. This may lead to out of memory aborts, because the allocated secure memory is limited (currently 32kB). The limit of libgcrypt can be adjusted by the third parameter `max_secmem`, however, probably there are restrictions of the operating system (cf. RLIMIT_MEMLOCK).

Additionally, the function `version_libTMCG` returns a string containing the version number of the library in a common format. It is strongly recommended to check, whether the installed version matches your requirements.

`const std::string` *version_libTMCG* () [Function]

This function returns the version of the library in the format *major.minor.revision*.

Last but not least, there is a function `identifier_libTMCG` which returns an identifier of LibTMCG including the version, copyright mark and license.

`const std::string` *identifier_libTMCG* () [Function]

This function returns an identifier of the library.

2 Application Programming Interface

Now we start with a description of some important global symbols and structures.

2.1 Preprocessor Defined Global Symbols

Please note that the following macros are fixed at compile time of LibTMCG and cannot be changed by your application. They are only provided here for informational purposes.

`TMCG_MR_ITERATIONS` [Macro]
: Defines the number of iterations for the Miller-Rabin primality test. The default value is `64` which implies a soundness error probability $\leq 4^{-64}$.

`TMCG_MAX_ZNP_ITERATIONS` [Macro]
: Defines the maximum number of iterations for the prover in cut-and-choose style zero-knowledge protocols of Schindelhauer's toolbox. The default value is `80` which limits the soundness error probability to $\geq 2^{-80}$, however, it protects against some obvious denial-of-service attacks from a malicious verifier.

`TMCG_GROTH_L_E` [Macro]
: Defines the security parameter ℓ_e of Groth's (interactive) shuffle argument [Gr05]. The default value is `80` which implies a soundness error probability $\leq 2^{-80}$. For the intended purposes of LibTMCG this seems to be reasonable.

`TMCG_DDH_SIZE` [Macro]
: Defines the security parameter (finite field size in bit) of the group G which is used by the card encoding scheme of Barnett and Smart [BS03]. The underlying assumptions are DDH, CDH, and DLOG. The default value is `2048`.

`TMCG_DLSE_SIZE` [Macro]
: Defines the security parameter (subgroup size in bit) of the group G which is used by the card encoding scheme of Barnett and Smart [BS03]. The underlying assumptions are DLSE (related to DDH) and DLOG. The default value is `256`.

`TMCG_AIO_HIDE_SIZE` [Macro]
: Defines the security parameter for hiding the length of integers in derived classes from `aiounicast`. The default value is `256`.

`TMCG_GCRY_MD_ALGO` [Macro]
: Defines the main message digest algorithm (i.e. hash function $h()$) for digital signatures with PRab [BR96] and mask generation for Rabin encryption with SEAP [Bo01] in `TMCG_SecretKey`. This algorithm is also used for the construction of a special hash function $g()$, which is needed for the Fiat-Shamir heuristic [FS87]. Recently we switched[1] to the hash function SHA-256 (default value `GCRY_MD_SHA256`[2]) for improved collision resistance. Thus we gain a security level of approximately 2^{128}, assuming that a birthday-attack is the best known attack against this message digest.

  Please note that the security of the non-interactive zero-knowledge proofs resp. arguments (NIZK) is proved in the so-called random oracle model (ROM), i.e., we suppose that the instantiated hash function $g()$ behaves like an ideal random function (which obviously cannot hold in a real world scenario with deterministic computations). However, this assumption seems to be reasonable, if the underlying hash function is collision-resistant and if it is carefully implemented with respect to other instantiations [BR93].

---

[1] In former versions of LibTMCG the default value of this symbol was `GCRY_MD_RMD160`, i.e. the hash algorithm RIPEMD-160 (see Dobbertin, Bosselaers, Preneel: *RIPEMD-160, a strengthened version of RIPEMD*, 1996), which is a function that has only an output length of 160 bit.

[2] This is also a constant defined by the GNU Crypto Library.

TMCG_GCRY_MAC_ALGO [Macro]

Defines the message authentication algorithm for authenticated channels established by the class aiounicast. The default value is GCRY_MAC_HMAC_SHA256[3], i.e. the HMAC based scheme with hashing algorithm SHA-256.

TMCG_GCRY_ENC_ALGO [Macro]

Defines the symmetric encryption algorithm (sometimes also called cipher) for private channels established by the class aiounicast. The default value is GCRY_CIPHER_AES256[4], i.e. the cipher AES256, which is used by LibTMCG in CFB (Cipher Feedback) mode.

TMCG_KEYID_SIZE [Macro]

Defines the length (in characters w.r.t. TMCG_MPZ_IO_BASE) for the distinctive suffix of the unique TMCG key identifier. The default value is 8 which spans a reasonable name space for at least 2^{20} different TMCG keys (see TMCG_PublicKey). However, sometimes it is required to use even smaller sizes due to artificial protocol restrictions (e.g. the IRC nickname is sometimes restricted to 9 characters).

Each key identifier starts with the string "ID" followed by the decimal encoded value of TMCG_KEYID_SIZE and the appended carret symbol "^". The final suffix contains TMCG_KEYID_SIZE alphanumerical characters from the self signature of TMCG key. This signature has enough entropy included to be used as unique key identifier.

TMCG_KEY_NIZK_STAGE1 [Macro]

Defines the security parameter (number of iterations) of the NIZK proof [GMR98] (stage 1) which convince all verifiers that the TMCG key was correctly generated. The default value is 16 which implies a soundness error probability $\leq d^{-16}$, where $d = \gcd(m, \phi(m))$ and m is part of the public key. This parameter is only relevant for the card encoding scheme of Schindelhauer, where the key has a very special format.

TMCG_KEY_NIZK_STAGE2 [Macro]

Defines the security parameter (number of iterations) of the NIZK proof [GMR98] (stage 2) which convince all verifiers that the TMCG key was correctly generated. The default value is 128 which implies a soundness error probability $\leq 2^{-128}$. This parameter is only relevant for the card encoding scheme of Schindelhauer.

TMCG_KEY_NIZK_STAGE3 [Macro]

Defines the security parameter (number of iterations) of the NIZK proof [Sc98] (stage 3) which convince all verifiers that the TMCG key was correctly generated. The default value is 128 which implies a soundness error probability $\leq 2^{-128}$. This parameter is only relevant for the card encoding scheme of Schindelhauer.

TMCG_LIBGCRYPT_VERSION [Macro]

Defines the required minimum version number of the GNU Crypto Library. The default value is "1.6.0". During the initialization of LibTMCG (see init_libTMCG) it is checked, whether the version number of the linked shared object fulfills this condition.

TMCG_LIBGMP_VERSION [Macro]

Defines the required minimum version number of the GNU Multiple Precision Arithmetic Library. The default value is "4.2.0". During the initialization of LibTMCG (see init_libTMCG) it is checked, whether the version number provided by the header file gmp.h and used at compile time of LibTMCG fulfills this condition.

[3] This is also a constant defined by the GNU Crypto Library.

[4] This is also a constant defined by the GNU Crypto Library.

`TMCG_MAX_CARDS` [Macro]
Defines the maximum number of stackable cards. The default value is `1024`.

`TMCG_MAX_PLAYERS` [Macro]
Defines the maximum number of players. The default value is `32`. This parameter is only relevant for the card encoding scheme of Schindelhauer.

`TMCG_MAX_TYPEBITS` [Macro]
Defines the maximum number of bits to represent the card type in the scheme of Schindelhauer. On the other hand, this value determines the maximum size of the message space in the scheme of Barnett and Smart. The default value is `10` which implies that 1024 different card types are possible. For each type some memory will be allocated, thus this value should modified very carefully.

`TMCG_MPZ_IO_BASE` [Macro]
Defines the input and output base of the `std::iostream` operators `<<` and `>>` which is used to encode large integers (`mpz_t`). The former value was `36` which was some years ago the largest base supported by the GNU Multiple Precision Arithmetic Library. Since version 1.2.0 of LibTMCG the new default value is `62`.

`TMCG_PRAB_K0` [Macro]
Defines the security parameter k_0 (in characters) of the PRab scheme [BR96]. The default value is `20` which implies a security level around 2^{80}.

`TMCG_QRA_SIZE` [Macro]
Defines the security parameter (size of the modulus $m = p \cdot q$ in bit) of the TMCG key. The underlying assumptions are QRA and FACTOR. The default value is `2048`. This parameter is only relevant for TMCG keys and Schindelhauer's encoding scheme.

`TMCG_SAEP_S0` [Macro]
Defines the security parameter s_0 (in characters) of the Rabin-SAEP scheme [Bo01]. The default value is `20` which implies a security level around 2^{80} against CCA (chosen-ciphertext attack).

`TMCG_HASH_COMMITMENT` [Macro]
Defines whether shortened commitments are used in the shuffle verification procedure of Schindelhauer [Sc98]. The default value is `true`, because this will decrease the communication complexity significantly. However, as an immediate consequence the soundness property is violated, if the hash function `TMCG_GCRY_MD_ALGO` is broken.

`TMCG_MAX_FPOWM_T` [Macro]
Defines the maximum size of admissible exponents (in bit) used by our fast exponentiation procedures. The default value is `2048`. Note that this parameter has a strong influence on the amount of memory allocated by LibTMCG since it determines the size of the precomputed tables. However, it should be at least greater or equal than `TMCG_DDH_SIZE` and `TMCG_QRA_SIZE` in order to support the possible exponents of common finite field sizes.

`TMCG_MAX_FPOWM_N` [Macro]
Define the maximum number of different bases for doing the above precomputation. This value is a trade-off between fast exponentiation for all possible bases and memory allocation. Currently it is only relevant for the generators $g_1, \ldots, g_n$ in Groth's variant of Pedersen commitment scheme (see Section 2.2.3.4 [GrothVSSHE], page 34). The default value is `256`.

`TMCG_MAX_SSRANDOMM_CACHE` [Macro]
Define the maximum size of the cache for function `mpz_ssrandomm`. The cache must be proper initialized and is useful in interactive protocols, where entropy is limited and a lot of very

secure randomness is required immediately. Thus some values should be aquired and cached before the protocol starts. The default value is `256`.

2.2 Basic Structures

This section describes all public data types, communication interfaces, and classes of high-level protocols that are necessary to create a secure card game. Private methods and only internally used members are not explained.

2.2.1 Data Types

LibTMCG provides several data structures for cards, stacks, and cryptographic keys.

2.2.1.1 Encoding Schemes for Cards

There exist two different encoding schemes that can be used for the digital representation of playing cards. In the scheme of Schindelhauer [Sc98] the type of a card is shared among the players through bit-wise representation by quadratic (non-)residues. Thus the security relies on the well-known QRA (Quadratic Residuosity Assumption). Unfortunately, the size of a card grows linearly in the number of players and logarithmically in the number of card types. Recently the much more efficient solution of Barnett and Smart [BS03] has been implemented. This encoding works on a cyclic group of prime order and requires that the DDH (Decisional Diffie-Hellman Assumption) holds there.

For both schemes LibTMCG provides a structure whose name contains the suffix `Card`. This data type is used to represent an open or even a masked card. Further, there is a corresponding structure whose name contains the suffix `CardSecret`. This data type is used to represent the secret values involved in a card masking operation.

Because of the reduced computational and communication complexity (see [St05] for more details) the usage of the second card encoding scheme, i.e. `VTMF_Card` and `VTMF_CardSecret`, is highly recommended.

`TMCG_Card` [Data type]

This `struct` represents a card in the encoding scheme of Schindelhauer [Sc98]. The type of the card is shared among the players by quadratic residues and non-residues, respectively. Thus the security relies on the Quadratic Residuosity Assumption.

`std::vector< std::vector<MP_INT> > z` [Member of `TMCG_Card`]

This $k \times w$-matrix encodes the type of the corresponding card in a shared way. For each of the k players there is a separate row and for each of the w bits in the binary representation of the type there is a column. The elements are numbers from the group $\mathbf{Z}^{\circ}_{m_i}$ where m_i is the public modulus of the ith player.

`TMCG_Card ()` [Constructor on `TMCG_Card`]

This default constructor initializes the card with an empty 1×1-matrix. Later the method `TMCG_Card::resize` can be used to enlarge the card representation.

`TMCG_Card (size_t` *k*, `size_t` *w*`)` [Constructor on `TMCG_Card`]

This constructor initializes the card with an empty $k \times w$-matrix. The parameter k is the number of players and w is the maximum number of bits used by the binary representation of the card type.

`TMCG_Card (const TMCG_Card&` *that*`)` [Constructor on `TMCG_Card`]

This is a simple copy-constructor and *that* is the card to be copied.

`TMCG_Card& = (const TMCG_Card&` *that*`)` [Operator on `TMCG_Card`]

This is a simple assignment-operator and *that* is the card to be assigned.

`bool == (const TMCG_Card& that)` [Operator on `TMCG_Card`]
This operator tests two card representations for equality.

`bool != (const TMCG_Card& that)` [Operator on `TMCG_Card`]
This operator tests two card representations for inequality.

`void resize (size_t k, size_t w)` [Method on `TMCG_Card`]
This method resizes the representation of the card. The current content of the member `z` will be released and a new $k \times w$-matrix is created. The parameter k is the number of players and w is the maximum number of bits used by the binary representation of the card type.

`bool import (std::string s)` [Method on `TMCG_Card`]
This method imports the content of the member `z` from the correctly formatted input string *s*. It returns `true`, if the import was successful.

`~TMCG_Card ()` [Destructor on `TMCG_Card`]
This destructor releases all occupied resources.

`std::ostream& << (std::ostream& out, const TMCG_Card& card)` [Operator on `TMCG_Card`]
This operator exports the content of the member `z` (of the given `TMCG_Card` *card*) to the output stream *out*.

`std::istream& >> (std::istream& in, TMCG_Card& card)` [Operator on `TMCG_Card`]
This operator imports the content of the member `z` (of the given `TMCG_Card` *card*) from the input stream *in*. The data has to be delimited by a newline character. The `failbit` of the stream is set, if any parse error occurred.

`TMCG_CardSecret` [Data type]
This `struct` represents the secret used for a card masking operation in the original encoding scheme of Schindelhauer [Sc98].

`std::vector< std::vector<MP_INT> > r` [Member of `TMCG_CardSecret`]
This $k \times w$-matrix encodes the first part of the secret. For each of the k players there is a separate row and for each of the w bits in the binary representation of the corresponding card type there is a column. The elements are numbers from the group $\mathbf{Z}^{\circ}_{m_i}$ where m_i is the public modulus of the ith player.

`std::vector< std::vector<MP_INT> > b` [Member of `TMCG_CardSecret`]
This $k \times w$-matrix encodes the second part of the secret. For each of the k players there is a separate row and for each of the w bits in the binary representation of the corresponding card type there is a column. The elements are simply numbers from $\{0, 1\}$.

`TMCG_CardSecret ()` [Constructor on `TMCG_CardSecret`]
This default constructor initializes both members with an empty 1×1-matrix. Later the method `TMCG_CardSecret::resize` can be used to enlarge the card representation.

`TMCG_CardSecret (size_t k, size_t w)` [Constructor on `TMCG_CardSecret`]
This constructor initializes both members with an empty $k \times w$-matrix. The parameter k is the number of players and w is the maximum number of bits used by the binary representation of the corresponding card type.

`TMCG_CardSecret (const TMCG_CardSecret& that)` [Constructor on `TMCG_CardSecret`]
This is a simple copy-constructor and *that* is the secret to be copied.

`TMCG_CardSecret& = (const TMCG_CardSecret& that)` [Operator on `TMCG_CardSecret`]

This is a simple assignment-operator and *that* is the secret to be assigned.

`void resize (size_t k, size_t w)` [Method on `TMCG_CardSecret`]

This method resizes the representation of the secret. The current content of the members `r` and `b` will be released and new $k \times w$-matrices are created. The parameter k is the number of players and w is the maximum number of bits used by the binary representation of the corresponding card type.

`bool import (std::string s)` [Method on `TMCG_CardSecret`]

This method imports the content of the members `r` and `b` from the correctly formatted input string *s*. It returns `true`, if the import was successful.

`~TMCG_CardSecret ()` [Destructor on `TMCG_CardSecret`]

This destructor releases all occupied resources.

`std::ostream& << (std::ostream& out, const TMCG_CardSecret& cardsecret)` [Operator on `TMCG_CardSecret`]

This operator exports the content of the members `r` and `b` (of the given `TMCG_CardSecret` *cardsecret*) to the output stream *out*.

`std::istream& >> (std::istream& in, TMCG_CardSecret& cardsecret)` [Operator on `TMCG_CardSecret`]

This operator imports the content of the members `r` and `b` (of the given `TMCG_CardSecret` *cardsecret*) from the input stream *in*. The data has to be delimited by a newline character. The `failbit` of the stream is set, if any parse error occurred.

`VTMF_Card` [Data type]

This `struct` represents a card in the encoding scheme of Barnett and Smart [BS03]. Here we use the discrete logarithm based instantiation of their general cryptographic primitive VTMF (Verifiable k-out-of-k Threshold Masking Function). The security relies on the DDH assumption in the underlying abelian group G.

`mpz_t c_1` [Member of `VTMF_Card`]

This is the first part of the encrypted card type. It is an element from the underlying group G.

`mpz_t c_2` [Member of `VTMF_Card`]

This is the second part of the encrypted card type. It is also an element from the underlying group G.

`VTMF_Card ()` [Constructor on `VTMF_Card`]

This default constructor initializes an empty card where the members `c_1` and `c_2` are set to zero.

`VTMF_Card (const VTMF_Card& that)` [Constructor on `VTMF_Card`]

This is a simple copy-constructor and *that* is the card to be copied.

`VTMF_Card& = (const VTMF_Card& that)` [Operator on `VTMF_Card`]

This is a simple assignment-operator and *that* is the card to be assigned.

`bool == (const VTMF_Card& that)` [Operator on `VTMF_Card`]

This operator tests two card representations for equality.

`bool != (const VTMF_Card& that)` [Operator on `VTMF_Card`]

This operator tests two card representations for inequality.

`bool import (std::string s)` [Method on `VTMF_Card`]

This method imports the content of the members `c_1` and `c_2` from a correctly formatted input string *s*. It returns `true`, if the import was successful.

`~VTMF_Card ()` [Destructor on `VTMF_Card`]

This destructor releases all occupied resources.

`std::ostream& << (std::ostream& out, const VTMF_Card& card)` [Operator on `VTMF_Card`]

This operator exports the content of the members `c_1` and `c_2` (of the given `VTMF_Card` *card*) to the output stream *out*.

`std::istream& >> (std::istream& in, VTMF_Card& card)` [Operator on `VTMF_Card`]

This operator imports the content of the members `c_1` and `c_2` (of the given `VTMF_Card` *card*) from the input stream *in*. The data has to be delimited by a newline character. The `failbit` of the stream is set, if any parse error occurred.

`VTMF_CardSecret` [Data type]

This `struct` represents the secrets used in the card masking operation by the encoding scheme of Barnett and Smart [BS03].

`mpz_t r` [Member of `VTMF_CardSecret`]

This member is the exponent (randomizer) used in the masking operation. It should be chosen uniformly and randomly from $\mathbf{Z}_q$ where q is the order of the finite abelian group G for which the DDH assumption holds.

According to the results of Koshiba and Kurosawa (see *Short Exponent Diffie-Hellman Problems*, PKC 2004, LNCS 2947) the length of this exponent can be shorten to a more efficient size (e.g. 160 bit), if the corresponding generator of G is adjusted as well. Under the additional DLSE (Discrete Logarithm with Short Exponents) assumption the DDH problem in G seems to be still hard. By such an optimization trick we gain a great performance advantage for almost all modular exponentiations that are computed during the masking operation, if the VTMF primitive was instantiated by the later explained class `BarnettSmartVTMF_dlog_GroupQR`. Furthermore, the size of the card secret is substantially reduced which results in an improved communication complexity.

`VTMF_CardSecret ()` [Constructor on `VTMF_CardSecret`]

This default constructor initializes the secret with an empty member `r`.

`VTMF_CardSecret (const VTMF_CardSecret& that)` [Constructor on `VTMF_CardSecret`]

This is a simple copy-constructor and *that* is the secret to be copied.

`VTMF_CardSecret& = (const VTMF_CardSecret& that)` [Operator on `VTMF_CardSecret`]

This is a simple assignment-operator and *that* is the secret to be assigned.

`bool import (std::string s)` [Method on `VTMF_CardSecret`]

This method imports the content of the member `r` from the correctly formatted input string *s*. It returns `true`, if the import was successful.

`~VTMF_CardSecret ()` [Destructor on `VTMF_CardSecret`]

This destructor releases all occupied resources.

`std::ostream& << (std::ostream& out, const VTMF_CardSecret& cardsecret)` [Operator on `VTMF_CardSecret`]

This operator exports the content of the member `r` (of the given `VTMF_CardSecret` *cardsecret*) to the output stream *out*.

`std::istream& >> (std::istream& in, VTMF_CardSecret& cardsecret)` [Operator on `VTMF_CardSecret`]

This operator imports the content of the member `r` (of the given `VTMF_CardSecret` *cardsecret*) from the input stream *in*. The data has to be delimited by a newline character. The `failbit` of the stream is set, if any parse error occurred.

2.2.1.2 Stacks

All of the following data types are generic containers that can be instantiated as C++ templates with the former explained `Card` and `CardSecret` data types, respectively. Note the maximum number of stackable data is upper-bounded by `TMCG_MAX_CARDS`. There is no error reported, if this limit is exceeded.

`TMCG_Stack<CardType>` [Data type]

This `struct` is a simple container for cards of the specified *CardType*. Currently, the elements can be either of type `TMCG_Card` or `VTMF_Card` depending on which kind of encoding scheme is used. The `TMCG_Stack` structure is mainly used to represent a stack of masked cards, i.e., playing cards that are stacked in a face-down manner. It can be either a public stack where all participants have access to or even a private stack, e.g. the players' hand. If the corresponding card types are known it can also serve as an "open stack", although `TMCG_OpenStack` is more suitable in that case.

`std::vector<CardType> stack` [Member of `TMCG_Stack`]

This is the container that is used internally for storing the cards.

`TMCG_Stack ()` [Constructor on `TMCG_Stack`]

This default constructor initializes an empty stack.

`TMCG_Stack& = (const TMCG_Stack<CardType>& that)` [Operator on `TMCG_Stack`]

This is a simple assignment-operator and *that* is the stack to be assigned.

`bool == (const TMCG_Stack<CardType>& that)` [Operator on `TMCG_Stack`]

This operator tests two stacks for equality. It checks whether the sizes of the stacks and the contained cards are equal with respect to the implied order.

`bool != (const TMCG_Stack<CardType>& that)` [Operator on `TMCG_Stack`]

This operator tests two stacks for inequality. It returns `true`, if either the sizes do not match or at least two corresponding cards are not equal.

`const CardType& [] (const size_t n)` [Operator on `TMCG_Stack`]

This operator provides read-only random access to the contained cards. It returns a const-reference to the *n*th card from the top of the stack.

`CardType& [] (const size_t n)` [Operator on `TMCG_Stack`]

This operator provides random access to the contained cards. It returns a reference to the *n*th card from the top of the stack.

`size_t size ()` [Method on `TMCG_Stack`]

This method returns the size of the stack.

`void push (const CardType& c)` [Method on `TMCG_Stack`]

This method pushes the card *c* to the back of the stack.

`void push (const TMCG_Stack<CardType>& s)` [Method on `TMCG_Stack`]

This method pushes the stack *s* to the back of the stack.

`void push (const TMCG_OpenStack<CardType>& s)` [Method on `TMCG_Stack`]
: This method pushes the cards of the open stack *s* to the back of the stack.

`bool empty ()` [Method on `TMCG_Stack`]
: This method returns `true`, if the stack is empty.

`bool pop (CardType& c)` [Method on `TMCG_Stack`]
: This method removes a card from the back and stores the data in *c*. It returns `true`, if the stack was not empty and thus *c* contains useful data.

`void clear ()` [Method on `TMCG_Stack`]
: This method clears the stack, i.e., it removes all cards.

`bool find (const CardType& c)` [Method on `TMCG_Stack`]
: This method returns `true`, if the card *c* was found in the stack.

`bool remove (const CardType& c)` [Method on `TMCG_Stack`]
: This method removes the top-most card from the stack which is equal to *c*. It returns `true`, if the card was found and successfully removed.

`size_t removeAll (const CardType& c)` [Method on `TMCG_Stack`]
: This method removes every card from the stack which is equal to *c*. It returns the number of removed cards.

`bool import (std::string s)` [Method on `TMCG_Stack`]
: This method imports the stack from the correctly formatted input string *s*. It returns `true`, if the import was successful.

`~TMCG_Stack ()` [Destructor on `TMCG_Stack`]
: This destructor releases all occupied resources.

`std::ostream& << (std::ostream& out, const TMCG_Stack<CardType>& stack)` [Operator on `TMCG_Stack`]
: This operator exports the given *stack* to the output stream *out*.

`std::istream& >> (std::istream& in, TMCG_Stack<CardType>& stack)` [Operator on `TMCG_Stack`]
: This operator imports the given *stack* from the input stream *in*. The data has to be delimited by a newline character. The `failbit` of the stream is set, if any parse error occurred.

`TMCG_OpenStack<CardType>` [Data type]
: This `struct` is a simple container for cards of the specified *`CardType`* whose types are known. The elements are pairs where the first component is the type and the second component is the corresponding card. The card type is represented by a `size_t` integer. Currently, the cards can be either of type `TMCG_Card` or `VTMF_Card` depending on which kind of encoding scheme is used.

`std::vector<std::pair<size_t, CardType> > stack` [Member of `TMCG_OpenStack`]
: This is the container that is used internally for storing the pairs.

`TMCG_OpenStack ()` [Constructor on `TMCG_OpenStack`]
: This default constructor initializes an empty stack.

`TMCG_OpenStack& = (const TMCG_OpenStack<CardType>& that)` [Operator on `TMCG_OpenStack`]
: This is a simple assignment-operator and *that* is the stack to be assigned.

`bool == (const TMCG_OpenStack<CardType>& that)` [Operator on `TMCG_OpenStack`]

This operator tests two stacks for equality. It checks whether the types, the sizes, and the contained cards are equal with respect to the stack order.

`bool != (const TMCG_OpenStack<CardType>& that)` [Operator on `TMCG_OpenStack`]

This operator tests two stacks for inequality. It returns `true`, if either the sizes resp. types do not match or at least two corresponding cards are not equal.

`const std::pair<size_t, CardType>& [] (const size_t n)` [Operator on `TMCG_OpenStack`]

This operator provides read-only random access to the contained pairs. It returns a const-reference to the *n*th pair from the top of the stack.

`std::pair<size_t, CardType>& [] (const size_t n)` [Operator on `TMCG_OpenStack`]

This operator provides random access to the contained pairs. It returns a reference to the *n*th pair from the top of the stack.

`size_t size ()` [Method on `TMCG_OpenStack`]

This method returns the size of the stack.

`void push (const std::pair<size_t, CardType>& p)` [Method on `TMCG_OpenStack`]

This method pushes the pair *p* to the back of the stack. The first component is the type and the second component is the corresponding card representation.

`void push (const size_t type, const CardType& c)` [Method on `TMCG_OpenStack`]

This method pushes a pair to the back of the stack. The parameter *type* is the card type and *c* is the corresponding card representation.

`void push (const TMCG_OpenStack<CardType>& s)` [Method on `TMCG_OpenStack`]

This method pushes the pairs of the stack *s* to the back of this stack.

`bool empty ()` [Method on `TMCG_OpenStack`]

This method returns `true`, if the stack is empty.

`bool pop (size_t& type, CardType& c)` [Method on `TMCG_OpenStack`]

This method removes a pair from the back of the stack. It stores the card type in *type* and the representation in *c*. It returns `true`, if the stack was not empty and thus *type* and *c* contain useful data.

`void clear ()` [Method on `TMCG_OpenStack`]

This method clears the stack, i.e., it removes all pairs.

`bool find (const size_t type)` [Method on `TMCG_OpenStack`]

This method returns `true`, if a pair with the first component *type* was found in the stack.

`bool remove (const size_t type)` [Method on `TMCG_OpenStack`]

This method removes the top-most pair with the first component *type* from the stack. It returns `true`, if such a pair was found and successfully removed.

`size_t removeAll (const size_t type)` [Method on `TMCG_OpenStack`]

This method removes every pair from the stack whose first component is equal to *type*. Further it returns the number of removed pairs.

`bool move (const size_t type, TMCG_Stack<CardType>& s)` [Method on `TMCG_OpenStack`]

This method moves the top-most card representation of the given *type* to another stack *s*. It returns `true`, if such a pair was found and successfully moved.

`~TMCG_OpenStack ()` [Destructor on `TMCG_OpenStack`]

This destructor releases all occupied resources.

`TMCG_StackSecret<CardSecretType>` [Data type]

This `struct` is a simple container for the secrets involved in the masking operation of cards. Additionally, the permutation of a corresponding shuffle of the stack is stored. The elements are pairs where the first component is a permutation index of type `size_t` and the second component is a card secret of the specified *CardSecretType*. Currently, such secrets can be either of type `TMCG_CardSecret` or `VTMF_CardSecret` depending on which kind of encoding scheme is used.

`std::vector<std::pair<size_t, CardSecretType> > stack` [Member of `TMCG_StackSecret`]

This is the container that is used internally for storing the pairs.

`TMCG_StackSecret ()` [Constructor on `TMCG_StackSecret`]

This default constructor initializes an empty stack secret.

`TMCG_StackSecret& = (const TMCG_StackSecret<CardSecretType>& that)` [Operator on `TMCG_StackSecret`]

This is a simple assignment-operator and *that* is the stack secret to be assigned.

`const std::pair<size_t, CardSecretType>& [] (const size_t n)` [Operator on `TMCG_StackSecret`]

This operator provides read-only random access to the contained pairs. It returns a const-reference to the *n*th pair from the top of the stack secret.

`std::pair<size_t, CardSecretType>& [] (const size_t n)` [Operator on `TMCG_StackSecret`]

This operator provides random access to the contained pairs. It returns a reference to the *n*th pair from the top of the stack secret.

`size_t size ()` [Method on `TMCG_StackSecret`]

This method returns the size of the stack secret.

`void push (const size_t index, const CardSecretType& cs)` [Method on `TMCG_StackSecret`]

This method pushes a pair to the back of the stack secret. The parameter *index* is the permutation index and *cs* is the corresponding card secret.

`void clear ()` [Method on `TMCG_StackSecret`]

This method clears the stack secret, i.e., it removes all pairs.

`size_t find_position (const size_t index)` [Method on `TMCG_StackSecret`]

This method searches for a given permutation index in the stack secret. It returns the corresponding position[5] in the stack secret, if the *index* was found. Otherwise, the size of the stack secret is returned. Please note that in this case the returned value is not a valid position for an access to the stack secret.

[5] According to the behavior of the `[]`-operator, the zero denotes always the top-most position.

`bool find (const size_t index)` [Method on `TMCG_StackSecret`]

This method searches for a given permutation index in the stack secret. It returns `true`, if such an *index* was found.

`bool import (std::string s)` [Method on `TMCG_StackSecret`]

This method imports the stack secret from a correctly formatted input string *s*. It returns `true`, if the import was successful.

`~TMCG_StackSecret ()` [Destructor on `TMCG_StackSecret`]

This destructor releases all occupied resources.

`std::ostream& << (std::ostream& out, const TMCG_StackSecret<CardSecretType>& stacksecret)` [Operator on `TMCG_StackSecret`]

This operator exports the given *stacksecret* to the output stream *out*.

`std::istream& >> (std::istream& in, TMCG_StackSecret<CardSecretType>& stacksecret)` [Operator on `TMCG_StackSecret`]

This operator imports the given *stacksecret* from the input stream *in*. The data has to be delimited by a newline character. The `failbit` of the stream is set, if any parse error occurred.

2.2.1.3 Cryptographic Keys

LibTMCG provides corresponding data types for keys used by the encoding scheme of Schindelhauer [Sc98], because in this scheme it is not efficient to perform the process of key generation for every new game session. These keys are called TMCG keys. However, they also can be utilized to encrypt and sign messages for the more general reasons of confidentiality and integrity, even if the card encoding scheme of Schindelhauer is not used. Therefore these structures may be of independent interest, for example to establish authenticated communication channels between players. However, like for all public key cryptosystems a trusted PKI (Public Key Infrastructure) is needed. This might not be a serious problem in distributed game environments, because the players can include key fingerprints in their individual profile or a service provider can issue public key certificates.

`TMCG_SecretKey` [Data type]

This `struct` represents the secret part of the key. The underlying public key cryptosystem is due to Rabin (see *Digitalized Signatures and Public-Key Functions as Intractable as Factorization*, MIT Technical Report 212, 1979) and Williams (see *A modification of the RSA public-key encryption procedure*, IEEE Transactions on Information Theory, 26(6):726–729, 1980) with minor modifications for encryption padding (SAEP scheme of Boneh [Bo01]) and digital signatures (PRab scheme of Bellare and Rogaway [BR96]).

`std::string name` [Member of `TMCG_SecretKey`]

This string contains the name or a pseudonym of the key owner.

`std::string email` [Member of `TMCG_SecretKey`]

This string contains the email address of the key owner.

`std::string type` [Member of `TMCG_SecretKey`]

This string contains information about the key type. The common prefix is `TMCG/RABIN`. It is followed by the decimal encoded bit size of the modulus m. The suffix `NIZK` signals that the correctness of the key is shown by an appended non-interactive zero-knowledge proof. The single parts of the description are separated by underscore characters `_`, e.g., `TMCG/RABIN_2048_NIZK` has the correct form. The suffix can be left empty, if the key is only used for encryption and signing (so-called non-NIZK key) without card encoding.

`std::string nizk` [Member of `TMCG_SecretKey`]
This string contains two stages of the non-interactive zero-knowledge proof of Gennaro, Micciancio, and Rabin [GMR98]. The proof shows that m was correctly generated as product of at most two primes and both are congruent to 3 (modulo 4). Further there is another non-interactive zero-knowledge proof appended which shows that the condition $y \in \mathbf{NQR}^{\circ}_{m}$ holds.

`std::string sig` [Member of `TMCG_SecretKey`]
This string contains a self signature of the public key.

`mpz_t m` [Member of `TMCG_SecretKey`]
This is the public modulus $m = p \cdot q$ which is the product of two secret primes p and q. The size of m is determined by the security parameter `TMCG_QRA_SIZE`.

`mpz_t y` [Member of `TMCG_SecretKey`]
This is the public quadratic non-residue $y \in \mathbf{NQR}^{\circ}_{m}$ which is used in several zero-knowledge proofs of Schindelhauer's encoding scheme [Sc98].

`mpz_t p` [Member of `TMCG_SecretKey`]
This is the secret prime number p which is a factor of the modulus m.

`mpz_t q` [Member of `TMCG_SecretKey`]
This is the secret prime number q which is a factor of the modulus m.

`TMCG_SecretKey ()` [Constructor on `TMCG_SecretKey`]
This default constructor initializes an empty secret key.

`TMCG_SecretKey (const std::string&` *n*`, const std::string&` *e*`, const unsigned long int` *keysize* `=TMCG_QRA_SIZE, const bool` *nizk_key* `=true)` [Constructor on `TMCG_SecretKey`]
This constructor generates a new secret key, where *n* is the name or a pseudonym of the owner, *e* is a corresponding email address, *keysize* is the desired bit length of the modulus m, and *nizk_key* indicates whether or not a NIZK proof will be appended. The default value of the third argument is set to `TMCG_QRA_SIZE`, if *keysize* is omitted in the call. The default value of the fourth argument is set to `true`, whenever it is omitted in the call. Depending on *keysize* and *nizk_key* the generation is a very time-consuming task that should be taken into account by the application designer.

`TMCG_SecretKey (const std::string&` *s*`)` [Constructor on `TMCG_SecretKey`]
This constructor initializes the key from a correctly formatted input string *s*.

`TMCG_SecretKey (const TMCG_SecretKey&` *that*`)` [Constructor on `TMCG_SecretKey`]
This is a simple copy-constructor and *that* is the key to be copied.

`TMCG_SecretKey& = (const TMCG_SecretKey&` *that*`)` [Operator on `TMCG_SecretKey`]
This is a simple assignment-operator and *that* is the key to be assigned.

`bool check ()` [Method on `TMCG_SecretKey`]
This method tests whether the self signature is valid and whether the non-interactive zero-knowledge proofs are sound. It returns `true`, if all checks have been successfully passed. Due to the computational complexity of the verification procedure these checks are a very time-consuming task.

`std::string fingerprint ()` [Method on `TMCG_SecretKey`]
This method returns the fingerprint of the key. The fingerprint is the hexadecimal notation of the hash value (using algorithm `TMCG_GCRY_MD_ALGO`) on the concatenated members `name`, `email`, `type`, `m`, `y`, `nizk`, and `sig`.

`std::string selfid ()` [Method on `TMCG_SecretKey`]
This method returns the real value of the self signature. The string `ERROR` is returned, if any parse error occurred. The string `SELFSIG-SELFSIG-SELFSIG-SELFSIG-SELFSIG-SELFSIG` is returned, if the self signature `sig` was empty.

`std::string keyid (const size_t` *size* `=TMCG_KEYID_SIZE)` [Method on `TMCG_SecretKey`]
This method returns the unique key identifier of length *size*. The default value of the first argument is set to `TMCG_KEYID_SIZE`, if *size* is omitted in the call.

`size_t keyid_size (const std::string&` *s*`)` [Method on `TMCG_SecretKey`]
This method returns the length of the unique key identifier *s*. Zero is returned, if any parse error occurred.

`std::string sigid (std::string` *s*`)` [Method on `TMCG_SecretKey`]
This method returns the unique key identifier which is included in the signature *s*. The string `ERROR` is returned, if any parse error occurred.

`bool import (std::string` *s*`)` [Method on `TMCG_SecretKey`]
This method imports the key from a correctly formatted input string *s*. It returns `true`, if the import was successful.

`bool decrypt (unsigned char*` *value*`, std::string` *s*`)` [Method on `TMCG_SecretKey`]
This method decrypts the given encryption packet *s* and stores the content in *value* which is a pointer to a character array of size `TMCG_SAEP_S0`. The method returns `true`, if the decryption was successful.

`std::string sign (const std::string&` *data*`)` [Method on `TMCG_SecretKey`]
This method returns a digital signature on *data*.

`std::string encrypt (const unsigned char*` *value*`)` [Method on `TMCG_SecretKey`]
This method encrypts the content of *value* which is a pointer to a character array of size `TMCG_SAEP_S0`. The method returns a corresponding encryption packet that can be decrypted by the owner of the secret key.

`bool verify (const std::string&` *data*`, std::string` *s*`)` [Method on `TMCG_SecretKey`]
This method verifies whether the signature *s* on *data* is valid or not. It returns `true`, if everything was sound.

`~TMCG_SecretKey ()` [Destructor on `TMCG_SecretKey`]
This destructor releases all occupied resources.

`std::ostream& << (std::ostream&` *out*`, const TMCG_SecretKey&` *key*`)` [Operator on `TMCG_SecretKey`]
This operator exports the given *key* to the output stream *out*.

`std::istream& >> (std::istream& in, TMCG_SecretKey& key)` [Operator on `TMCG_SecretKey`]

This operator imports the given *key* from the input stream *in*. The data has to be delimited by a newline character. The `failbit` is set, if any parse error occurred.

`TMCG_PublicKey` [Data type]

This `struct` represents the public part of the TMCG key.

`std::string name` [Member of `TMCG_PublicKey`]

This string contains the name or a pseudonym of the key owner.

`std::string email` [Member of `TMCG_PublicKey`]

This string contains the email address of the key owner.

`std::string type` [Member of `TMCG_PublicKey`]

This string contains information about the key type. The common prefix is `TMCG/RABIN`. It is followed by the decimal encoded bit size of the modulus m. The suffix `NIZK` signals that the correctness of the key is shown by an appended non-interactive zero-knowledge proof. The single parts of the string are separated by underscore characters `_`, e.g., `TMCG/RABIN_2048_NIZK` has the correct form. However, the suffix can be left empty, if the key is only used for encryption and signing.

`std::string nizk` [Member of `TMCG_PublicKey`]

This string contains two stages of non-interactive zero-knowledge proof of Gennaro, Micciancio and Rabin [GMR98]. This gives strong evidence that m was generated correctly. Further there is another non-interactive zero-knowledge proof appended which shows that the condition $y \in \mathbf{NQR}^\circ_m$ holds.

`std::string sig` [Member of `TMCG_PublicKey`]

This string contains the self signature of the public key.

`mpz_t m` [Member of `TMCG_PublicKey`]

This is the public modulus $m = p \cdot q$ which is the product of two secret primes p and q. The size of m is determined by the security parameter `TMCG_QRA_SIZE`.

`mpz_t y` [Member of `TMCG_PublicKey`]

This is the public quadratic non-residue $y \in \mathbf{NQR}^\circ_m$ which is used by several zero-knowledge proofs of the toolbox.

`TMCG_PublicKey ()` [Constructor on `TMCG_PublicKey`]

This default constructor initializes an empty public key.

`TMCG_PublicKey (const TMCG_SecretKey& skey)` [Constructor on `TMCG_PublicKey`]

This constructor initializes the key using public values of the secret key *skey*.

`TMCG_PublicKey (const TMCG_PublicKey& pkey)` [Constructor on `TMCG_PublicKey`]

This is a simple copy-constructor and *pkey* is the key to be copied.

`TMCG_PublicKey& = (const TMCG_PublicKey& that)` [Operator on `TMCG_PublicKey`]

This is a simple assignment-operator and *that* is the key to be assigned.

`bool check ()` [Method on `TMCG_PublicKey`]

This method tests whether the self signature is valid and whether the non-interactive zero-knowledge proofs are sound. It returns `true`, if all checks have been successfully passed. Due to the computational complexity of the verification procedure these checks are an extremely time-consuming task.

`std::string fingerprint ()` [Method on `TMCG_PublicKey`]
This method returns the fingerprint of the key. The fingerprint is the hexadecimal notation of the hash value (using algorithm `TMCG_GCRY_MD_ALGO`) on the concatenated members `name`, `email`, `type`, `m`, `y`, `nizk`, and `sig`.

`std::string selfid ()` [Method on `TMCG_PublicKey`]
This method returns the real value of the self signature. The string `ERROR` is returned, if any parse error occurred. The string `SELFSIG-SELFSIG-SELFSIG-SELFSIG-SELFSIG-SELFSIG` is returned, if the self signature `sig` was empty.

`std::string keyid (const size_t` *size* `=TMCG_KEYID_SIZE)` [Method on `TMCG_PublicKey`]
This method returns the unique key identifier of length *size*. The default value of the first argument is set to `TMCG_KEYID_SIZE`, if *size* is omitted in the call.

`size_t keyid_size (const std::string&` *s*`)` [Method on `TMCG_PublicKey`]
This method returns the length of the unique key identifier *s*. Zero is returned, if any parse error occurred.

`std::string sigid (std::string` *s*`)` [Method on `TMCG_PublicKey`]
This method returns the unique key identifier which is included in the signature *s*. The string `ERROR` is returned, if any parse error occurred.

`bool import (std::string` *s*`)` [Method on `TMCG_PublicKey`]
This method imports the key from a correctly formatted input string *s*. It returns `true`, if the import was successful.

`std::string encrypt (const unsigned char*` *value*`)` [Method on `TMCG_PublicKey`]
This method encrypts the content of *value* which is a pointer to a character array of size `TMCG_SAEP_S0`. The method returns a corresponding encryption packet that can be decrypted by the owner of the secret key.

`bool verify (const std::string&` *data*`, std::string` *s*`)` [Method on `TMCG_PublicKey`]
This method verifies whether the signature *s* on *data* is valid or not. It returns `true`, if everything was sound.

`~TMCG_PublicKey ()` [Destructor on `TMCG_PublicKey`]
This destructor releases all occupied resources.

`std::ostream& << (std::ostream&` *out*`, const TMCG_PublicKey&` *key*`)` [Operator on `TMCG_PublicKey`]
This operator exports the given *key* to the output stream *out*.

`std::istream& >> (std::istream&` *in*`, TMCG_PublicKey&` *key*`)` [Operator on `TMCG_PublicKey`]
This operator imports the given *key* from the input stream *in*. The data has to be delimited by a newline character. The `failbit` is set, if any parse error occurred.

`TMCG_PublicKeyRing` [Data type]
This `struct` is just a simple container for TMCG public keys. There are no particular methods provided by `TMCG_PublicKeyRing`. You have to use the regular interface of the STL container `std::vector` to access the single keys of the ring.

`std::vector<TMCG_PublicKey> keys` [Member of `TMCG_PublicKeyRing`]
This is the real container that is used to store the keys.

TMCG_PublicKeyRing () [Constructor on TMCG_PublicKeyRing]
This default constructor initializes an empty public key ring.

TMCG_PublicKeyRing (size_t *n*) [Constructor on TMCG_PublicKeyRing]
This constructor initializes the container for storing exactly n keys.

~TMCG_PublicKeyRing () [Destructor on TMCG_PublicKeyRing]
This destructor releases all occupied resources.

2.2.2 Communication Interfaces

The base class aiounicast and the corresponding derived classes aiounicast_nonblock and aiounicast_select provide a simple communication interface for asynchronous point-to-point communication channels. They can be used to transfer data of type mpz_t (big integers, see libgmp for explanation of this data type) between up to n parties, that are connected by sockets, pipes or any other file descriptor based input/output mechanism.

Moreover, the channels can be authenticated by a message authentication code and encrypted by a symmetric cipher. The deployed algorithms are defined by global symbols (TMCG_GCRY_MAC_ALGO and TMCG_GCRY_ENC_ALGO, respectively) and fixed at compile time of LibTMCG.

aiounicast [Class]
This class is only an abstract interface and cannot be instantiated directly. We explain some basic class members that are useful for an application programmer.

static const time_t aio_timeout_very_short [Member of aiounicast]
This constant defines a very short time interval of only one second.

static const time_t aio_timeout_short [Member of aiounicast]
This constant defines a short time interval of 15 seconds.

static const time_t aio_timeout_middle [Member of aiounicast]
This constant defines a middle time interval of 30 seconds.

static const time_t aio_timeout_long [Member of aiounicast]
This constant defines a long time interval of 90 seconds.

static const time_t aio_timeout_very_long [Member of aiounicast]
This constant defines a very long time interval of 180 seconds.

static const time_t aio_timeout_extremly_long [Member of aiounicast]
This constant defines an extremly long time interval of 300 seconds.

static const size_t aio_scheduler_roundrobin [Member of aiounicast]
This constant represents the round-robin scheduler for message processing.

static const size_t aio_scheduler_random [Member of aiounicast]
This constant represents the random select scheduler for message processing.

static const size_t aio_scheduler_direct [Member of aiounicast]
This constant represents the constant select scheduler for message processing.

const size_t n [Member of aiounicast]
This is the total number of parties n involved in the communication.

const size_t j [Member of aiounicast]
This is an uniqe index of the party running this instance.

`std::map<size_t, int> fd_in` [Member of `aiounicast`]
The input file descriptors of point-to-point links to all parties.

`std::map<size_t, int> fd_out` [Member of `aiounicast`]
The ouput file descriptors of point-to-point links to all parties.

`size_t numWrite` [Member of `aiounicast`]
The total number of bytes written to point-to-point links.

`size_t numRead` [Member of `aiounicast`]
The total number of bytes read from point-to-point links.

`size_t numEncrypted` [Member of `aiounicast`]
The total number of bytes that have been encrypted yet.

`size_t numDecrypted` [Member of `aiounicast`]
The total number of bytes that have been decrypted yet.

`size_t numAuthenticated` [Member of `aiounicast`]
The total number of bytes that have been authenticated yet.

Note that the header files `aiounicast_nonblock.hh` or `aiounicast_select.hh` must be included in addition to `libTMCG.hh`. The use of class `aiounicast_select` is strongly recommended.

`aiounicast_nonblock` [Class]
This class works with non-blocking file descriptors, i.e., the pipes or sockets have to be opened with the `O_NONBLOCK` flag. The methods use continiuous polling on the descriptors to achieve asynchronous I/O that results in exorbitant CPU load. The class should be used only, if no select system call is available or appropriate for the application.

`aiounicast_nonblock (const size_t` *n_in*, `const size_t` *j_in*, `const std::vector<int>&` *fd_in_in*, `const std::vector<int>&` *fd_out_in*, `const std::vector<std::string>&` *key_in*, `const size_t` *aio_default_scheduler_in* `=aio_scheduler_roundrobin`, `const time_t` *aio_default_timeout_in* `=aio_timeout_long`, `const bool` *aio_is_authenticated_in* `=true`, `const bool` *aio_is_encrypted_in* `=true)` [Constructor on `aiounicast_nonblock`]
The constructor initializes internal queues and data structures for asynchronous point-to-point channels connecting n parties (i.e. *n_in*). The index of the calling party within this set is given by *j_in*. It is followed by a vector *fd_in_in* of exactly n input file descriptors that are ready for reading and writing, and by a vector *fd_out_in* of exactly n output file descriptors. Finally, the vector *key_in* with exactly n passphrases[6] or pre-shared keys is neccesary, if *aio_is_authenticated_in* or *aio_is_encrypted_in* is set `true`, which is the default behaviour. The default values for timeout (in seconds) and the receive scheduler can be modified carefully according to the desired usage scenario.

`bool Send (mpz_srcptr` *m*, `const size_t` *i_in*, `time_t` *timeout* `=aio_timeout_default)` [Method on `aiounicast_nonblock`]
This method sends an integer m over the corresponding point-to-point link to the party with index *i_in*. In presence of the third argument this transmission is tried for at most *timeout* seconds. Otherwise, the default timeout given to the constructor is applied.

The method returns `false`, if sending fails, and error messages are written to `std::cerr`.

[6] The key derivation function PBKDF2 is applied with an iteration count of 25.000 and a different constant salt to derive the authentication and the encryption key, respectively.

bool Send (const std::vector<mpz_srcptr>& *m*, const size_t *i_in*, time_t *timeout* =aio_timeout_default) [Method on aiounicast_nonblock]

This method works as above, however, a vector *m* of integers is sent.

bool Receive (mpz_ptr *m*, size_t& *i_out*, size_t *scheduler* =aio_scheduler_default, time_t *timeout* =aio_timeout_default) [Method on aiounicast_nonblock]

This method receives an integer *m* over the point-to-point links from any party. The index of the sender is returned in *i_out*. In presence of the third argument it waits for at most *timeout* seconds. Otherwise, the default timeout given to the constructor is applied.

The method returns false, if receiving fails. Only in critical cases some error messages are written to std::cerr.

bool Receive (std::vector<mpz_ptr>& *m*, size_t& *i_out*, size_t *scheduler* =aio_scheduler_default, time_t *timeout* =aio_timeout_default) [Method on aiounicast_nonblock]

This method works as above, however, a vector *m* of integers is received.

~aiounicast_nonblock () [Destructor on aiounicast_nonblock]

This destructor releases all occupied resources.

aiounicast_select [Class]

This class works with arbitrary file descriptors. It uses the select interface of the operation system with negligible timeout of 1000us to achieve asynchronous I/O. This results in a reasonable CPU load in comparison with aiounicast_nonblock.

aiounicast_select (const size_t *n_in*, const size_t *j_in*, const std::vector<int>& *fd_in_in*, const std::vector<int>& *fd_out_in*, const std::vector<std::string>& *key_in*, const size_t *aio_default_scheduler_in* =aio_scheduler_roundrobin, const time_t *aio_default_timeout_in* =aio_timeout_long, const bool *aio_is_authenticated_in* =true, const bool *aio_is_encrypted_in* =true) [Constructor on aiounicast_select]

The constructor initializes internal queues and data structures for asynchronous point-to-point channels connecting n parties (i.e. *n_in*). The index of the calling party within this set is given by *j_in*. It is followed by a vector *fd_in_in* of exactly n input file descriptors that are ready for reading and writing, and by a vector *fd_out_in* of exactly n output file descriptors. Finally, the vector *key_in* with exactly n passphrases[7] or pre-shared keys is neccesary, if *aio_is_authenticated_in* or *aio_is_encrypted_in* is set true, which is the default behaviour. The default values for timeout (in seconds) and the receive scheduler can be modified carefully according to the desired usage scenario.

bool Send (mpz_srcptr *m*, const size_t *i_in*, time_t *timeout* =aio_timeout_default) [Method on aiounicast_select]

This method sends an integer *m* over the corresponding point-to-point link to the party with index *i_in*. In presence of the third argument this transmission is tried for at most *timeout* seconds. Otherwise, the default timeout given to the constructor is applied.

The method returns false, if sending fails, and error messages are written to std::cerr.

[7] The key derivation function PBKDF2 is applied with an iteration count of 25.000 and a different constant salt to derive the authentication and the encryption key, respectively.

`bool Send (const std::vector<mpz_srcptr>& m, const size_t i_in, time_t timeout =aio_timeout_default)` [Method on `aiounicast_select`]

This method works as above, however, a vector *m* of integers is sent.

`bool Receive (mpz_ptr m, size_t& i_out, size_t scheduler =aio_scheduler_default, time_t timeout =aio_timeout_default)` [Method on `aiounicast_select`]

This method receives an integer *m* over the point-to-point links from any party. The index of the sender is returned in *i_out*. In presence of the third argument it waits for at most *timeout* seconds. Otherwise, the default timeout given to the constructor is applied.

The method returns `false`, if receiving fails. Only in critical cases some error messages are written to `std::cerr`.

`bool Receive (std::vector<mpz_ptr>& m, size_t& i_out, size_t scheduler =aio_scheduler_default, time_t timeout =aio_timeout_default)` [Method on `aiounicast_select`]

This method works as above, however, a vector *m* of integers is received.

`~aiounicast_select ()` [Destructor on `aiounicast_select`]

This destructor releases all occupied resources.

2.2.3 Classes

LibTMCG consists of several C++ classes. Some of them are only extensions or optimizations, but other provide necessary interfaces to perform the basic operations in secure card games, e.g., the creation of open cards, the masking of cards, the opening of masked cards, the verifiable secret shuffle of a stack, and more general tasks like distributed key generation procedures. Each class implements the some functionality of the corresponding research paper [CKPS01, BS03, JL00, Gr05, HSSV09, Sc98]. The author names are a prefix of the class name and the following part is an abbreviation of (a part of) the title, respectively.

2.2.3.1 Secure and Efficient Asynchronous Broadcast Protocols

This part of LibTMCG provides an implementation of *reliable broadcast*, which is actually based on an optimized variant of Bracha's double-echo broadcast protocol. It works without further authentication mechanisms (e.g. digital signatures) and thus guarantees the desired properties (i.e. validity, consistency, and totality[8]) of reliable broadcast only, if the number of faulty or even malicious players t is strictly less than one third of all parties n, i.e. $t < n/3$. Please note that without further assumptions this condition is rather optimal for asynchronous communication and thus has crucial impact for liveness of the high-level protocols using it.

We describe only those classes, methods, and members that might be of interest for an application programmer.

`CachinKursawePetzoldShoupRBC` [Class]

This class implements the protocol RBC by Cachin, Kursawe, Petzold, and Shoup [CKPS01] for a reliable broadcast in the asynchronous communication model, where $t < n/3$ holds. Additionally, a FIFO-ordered delivery mechanism based on sequence numbers has been implemented.

`size_t n` [Member of `CachinKursawePetzoldShoupRBC`]

This is the total number of parties n involved in this protocol.

[8] Totality ensures that all correct parties either deliver a message or don't. In the literature consistency and totality properties are often combined into a single condition called *agreement*.

`size_t t` [Member of `CachinKursawePetzoldShoupRBC`]
This is the number of possible faulty parties t.

`size_t j` [Member of `CachinKursawePetzoldShoupRBC`]
This is an uniqe index of the party running this instance.

`CachinKursawePetzoldShoupRBC` [Constructor on `CachinKursawePetzoldShoupRBC`]
(`const size_t` *n_in*, `const size_t` *t_in*, `const size_t` *j_in*, `aiounicast*` *aiou_in*, `const size_t` *aio_default_scheduler_in* `=aiounicast::aio_scheduler_roundrobin`, `const time_t` *aio_default_timeout_in* `=aiounicast::aio_timeout_very_long`)

The constructor initializes an instance for a reliable broadcast channel of n parties. This total number of parties is given in the first argument *n_in*. The number of possible faulty or even malicious parties t (given in the second argument *t_in*) must not exceed $n/3$. Otherwise a warning is printed to `std::cerr` and the liveness of the protocol RBC is not guaranteed. Thus, it is recommended to set *t_in* to the asynchronous maximum $(n-1)/3$. The third argument *j_in* is an index of the party running this instance. Finally, the constructor needs as fourth argument *aiou_in* a reference to already established point-to-point channels (see Section 2.2.2 [Communication Interfaces], page 24), which should exclusively[9] used for this broadcast channels. The default values for timeout (in seconds) and the deliver scheduler can be modified carefully with respect to the usage scenario.

`void setID (const std::string` *ID_in*`)` [Method on `CachinKursawePetzoldShoupRBC`]
Broadcast channels are parameterized by a *tag* called ID, that is contained in every message. This method sets the tag to *ID_in*, which should be equal for all parties for the desired channel.

`void unsetID ()` [Method on `CachinKursawePetzoldShoupRBC`]
This method unset the current channel tag and returns to the previous value. This is commonly used to return from a channel of a subprotcol to the channel of the calling protocol.

`void Broadcast (mpz_srcptr` *m*, `const bool` *simulate_faulty_behaviour* `=false)` [Method on `CachinKursawePetzoldShoupRBC`]
This method broadcasts the integer m to all parties.

`bool Deliver (mpz_ptr` *m*, `size_t&` *i_out*, `size_t` *scheduler* `=aiounicast::aio_scheduler_default`, `time_t` *timeout* `=aiounicast::aio_timeout_default)` [Method on `CachinKursawePetzoldShoupRBC`]
This method delivers a broadcasted integer m from any party using deliver scheduler *scheduler*. The index of the sender is returned in *i_out*. In presence of the fourth argument it waits for at most *timeout* seconds. Otherwise, the default timeout given to the constructor is applied.

The method returns `false`, if delivering fails. Only in some critical cases error messages are written to `std::cerr`.

`bool DeliverFrom (mpz_ptr` *m*, `const size_t` *i_in*, `size_t` *scheduler* `=aiounicast::aio_scheduler_default`, `time_t` *timeout* `=aiounicast::aio_timeout_default)` [Method on `CachinKursawePetzoldShoupRBC`]
This method delivers a broadcasted integer m from a specified party with index *i_in* using deliver scheduler *scheduler*. In presence of the fourth argument it waits at most for *timeout* seconds. Otherwise, the default timeout given to the constructor is applied.

[9] These channels should be authenticated such that network attacks or errors can be detetcted.

The method returns `false`, if delivering fails. Only in some critical cases error messages are written to `std::cerr`.

`bool Sync (time_t` *timeout* `=aiounicast::aio_timeout_default, const std::string` *tag* `="")` [Method on `CachinKursawePetzoldShoupRBC`]

This method continues the execution of RBC protocol such that the requests of other waiting parties are satisfied. In presence of the first argument it waits approximately for $(t{+}1)\cdot$ *timeout* seconds while trying to synchronize all parties based on their corresponding local Unix Epoch time. Otherwise, the default timeout given to the constructor is applied. Each synchronization point is required to be unique. Thus, a string called *tag* with a description of the synchronization point can be supplied as second argument of this method.

The method returns `false`, if synchronization is failed.

`~CachinKursawePetzoldShoupRBC ()` [Destructor on `CachinKursawePetzoldShoupRBC`]

This destructor releases all occupied resources.

2.2.3.2 Verifiable *k*-out-of-*k* Threshold Masking Function

The two classes of this subsection are concrete instantiations of Barnett and Smart's VTMF primitive [BS03]. More formally, the authors specify four different protocols:

- Key Generation Protocol
- Verifiable Masking Protocol
- Verifiable Re-masking Protocol
- Verifiable Decryption Protocol

Each protocol uses low-level operations on an appropriately chosen algebraic group G. The choice of this group is crucial to the security of the card encoding scheme and thus also to the security of high-level operations on cards resp. stacks.

There are just a few methods and members of these classes that might be of general interest for an application programmer, e.g. the methods of the key generation protocol. The other stuff is only used internally by high-level operations of `SchindelhauerTMCG`. Therefore this manual omits the description of such internal functions and members.

`BarnettSmartVTMF_dlog` [Class]

This class implements the discrete logarithm instantiation of the VTMF primitive in the field $\mathbf{Z}/p\mathbf{Z}$, where p is a large prime number. The mathematical computations are performed in the finite cyclic subgroup G of prime order q such that $p = kq + 1$ holds for some $k \in \mathbf{Z}$. The security relies on the DDH assumption in G, i.e., the distribution $\{g^a, g^b, g^{ab}\}$ is computationally indistinguishable from $\{g^a, g^b, g^c\}$, where g is a generator of G and a, b, c are chosen at random from $\mathbf{Z}_q$. Currently, this well-established assumption is believed to hold, if p and q are chosen according to the predefined security parameters of LibTMCG.

`mpz_t p` [Member of `BarnettSmartVTMF_dlog`]

This is the public prime number p which defines the underlying finite field $\mathbf{Z}/p\mathbf{Z}$.

`mpz_t q` [Member of `BarnettSmartVTMF_dlog`]

This is the public prime number q which defines the underlying cyclic group G. G is a subgroup of $\mathbf{Z}/p\mathbf{Z}$ and is exactly of order q.

`mpz_t g` [Member of `BarnettSmartVTMF_dlog`]

This is the fixed public generator g of the underlying group G.

`mpz_t k` [Member of `BarnettSmartVTMF_dlog`]
This is a public integer k such that $p = kq + 1$ holds.

`mpz_t h` [Member of `BarnettSmartVTMF_dlog`]
This is the common public key $h = \prod_{i=1}^{k} h_i$ which contains the public keys h_i of each player P_i. Note that in the above formula k denotes the number of players.

`mpz_t h_i` [Member of `BarnettSmartVTMF_dlog`]
This is the public key h_i of this player instance.

`BarnettSmartVTMF_dlog (const unsigned long int` *fieldsize* `=TMCG_DDH_SIZE, const unsigned long int` *subgroupsize* `=TMCG_DLSE_SIZE, const bool` *canonical_g_usage* `=false, const bool` *initialize_group* `=true)` [Constructor on `BarnettSmartVTMF_dlog`]
This constructor creates a new VTMF instance. That means, the primes p and q are randomly and uniformly chosen such that they have length *fieldsize* bit and *subgroupsize* bit, respectively. Further, either a generator g for the unique subgroup of order q is chosen at random or, if *canonical_g_usage* is set `true`, the generator g is chosen in a verifiable way (cf. FIPS 186-3 A.2.3). If the arguments are omitted, then *fieldsize*, *subgroupsize* and *canonical_g_usage* are set to their default values `TMCG_DDH_SIZE`, `TMCG_DLSE_SIZE`, and `false`, respectively. The argument *initialize_group* should be always set `true`. Depending on *fieldsize* and *subgroupsize* the group generation is a very time-consuming task that should be taken into account by the application designer.

`BarnettSmartVTMF_dlog (std::istream&` *in*`, const unsigned long int` *fieldsize* `=TMCG_DDH_SIZE, const unsigned long int` *subgroupsize* `=TMCG_DLSE_SIZE, bool` *canonical_g_usage* `=false, const bool` *precompute* `=true)` [Constructor on `BarnettSmartVTMF_dlog`]
This constructor initializes the VTMF instance from a correctly formatted input stream *in*. For example, such a stream can be generated by calling the method `PublishGroup` of an already created instance. The arguments *fieldsize*, *subgroupsize*, and *canonical_g_usage* are stored for later following usage, e.g. by the method `CheckGroup` as explained below. The argument *precompute* should be always set `true`. If these arguments are omitted, then they are set to the default values `TMCG_DDH_SIZE`, `TMCG_DLSE_SIZE`, `false`, and `true` respectively.

`bool CheckGroup ()` [Method on `BarnettSmartVTMF_dlog`]
This method checks whether p and q have appropriate sizes with respect to the bit lengths given during the initialization of the corresponding instance. Further, it checks whether p has the correct form (i.e. $p = kq + 1$), whether p and q are probable prime, and whether g is a generator of the subgroup G. If *canonical_g_usage* is set `true` during the call of constructor, then it additionally checks whether g was generated in a verifiable way (cf. FIPS 186-3 A.2.3). It returns `true`, if all of these checks have been passed successfully.

`void PublishGroup (std::ostream&` *out*`)` [Method on `BarnettSmartVTMF_dlog`]
This method exports all necessary group parameters of G to the given output stream *out*, so other VTMF instances of G can be initialized, e.g. with the second constructor of `BarnettSmartVTMF_dlog`.

`void KeyGenerationProtocol_GenerateKey ()` [Method on `BarnettSmartVTMF_dlog`]
This method generates a VTMF key pair and stores the numbers internally for a later following usage. It must be called before any other part of the key generation protocol is executed. Otherwise, the produced results are wrong.

`void KeyGenerationProtocol_PublishKey (std::ostream& out)` [Method on `BarnettSmartVTMF_dlog`]

This method exports the public part h_i of the generated VTMF key pair to the given output stream *out*. Further, it appends a non-interactive zero-knowledge proof of knowledge (NIZK) which shows that the instance knows the secret part x_i such that $h_i \equiv g^{x_i} \pmod p$ holds. Due to the non-interactive nature of this proof the method has to be called only once while the computed output can be reused multiple times if necessary.

`bool KeyGenerationProtocol_UpdateKey (std::istream& in)` [Method on `BarnettSmartVTMF_dlog`]

This method reads the public part of a VTMF key and the NIZK from the input stream *in*. It appends the key to the common public key and returns `true`, if the given proof was sound. Otherwise, `false` is returned.

`bool KeyGenerationProtocol_RemoveKey (std::istream& in)` [Method on `BarnettSmartVTMF_dlog`]

This method reads the public part of a VTMF key and the corresponding NIZK from the input stream *in*. It removes the key from the common public key and returns `true`, if the key was previously appended by `KeyGenerationProtocol_UpdateKey` as explained above.

`void KeyGenerationProtocol_Finalize ()` [Method on `BarnettSmartVTMF_dlog`]

This method must be called after any update (`KeyGenerationProtocol_UpdateKey`) or removal (`KeyGenerationProtocol_RemoveKey`) has been performed on the common public key.

`~BarnettSmartVTMF_dlog ()` [Destructor on `BarnettSmartVTMF_dlog`]

This destructor releases all occupied resources.

`BarnettSmartVTMF_dlog_GroupQR` [Subclass of `BarnettSmartVTMF_dlog`]

This subclass implements the discrete logarithm instantiation of the VTMF primitive in the field $\mathbf{Z}/p\mathbf{Z}$, where p is a large prime number. The mathematical computations are performed in a special finite cyclic subgroup G (quadratic residues modulo p) of prime order q, where $p = 2q + 1$ holds. The security also relies on the DDH assumption w.r.t. G, i.e., the distribution $\{g^a, g^b, g^{ab}\}$ is computationally indistinguishable from $\{g^a, g^b, g^c\}$, where g is a generator of G and a, b, c are chosen at random from $\mathbf{Z}_q$. Currently, this well-established assumption is believed to hold, if p and q are chosen according to the predefined security parameters of LibTMCG.

`mpz_t p` [Member of `BarnettSmartVTMF_dlog`]

This is the public prime number p which defines the underlying finite field $\mathbf{Z}/p\mathbf{Z}$.

`mpz_t q` [Member of `BarnettSmartVTMF_dlog`]

This is the public prime number q which defines the underlying cyclic group G. G denotes the unique subgroup of quadratic residues modulo p which is exactly of order q, if $p = 2q+1$ holds.

`mpz_t g` [Member of `BarnettSmartVTMF_dlog`]

This is the fixed public generator g of the underlying group G.

`mpz_t k` [Member of `BarnettSmartVTMF_dlog`]

This integer is fixed here by $k = 2$.

`mpz_t h` [Member of `BarnettSmartVTMF_dlog`]

This is the common public key $h = \prod_{i=1}^{k} h_i$ which contains the public keys h_i of each player P_i. Note that in the above formula k denotes the number of players.

`mpz_t h_i` [Member of `BarnettSmartVTMF_dlog`]

This is the public key h_i of this player instance.

`BarnettSmartVTMF_dlog_GroupQR` [Constructor on `BarnettSmartVTMF_dlog_GroupQR`]
`(const unsigned long int` *fieldsize* `=TMCG_DDH_SIZE, const unsigned long int` *exponentsize* `=TMCG_DLSE_SIZE)`

This constructor creates a new VTMF instance. That means, the safe prime p is randomly and uniformly chosen such that it has a length of *fieldsize* bit. Further, the generator g is initially set up by 2 and then shifted by *fieldsize* − *exponentsize* bit positions, according to the procedure described by Koshiba and Kurosawa (see *Short Exponent Diffie-Hellman Problems*, PKC 2004, LNCS 2947). If the arguments of the constructor are omitted, then *fieldsize* and *exponentsize* are set to their default values `TMCG_DDH_SIZE` and `TMCG_DLSE_SIZE`, respectively. Depending on *fieldsize* and *exponentsize* the group generation is a very time-consuming task that should be taken into account by the application designer.

`BarnettSmartVTMF_dlog_GroupQR` [Constructor on `BarnettSmartVTMF_dlog_GroupQR`]
`(std::istream&` *in*, `const unsigned long int` *fieldsize* `=TMCG_DDH_SIZE, const unsigned long int` *exponentsize* `=TMCG_DLSE_SIZE)`

This constructor initializes the VTMF instance from a correctly formatted input stream *in*. For example, such a stream can be generated by calling the method `PublishGroup` of an already created instance. The arguments *fieldsize* and *exponentsize* are stored for later following usage, e.g. by the method `CheckGroup` as explained below. If these arguments are omitted, then they are set to the default values `TMCG_DDH_SIZE` and `TMCG_DLSE_SIZE`, respectively.

`bool CheckGroup ()` [Method on `BarnettSmartVTMF_dlog_GroupQR`]

This method checks whether p and q have appropriate sizes with respect to the bit lengths given during the initialization of the corresponding instance. Further, it checks whether p has the correct form (i.e. $p = 2q + 1$), whether p and q are probable prime, and whether g is a generator of the subgroup G. It returns `true`, if all of these checks have been passed successfully.

`~BarnettSmartVTMF_dlog_GroupQR` [Destructor on `BarnettSmartVTMF_dlog_GroupQR`]
`()`

This destructor releases all occupied resources.

2.2.3.3 Adaptively Secure Threshold Cryptography

Jarecki and Lysyanskaya [JL00] have introduced some useful building blocks in order to gain security against an adaptive adversary for threshold cryptography.

`JareckiLysyanskayaEDCF` [Class]

This class provides the erasure-free distributed coinflip (EDCF) protocol. It also needs a group G_q of prime order q where the discrete logarithm problem is computationally hard. The protocol produces a public value $a = \sum_{i=1}^{n} a_i \bmod q$ such that $0 \leq a < q$ is random and uniformly distributed, if at least one party $P_i, 1 \leq i \leq n$ has chosen their corresponding coin share $a_i \in \mathbf{Z}_q$ uniformly at random.

The coinflip protocol is useful in order to transform a public-coin honest-verifier zero-knowledge proof of knowledge (HVZKP) into interactive proof resp. argument which preserve the zero-knowledge property even in case of malicious verifiers. Such proof systems are called *simultaneous* zero-knowledge proofs of knowledge. The underlying general model of Jarecki and Lysyanskaya [JL00] considers a synchronous communication network of n players with access to a reliable broadcast channel, where an adaptive adversary can corrupt up to a minority $t < n/2$ of the players.

`mpz_t p` [Member of `JareckiLysyanskayaEDCF`]
This is the public prime number p which defines the underlying finite field $\mathbf{Z}/p\mathbf{Z}$.

`mpz_t q` [Member of `JareckiLysyanskayaEDCF`]
This is the public prime number q which defines the underlying cyclic group G_q. Note that G_q is a subgroup of $\mathbf{Z}/p\mathbf{Z}$ and it must be chosen to have order q.

`mpz_t g` [Member of `JareckiLysyanskayaEDCF`]
This is the fixed public generator g of the underlying group G_q.

`mpz_t h` [Member of `JareckiLysyanskayaEDCF`]
This is the common public value $h \in G_q$ such that nobody knows $\log_g h$. It can be obtained by the above key generation protocol (see Section 2.2.3.2 [BarnettSmartVTMF], page 29).

> **Jarecki and Lysyanskaya [JL00]:** "When secure channels are present, h can be obtained by using general techniques of multi-party computation [BGW88, CDD+99]. When secure channel are not there, and implementing them by erasure is not an option, we can use another protocol, where each player generates his share h_i of h, and then all players, in parallel, prove knowledge of $\log_g h_i$ to each other."

`size_t n` [Member of `JareckiLysyanskayaEDCF`]
This is the total number of parties n involved in this protocol.

`size_t t` [Member of `JareckiLysyanskayaEDCF`]
This is the maximum number of faulty parties t (reconstruction threshold).

`JareckiLysyanskayaEDCF (const size_t` *n_in*, `const size_t` *t_in*, `mpz_srcptr` *p_CRS*, `mpz_srcptr` *q_CRS*, `mpz_srcptr` *g_CRS*, `mpz_srcptr` *h_CRS*, `const unsigned long int` *fieldsize* `=TMCG_DDH_SIZE`, `const unsigned long int` *subgroupsize* `=TMCG_DLSE_SIZE)` [Constructor on `JareckiLysyanskayaEDCF`]
This constructor creates a new EDCF instance. That means, the required primes p and q and the generators g and h are initialized from the given arguments *p_CRS*, *q_CRS*, *g_CRS*, and *h_CRS*, respectively. *n_in* is the total number of participating players, for which at most *t_in* are faulty or act malicious during the protocol execution.

`bool CheckGroup ()` [Method on `JareckiLysyanskayaEDCF`]
This method checks whether p and q have appropriate sizes with respect to the bit lengths given during the initialization of the corresponding instance. Further, it checks whether p has the correct form (i.e. $p = kq + 1$), whether p and q are probable prime, and whether g resp. h are different generators of the subgroup G_q. It returns `true`, if all of these checks have been passed successfully.

`bool Flip (const size_t` *i*, `mpz_ptr` *a*, `aiounicast*` *aiou*, `CachinKursawePetzoldShoupRBC*` *rbc*, `std::ostream&` *err*, `const bool` *simulate_faulty_behaviour* `=false)` [Method on `JareckiLysyanskayaEDCF`]
This method starts the protocol which produces a public value $a = \sum_{i=1}^{n} a_i \bmod q$ such that $0 \leq a < q$ is random and uniformly distributed, if at least one party $P_i, 1 \leq i \leq n$ has chosen their corresponding share $a_i \in \mathbf{Z}_q$ uniformly at random. If it returns `true`, then a contains this common random value. The argument i is an index of the running instance with respect to already initialized instances of asynchronous point-to-point channels *aiou* and a reliable broadcast channel *rbc*. Logging and debug messages are printed to the provided output stream *err*.

`bool Flip_twoparty (const size_t` *i*, `mpz_ptr` *a*, `std::istream&` *in*, `std::ostream&` *out*, `std::ostream&` *err*, `const bool` *simulate_faulty_behaviour* `=false)` [Method on `JareckiLysyanskayaEDCF`]

This is the two-party version of the above method. Thus there are only an input stream *in* and output stream *out* for communication between the players. The other arguments are as above.

`~JareckiLysyanskayaEDCF ()` [Destructor on `JareckiLysyanskayaEDCF`]

This destructor releases all occupied resources.

2.2.3.4 Verifiable Secret Shuffle of Homomorphic Encryptions

Recently, Groth [Gr05, Gr10] has proposed a very efficient solution to perform a verifiable shuffle of homomorphically encrypted values. He describes an honest verifier zero-knowledge argument which shows the correctness of a shuffle. Beside other applications (e.g. verifiable mix networks, electronic voting) his protocol can be used to show (with overwhelming probability) that the secret shuffle of a deck of cards was performed correctly. The computational complexity and the produced communication traffic are superior to previously deployed techniques (e.g. Schindelhauer's cut-and-choose method). LibTMCG provides the first known implementation of Groth's famous protocol. However, it can only be used along with the VTMF card encoding scheme of Barnett and Smart [BS03] based on the hardness of computing discrete logarithms.

Our implementation uses a generalized variant [Gr05, Gr10] of the statistically hiding and computationally binding homomorphic commitment scheme due to Pedersen (see *Non-interactive and Information-theoretic Secure Verifiable Secret Sharing*, CRYPTO '91, LNCS 576, 1992). The binding property relies on the hardness of computing discrete logarithms in G w.r.t. random bases $g_1, \ldots, g_n$ and thus a commitment is only binding for computationally bounded provers.[10] But this choice seems to be reasonable for the intention of LibTMCG, because all players are supposed to be computationally bounded. The security parameters of the commitment scheme (in particular the group G) are determined by the corresponding VTMF instance.

Since version 1.2.0 of LibTMCG we use a two-party version of a distributed coin flipping protocol by Jarecki and Lysyanskaya [JL00] to protect against malicious verifiers attacking the zero-knowledge property. Since version 1.3.0 there is an additional method for generating the bases $g_1, \ldots, g_n$ of the Pedersen commitment scheme by distributed coin flipping and a verifiable generation procedure similar to FIPS 186-3 A.2.3. This step is important in order to ensure, that a malicious prover cannot compute $\log_{g_i} h$ resp. $\log_h g_i$, for some $i = 1, \ldots, n$, and thus erroneously pass the shuffle verification. It improves our former security model which considered only a passive adversary.

Further, to the best of our knowledge it is not known, whether Groth's protocol retains the zero-knowledge property when it is executed in a concurrent setting. Thus the application programmer should be careful and avoid parallel invocations of the same instance.

`GrothVSSHE` [Class]

This class provides the low-level interface for Groth's protocol. There are just a few methods that might be of general interest. All other components are only used internally by high-level operations and thus their description is omitted here.

---

[10] Strictly speaking, due to this reason Groth's protocol is a zero-knowledge *argument* instead of a zero-knowledge *proof*. However, for convenience we will not distinguish between these terms here.

`GrothVSSHE (size_t` *n*, `mpz_srcptr` *p_ENC*, `mpz_srcptr` *q_ENC*, `mpz_srcptr` *k_ENC*, `mpz_srcptr` *g_ENC*, `mpz_srcptr` *h_ENC*, `unsigned long int` *ell_e* `=TMCG_GROTH_L_E`, `unsigned long int` *fieldsize* `=TMCG_DDH_SIZE`, `unsigned long int` *subgroupsize* `=TMCG_DLSE_SIZE)` [Constructor on `GrothVSSHE`]

This constructor creates a new instance. The low-level operations are later used to show the correctness of a shuffle of at most *n* cards. The protocol and some parameters of the commitment scheme are initialized by the members of the corresponding VTMF instance. Consequently, *p_ENC* is the prime number p which determines the field $\mathbf{Z}/p\mathbf{Z}$, *q_ENC* is the order of the underlying subgroup G, i.e. the prime number q, and *k_ENC* is the integer such that $p = qk + 1$ holds. Further, *g_ENC* is the generator g of this subgroup, and finally *h_ENC* is the common public key h. The positive integer *ell_e* is the security parameter which controls the soundness error probability ($2^{-\ell_e}$) of the protocol. The default value is defined by `TMCG_GROTH_L_E`, if this argument is omitted. The *fieldsize* and the *subgroupsize* are supplied to internal classes and are only of interest, if *p_ENC* or *q_ENC* have lengths different from the default. If these arguments are omitted, they are set to `TMCG_DDH_SIZE` and `TMCG_DLSE_SIZE`, respectively.

This constructor should be instantiated only once by the session leader. All other instances can be created by the second constructor. Further, it is very important that the VTMF key generation protocol has been finished before the value of h is passed to the constructors. Otherwise, the correctness verification of the shuffle will fail.

Note that the generators $g_1, \ldots, g_n$ of the Pedersen commitment scheme are randomly and uniformly chosen from $\mathbf{Z}_q$ by the session leader. However, this is not verifiable by other parties and a malicious leader can choose $g_j := h^{\xi_j} \bmod p$ for some secret $\xi_j \in \mathbf{Z}_q$ where $1 \leq j \leq n$. Thus it is importand to call `SetupGenerators_publiccoin` during game initialization before any shuffle verification is performed.

`GrothVSSHE (size_t` *n*, `std::istream&` *in*, `unsigned long int` *ell_e* `=TMCG_GROTH_L_E`, `unsigned long int` *fieldsize* `=TMCG_DDH_SIZE`, `unsigned long int` *subgroupsize* `=TMCG_DLSE_SIZE)` [Constructor on `GrothVSSHE`]

This constructor initializes the instance from a correctly formatted input stream *in*. For example, such a stream can be generated by calling the method `PublishGroup` of an already created instance. Later the instance can be used to show the correctness of a shuffle of at most *n* cards. The positive integer *ell_e* controls the soundness error probability of the protocol. The default value is defined by `TMCG_GROTH_L_E`, if this argument is omitted.

Note that the generators $g_1, \ldots, g_n$ of the Pedersen commitment scheme are randomly and uniformly chosen from $\mathbf{Z}_q$ by the session leader. However, this is not verifiable by other parties and a malicious leader can choose $g_j := h^{\xi_j} \bmod p$ for some secret $\xi_j \in \mathbf{Z}_q$ and $1 \leq j \leq n$. Thus it is necessary to call the method `SetupGenerators_publiccoin` before any shuffle verification is performed.

`void SetupGenerators_publiccoin (mpz_srcptr` *a*`)` [Method on `GrothVSSHE`]

This is a simple method to setup the generators $g_1, \ldots, g_n$ of the internal Pedersen commitment scheme by using a common random value *a* for a verifiable generation procedure similar to FIPS 186-3 A.2.3. Note that the same *a* must be used by all participants and that this value should be different for each game session.

`bool SetupGenerators_publiccoin (size_t` *whoami*, `aiounicast*` *aiou*, `CachinKursawePetzoldShoupRBC*` *rbc*, `JareckiLysyanskayaEDCF*` *edcf*, `std::ostream&` *err*`)` [Method on `GrothVSSHE`]

This method setup the generators $g_1, \ldots, g_n$ of the internal Pedersen commitment scheme by using a distributed coinflip protocol [JL00] and a verifiable generation procedure similar to FIPS 186-3 A.2.3. Assuming at least one honest player these values are randomly and uniformly chosen from $\mathbf{Z}_q$ such that $\log_{g_i} h$ and $\log_h g_i$ are unkown, for all $i = 1, \ldots, n$. The argument *whoami* is an index of the running instance with respect to already initialized instances of asynchronous point-to-point channels *aiou* and a reliable broadcast channel *rbc*. Logging and debug messages are printed to the provided output stream *err*. The method returns `true`, if all generators have been setup successfully.

`bool CheckGroup ()` [Method on `GrothVSSHE`]

This method checks whether the initialized commitment scheme is sound. It returns `true`, if all tests have been passed successfully.

`void PublishGroup (std::ostream& out)` [Method on `GrothVSSHE`]

This method exports the instance configuration to the output stream *out* such that other instances can be initialized, e.g. with the second constructor.

`~GrothVSSHE ()` [Destructor on `GrothVSSHE`]

This destructor releases all occupied resources.

2.2.3.5 Verifiable Rotation of Homomorphic Encryptions

De Hoogh, Schoenmakers, Skoric, and Villegas [HSSV09] has proposed an efficient solution to perform a verifiable rotation (also known as cyclic shift) of homomorphically encrypted values. Other solutions (e.g. Reiter and Wang, *Fragile Mixing*, ACM CCS, 2004) do not provide that level of efficency. LibTMCG provides the first known implementation of their protocol. It can only be used with the VTMF card encoding scheme of Barnett and Smart [BS03].

Further, to the best of our knowledge it is not known, whether their protocol retains the zero-knowledge property when it is executed in a concurrent setting. Thus the application programmer should be careful and avoid parallel invocations of the same instance.

`HooghSchoenmakersSkoricVillegasVRHE` [Class]

This class provides the low-level interface for their protocol. There are just a few methods that might be of general interest. All other components are only used internally by high-level operations and thus their description is omitted here.

`HooghSchoenmakersSkoricVillegasVRHE` [Constructor on `HooghSchoenmakersSkoricVillegasVRHE`]
`(mpz_srcptr p_ENC, mpz_srcptr q_ENC, mpz_srcptr k_ENC, mpz_srcptr g_ENC, mpz_srcptr h_ENC, unsigned long int fieldsize =TMCG_DDH_SIZE, unsigned long int subgroupsize =TMCG_DLSE_SIZE)`

This constructor creates a new instance. The low-level operations are later used to show the correctness of a rotation of the cards. The protocol and some of its parameters are initialized by the members of the corresponding VTMF instance. Consequently, *p_ENC* is the prime number p which determines the field $\mathbf{Z}/p\mathbf{Z}$, *q_ENC* is the order of the underlying subgroup G, i.e. the prime number q, and *k_ENC* is the integer such that $p = qk+1$ holds. Further, *g_ENC* is the generator g, and finally *h_ENC* is the common public key h. The *fieldsize* and the *subgroupsize* are supplied to internal classes and are only of interest, if *p_ENC* or *q_ENC* have lengths different from the default. If these arguments are omitted, they are set to `TMCG_DDH_SIZE` and `TMCG_DLSE_SIZE`, respectively.

This constructor should be instantiated only once by the session leader. All other instances must be created by the second constructor. Further, it is very important that the VTMF key generation protocol has been finished before the value of h is passed to the constructor. Otherwise, the correctness verification will definitely fail.

HooghSchoenmakersSkoricVillegasVRHE [Constructor on HooghSchoenmakersSkoricVillegasVRHE]
(std::istream& *in*, unsigned long int *fieldsize* =TMCG_DDH_SIZE,
unsigned long int *subgroupsize* =TMCG_DLSE_SIZE)

This constructor initializes the instance from a correctly formatted input stream *in*. For example, such a stream can be generated by calling the method PublishGroup of an already created instance. Later the instance can be used to show the correctness of a rotation.

bool CheckGroup () [Method on HooghSchoenmakersSkoricVillegasVRHE]

This method checks whether the initialized commitment scheme is sound. It returns true, if all tests have been passed successfully.

void PublishGroup [Method on HooghSchoenmakersSkoricVillegasVRHE]
(std::ostream& *out*)

This method exports the instance configuration to the output stream *out* such that other instances can be initialized, e.g. with the second constructor.

~HooghSchoenmakersSkoricVillegasVRHE [Destructor on HooghSchoenmakersSkoricVillegasVRHE]
()

This destructor releases all occupied resources.

2.2.3.6 Toolbox for Mental Card Games

This section explains the main class of LibTMCG which provides some "high-level operations" from Schindelhauer's toolbox [Sc98]. Even if the more efficient card encoding scheme of Barnett and Smart [BS03] will deployed in your application, at least one instance of the following class must be created to perform any card or stack operations.

SchindelhauerTMCG [Class]

This class implements the main core of Schindelhauer's toolbox, i.e. important functions like masking, opening, and shuffling of cards and stacks, respectively. Some exotic operations are still missing, e.g., the possibility to insert a masked card secretly into a stack or the verifiable subset properties of stacks. All implemented operations are available for the original encoding scheme of Schindelhauer (see Section 2.2.1 [Data Types], page 11) and, of course, for the more efficient encoding scheme of Barnett and Smart (see Section 2.2.3.2 [BarnettSmartVTMF], page 29) as well.

unsigned long int TMCG_SecurityLevel [Member of SchindelhauerTMCG]

This read-only nonnegative integer represents the security parameter κ which was given to the constructor of this class. It defines the number of sequential protocol iterations and hence the soundness error probability ($2^{-\kappa}$) of the zero-knowledge proofs in the encoding scheme of Schindelhauer. Further it defines the soundness error probability (also $2^{-\kappa}$) of the shuffle argument in the encoding scheme of Barnett and Smart, if the efficient protocols of Groth [Gr05, Gr10] and others [HSSV09] are not used.

size_t TMCG_Players [Member of SchindelhauerTMCG]

This read-only nonnegative integer represents the number of players as given to the constructor of this class.

size_t TMCG_TypeBits [Member of SchindelhauerTMCG]

This read-only nonnegative integer contains the number of bits that are necessary to encode the card types in the binary representation. It was given as an argument to the constructor of this class.

`SchindelhauerTMCG (const unsigned long int security, const size_t k, const size_t w)` [Constructor on `SchindelhauerTMCG`]

This constructor creates an instance, where *security* is a nonnegative integer that represents the security parameter κ. The parameter k is the number of players and w is the number of bits which are necessary to represent all possible card types in a binary representation.

The integer κ controls the maximum soundness error probability ($2^{-\kappa}$) of the zero-knowledge proofs in the encoding scheme of Schindelhauer. Specifically, *security* defines the number of sequential iterations of the involved protocols and thus has a major impact on the computational and communication complexity. If the encoding scheme of Barnett and Smart [BS03] is used, then it only defines the soundness error probability (also $2^{-\kappa}$) of the corresponding shuffle proof. However, if the efficient shuffle verification protocol of Groth [Gr05] is used, then the parameter *security* is dispensable, because the parameter *ell_e* given during instantiation of `GrothVSSHE` (e.g. the LibTMCG default security parameter `TMCG_GROTH_L_E`) determines this soundness error probability ($2^{-\ell_e}$). The similar holds for the verifiable rotation protocol [HSSV09], however, in this case there is no explicit security parameter for the soundness error.

Unfortunately, the parameters k and w have a major impact on the complexity in the encoding scheme of Schindelhauer, too. Therefore you should always use reasonable values here. For example, to create a deck with M different card types simply set w to $\lceil \log_2 M \rceil$ which is an tight upper-bound for the applied binary representation. Furthermore, set k to the number of players which are really involved and not to a possible maximum value. Note that k and w are limited by the global constants `TMCG_MAX_PLAYERS` and `TMCG_MAX_TYPEBITS`, respectively.

`void TMCG_CreateOpenCard (TMCG_Card& c, const TMCG_PublicKeyRing& ring, const size_t type)` [Method on `SchindelhauerTMCG`]

This method initializes the open card c with the given *type* using the encoding scheme of Schindelhauer. The *type* MUST be an integer from the interval $[0, 2^w - 1]$, where w is the number given to the constructor of this class. The w MUST be the same number as used at creation of c (see Section 2.2.1 [Data Types], page 11). The parameter *ring* is a container with exactly k public keys, where k is the number given to the constructor of this class. The k MUST be the same number as used at the creation of c.

`void TMCG_CreateOpenCard (VTMF_Card& c, BarnettSmartVTMF_dlog* vtmf, const size_t type)` [Method on `SchindelhauerTMCG`]

This method initializes the open card c with the given *type* using the encoding scheme of Barnett and Smart. The *type* MUST be an integer from the interval $[0, 2^w - 1]$, where w is the number given to the constructor of this class. The parameter *vtmf* is a pointer to an already initialized VTMF instance, i.e. the key generation protocol was successfully finished (see Section 2.2.3.2 [BarnettSmartVTMF], page 29, and `BarnettSmartVTMF_dlog_GroupQR`, respectively).

`void TMCG_CreateCardSecret (TMCG_CardSecret& cs, const TMCG_PublicKeyRing& ring, const size_t index)` [Method on `SchindelhauerTMCG`]

This method initializes the card secret *cs* with random values which is necessary to perform later a masking operation on a card. The parameter *ring* is a container with exactly k public keys, where k is the number given to the constructor of this class. It MUST be the same number as used at the creation of *cs* (see Section 2.2.1 [Data Types], page 11). The parameter *index* is from the interval $[0, k - 1]$ and determines the position of the players public key in the container *ring*.

`void TMCG_CreateCardSecret (VTMF_CardSecret& cs, BarnettSmartVTMF_dlog* vtmf)` [Method on `SchindelhauerTMCG`]

This method initializes the card secret *cs* with a random value which is necessary to perform later a masking operation on a card. The parameter *vtmf* is a pointer to an already initialized VTMF instance, i.e. the key generation protocol MUST be successfully finished (see Section 2.2.3.2 [BarnettSmartVTMF], page 29).

`void TMCG_CreatePrivateCard (TMCG_Card& c, TMCG_CardSecret& cs, const TMCG_PublicKeyRing& ring, const size_t index, const size_t type)` [Method on `SchindelhauerTMCG`]

This method initializes a masked card *c* with the given *type* and a corresponding card secret *cs* using the encoding scheme of Schindelhauer. The *type* MUST be an integer from the interval $[0, 2^w - 1]$, where w is the number given to the constructor of this class. The w MUST be the same number as used at creation of *c* and *cs* (see Section 2.2.1 [Data Types], page 11). The parameter *ring* is a container with exactly k public keys, where k is the number given to the constructor of this class. The k MUST be the same number as used at the creation of *c* and *cs*. The parameter *index* is from the interval $[0, k - 1]$ and determines the position of the players public key in the container *ring*. Internally, `TMCG_CreatePrivateCard` calls

1. `TMCG_CreateOpenCard` to initialize *c* with *type*,
2. `TMCG_CreateCardSecret` to initialize *cs* with random values, and
3. `TMCG_MaskCard` to mask *c* with the secret *cs*.

`void TMCG_CreatePrivateCard (VTMF_Card& c, VTMF_CardSecret& cs, BarnettSmartVTMF_dlog* vtmf, const size_t type)` [Method on `SchindelhauerTMCG`]

This method initializes a masked card *c* with the given *type* and a corresponding card secret *cs* using the encoding scheme of Barnett and Smart. The *type* MUST be an integer from the interval $[0, 2^w - 1]$, where w is the number given to the constructor of this class. The parameter *vtmf* is a pointer to an already initialized VTMF instance, i.e. the key generation protocol MUST be successfully finished (see Section 2.2.3.2 [BarnettSmartVTMF], page 29). Specifically, `TMCG_CreatePrivateCard` directly executes the masking operation of the verifiable masking protocol.

`void TMCG_MaskCard (const TMCG_Card& c, TMCG_Card& cc, const TMCG_CardSecret& cs, const TMCG_PublicKeyRing& ring, const bool TimingAttackProtection =true)` [Method on `SchindelhauerTMCG`]

This method performs a masking operation on the open or already masked card *c* using the encoding scheme of Schindelhauer. Finally it returns the result in *cc*. The parameter *cs* MUST be an initialized fresh card secret which has NEVER been involved in a masking operation before. The parameters *c*, *cc*, and *cs* MUST be created such that their k and w corresponds to the numbers given to the constructor of this class, respectively. The parameter *ring* is a container with exactly k public keys. The protection against timing attacks is turned on, if *TimingAttackProtection* is set to `true`.

`void TMCG_MaskCard (const VTMF_Card& c, VTMF_Card& cc, const VTMF_CardSecret& cs, BarnettSmartVTMF_dlog* vtmf, const bool TimingAttackProtection =true)` [Method on `SchindelhauerTMCG`]

This method performs a masking operation on the open or already masked card *c* using the encoding scheme of Barnett and Smart. Finally it returns the result in *cc*. Specifically, `TMCG_MaskCard` directly executes the masking operation of the verifiable re-masking protocol. The parameter *cs* MUST be an initialized fresh card secret which has NEVER been involved in a masking operation before. The parameter *vtmf* is a pointer to an already initialized VTMF instance, i.e. the key generation protocol MUST be successfully finished (see Section 2.2.3.2 [BarnettSmartVTMF], page 29). The protection against timing attacks is turned on, if *TimingAttackProtection* is set to `true`.

`void TMCG_ProveMaskCard (const TMCG_Card&` *c*, `const TMCG_Card&` *cc*, `const TMCG_CardSecret&` *cs*, `const TMCG_PublicKeyRing&` *ring*, `std::istream&` *in*, `std::ostream&` *out*`)` [Method on `SchindelhauerTMCG`]

This method should be called by the prover after `TMCG_MaskCard` to show that he performed the masking operation correctly. The parameters *c*, *cc*, and *cs* are the input, the result, and the used card secret of `TMCG_MaskCard`, respectively. They MUST be created such that their *k* resp. *w* corresponds to the numbers given to the constructor of this class. The parameter *ring* is a container with exactly *k* public keys. The input/output protocol messages from and to the verifier are transmitted on the streams *in* and *out*, respectively.

`void TMCG_ProveMaskCard (const VTMF_Card&` *c*, `const VTMF_Card&` *cc*, `const VTMF_CardSecret&` *cs*, `BarnettSmartVTMF_dlog*` *vtmf*, `std::istream&` *in*, `std::ostream&` *out*`)` [Method on `SchindelhauerTMCG`]

This method should be executed by the prover after calling `TMCG_MaskCard` to show that he performed the masking operation correctly. Specifically, `TMCG_ProveMaskCard` directly calls the prove operation of the verifiable re-masking protocol. The parameters *c*, *cc*, and *cs* are the input, the result, and the used card secret of `TMCG_MaskCard`, respectively. The parameter *vtmf* is a pointer to an already initialized VTMF instance, i.e. the key generation protocol MUST be successfully finished. The input/output protocol messages from and to the verifier are transmitted on the streams *in* and *out*, respectively.

`bool TMCG_VerifyMaskCard (const TMCG_Card&` *c*, `const TMCG_Card&` *cc*, `const TMCG_PublicKeyRing&` *ring*, `std::istream&` *in*, `std::ostream&` *out*`)` [Method on `SchindelhauerTMCG`]

This method should be executed by the verifier to check whether or not a masking operation was performed correctly. The parameters *c* and *cc* are the input and the result of `TMCG_MaskCard`, respectively. They MUST be created such that their *k* resp. *w* corresponds to the numbers given to the constructor of this class. The parameter *ring* is a container with exactly *k* public keys. The input/output protocol messages from and to the prover are transmitted on the streams *in* and *out*, respectively. The method returns `true`, if everything was sound.

`bool TMCG_VerifyMaskCard (const VTMF_Card&` *c*, `const VTMF_Card&` *cc*, `BarnettSmartVTMF_dlog*` *vtmf*, `std::istream&` *in*, `std::ostream&` *out*`)` [Method on `SchindelhauerTMCG`]

This method should be executed by the verifier to check whether or not a masking operation was performed correctly. Specifically, `TMCG_VerifyMaskCard` directly calls the verify operation of the verifiable re-masking protocol. The parameters *c* and *cc* are the input and the result of `TMCG_MaskCard`, respectively. The parameter *vtmf* is a pointer to an already initialized VTMF instance, i.e. the key generation protocol MUST be successfully finished. The input/output protocol messages from and to the prover are transmitted on the streams *in* and *out*, respectively. The method returns `true`, if everything was sound.

`void TMCG_ProveCardSecret (const TMCG_Card&` *c*, `const TMCG_SecretKey&` *key*, `const size_t` *index*, `std::istream&` *in*, `std::ostream&` *out*`)` [Method on `SchindelhauerTMCG`]

This method is used to reveal the card type of *c* to a verifier. Every player must execute this method as prover. The card *c* MUST be created such that its *k* resp. *w* corresponds to the numbers given to the constructor of this class. The parameter *key* is the corresponding secret key (see Section 2.2.1 [Data Types], page 11) of the prover. The parameter *index* is from the interval $[0, k-1]$ and contains the position of the provers public key in the container *ring* (same as in `TMCG_CreateCardSecret`). The input/output protocol messages from and to the verifier are transmitted on the streams *in* and *out*, respectively.

`void TMCG_ProveCardSecret (const VTMF_Card& c, BarnettSmartVTMF_dlog* vtmf, std::istream& in, std::ostream& out)` [Method on `SchindelhauerTMCG`]

This method is used to reveal the card type of *c* to a verifier. Every player must execute this method as prover. Specifically, `TMCG_ProveCardSecret` directly calls the prove operation of the verifiable decryption protocol. The parameter *vtmf* is a pointer to an already initialized VTMF instance, i.e. the key generation protocol MUST be successfully finished. The input/output protocol messages from and to the verifier are transmitted on the streams *in* and *out*, respectively.

`bool TMCG_VerifyCardSecret (const TMCG_Card& c, TMCG_CardSecret& cs, const TMCG_PublicKey& key, const size_t index, std::istream& in, std::ostream& out)` [Method on `SchindelhauerTMCG`]

This method is used to verify and accumulate card type information regarding *c* that are supplied by a prover. It is the opposite method of `TMCG_ProveCardSecret` and must be executed by the player who wants to know the type. The secrets provided by the single provers are accumulated in the parameter *cs*. Thus *c* and *cs* MUST be created such that their k resp. w corresponds to the numbers given to the constructor of this class. The parameter *key* is the corresponding public key (see Section 2.2.1 [Data Types], page 11) of the prover. The parameter *index* is from the interval $[0, k-1]$ and contains the position of the provers public key in the container *ring* (same as in `TMCG_CreateCardSecret`). The input/output protocol messages from and to the prover are transmitted on the streams *in* and *out*, respectively.

`bool TMCG_VerifyCardSecret (const VTMF_Card& c, BarnettSmartVTMF_dlog* vtmf, std::istream& in, std::ostream& out)` [Method on `SchindelhauerTMCG`]

This method is used to verify and accumulate card type information regarding *c* that are supplied by a prover. It is the opposite method of `TMCG_ProveCardSecret` and must be executed by the player who wants to know the type. The secrets provided by the single provers are accumulated internally, thus this method cannot be interleaved with the opening of other cards. Specifically, `TMCG_VerifyCardSecret` directly calls the verify and update operation of the verifiable decryption protocol. The parameter *vtmf* is a pointer to an already initialized VTMF instance, i.e. the key generation protocol MUST be successfully finished. The input/output protocol messages from and to the verifier are transmitted on the streams *in* and *out*, respectively.

`void TMCG_SelfCardSecret (const TMCG_Card& c, TMCG_CardSecret& cs, const TMCG_SecretKey& key, const size_t index)` [Method on `SchindelhauerTMCG`]

This method is used to compute and accumulate card type information regarding *c*. Analogously to `TMCG_VerifyCardSecret` it must be executed by the player who wants to know the type of *c*. The information is accumulated in the parameter *cs*. Thus *c* and *cs* MUST be created such that their k resp. w corresponds to the numbers given to the constructor of this class. The parameter *key* is the corresponding secret key (see Section 2.2.1 [Data Types], page 11) of the player. The parameter *index* is from the interval $[0, k-1]$ and contains the position of the players public key in the container *ring* (same as in `TMCG_CreateCardSecret`).

void TMCG_SelfCardSecret (const VTMF_Card& *c*, BarnettSmartVTMF_dlog* *vtmf*) [Method on SchindelhauerTMCG]

This method is used to compute and accumulate card type information regarding *c*. It MUST be called by the player who wants to know the type of *c* BEFORE TMCG_VerifyCardSecret and TMCG_TypeOfCard are executed. The secrets provided by the player are accumulated internally, thus this method cannot be interleaved with the opening of other cards. Specifically, TMCG_SelfCardSecret directly calls the initialize operation of the verifiable decryption protocol. The parameter *vtmf* is a pointer to an already initialized VTMF instance, i.e. the key generation protocol MUST be successfully finished.

size_t TMCG_TypeOfCard (const TMCG_CardSecret& *cs*) [Method on SchindelhauerTMCG]

This method returns the type of a masked card provided that the type information were properly accumulated in *cs* before (by calling TMCG_SelfCardSecret and TMCG_VerifyCardSecret, respectively).

size_t TMCG_TypeOfCard (const VTMF_Card& *c*, BarnettSmartVTMF_dlog* *vtmf*) [Method on SchindelhauerTMCG]

This method returns the type of a masked card *c* provided that the type information regarding *c* were properly accumulated internally before (by calling TMCG_SelfCardSecret and TMCG_VerifyCardSecret, respectively). It returns the value TMCG_MaxCardType, if the opening operation failed or if the card type was not among the set of valid types. This method MUST be performed by the player who wants to know the type AFTER TMCG_SelfCardSecret and TMCG_VerifyCardSecret are executed. Specifically, TMCG_TypeOfCard directly calls the finalize operation of the verifiable decryption protocol. The parameter *vtmf* is a pointer to an already initialized VTMF instance, i.e. the key generation protocol MUST be successfully finished.

size_t TMCG_CreateStackSecret (TMCG_StackSecret<TMCG_CardSecret>& *ss*, const bool *cyclic*, const TMCG_PublicKeyRing& *ring*, const size_t *index*, const size_t *size*) [Method on SchindelhauerTMCG]

This method initializes the stack secret *ss* with a randomly and uniformly chosen permutation (using the algorithm of Knuth) and fresh card secrets. Later this stack secret can be used to perform a secret shuffle operation on a stack. If the parameter *cyclic* is set to true, then the permutation is only a cyclic shift which might be of interest for particular operations, e.g. cutting the deck. The parameter *ring* is a container with exactly k public keys, where k is the number given to the constructor of this class. The parameter *index* is from the interval $[0, k-1]$ and contains the position of the players public key in the container *ring*. The parameter *size* determines the size of the created stack secret, i.e. the number of cards in the corresponding stack. The *size* is upper-bounded by TMCG_MAX_CARDS. The method returns the offset of the cyclic shift, if *cyclic* was set to true. Otherwise, the value 0 is returned.

size_t TMCG_CreateStackSecret (TMCG_StackSecret<VTMF_CardSecret>& *ss*, const bool *cyclic*, const size_t *size*, BarnettSmartVTMF_dlog* *vtmf*) [Method on SchindelhauerTMCG]

This method initializes the stack secret *ss* with a randomly and uniformly chosen permutation (using the algorithm of Knuth) and fresh card secrets. Later this stack secret can be used to perform a secret shuffle operation on a stack. If the parameter *cyclic* is set to true, then the permutation is only a cyclic shift which might be of interest for particular operations, e.g. cutting the deck. The parameter *size* determines the size of the created stack secret, i.e. the number of cards in the corresponding stack. The *size* is upper-bounded by TMCG_MAX_CARDS. The parameter *vtmf* is a pointer to an already initialized VTMF instance, i.e. the key generation protocol MUST be successfully finished.

The method returns the offset of the cyclic shift, if *cyclic* was set to `true`. Otherwise, the value `0` is returned.

`void TMCG_CreateStackSecret (TMCG_StackSecret<TMCG_CardSecret>& ss, const std::vector<size_t>& pi, const TMCG_PublicKeyRing& ring, const size_t index, const size_t size)` [Method on `SchindelhauerTMCG`]

This method initializes the stack secret *ss* with a given permutation *pi* and fresh card secrets. Later this stack secret can be used to perform a secret shuffle operation on a stack. The parameter *ring* is a container with exactly k public keys, where k is the number given to the constructor of this class. The parameter *index* is from the interval $[0, k-1]$ and contains the position of the players public key in the container *ring*. The parameter *size* determines the size of the created stack secret, i.e. the number of cards in the corresponding stack. The *size* is upper-bounded by `TMCG_MAX_CARDS`.

`void TMCG_CreateStackSecret (TMCG_StackSecret<VTMF_CardSecret>& ss, const std::vector<size_t>& pi, const size_t size, BarnettSmartVTMF_dlog* vtmf)` [Method on `SchindelhauerTMCG`]

This method initializes the stack secret *ss* with a given permutation *pi* and fresh card secrets. Later this stack secret can be used to perform a secret shuffle operation on a stack. The parameter *size* determines the size of the created stack secret, i.e. the number of cards in the corresponding stack. The *size* is upper-bounded by `TMCG_MAX_CARDS`. The parameter *vtmf* is a pointer to an already initialized VTMF instance, i.e. the key generation protocol MUST be successfully finished.

`void TMCG_MixStack (const TMCG_Stack<TMCG_Card>& s, TMCG_Stack<TMCG_Card>& s2, const TMCG_StackSecret<TMCG_CardSecret>& ss, const TMCG_PublicKeyRing& ring, const bool TimingAttackProtection =true)` [Method on `SchindelhauerTMCG`]

This method shuffles a given stack *s* according to the previously created stack secret *ss* (see Section 2.2.1 [Data Types], page 11). The result of the shuffle is returned in *s2*. The parameter *ss* MUST be a fresh stack secret which has NEVER been involved in a shuffle operation before. The parameters *s* and *ss* MUST be of the same size. The parameter *ring* is a container with exactly k public keys, where k is the number given to the constructor of this class. The protection against timing attacks is turned on, if *TimingAttackProtection* is set to `true`.

`void TMCG_MixStack (const TMCG_Stack<VTMF_Card>& s, TMCG_Stack<VTMF_Card>& s2, const TMCG_StackSecret<VTMF_CardSecret>& ss, BarnettSmartVTMF_dlog* vtmf, const bool TimingAttackProtection =true)` [Method on `SchindelhauerTMCG`]

This method shuffles a given stack *s* according to the previously created stack secret *ss* (see Section 2.2.1 [Data Types], page 11). The result of the shuffle is returned in *s2*. The parameter *ss* MUST be a fresh stack secret which has NEVER been involved in a shuffle operation before. The parameters *s* and *ss* MUST be of the same size. The parameter *vtmf* is a pointer to an already initialized VTMF instance, i.e. the key generation protocol MUST be successfully finished. The protection against timing attacks is turned on, if *TimingAttackProtection* is set to `true`.

void TMCG_ProveStackEquality (const TMCG_Stack<TMCG_Card>& *s*, const TMCG_Stack<TMCG_Card>& *s2*, const TMCG_StackSecret<TMCG_CardSecret>& *ss*, const bool *cyclic*, const TMCG_PublicKeyRing& *ring*, const size_t *index*, std::istream& *in*, std::ostream& *out*) [Method on SchindelhauerTMCG]

This method should be called by the prover after TMCG_MixStack to show that he performed the shuffle operation correctly. The parameters *s*, *s2*, and *ss* are the input, the result, and the used stack secret of TMCG_MixStack, respectively. Of course, the parameters *s*, *s2*, and *ss* MUST be of the same size. The parameter *cyclic* determines whether a cyclic shift or a full permutation was used to shuffle the stack. The parameter *ring* is a container with exactly k public keys, where k is the number given to the constructor of this class. The parameter *index* is from the interval $[0, k-1]$ and contains the position of the provers public key in the container *ring*. The input/output protocol messages from and to the verifier are transmitted on the streams *in* and *out*, respectively.

void TMCG_ProveStackEquality (const TMCG_Stack<VTMF_Card>& *s*, const TMCG_Stack<VTMF_Card>& *s2*, const TMCG_StackSecret<VTMF_CardSecret>& *ss*, const bool *cyclic*, BarnettSmartVTMF_dlog* *vtmf*, std::istream& *in*, std::ostream& *out*) [Method on SchindelhauerTMCG]

This method should be called by the prover after TMCG_MixStack to show that he performed the shuffle operation correctly. The parameters *s*, *s2*, and *ss* are the input, the result, and the used stack secret of TMCG_MixStack, respectively. Of course, the parameters *s*, *s2*, and *ss* MUST be of the same size. The parameter *cyclic* determines whether a cyclic shift or a full permutation was used to shuffle the stack. The parameter *vtmf* is a pointer to an already initialized VTMF instance, i.e. the key generation protocol MUST be successfully finished. The input/output protocol messages from and to the verifier are transmitted on the streams *in* and *out*, respectively.

void TMCG_ProveStackEquality_Groth (const TMCG_Stack<VTMF_Card>& *s*, const TMCG_Stack<VTMF_Card>& *s2*, const TMCG_StackSecret<VTMF_CardSecret>& *ss*, BarnettSmartVTMF_dlog* *vtmf*, GrothVSSHE* *vsshe*, std::istream& *in*, std::ostream& *out*) [Method on SchindelhauerTMCG]

This is a method like above. The only difference is that the more efficient interactive shuffle verification protocol of Groth [Gr05] is used. Thus *vsshe* is a pointer to a proper initialized instance of GrothVSSHE. The rest of the arguments are the same.

void TMCG_ProveStackEquality_Groth_noninteractive (const TMCG_Stack<VTMF_Card>& *s*, const TMCG_Stack<VTMF_Card>& *s2*, const TMCG_StackSecret<VTMF_CardSecret>& *ss*, BarnettSmartVTMF_dlog* *vtmf*, GrothVSSHE* *vsshe*, std::ostream& *out*) [Method on SchindelhauerTMCG]

This is a method like above. The difference is that the non-interactive version of the shuffle verification protocol is used. Thus only an output stream *out* is given, for example std::stringstream can be appropriate here. Again *vsshe* is a pointer to a proper initialized instance of GrothVSSHE. The rest of the arguments are the same.

void TMCG_ProveStackEquality_Hoogh (const TMCG_Stack<VTMF_Card>& *s*, const TMCG_Stack<VTMF_Card>& *s2*, const TMCG_StackSecret<VTMF_CardSecret>& *ss*, BarnettSmartVTMF_dlog* *vtmf*, HooghSchoenmakersSkoricVillegasVRHE* *vrhe*, std::istream& *in*, std::ostream& *out*) [Method on SchindelhauerTMCG]

This is a method like above. The only difference is that the more efficient rotation verification protocol [HSSV09] is used. Thus *vrhe* is a pointer to an initialized instance of HooghSchoenmakersSkoricVillegasVRHE. The rest of the arguments are the same.

void TMCG_ProveStackEquality_Hoogh_noninteractive (const TMCG_Stack<VTMF_Card>& *s*, const TMCG_Stack<VTMF_Card>& *s2*, const TMCG_StackSecret<VTMF_CardSecret>& *ss*, BarnettSmartVTMF_dlog* *vtmf*, HooghSchoenmakersSkoricVillegasVRHE* *vrhe*, std::ostream& *out*) [Method on SchindelhauerTMCG]

This is a method like above. The difference is that the non-interactive version of the rotation verification protocol is used. Thus only an output stream *out* is given, for example std::stringstream can be appropriate here. Again *vrhe* is a pointer to an initialized instance of HooghSchoenmakersSkoricVillegasVRHE. The rest of the arguments are the same.

bool TMCG_VerifyStackEquality (const TMCG_Stack<TMCG_Card>& *s*, const TMCG_Stack<TMCG_Card>& *s2*, const bool *cyclic*, const TMCG_PublicKeyRing& *ring*, std::istream& *in*, std::ostream& *out*) [Method on SchindelhauerTMCG]

This method should be executed by the verifier to check whether or not a shuffle operation was performed correctly. The parameters *s* and *s2* are the input and the result of TMCG_MixStack, respectively. Of course, the parameters *s* and *s2* should be of the same size. The parameter *cyclic* determines whether a cyclic shift or a full permutation was used to shuffle the stack. The parameter *ring* is a container with exactly k public keys, where k is the number given to the constructor of this class. The input/output protocol messages from and to the prover are transmitted on the streams *in* and *out*, respectively. This method returns true, if the shuffle operation was successfully verified.

bool TMCG_VerifyStackEquality (const TMCG_Stack<VTMF_Card>& *s*, const TMCG_Stack<VTMF_Card>& *s2*, const bool *cyclic*, BarnettSmartVTMF_dlog* *vtmf*, std::istream& *in*, std::ostream& *out*) [Method on SchindelhauerTMCG]

This method should be executed by the verifier to check whether or not a shuffle operation was performed correctly. The parameters *s* and *s2* are the input and the result of TMCG_MixStack, respectively. Of course, the parameters *s* and *s2* should be of the same size. The parameter *cyclic* determines whether a cyclic shift or a full permutation was used to shuffle the stack. The parameter *vtmf* is a pointer to an already initialized VTMF instance, i.e. the key generation protocol MUST be successfully finished. The input/output protocol messages from and to the verifier are transmitted on the streams *in* and *out*, respectively. This method returns true, if the shuffle operation was successfully verified.

bool TMCG_VerifyStackEquality_Groth (const TMCG_Stack<VTMF_Card>& *s*, const TMCG_Stack<VTMF_Card>& *s2*, BarnettSmartVTMF_dlog* *vtmf*, GrothVSSHE* *vsshe*, std::istream& *in*, std::ostream& *out*) [Method on SchindelhauerTMCG]

This is a method like above. The only difference is that the more efficient shuffle verification protocol of Groth is used. Thus *vsshe* is a pointer to an initialized instance of GrothVSSHE. The rest of the arguments and the returned values are the same.

bool TMCG_VerifyStackEquality_Groth_noninteractive (const TMCG_Stack<VTMF_Card>& *s*, const TMCG_Stack<VTMF_Card>& *s2*, BarnettSmartVTMF_dlog* *vtmf*, GrothVSSHE* *vsshe*, std::istream& *in*) [Method on SchindelhauerTMCG]

This is a method like above. The difference is that the non-interactive version of the shuffle verification protocol is used. Thus only an input stream *in* is given, for example std::stringstream can be appropriate here. Again *vsshe* is a pointer to an initialized instance of GrothVSSHE. The rest of the arguments and the returned values are the same.

`bool TMCG_VerifyStackEquality_Hoogh (const TMCG_Stack<VTMF_Card>& s, const TMCG_Stack<VTMF_Card>& s2, BarnettSmartVTMF_dlog* vtmf, HooghSchoenmakersSkoricVillegasVRHE* vrhe, std::istream& in, std::ostream& out)` [Method on `SchindelhauerTMCG`]

This is a method like above. The only difference is that the more efficient rotation verification protocol [HSSV09] is used. Thus *vrhe* is a pointer to an initialized instance of `HooghSchoenmakersSkoricVillegasVRHE`. The rest of the arguments and the returned values are the same.

`bool TMCG_VerifyStackEquality_Hoogh_noninteractive (const TMCG_Stack<VTMF_Card>& s, const TMCG_Stack<VTMF_Card>& s2, BarnettSmartVTMF_dlog* vtmf, HooghSchoenmakersSkoricVillegasVRHE* vrhe, std::istream& in)` [Method on `SchindelhauerTMCG`]

This is a method like above. The difference is that the non-interactive version of the rotation verification protocol is used. Thus only an input stream *in* is given, for example `std::stringstream` can be appropriate here. Again *vrhe* is a pointer to an initialized instance of `HooghSchoenmakersSkoricVillegasVRHE`. The rest of the arguments and the returned values are the same.

`~SchindelhauerTMCG ()` [Destructor on `SchindelhauerTMCG`]

This destructor releases all occupied resources.

3 Examples

The following examples explain most of the steps that are necessary to create a secure and verifiable card game with LibTMCG. We consider an application with five permanent players (denoted by P_0, P_1, P_2, P_3, and P_4) and a regular deck of 52 different cards. For convenience only the more efficient card encoding scheme of Barnett and Smart [BS03] is described. Additionally, we complete our exposition with code fragments which show the application of the fast shuffle verification protocol due to Groth [Gr05, Gr10] with an interactive or even non-interactive instantiation of the zero-knowledge proofs. On modern computers this approach achieves good real world performance and simultaneously keeps the cheating probability negligible.

Throughout the remaining pages we assume that all players are pairwise connected by authenticated communication channels. These channels are organized in input resp. output streams, where `input_stream[i]` resp. `output_stream[i]` denote the corresponding `std::istream` resp. `std::ostream` instance for the communication with player P_i.[1]

3.1 Library Initialization

The very first step that should be done is the initialization of LibTMCG. You can simply perform this task by calling the function `init_libTMCG` and evaluating the return code.

```
if (!init_libTMCG())
  std::cerr << "Initialization of LibTMCG failed!" << std::endl;
```

Additionally, in most cases it is useful to check the installed version of the library by comparing the desired value with the returned string of the function `version_libTMCG`.

3.2 Setup Communication Channels

Some multiparty protocols require additional asynchronous point-to-point communication channels (authenticated and private) and a reliable broadcast channel. The following example shows, how to setup these channels for player P_i:

```
// create asynchronous private unicast channels
aiounicast_select *aiou = new aiounicast_select(5, i, uP_in, uP_out, uP_key,
  aiounicast::aio_scheduler_roundrobin, aiounicast::aio_timeout_short);

// create asynchronous private broadcast channels
aiounicast_select *aiou2 = new aiounicast_select(5, i, bP_in, bP_out, bP_key,
  aiounicast::aio_scheduler_roundrobin, aiounicast::aio_timeout_short);

// create an instance of a reliable broadcast protocol (RBC)
std::string myID = "example-poker-libTMCG-reference-manual";
CachinKursawePetzoldShoupRBC *rbc = new CachinKursawePetzoldShoupRBC(5, 1, i,
  aiou2, aiounicast::aio_scheduler_roundrobin, aiounicast::aio_timeout_short);
rbc->setID(myID);
```

We assume that pairwise private keys (e.g. passphrases) have been exchanged (i.e. vector `uP_key` resp. `bP_key`) and point-to-point links (i.e. input file descriptors in vector `uP_in` resp. `bP_in` and output file descriptors in vector `uP_out` resp. `bP_out`) have been already established.

[1] We assume that the players are ordered in a natural way such that we can use an uniform nomenclature.

3.3 Session Initialization and Key Generation

In the next step we create an instance of the class `SchindelhauerTMCG`. The first parameter determines the number of protocol iterations κ which upper-bounds the cheating probability by $2^{-\kappa}$. In our example the used value `64` defines a maximum cheating probability of $5.421010862 \cdot 10^{-20}$ which is reasonable small for our purposes.[2] The second parameter passes the number of players to the instance which is simply `5` in our case. The last argument defines the number of bits that are necessary to encode all card types in a binary representation. The given value `6` allows the encoding of $2^6 = 64$ different card types at maximum. This is enough to form our deck of 52 cards.

```
SchindelhauerTMCG *tmcg = new SchindelhauerTMCG(64, 5, 6);
```

In our example we would like to use the more efficient encoding scheme of Barnett and Smart, thus we create an instance of `BarnettSmartVTMF_dlog`. However, a particular player has to act as a *leader* who performs the generation of the group G as a common refrence. In our case P_0 will be the session leader. First, he executes the constructor of the class `BarnettSmartVTMF_dlog` that may take some time.

```
BarnettSmartVTMF_dlog *vtmf = new BarnettSmartVTMF_dlog();
```

Afterwards he checks the generated group G and sends the public parameters to all other players (their corresponding stream indices are `1`, `2`, `3`, and `4`, respectively).

```
if (!vtmf->CheckGroup())
  std::cerr << "Group G was not correctly generated!" << std::endl;
for (size_t i = 1; i < 5; i++)
  vtmf->PublishGroup(output_stream[i]);
```

The other players receive the group parameters from P_0 and use them to initialize their corresponding instances of `BarnettSmartVTMF_dlog`. It is very important that they also check, whether the group G was correctly generated by the leader.

```
BarnettSmartVTMF_dlog *vtmf =
  new BarnettSmartVTMF_dlog(input_stream[0]);
if (!vtmf->CheckGroup())
  std::cerr << "Group G was not correctly generated!" << std::endl;
```

Afterwards the key generation protocol is carried out. First, every player generates his own VTMF key. The private key material is stored internally and will never be exposed.

```
vtmf->KeyGenerationProtocol_GenerateKey();
```

Then every player P_j sends the public part of his VTMF key along with a non-interactive zero-knowledge proof of knowledge (NIZK) to each other player. The appended proof shows that he indeed knows the corresponding secret key. However, due to the non-interactive nature of this proof we have to be careful, if the same group G is eventually used again. It is even better to generate a fresh group (common reference) and key for each new game session.

[2] If we use the encoding scheme of Barnett and Smart and only Groth's shuffle protocol during the game, then the error probability is even smaller, because the security parameters of them are fixed within LibTMCG (see Section 2.1 [Preprocessor Defined Global Symbols], page 8).

```
for (size_t i = 0; i < 5; i++)
{
  if (i != j)
    vtmf->KeyGenerationProtocol_PublishKey(output_stream[i]);
}
```

After sending, P_j receives the public keys of the other players. Of course she checks, whether these keys are correctly generated, and she updates the common public key h.

```
for (size_t i = 0; i < 5; i++)
{
  if (i != j)
  {
    if (!vtmf->KeyGenerationProtocol_UpdateKey(input_stream[i]))
      std::cerr << "Public key was not correctly generated!" << std::endl;
  }
}
```

Finally, every player must finalize the key generation protocol.

```
vtmf->KeyGenerationProtocol_Finalize();
```

For some sophisticated parts of LibTMCG a distributed coin flipping protocol is neccesary. It protects the honest-verifier zero-knowledge proofs or arguments against malicious verifiers. So, all players should execute as an initialization procedure:

```
JareckiLysyanskayaEDCF *edcf;
edcf = new JareckiLysyanskayaEDCF(5, 5, vtmf->p, vtmf->q, vtmf->g, vtmf->h);
if (!edcf->CheckGroup())
  std::cerr << "Group G was not correctly generated!" << std::endl;
```

If we want to use the more efficient shuffle verification protocol of Groth, then P_0 must also create an instance of `GrothVSSHE`. The first argument determines the maximum stack size of which the correctness of a shuffle will be proven. The other parameters are obtained from the former created VTMF instance `vtmf`. It is important that the key generation protocol has been finalized before the common public key h (i.e. `vtmf->h`) is passed, because this value is checked within.

```
GrothVSSHE *vsshe = new GrothVSSHE(52, vtmf->p, vtmf->q, vtmf->k,
  vtmf->g, vtmf->h);
```

Again, P_0 will send the public parameters of the VSSHE instance to all other players.

```
for (size_t i = 1; i < 5; i++)
  vsshe->PublishGroup(output_stream[i]);
```

The other players receive these parameters from the leader and use them to initialize their corresponding instances of `GrothVSSHE`. Again, it is important to check, whether the parameters were correctly chosen by the leader.

```
GrothVSSHE *vsshe = new GrothVSSHE(52, input_stream[0]);
if (!vsshe->CheckGroup())
  std::cerr << "VSSHE was not correctly generated!" << std::endl;
if (mpz_cmp(vtmf->h, vsshe->com->h))
  std::cerr << "VSSHE: Common public key does not match!" << std::endl;
if (mpz_cmp(vtmf->q, vsshe->com->q))
  std::cerr << "VSSHE: Subgroup order does not match!" << std::endl;
if (mpz_cmp(vtmf->p, vsshe->p) || mpz_cmp(vtmf->q, vsshe->q) ||
  mpz_cmp(vtmf->g, vsshe->g) || mpz_cmp(vtmf->h, vsshe->h))
    std::cerr << "VSSHE: Encryption scheme does not match!" << std::endl;
```

Last but not least the setup of some internal generators must be accomplished by all players in a verifiable way (see Section 2.2.3.4 [GrothVSSHE], page 34).[3]

```
std::stringstream err_log;
if (!vsshe->SetupGenerators_publiccoin(whoami, aiou, rbc, edcf, err_log))
  std::cerr << "VSSHE: SetupGenerators_publiccoin() failed!" << std::endl;
// synchronize
rbc->Sync(aiounicast::aio_timeout_short);
```

3.4 Operations on Cards

Now we are ready to perform several operations on cards. We start with some basic stuff which might be of interest in particular situations. However, it is often more convenient to work directly with stacks, as explained later.

3.4.1 Creating an Open Card

The creation of an open card is very simple. The following code creates a card of type 7.

```
VTMF_Card c;
tmcg->TMCG_CreateOpenCard(c, vtmf, 7);
```

3.4.2 Masking and Re-masking of a Card

Now the previously created card `c` will be masked to hide its type. Then `cc` is sent to P_1.

```
VTMF_Card cc;
VTMF_CardSecret cs;
tmcg->TMCG_CreateCardSecret(cs, vtmf);
tmcg->TMCG_MaskCard(c, cc, cs, vtmf);
out_stream[1] << cc << std::endl;
```

P_1 receives the card `cc`, re-masks them, and sends the result `ccc` back to the player P_0. Further he proves that the masking operation was performed correctly.

[3] There is also the possibility to use the simple variant of `SetupGenerators_publiccoin` with the already generated public key h as a common random value. However, this value should be refreshed periodically.

```
VTMF_Card cc, ccc;
VTMF_CardSecret ccs;
in_stream[0] >> cc;
if (!in_stream[0].good())
  std::cerr << "Read or parse error!" << std::endl;
tmcg->TMCG_CreateCardSecret(ccs, vtmf);
tmcg->TMCG_MaskCard(cc, ccc, ccs, vtmf);
out_stream[0] << ccc << std::endl;
tmcg->TMCG_ProveMaskCard(cc, ccc, ccs, vtmf, in_stream[0], out_stream[0]);
```

P_0 receives the card, verifies the proof, and sends the card to all other players.

```
VTMF_Card ccc;
in_stream[1] >> ccc;
if (!tmcg->TMCG_VerifyMaskCard(cc, ccc, vtmf, in_stream[1], out_stream[1]))
  std::cerr << "Verification failed!" << std::endl;
for (size_t i = 1; i < 5; i++)
  out_stream[i] << ccc << std::endl;
```

Finally, all other players receive and store the masked card `ccc`.

3.4.3 Opening a Masked Card

Suppose that P_1 would like to know the type of the masked card `ccc`. Of course, P_0 could simply reveal it, but that isn't verifiable. Anyway, if all players cooperate, then P_1 can compute the type in a verifiable way. First, every player (except P_1) will execute the following code.

```
tmcg->TMCG_ProveCardSecret(ccc, vtmf, in_stream[1], out_stream[1]);
```

This sends the necessary data to P_1 and proves their correctness. On the other hand, P_1 will execute the following commands exactly in the given order. Finally, he obtain the card type in the variable `type`. Note that the corresponding function `TMCG_VerifyCardSecret` is not called for his own index `1`.

```
tmcg->TMCG_SelfCardSecret(ccc, vtmf);
for (size_t i = 0; i < 5; i++)
{
  if (i == 1)
    continue;
  if (!tmcg->TMCG_VerifyCardSecret(ccc, vtmf, in_stream[i], out_stream[i]))
    std::cerr << "Verification failed!" << std::endl;
}
type = tmcg->TMCG_TypeOfCard(ccc, vtmf);
```

Please notice that first `TMCG_SelfCardSecret` is called, then `TMCG_VerifyCardSecret`, and finally `TMCG_TypeOfCard`.

3.5 Operations on Stacks

There exist a lot of basic operations on stacks, e.g. pushing a card to a stack or importing a stack. These functions are to simple for explaining them here, but they are used implicitly. However, a short description can be found in the API part of the manual (see Section 2.2.1 [Data Types], page 11).

3.5.1 Creating the Deck

A quite common operation is the creation of a card deck. The deck will initially be represented by an open stack (see TMCG_OpenStack) called deck. Every player creates his own instance of the deck, which consists of 52 open cards of different type in our example.

```
TMCG_OpenStack<VTMF_Card> deck;
for (size_t type = 0; type < 52; type++)
{
  VTMF_Card c;
  tmcg->TMCG_CreateOpenCard(c, vtmf, type);
  deck.push(type, c);
}
```

Note that the instances of the deck must be consistent for all players, that means, the order of the open cards and their types must be exactly the same for all players.

Finally, we copy the deck to a regular stack s for further processing:

```
TMCG_Stack<VTMF_Card> s;
s.push(deck);
```

3.5.2 Shuffling the Deck

Every player must perform a shuffle of the deck, because only such a procedure guarantees that no coalition has influence on the outcome. Thus we build a shuffle chain (e.g. using the strict total order $P_i < P_j$, if and only if $i < j$) such that each player shuffles the deck once.

First the regular stack s is initialized with open cards from deck. Then each player shuffles the stack (see Section 2.2.3.6 [SchindelhauerTMCG], page 37, i.e, TMCG_MixStack) using randomness (see TMCG_CreateStackSecret) and proves the correctness of this operation (see TMCG_ProveStackEquality). Consequently, every player should verify these proofs (see TMCG_VerifyStackEquality) and complain deviations immediately. Finally, the stack s contains the shuffled result. Consider the following code fragment for the player P_j.

```
for (size_t i = 0; i < 5; i++)
{
  TMCG_Stack<VTMF_Card> s2;
  if (i == j)
  {
    TMCG_StackSecret<VTMF_CardSecret> ss;
    tmcg->TMCG_CreateStackSecret(ss, false, s.size(), vtmf);
    tmcg->TMCG_MixStack(s, s2, ss, vtmf);
    for (size_t i2 = 0; i2 < 5; i2++)
    {
      if (i2 == j)
        continue;
      out_stream[i2] << s2 << std::endl;
      tmcg->TMCG_ProveStackEquality(s, s2, ss, false, vtmf,
        in_stream[i2], out_stream[i2]);
    }
  }
  else
  {
    in_stream[i] >> s2;
    if (!in_stream[i].good())
      std::cerr << "Read or parse error!" << std::endl;
    if (!tmcg->TMCG_VerifyStackEquality(s, s2, false, vtmf,
      in_stream[i], out_stream[i]))
        std::cerr << "Verification failed!" << std::endl;
  }
  s = s2;
}
```

If you want to use the more efficient shuffle verification protocol of Groth, then you have to replace `TMCG_ProveStackEquality` and `TMCG_VerifyStackEquality` by `TMCG_ProveStackEquality_Groth` and `TMCG_VerifyStackEquality_Groth`, respectively.[4]

3.5.3 Drawing a Card from the Deck

Now every player has the same shuffled deck `s` and nobody knows in which order the 52 cards are stacked. Therefore you can simply use any drawing strategy to obtain a players hand. For example, look at the following code that draws two cards from `s` for each player.

```
TMCG_Stack<VTMF_Card> hand[5];
for (size_t i = 0; i < 5; i++)
{
  VTMF_Card c1, c2;
  s.pop(c1), s.pop(c2);
  hand[i].push(c1), hand[i].push(c2);
}
```

[4] The non-interactive version of Groth's protocol (see `TMCG_ProveStackEquality_Groth_noninteractive` and `TMCG_VerifyStackEquality_Groth_noninteractive`) provides an even more efficient implementation, because the prover has to compute the argument of correctness only once. Additionallly, it protects agains malicious verifiers and reduces the communication complexity, i.e. instead of $O(n^2)$ the prover must perform only $O(n)$ steps. Thus this approach is strongly recommended. However, the security then relies on the random oracle assumption. Please have a look at the included source code `tests/t-poker-noninteractive.cc` to get a clue.

Further, probably you want disclose the card types to the corresponding player. Consider the code fragment for the player P_j: Every player receives the necessary information from the other players and she computes the card types of her hand `hand[j]`. Finally, these types are stored together with the masked cards in the open stack `private_hand`.

```
TMCG_OpenStack<VTMF_Card> private_hand;
for (size_t i = 0; i < 5; i++)
{
  if (i == j)
  {
    for (size_t k = 0; k < hand[j].size(); k++)
    {
      tmcg->TMCG_SelfCardSecret(hand[j][k], vtmf);
      for (size_t i2 = 0; i2 < 5; i2++)
      {
        if (i2 == j)
          continue;
        if (!tmcg->TMCG_VerifyCardSecret(hand[j][k], vtmf,
          in_stream[i2], out_stream[i2]))
            std::cerr << "Verification failed!" << std::endl;
      }
      private_hand.push(tmcg->TMCG_TypeOfCard(hand[j][k], vtmf),
        hand[j][k]);
    }
  }
  else
  {
    for (size_t k = 0; k < hand[i].size(); k++)
    {
      tmcg->TMCG_ProveCardSecret(hand[i][k], vtmf,
        in_stream[i], out_stream[i]);
    }
  }
}
```

The example can be modified in a straightforward way to publicly disclose a card from a players hand or from the remaining stack `s`, i.e. to lay down the card face-up on the table.

3.6 Quit a Session

In the last step you should release all occupied resources.

```
delete vsshe, delete edcf, delete vtmf, delete tmcg;
delete rbc, delete aiou2, delete aiou;
```

4 Tools

LibTMCG provides some additional protocols that may be of independent interest.

4.1 Distributed Key Generation and Threshold Cryptography

We have implemented a robust and secure protocol for Distributed Key Generation (DKG) of public-key cryptosystems (see Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin: *Secure Distributed Key Generation for Discrete-Log Based Cryptosystems*, Journal of Cryptology, Vol. 20 Nr. 1, Springer 2007). Moreover, LibTMCG also provides a robust and secure protocol for threshold DSA/DSS (see Ran Canetti, Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin: *Adaptive Security for Threshold Cryptosystems*, Advances in Cryptology – Proceedings of CRYPTO '99, Lecture Notes in Computer Science 1666, Springer 1999). Robustness and security means that up to $t \leq n/2$ resp. $t \leq n/3$ parties can act maliciously and the protocols still produce some result (e.g. a valid DSA/DSS signature on a given hash value).

The current implementation is in *experimental state* and should not be used in production environments. Motivation, cryptographical background and some usage scenarios have been presented recently at 26th Krypto-Tag (GI Working Group) and Datengarten/81 (CCCB). Please consult the slides for a first overview. The former DKG tools have been removed from this release. These programs are continued in a separate package called *Distributed Privacy Guard* (DKGPG).

Please report any bugs to the maintainer of LibTMCG. Every help with development or testing of these DKG protocols and programs is very welcome!

Appendix A Licenses

A.1 GNU General Public License

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Lesser General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with

modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

   In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.
10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.
11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) yyyy  name of author

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'show c' for details.
```

The hypothetical commands '`show w`' and '`show c`' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than '`show w`' and '`show c`'; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the program
'Gnomovision' (which makes passes at compilers) written by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License.

A.2 GNU Free Documentation License

Version 1.3, 3 November 2008

0. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

   This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

   We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

   This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

   A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

   A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

   The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

   The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

   A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

   You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

   You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
   B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
   C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
   D. Preserve all the copyright notices of the Document.
   E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
   F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
   G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
   H. Include an unaltered copy of this License.
   I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
   J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History"

section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    "Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Appendix B General and API Index

General Index

API Index

!

<

=

>

[

~

A

B

C

T

U

V