# libgroupsig: An extensible C library for group signatures

Jesus Diaz, David Arroyo, and Francisco B. Rodriguez

Escuela Politécnica Superior
Universidad Autónoma de Madrid
{j.diaz,david.arroyo,f.rodriguez}@uam.es

**Abstract.** One major need in the context of Privacy Enhancing Technologies (PETs) is to bridge theoretical proposals and practical implementations. In order to foster easy deployment of PETs, the crux is on proposing standard and well-defined programming interfaces. This need is not completely fulfilled in the case of group signatures. Group signatures are key cryptographic primitives to build up privacy respectful protocols and endorsing fair management of anonymity. To the best of our knowledge, currently there exists no abstract and unified programming interface definition for group signatures. In this work we address this matter and propose a programming interface definition enclosing the functionality of current group signatures schemes. Furthermore, for the sake of abstraction and generalization, we have also endowed our interface with the means to include new group signatures schemes. Finally, we have considered three well known group signature schemes to implement an open source library of the interface using C programming language. We have also performed an analysis of the software implementation with respect to different values of the key size and other parameters of the group signatures interface.

## 1 Introduction

While the general principles of cryptographic primitives may be clear to programmers, their internal details are not usually so well understood [19]. In addition, most theoretical proposals in cryptography are oriented to very specific scenarios which makes difficult their inclusion in a wider set of practical contexts. Certainly, in most cases where a software implementation is provided to backup a theoretical contribution, it is very difficult to adapt it and (re-)use it in more complex systems demanding the functionality that this software provides [1]. An example of this situation is depicted by the creation and use of group signatures schemes [14]. Although there exist some implementations, either they seem to be currently unmaintained (like `libgs`[1]); or they just implement specific schemes (like the `FTMGS` library[2]) and thus they are only suitable for contexts

[1] `http://www.ing.unibs.it/ntw/tools/pp2db/`. Last access on August 5th, 2015.
[2] `http://www.lcc.uma.es/~vicente/swprj/index.html#libftmgs`. Last access on December 18th, 2014.

consistent with the properties of the implemented scheme. In order to overcome this problem we have adopted the recommendations in [31], and thus we have analyzed the most relevant works in the field of group signatures to extract the underlying principles and the involved functionalities. Correspondingly, we have designed an Application Programming Interface (API) and implemented a prototype using C programming language. As we will discuss, our the abstraction we have defined is basically an extension of the one described in the ISO/IEC standards [24, 25]. Regarding the implementation, we have used three types of group signatures as bottom line. Nevertheless, in the API design we have taken into account that specific group signature schemes may bear only a subset of the functionalities associated to the group signature primitive. As a result, our library supports the addition of new schemes without the need of modifying the existing code. On the basis of the main conclusions in [21], our implementation is open source[3] to enable its use in advanced privacy respectful systems, and to promote its revision and improvement through the collaboration of the whole PETs community.

The remainder of this work is organized as follows. In Section 2 we summarize the main advancements in the functionality of group signatures, by some of the most important schemes and make a review of existing implementations and standards on group signatures. In Section 3 we extrapolate this main functionality in order to create a unified interface which should be adaptable to the most important variants. Section 4 introduces the API of `libgroupsig`, detailing its architecture and main functions, and also pointing out some implementation notes. Section 5 describes the results of the performed benchmarks, and references further work in which we have also tested `libgroupsig`. Section 6 concludes with some future work. A includes some code snippets showing the invocation of several main functions, and B briefly describes how to extend the library by adding new schemes. Additionally, further (and constantly updated) documentation on the library is available at `bitbucket.org/jdiazvico/libgroupsig/`.

## 2 Related work

Group signatures were first proposed in [14] and, like conventional digital signatures, they are used to prove that the owner of a specific secret has been the source of some information. However, unlike their conventional counterpart, group signatures hide this owner among a set (group) of possible owners. Hence, group signatures can be used as a means to provide some sort of anonymity.

**Group signatures basics** After the initial proposal by Chaum and van Heyst [14], group signatures were further formalized including a security model [5]. This formalization assumed static groups, meaning that the group members are defined during the setup phase. This setting was subsequently extended to

[3] Available at `https://bitbucket.org/jdiazvico/libgroupsig/`. Last access on August 5th, 2015.

the case of dynamic groups in [6] and, independently, in [28, 27] both defining equivalent security models.

Focusing on the functionality provided by group signatures, several variations have been proposed to add new features around the central property of anonymity. For instance, there is typically a *Group Manager* who controls some secret information that allows her to revoke this anonymity and fetch the identity of the issuer of a group signature (this is called *opening* a group signature). But there also exist schemes, like *ring signatures* [34], that provide unconditional anonymity, meaning that the signature-opening functionality cannot be performed. Schemes as those in [27, 15] add an extra trapdoor besides the one used for opening group signatures, so that an authority (either the Group Manager or some subsidiary authority) is able to link signatures made by the same group member. In those schemes the authority performs this tracing using a *tracing trapdoor* instead of using her identity. This type of signatures are consequently named *traceable signatures*. It is also possible to apply Zero-Knowledge protocols [22] to claim having issued a specific group signature [27]. Even though the term is not used in the original paper, we could name variations supporting this functionality *claimable group signatures*. In [8] the trust placed in the Group Manager is divided across several authorities, which need to combine their secrets in order to be able to open some group signature. The authors call the result *fair signatures* or, rather, fair traceable signatures, since their proposal is based on traceable signatures (and also supports tracing). Besides these extensions to their functionality, their efficiency has also been refined in many ways. A detailed overview of the evolution of the computational and communication costs of the different schemes of group signatures is available at [29, Sec. 1.1].

**Standards and implementations of group signatures** Group signatures have been standardized in ISO/IEC 20008, which defines the general setting and main operations [24], and a total of 7 schemes with opening and linking capabilities [25]. Several implementations of different group signature schemes are currently available online. The group signature scheme in [36] is implemented as an example of use of the `PBC` library[4], and the proposal in [9] is implemented in C within the `PBC_sig` library[5]. The group signature defined in [9] and [16] are implemented in Python within the Charm framework[6] [1]. The scheme introduced in [8] is implemented in C in the `FTMGS` library[7], whereas the signatures described in [11] and [4] are implemented in the `libgs` library using Java, as part of the PP2db project[8]. The Java framework in [33] implements group signatures given in [13], [10] and [26]. A variant of the group signature scheme in

[4] `http://crypto.stanford.edu/pbc/`. Last access on August 5th, 2015.
[5] `http://crypto.stanford.edu/pbc/sig/`. Last access on August 5th, 2015.
[6] `https://code.google.com/p/charm-crypto/`. Last access on August 5th, 2015.
[7] `http://www.lcc.uma.es/~vicente/swprj/index.html#libftmgs`. Last access on December 18th, 2014.
[8] `http://www.ing.unibs.it/ntw/tools/pp2db/`. Last access on August 5th, 2015.

[20] is provided in the cryptonote's library[9]. Finally, the group signature scheme defined in [30] is included in the Crypto-book prototype[10].

From the previous implementations, it is worth noting that most of them are ad hoc implementations of specific schemes, programmed without the requirement of providing a common API for other equivalent schemes. In this concern the Charm framework is an exception. This framework has been created for rapid prototyping of cryptographic systems. In the specific case of group (or ring) signatures, however, it implements only the schemes in [9] and [16]. Those proposals only provide a subset of the functionality available from more general group signature schemes (see Section 3). Therefore, it is necessary to implement schemes with a richer functionality set in order to test whether or not the provided API would be practical enough for complex systems or applications. It is also worth noting the Java framework in [33]. However, in this case, despite implementing three different schemes within the ISO/IEC 20008 standard [25], the provided API is not uniform among schemes, and the functionality for interacting with each scheme depends on its internals. Thus, despite being a relevant effort for comparing the performance of different schemes, it does not provide an appropriate interface for practical systems or applications.

## 3 The basis for a common interface

According to the previous introduction to group signatures, the different operations may imply subtle differences through their implementation in different schemes. For example, it is possible to find differences in the way those operations are performed, or the implications of those operations with respect to the privacy of the group member. In other cases, some group signature schemes may just implement a subset of the above mentioned functionality. However, it is possible to abstract the functionality from the study of the main primitives. In Figure 1 we sketch an abstraction of all the operations provided by group signatures, mostly matching the ones described in [27, 24, 25]. Each operation is described as follows:

**Setup.** Generates and initializes the group and manager keys for any arbitrary group. All operations below are always related to a group initialized with this operation.

**Join.** The process by means of which new members join the group. It is typically divided in a phase run by the new member, who obtains a member key, and a phase run by the group manager, who updates the Group Membership List (GML).

**Sign.** The process of issuing a group signature.

**Verify.** The process for verifying a group signature.

**Claim.** The process for claiming ownership of a group signature.

[9] `https://github.com/AlbertWerner/cryptonotecoin`. Last access on August 5th, 2015.

[10] `https://github.com/jyale/crypto-book/`. Last access on August 5th, 2015.

**Equality proving.** The process by means of which the issuer of a set of group signatures proves that she has issued all the signatures within the set. This may be seen as a generalization of the claiming process, and is specially interesting in schemes allowing Zero-Knowledge claims, like [27, 15].

**Claim Verify.** The process of verifying a claim of a group signature.

**Verification of equality.** The verification counterpart for equality proving.

**Open.** Used for extracting the identity of the issuer of a specific group signature, possibly producing a proof of opening.

**Open verify.** Optionally, some schemes implement this operation, for verifying the correctness of the proof produced by the open process.

**Reveal.** Employed for extracting the tracing trapdoor of a group member.

**Trace.** Checks whether a group signature has been issued by a group member who has been somehow revoked, e.g., included in a Certificate Revocation List (CRL).

It is worth emphasizing that this abstraction is basically an extension of the one proposed in [24, 25]. In the ISO/IEC standards, a general model is given for *anonymous digital signature mechanisms* using a group public key (i.e., group signatures) and *anonymous digital signature mechanisms* using multiple public keys (i.e., ring signatures). For group signatures, the operations defined in [24, 25] are:

**Key generation.** In [24, 25], this process covers both our **setup** and **join** processes.

**Signature.** Equivalent to our **sign** process.

**Verification.** Equivalent to our **verify** process.

**Opening.** Equivalent to our **open** process. In [24, 25], there is an optional process related to **open** that allows to verify the correctness of the output of **open**. We have named here this process **open verify**.

**Linking.** Equivalent to our **trace** process.

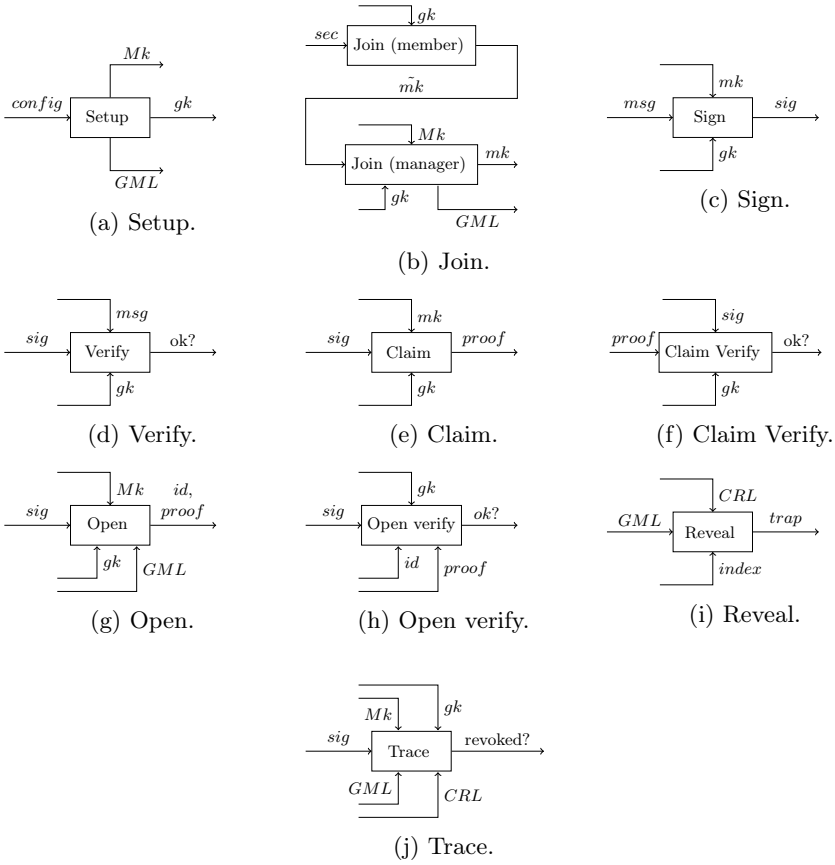
**Revocation.** In our interface, this functionality is incorporated within the **open** and **reveal** processes, depending on the received parameters. Through the former it is possible to revoke the anonymity of a group member, and through the latter it is possible to revoke the unlinkability of a group member.

Additionally, we have included support for several additional operations that are not described in [24, 25], but are implemented by some schemes, like [27, 15].

**Claiming.** Allows a group member to claim, in Zero Knowledge, ownership of a group signature.

**Equality proving.** A generalization of the claiming operation, where the issuer of a group signature is able to claim ownership of two or more group signatures.

**Claim verification.** Allows any third party to verify a claim produced using the claiming operation.




(a) Setup. (b) Join. (c) Sign.

(d) Verify. (e) Claim. (f) Claim Verify.

(g) Open. (h) Open verify. (i) Reveal.

(j) Trace.

Fig. 1: Main operations in group signatures. Incoming arrows depict inputs and outgoing arrows depict outputs. $Mk$ stands for Group Manager key, $gk$ for group key, $mk$ for member key, where $\tilde{mk}$ is a partially complete member key. The input $config$ depicts arbitrary configuration parameters, $id$ is any arbitrary secret allowing the identification of group members and $trap$ is the trapdoor used for tracing group members. The other tokens are self-explanatory.

**Equality proof verification.** The verification counterpart of the equality proving operation.

Finally, concerning the revocation capabilities of group signature schemes (i.e., the *Revocation* functionality), it is also worth noting that from the previous description we can distinguish two types of privacy properties built upon group signature schemes: *anonymity* and *unlinkability* [32]. First, anonymity is the property ensuring that no one will be able to learn the real identity of the issuer of a group signature. This property may be revoked (for the schemes that support it) through the *open* action. Second, unlinkability is the property guaranteeing

that no one will be able to determine whether two or more group signatures have been issued by the same signer. Equivalently, this property may be revoked by revealing the tracing trapdoor via the *reveal* action, and then using this trapdoor as an input parameter to the *trace* action.

## 4 Group signatures API

Next, we briefly describe the general interface that we have defined for interacting with the functionality provided by the `libgroupsig` library, according to the main operations introduced in Section 3. Figure 2 shows an UML-like class diagram, that depicts the described API in a component-wise manner. The main component is the one aimed to the interaction with group signature schemes, named `groupsig` in Figure 2. The functions defined within this component are:

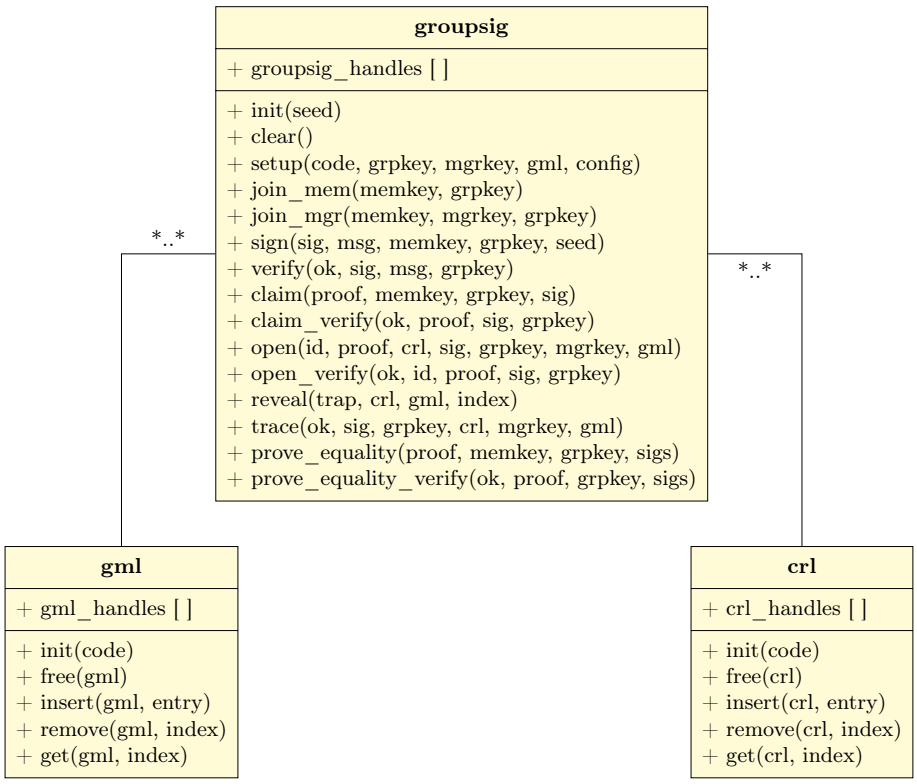



Fig. 2: UML-like class diagram for the `libgroupsig` API. For readability, we omit variable types.

`groupsig_init`. Initializes the library environment, including the internal Pseudo Random Number Generator.

`groupsig_clear`. Frees the internal variables initialized in the previous function.

`groupsig_setup`. Initializes the scheme with the specified code, filling the group key, manager key and GML. Uses the input parameters contained in the specified configuration structure for controlling the generation process.

`groupsig_join_mem`. Executes the join member part of the scheme. The member key will be updated with the member side generated keying information. Note that, in most schemes, there is typically a member side and a manager side of the join process, which may be used to prevent the manager from learning private tokens. If, nevertheless, the manager runs all the joining functionality, this function could just be left as a stub.

`groupsig_join_mgr`. Runs the manager side of join of the scheme. With it, the member key is completed, and a new entry related to the new member is added to the GML.

`groupsig_sign`. Runs the signing algorithm of the scheme and stores the resulting group signature in `sig`. The `seed` parameter is useful when reseeding the internal Pseudo Random Number Generator is necessary.

`groupsig_verify`. Verifies the given signature with the received message and group key.

`groupsig_claim`. Issues a Zero-Knowledge proof claiming having issued the specified signature for the given group and member keys.

`groupsig_claim_verify`. Verifies whether the given claim is correct for the specified group signature and group key.

`groupsig_open`. Returns the real identity of the issuer of the given signature and, optionally, a proof of opening. In our library, this may imply the addition of the identity into a CRL for members with revoked anonymity.

`groupsig_open_verify`. Verifies the proof of opening returned by `groupsig_open`.

`groupsig_reveal`. Reveals the tracing trapdoor of the member in position `index` within the given GML. In our library, this may imply the inclusion of the tracing trapdoor into a CRL for members with revoked unlinkability.

`groupsig_trace`. Determines whether or not the issuer of the specified signature has been revoked according to the given CRL.

`groupsig_prove_equality`. Creates a proof of equality of all the group signatures within the given set, using the specified member and group keys.

`groupsig_prove_equality_verify`. Verifies the given proof of equality, associated to the specified set of group signatures.

Besides the core functionality for group signatures, the library also includes two components for managing Group Membership Lists (GMLs) and Certificate Revocation Lists (CRLs). Also, for the sake of achieving a uniform API for all the schemes, we have not been completely strict on some matters. The following subsections summarize this.

### 4.1 GMLs and CRLs

The modules `gml` and `crl` are intended for the management of Group Membership Lists (GMLs) and Certificate Revocation Lists (CRLs), respectively. GMLs are lists of members, typically set up during the group initialization and updated each time a new member is added (or permanently removed). They contain important information that may be used when either anonymity or unlinkability revocation is required. CRLs are named after their equivalents in the X.509 infrastructure [35], but in this case they are employed within the extended setting created by group signatures [18]. That is, they are used for keeping a list of member keys for which either their anonymity or their unlinkability properties (or both) have been revoked (see Section 3).

The main operations provided within the `gml` and `crl` components are the ones typical of a list-like structure. Therefore, we allow the creation and liberation of these structures, the insertion (resp. removal) of new (resp. existing) elements through the insert (resp. remove) action, and the access to elements in the list through the get action.

### 4.2 Implementation notes

In the library, we have followed the abstraction outlined in Section 3 as interface for the group signatures functionality. Currently, the library incorporates three group signature schemes: KTY04 [27] and CPY06 [15], which are both traceable (group) signatures in the dynamic setting; and BBS04 [9], which is a group signature scheme in the dynamic setting, but without support for the (privacy respectful) tracing functionality. However, the library has been prepared so that adding and using new schemes is possible through the same interface. In this regard we have to take into account that not all group signatures actually provide the same functionality set. For the sake of a more unified API, we have been slightly loose when assigning names to each function, and simultaneously we have been cautious to avoid misleading potential users of our library.

For instance, in KTY04 and CPY06 there are two revocation functions: *open*, which given a group signature and (part of) the join transcript of a group member (stored within the Group Membership List in `libgroupsig`), returns the real identity of the signer; and *reveal*, which given (part of) the join transcript of a specific group member, returns a trapdoor that allows tracing him. In our library we refer to the respective parts of the join transcripts as *open trapdoor* and *tracing trapdoor*. However, BBS04 does not natively provide the same revocation options like that of KTY04 or CPY06, since it does not contain what we call tracing trapdoors. Nevertheless, tracing is still possible in BBS04, although in a less privacy respectful way. Indeed, what it is called *tracing* in BBS04 implies executing the *open* procedure (thus obtaining what we named *open trapdoor*) and looking for the signer's identity within a list of revoked members. Thus, it is not actually precise to use the term *reveal* with BBS04 to refer to the procedure defined with this name in KTY04 or CPY06. Nevertheless, in `libgroupsig` we use the term *reveal* to name a procedure that, given the part of the join transcript

of a specific group member used as tracing trapdoor, includes it within a CRL, which will be subsequently used for tracing. This allows us to create a unified API for similar functionality, although the inner cryptographic details (and privacy implications) may not be equivalent.

The library also contains additional modules for implementing functionality not directly related to group signatures, GMLs, or CRLs. This code is basically divided in mathematical functions (mostly some number theory algorithms) in a module named `math`; the `sys` module, which defines system-wide functions such as memory management functions, global constants and environment variables; and the `misc` module, which implements functionality for reading and printing information, type conversions, etc. There is also a component of the library for the management of group member identities in an abstract manner, i.e., for making GMLs independent on whether the programmer wants to include full names, job positions, etc. within the data stored in each GML entry.

`libgroupsig` is available under GNU LGPL at Bitbucket[11]. We have tested and applied it in our prototypes, but its development is still in an alpha stage and, by opening its source to the community, we expect to receive useful feedback to improve it.

## 5 Experimental evaluation

The acceptability of a software library is very dependent on the functionality achieved, but also on the efficiency of the final implementation. Therefore, after defining the interface, we have implemented it using the C programming language in order to analyze the costs associated to each of the supported actions explained in Section 4[12]. Since BBS04 and CPY06 use elliptic curve cryptography while KTY04 is RSA-based, we show the costs associated to key sizes providing roughly the same security level. Specifically, according to the NIST[13], the equivalences are as shown in Table 1. All the measurements have been obtained with a desktop PC (Intel Core i7-2600, 16GB DDR3 running Debian Wheezy), iterating 1000 times for each operation and using different keys in each iteration.

Figures 3b through 6 depict the costs associated to each of the main operations excluding tracing, for each of the implemented group signature schemes in our library. In all cases, the evolution starts to differ notably for keys larger than 3072 bits (for KTY04) and keys larger than 256 bits (for BBS04 and CPY06), with differences of at most a few tenth of seconds for smaller keys. Specifically,

[11] `https://bitbucket.org/jdiazvico/libgroupsig/`. Last access on August 5th, 2015.

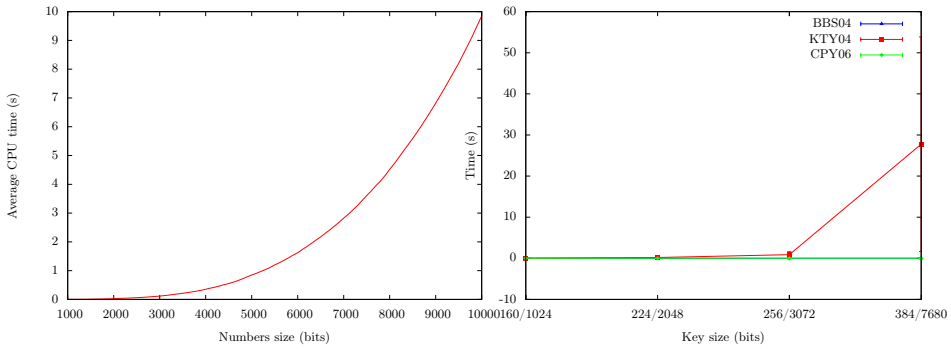
[12] We do not include measurements for the *prove equality* functionality (and its verification counterpart), which is a generalization of the claim action (resp., claim verify action). Thus, the specific cases give a good idea of the costs related to their more general counterparts.

[13] `http://www.nsa.gov/business/programs/elliptic_curve.shtml`. Last access on August 5th, 2015.

| RSA key size | ECC key size |
|---|---|
| 1024 | 160 |
| 2048 | 224 |
| 3072 | 256 |
| 7680 | 384 |

Table 1: Approximate key sizes providing equivalent security for ECC and RSA schemes.

for keys of size 7680 bits the costs of KTY04 increase abruptly, while the equivalent in BBS04 and CPY06 (384 bit keys) maintain a reasonable growth. The increase in KTY04 is most probably due to the costs associated of operating with larger numbers. Indeed, the three schemes rely on GNU GMP[14] (KTY04 uses it directly, while BBS04 and CPY06 through Ben Lynn's PBC library[15]). To verify this, we performed a profiling of GMP, based on the size of the employed numbers. Figure 3a shows the result. The profiling of GMP was performed in the same system as the one used for the analysis of `libgroupsig`. The tests involved 1000 iterations of GMP numbers ranging from 1000 to 10000 bits, increasing 100 bits per iteration (X-axis). Each iteration includes the basic operations: addition, multiplication, exponentiation and random number selection. The values in the Y-axis are the average running time for each iteration. It can be seen that the evolution of the CPU time in the profiling of GMP and that of the group signature schemes confirm our hypothesis.



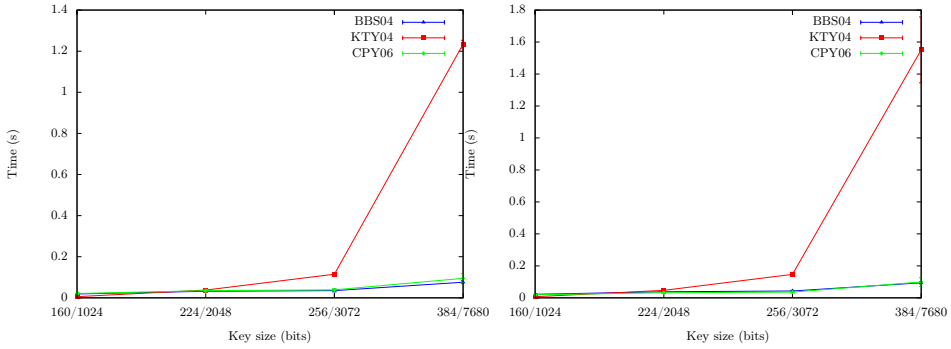

(a) Profiling of the GMP library.

(b) Costs of Join.

Fig. 3: Profiling of the GMP library (left) and costs of Join operations in KTY04, BBS04 and CPY06 (right). The increase in the costs of computing with GMP, depending on the size of the numbers is reflected in KTY04 in the Join operation (as well as in the operations shown below).

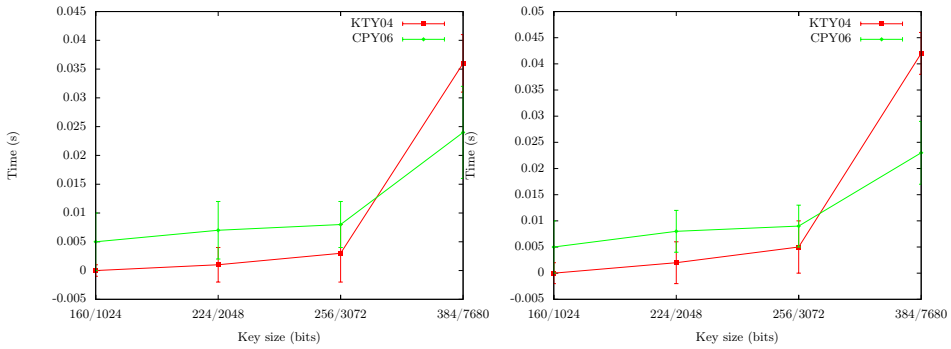
[14] `https://gmplib.org/`. Last access on August 5th, 2015.

[15] `http://crypto.stanford.edu/pbc/`. Last access on August 5th, 2015.




(a) Sign. (b) Verify.

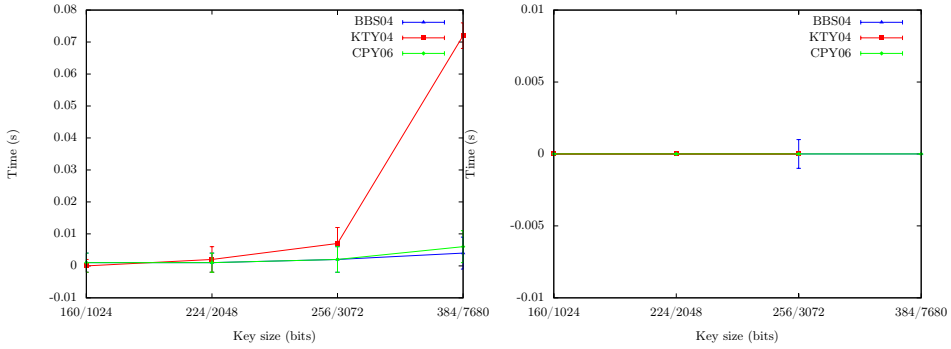
Fig. 4: Costs of Sign and Verify operations in KTY04, BBS04 and CPY06.




(a) Claim. (b) Claim Verify.

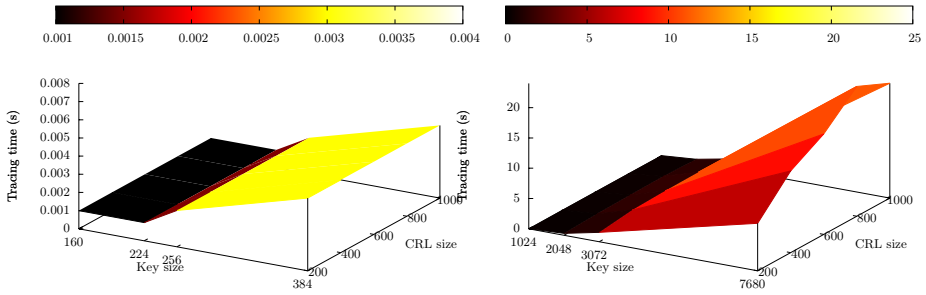
Fig. 5: Costs of Claim and Claim Verify operations in KTY04, BBS04 and CPY06.

For analyzing the costs of tracing, we need to consider both the size of the Certificate Revocation Lists (CRLs) and the keys. The graphs in Figure 7 show the evolution of the associated costs, given these parameters. BBS04 is by far the most efficient one (nevertheless, consider the observation made in Section 4.2), with costs always less than 0.008 seconds; KTY04 is also quite efficient up to keys of 3072 bits, but increases steeply from less than 5 seconds per tracing operation to more than 20 seconds when using keys of 7680 bits; finally, CPY06 is the most expensive in this operation, growing uniformly depending on the key and CRL size from 2 seconds to 18 seconds per tracing operation.

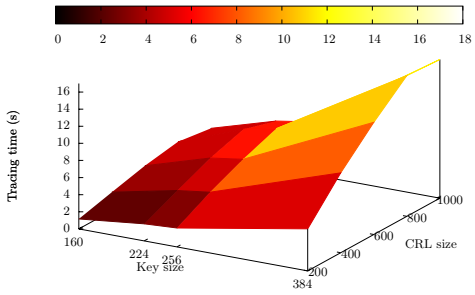



(a) Open. (b) Reveal.

Fig. 6: Costs of Open and Reveal operations in KTY04, BBS04 and CPY06.




(a) BBS04. (b) KTY04.




(c) CPY06.

Fig. 7: Costs of Trace in `libgroupsig`, for increasing key and CRL sizes.

Besides the tests summarized here, `libgroupsig` has also been employed in other research works where group signatures play a central role. Among other projects, we have used it for implementing a proof of concept of the extensions to X.509 in [17, 18].

## 6 Conclusion and future work

The current state of Information and Communication technologies demands the creation and proper implementation of new procedures to manage digital identities [3]. As we have underlined in [2], there is a lack of standard and thoroughly evaluated cryptographic libraries for the creation and use of non-conventional digital signatures. In this paper we have tackled this need for the concrete case of group signatures. This being the case, we have presented `libgroupsig`, a C library that provides a uniform API for different group signature schemes. As pointed out in Section 3, this API is basically an extension of the group signatures functionality described in the ISO/IEC standards [24, 25]. Moreover, the library supports the addition of new group signature schemes without needing to modify the already implemented code. This offers very interesting possibilities. For instance, new schemes may be seamlessly incorporated into our library (see B); or complex systems where privacy is of concern may use our library with a group signature scheme $A$, but switch to another scheme $B$ if needed with just updating a few tenths of lines of code at most (see A). To the best of our knowledge, no existing open source library provides equivalent possibilities.

Given the usefulness of group signatures as a building block for providing privacy, and the features of our library, we consider that our contribution may help in the development of advanced privacy respectful systems. In the past, open source libraries corresponding to advanced cryptographic primitives have served as catalysts for prototypes and complex cryptographic systems (see, e.g., the PBC library and all the projects that depend on it[16]). Keeping in mind the future work still pending we expect that our library might thus help in the creation of new advanced privacy respectful systems. The possibilities are many. For instance, we have already employed our library to implement the prototype of a comprehensive and privacy respectful e-commerce system, suitable for current e-commerce infrastructures. Also, one of the final applications we have in mind for this library is to conform the basis for X.509 extensions like the ones proposed in [7, 18]. This would undoubtedly suppose a great improvement of the X.509 PKI [35] towards supporting privacy respectful systems and applications. In [17, 18] we already applied this library for implementing a prototype of the mentioned extensions.

Nevertheless, as with every programming project, despite the library has reached a fully functional state, more work is required to improve it. First, the library is still in an alpha stage, and much testing is necessary for guaranteeing its correct functioning, including tests in as many different platforms as possible. In addition, the implementation of the currently supported schemes could

[16] `http://crypto.stanford.edu/pbc/who.html`. Last access on August 5th, 2015.

probably be improved (through code optimization) towards achieving better efficiency, and the inclusion of more schemes in the library will help in testing and refining its extensibility, and also help creating a richer range of schemes to choose from. For instance, note that none of the implemented schemes actually provides the functionality for verifying opening proofs. Although our API does support it, it is advisable to actually implement schemes (like [23]) that support this functionality. Future work will be also focused in incorporating more computationally schemes as those in [12] (where revocation is based on accumulators) and [29]. Finally, given that the final aim of this library is to be employed within cryptographic systems, a source code security audit is mandatory. In this regard it is relevant to underline that the publication of the library as open source contributes not only to its inclusion in more complex projects, but also its evaluation through the open source community.

## Acknowledgments

This work was supported by Comunidad de Madrid (Spain) under the project S2013/ICE-3095-CM (CIBERDINE), and by MINECO TIN2010-19607, TIN2012-30883, TIN2014-54580-R.

## References

[1] Joseph A. Akinyele, Christina Garman, Ian Miers, Matthew W. Pagano, Michael Rushanan, Matthew Green, and Aviel D. Rubin. Charm: a framework for rapidly prototyping cryptosystems. *J. Cryptographic Engineering*, 3(2):111–128, 2013.

[2] David Arroyo, Jesus Diaz, and VÃctor Gayoso. On the difficult tradeoff between security and privacy: Challenges for the management of digital identities. In Álvaro Herrero, Bruno Baruque, Javier Sedano, Héctor Quintián, and Emilio Corchado, editors, *International Joint Conference*, volume 369 of *Advances in Intelligent Systems and Computing*, pages 455–462. Springer International Publishing, 2015.

[3] David Arroyo, Jesus Diaz, and FranciscoB. Rodriguez. Non-conventional digital signatures and their implementations-a review. In Álvaro Herrero, Bruno Baruque, Javier Sedano, Héctor Quintián, and Emilio Corchado, editors, *International Joint Conference*, volume 369 of *Advances in Intelligent Systems and Computing*, pages 425–435. Springer International Publishing, 2015.

[4] Giuseppe Ateniese, Jan Camenisch, Marc Joye, and Gene Tsudik. A practical and provably secure coalition-resistant group signature scheme. In *Advances in Cryptology - CRYPTO 2000, 20th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2000, Proceedings*, pages 255–270, 2000.

[5] Mihir Bellare, Daniele Micciancio, and Bogdan Warinschi. Foundations of group signatures: Formal definitions, simplified requirements, and a construction based on general assumptions. In *Advances in Cryptology - EUROCRYPT 2003, International Conference on the Theory and Applications of Cryptographic Techniques, Warsaw, Poland, May 4-8, 2003, Proceedings*, pages 614–629, 2003.

[6] Mihir Bellare, Haixia Shi, and Chong Zhang. Foundations of group signatures: The case of dynamic groups. In *Topics in Cryptology - CT-RSA 2005, The Cryptographers' Track at the RSA Conference 2005, San Francisco, CA, USA, February 14-18, 2005, Proceedings*, pages 136–153, 2005.

[7] Vicente Benjumea, Seung Geol Choi, Javier Lopez, and Moti Yung. Anonymity 2.0 - X.509 extensions supporting privacy-friendly authentication. In *CANS*, pages 265–281, 2007.

[8] Vicente Benjumea, Seung Geol Choi, Javier Lopez, and Moti Yung. Fair traceable multi-group signatures. In *Financial Cryptography*, pages 231–246, 2008.

[9] Dan Boneh, Xavier Boyen, and Hovav Shacham. Short group signatures. In *CRYPTO*, pages 41–55, 2004.

[10] Ernest F. Brickell, Jan Camenisch, and Liqun Chen. Direct anonymous attestation. In *Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004, Washington, DC, USA, October 25-29, 2004*, pages 132–145, 2004.

[11] Jan Camenisch and Jens Groth. Group signatures: Better efficiency and new theoretical aspects. In *Security in Communication Networks, 4th International Conference, 2004, Italy, September 8-10, 2004, Revised Selected Papers*, pages 120–133, 2004.

[12] Jan Camenisch and Anna Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *CRYPTO*, pages 61–76, 2002.

[13] Sébastien Canard, Berry Schoenmakers, Martijn Stam, and Jacques Traoré. List signature schemes. *Discrete Applied Mathematics*, 154(2):189–201, 2006.

[14] David Chaum and Eugène van Heyst. Group signatures. In *EUROCRYPT*, pages 257–265, 1991.

[15] Seung Geol Choi, Kunsoo Park, and Moti Yung. Short traceable signatures based on bilinear pairings. In *IWSEC*, pages 88–103, 2006.

[16] Sherman SM Chow, Siu-Ming Yiu, and Lucas CK Hui. Efficient identity based ring signature. In *Applied Cryptography and Network Security*, pages 499–512. Springer, 2005.

[17] Jesus Diaz, David Arroyo, and Francisco B. Rodriguez. Anonymity revocation through standard infrastructures. In *EuroPKI*, pages 112–127, 2012.

[18] Jesus Diaz, David Arroyo, and Francisco B. Rodriguez. New X.509-based mechanisms for fair anonymity management. *Computers & Security*, 46(0):111 – 125, 2014.

[19] Niels Ferguson and Bruce Schneier. *Practical Cryptography*. John Wiley & Sons, Inc., New York, NY, USA, 1 edition, 2003.

[20] Eiichiro Fujisaki and Koutarou Suzuki. Traceable ring signature. In *Public Key Cryptography - PKC 2007, 10th International Conference on Practice and Theory in Public-Key Cryptography, China, April 16-20, 2007, Proceedings*, pages 181–200, 2007.

[21] Jaap henk Hoepman and Bart Jacobs. Increased security through open source. *Communications of the ACM*, 50:79–83, 2007.

[22] Ryan Henry. Efficient Zero-Knowledge Proofs and Applications, August 2014.

[23] Jung Yeon Hwang, Sokjoon Lee, Byung ho Chung, Hyun Sook Cho, and DaeHun Nyang. Short group signatures with controllable linkability. In *Lightweight Security Privacy: Devices, Protocols and Applications (LightSec), 2011 Workshop on*, pages 44–52, March 2011.

[24] ISO/IEC 20008-1: Information technology – Security techniques – Anonymous digital signatures – Part 1: General, 2013.

[25] ISO/IEC 20008-2: Information technology – Security techniques – Anonymous digital signatures – Part 2: Mechanisms using a group public key, 2013.

[26] Toshiyuki Isshiki, Kengo Mori, Kazue Sako, Isamu Teranishi, and Shoko Yonezawa. Using group signatures for identity management and its implementation. In *Proceedings of the 2006 Workshop on Digital Identity Management, Alexandria, VA, USA, November 3, 2006*, pages 73–78, 2006.

[27] Aggelos Kiayias, Yiannis Tsiounis, and Moti Yung. Traceable signatures. In *EUROCRYPT*, pages 571–589, 2004.

[28] Aggelos Kiayias and Moti Yung. Secure scalable group signature with dynamic joins and separable authorities. *IJSN*, 1(1/2):24–45, 2006.

[29] Benoît Libert, Thomas Peters, and Moti Yung. Group signatures with almost-for-free revocation. In *CRYPTO*, pages 571–589, 2012.

[30] Joseph K Liu, Victor K Wei, and Duncan S Wong. Linkable spontaneous anonymous group signature for ad hoc groups. In *Information Security and Privacy*. Springer, 2004.

[31] P. Neumann. Principled assuredly trustworthy composable architectures. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, 2004.

[32] Andreas Pfitzmann and Marit Hansen. A terminology for talking about privacy by data minimization: Anonymity, unlinkability, undetectability, unobservability, pseudonymity, and identity management.

[33] Klaus Potzmader, Johannes Winter, Daniel Hein, Christian Hanser, Peter Teufl, and Liqun Chen. Group signatures on mobile devices: Practical experiences. In *Trust and Trustworthy Computing - 6th International Conference, TRUST 2013, London, UK, June 17-19, 2013. Proceedings*, pages 47–64, 2013.

[34] Ronald L. Rivest, Adi Shamir, and Yael Tauman. How to leak a secret. In *ASIACRYPT*, pages 552–565, 2001.

[35] ITU-T Recommendation X.509. Information technology – open systems interconnection – the directory: Public-key and attribute certificate frameworks. Technical report, 11 2008.

[36] Fangguo Zhang and Kwangjo Kim. Id-based blind signature and ring signature from pairings. In *Advances in cryptology-ASIACRYPT 2002*, pages 533–547. Springer, 2002.

## A Using `libgroupsig`

In this section we explain how to configure, compile and make use of the library through a few simple code snippets for the main actions. `libgroupsig` requires glib (version 2.33 or compatible), openssl (version 1.0.1e-2 or compatible) for hashing functions, libgmp (version 2:5.0.5 or compatible) and the PBC library[17] (0.5.12 or compatible). Therefore, in order to use it, these libraries must be installed in the system. Also, the library uses the GNU build system[18]. Thus, in order to check the environment and generate the proper compilation scripts, the `configure` script must be run. Afterwards, the library is compiled with `make` and optionally installed with `make install`. A minimal set of auxiliary tools (located under the *tools* folder) for testing the library may be compiled with `make check`. Below we include a few code snippets, mostly extracted from the mentioned *tools* programs, showing some of the main functionality of the library. A detailed API documentation is available within the library's home page at Bitbucket[19]

*Group creation* The code snippet in Listing 8 shows how to create a group. Specifically, it creates the group and manager keys and an empty GML, using predefined configuration values for each supported group signature scheme. The initial `groupsig_init` call sets up library-wide structures (currently, it seeds random number generators). Subsequently, the group and manager keys, and GML are initialized. Finally, the group is created by filling up the initialized cryptographic tokens and setting scheme-wide data structures (e.g., PBC data structures for pairing based group signature schemes).

*Adding group members* This operation typically requires some precomputation by the new member and a finalization by the group manager. Thus, we have divided it accordingly. The result of each operation may just be transmitted over the network. However, for brevity, we include it in the snippet in Listing 9 as part of the same program. After successfully adding a new member, the GML (required parameter to the group manager side of the process) will be updated with the new member information.

[17] `http://crypto.stanford.edu/pbc/`. Last access on August 5th, 2015.
[18] `http://en.wikipedia.org/wiki/GNU_build_system`. Last access on August 5th, 2015.
[19] `https://bitbucket.org/jdiazvico/libgroupsig/`. Last access on August 5th, 2015.

```
/* Initialize environment */
if(groupsig_init(time(NULL)) == IERROR) { return IERROR; }

/* Set group signature scheme configuration parameters. */
if(cfg->scheme == GROUPSIG_KTY04_CODE) {
  KTY04_CONFIG_SET_DEFAULTS((kty04_config_t*) cfg->config,key_format);
} else if(cfg->scheme == GROUPSIG_BBS04_CODE ||
          cfg->scheme == GROUPSIG_CPY06_CODE) {
  CPY06_CONFIG_SET_DEFAULTS((cpy06_config_t*) cfg->config,key_format);
}

/* Initialize the group key, manager key and GML variables */
if(!(mgrkey = groupsig_mgr_key_init(cfg->scheme))) { return IERROR; }
if(!(grpkey = groupsig_grp_key_init(cfg->scheme))) { return IERROR; }
if(!(gml = gml_init(cfg->scheme))) { return IERROR; }

/* ''Construct'' the group */
if(groupsig_setup(cfg->scheme, grpkey, mgrkey, gml, cfg) == IERROR) {
  return IERROR;
}
```

Fig. 8: Group creation.

```
/* Initialize member key structure */
if(!(memkey = groupsig_mem_key_init(cfg->scheme))) { return IERROR;}

/* Member side of join */
if(groupsig_join_mem(memkey, grpkey) == IERROR) { return IERROR; }

/* Group manager side of join */
if(groupsig_join_mgr(gml, memkey, mgrkey, grpkey) == IERROR) {
  return IERROR;
}
```

Fig. 9: Addition of new group members.

*Signing messages and signature verification* Listing 10 shows the process for creating a group signature. It is worth emphasizing the last parameter which, in case of being other than `UINT_MAX`, specifies that the random number generator must be re-seeded using the specified value. Verification of a group signatures is shown in Listing 11.

*Opening signatures* With the open function, the real identity of the signer of a group signature is obtained. It requires the group signature itself and the group membership list besides, of course, the group manager key. Listing 12 shows how to call the function. Once obtained the identity of the signer, its member key may be revoked by including it in a CRL which, in turn, may be used to trace dishonest users, and even made public.

*Other functions* The previous functions represent the core of group signature schemes. However, specific schemes may implement additional functionality, like tracing dishonest users and claiming group signatures. For implementing these functions, the library provides handlers following the same style than the already

```
/* Initialize the group signature object */
if(!(sig = groupsig_signature_init(scheme))) {
  fprintf(stderr,
          "Error: failed to initialize the group signature.\n");
  return IERROR;
}

/* Sign the message: setting the seed to UINT_MAX forces to
   get a new pseudo random number for this signature instead
   of using a pre-fixed random number. */
if(groupsig_sign(sig, msg, memkey, grpkey, UINT_MAX) == IERROR) {
  fprintf(stderr, "Error: signing failure.\n");
  return IERROR;
}
```

Fig. 10: Issuing group signatures.

```
/* Verify group signature */
if(groupsig_verify(&bool, sig, msg, grpkey) == IERROR) {
  fprintf(stderr, "Error: verification failure.\n");
  return IERROR;
}
if(!bool) { fprintf(stdout, "WRONG signature.\n"); }
else { fprintf(stdout, "VALID signature.\n"); }
```

Fig. 11: Verifying group signatures.

```
/* Open group signature */
if((rc = groupsig_open(id,proof,sig,grpkey,mgrkey,gml)) == IERROR) {
  fprintf(stderr, "Error opening signature.\n");
  return IERROR;
}
```

Fig. 12: Opening group signatures.

introduced ones. Also, miscellaneous functionality is provided for importing, exporting and copying keys and signatures.

## B Extending `libgroupsig`

The library includes a script, named `libgroupsig.sh` and located under the `tools` directory, which allows the automated creation of the skeleton of a new group signature scheme. This option is invoked with the command `./libgroupsig.sh addscheme <scheme name>`, and creates a new subdirectory `$libroot$/groupsig/<scheme name>` containing this "empty" skeleton and updating a few library wide data structures. After running this command, the programmer would have to implement the actual functionality within the files created inside the new subdirectory (e.g. `setup.c`, `sign.c`, etc.).