Toward Generic Method for Server-Aided Cryptography

Sébastien Canard[1], Iwen Coisel[2], Julien Devigne[1,3], Cécilia Gallais[4], Thomas Peters[5], and Olivier Sanders[1,6]

[1] Orange Labs - Applied Crypto Group
42 rue des coutures - 14000 CAEN - France
[2] European Commission - Joint Research Centre (JRC)
Institute for the Protection and the Security of the Citizen - Digital Citizen Security
21027 ISPRA (VA) - Italy
[3] Université de Caen Basse-Normandie - Laboratoire GREYC
Esplanade de la Paix - 14000 Caen - France
[4] Tevalis
80 av. des Buttes de Cosmes - 35700 Rennes - France
[5] Université catholique de Louvain - ICTEAM/Crypto Group
1348 Louvain-la-Neuve - Belgium
[6] Ecole Normale Supérieure - Département d'Informatique
45 rue dUlm - 75230 Paris Cedex 05 - France

Abstract. Portable devices are very useful to access services from anywhere at any time. However, when the security underlying the service requires complex cryptography, implying the execution of several costly mathematical operations, these devices may become inadequate because of their limited capabilities. In this case, it is desirable to adapt the way to use cryptography. One possibility, which has been widely studied in many particular cases, is to propose a server-aided version of the executed cryptographic algorithm, where some well-chosen parts of the algorithm are delegated to a more powerful entity. As far as we know, nothing has been done to generically change a given well-known secure instance of a cryptographic primitive in its initial form to a secure server-aided version where the server (called the intermediary) may be corrupted by the adversary. In this paper, we propose an almost generic method to simplify the work of the operator who wants to construct this secure server-aided instance. In particular, we take into account the efficiency of the resulting server-aided instance by giving the best possible way to separate the different tasks of the instance so that the resulting time efficiency is optimal. Our methodology can be applied to most of public key cryptographic schemes.

1 Introduction

Constrained devices (*e.g.* mobile phones, smart cards or RFIDs) are more and more used in our daily life. Practical applications may require them to execute cryptographic algorithms. However, in return for their low-cost, these devices are

S. Qing et al. (Eds.): ICICS 2013, LNCS 8233, pp. 373–392, 2013.

generally resource constrained and/or do not implement all the necessary mathematical/cryptographic tools to perform such executions. This is not a major drawback in protocols requiring a low user's workload, but it can become appalling for some modern and complex protocols allying contradictory properties (such as anonymity and accountability, or confidentiality and sharing). Then, some applications may not be developed if the time taken by a device to execute required operations is too long. Thus, cryptographic protocols sometimes need to be further studied when executed in such environments. One solution is to use preprocessing (see *e.g.* [30]) which permits some data to be computed in advance so that the whole algorithm does not require heavy computation to be efficiently executed. This has the drawback of consuming a lot of space memory and may not be applicable all the time. Another possibility is to modify the cryptographic mechanism to fit the device restrictions. This has already been done in the RFID case [18,26] or when considering the integration of *e.g.* group and ring signatures in a smart card [10,32]. This approach sometimes necessitates important modifications of the initial algorithm, and may imply some stronger (and questionable!) assumptions such as *e.g.* the tamper-resistance one.

This paper focus on the approach which consists in speeding up the cryptographic operation by delegating a substantial part of computations to a more powerful entity, generally called a server or an *intermediary*.

Related Work. Many papers in the literature propose a way to outsource cryptographic operations to servers. Regarding efficiency, the result should be more efficient than the non-server-aided execution. Regarding security, the possibility to corrupt the intermediary should be taken into account in the server-aided version. This work has been done *e.g.* in the case of RSA [25,5], where the aim is to help the restricted device to perform a modular exponentiation with an RSA modulus[1], or in the case of the signature/authentication verification [23,19], for several existing schemes. Multi-party computation techniques (see *e.g.* [33]) permit several entities to jointly compute a function over their inputs, while each entity keeps its own input secret.

When dealing with more complicated protocols, especially those dealing with anonymity, a lot of research has also been carried out. For group signature schemes [12], Maitland and Boyd [24] and then Canard *et al.* [9] proposed variants of existing schemes where the group member is helped by some semi-trusted entity to produce a group signature. This trick is also part of the Direct Anonymous Attestation framework (see *e.g.* [8,13]).

Another approach, called *wallet with observers*, has been taken in the CAFE project [11,16]. Here, a powerful prover interacts with a non-trusted smart card to perform some computations, such that the prover is unlinkable *w.r.t.* the smart card.

Hohenberger and Lysyanskaya [20] have proposed a new security model where the server is necessarily split into two different components. For signature/authentication schemes, Girault and Lefranc [19] have given the theory for

[1] Even if most of them have later been broken [29,28,27].

server-aided verification. However, nothing has been done to generically transform a given secure instance of a cryptographic primitive in its initial form into a secure server-aided version where the server may be corrupted by the adversary. Such a transformation permits creating automatically the previous systems, with a potentially more efficient outcome.

Our contribution. In this paper, we provide an almost generic method to simplify the work of an operator for the above problem. More precisely, we focus on an entity $\mathcal{E}_0$ and its execution of a cryptographic algorithm ALG_0 underlying a secure instance of a cryptographic primitive. We first divide $\mathcal{E}_0$ into two roles: a trusted entity $\mathcal{T}$ which manages the inputs of ALG_0 but is not necessarily powerful (typically a smart card or a PC) and an intermediary $\mathcal{I}$ which is not necessarily trusted but is considered as more powerful (typically a mobile phone or a cloud server). Our aim is then to produce a secure server-aided variant which is as efficient as possible.

We first focus on the data manipulated inside ALG_0. All the inputs and outputs of the algorithm are known by $\mathcal{E}_0$ and, as we trust it, by $\mathcal{T}$ too. Regarding $\mathcal{I}$, this may be different since it depends on the possibility of corruption of $\mathcal{I}$ by the adversary (who can passively listen to the interactions between $\mathcal{T}$ and $\mathcal{I}$, can obtain the data given to $\mathcal{I}$ or can corrupt $\mathcal{I}$). Our method allows the operator to choose the power of the adversary on each expected security property, and automatically outputs, for most of manipulated data, whether this data can be *known* or *unknown* (called the status of the data) to $\mathcal{I}$.

We then consider the studied algorithm ALG_0 as a set of tasks. Then, depending on the status of the inputs and outputs, we decide whether each task can be performed by $\mathcal{T}$ alone, $\mathcal{I}$ alone or by both (using some well-known server-aided computations).

We finally provide an algorithm which outputs the best possible secure variant, depending on the time performances of both $\mathcal{T}$ and $\mathcal{I}$. All along the paper, we use as a running example the case of group signatures (to make a comparison with the work in [9]) but our method can also be applied to most existing cryptographic primitives.

Organization of the paper. The next section describes our framework and introduces the notion of intermediary. Section 3 describes how one can fill in the data status table according to the chosen security. Section 4 defines task statuses and explains how one can determine them. Section 5 is devoted to the description of the way to efficiently distribute the computations.

2 Background and Definitions

All along the paper, any entity is denoted using calligraphic typography (e.g. $\mathcal{E}$), an object or a data using sans font typography (e.g. d), algorithm using small capital letters (e.g. Alg), list using true type typography (e.g. `list`), and sets using greek letters (e.g. Ω). A task is always denoted t.

When a single entity is required to perform a procedure, it is generally called an algorithm, whereas it is called a protocol when interactions between several

entities are required. However, protocols can be split into as many parts (algorithms) executed by a single entity as needed. We thus only focus on the notion of algorithm in the following.

2.1 Definitions

Notion of Primitive. A *cryptographic primitive* Π describes the main guidelines of a cryptographic application. Informally, a primitive Π is defined by a set Ξ of entities, a set Ω of objects, a set $\Lambda = \{\text{Alg}_1, \ldots, \text{Alg}_v\}$ of algorithms and a set Σ of security properties. An instance of a given primitive Π is a precise description of all the algorithms and the associated objects which ensure the security properties of Π. We consider in the following that each algorithm is realized thanks to a sequence of *tasks*, defined below.

Considered Tasks. The instance of an algorithm Alg_i is a set of tasks, denoted Θ_i, related to cryptographic or mathematical operations, that formally describes how to reach the output of the focused algorithm, denoted out, from its input, denoted inp. In this paper, we group these tasks by types which are assigned an identifier: (1) pseudo-random generation (*e.g.* $r \in_R \mathbb{Z}_p^*$), (2) multi-linear combination evaluation (*e.g.* $s = a \cdot b + c$), (3) exponentiation evaluation (*e.g.* $T = g^x$), (4) group operation (*e.g.* $z = g \cdot h$), (5) pairing evaluation (*e.g.* $h = e(P, Q)$), (6) hash function evaluation (*e.g.* $\mathsf{hash}(m)$) and (7) communication. This list is totally arbitrary and our generic transformation still works if other types of tasks are introduced. As an example, exponentiations in a *regular* finite field and in an elliptic curve group might be considered as two different tasks while leading to the same result.

The Data. In an instance π of a primitive Π, data can come from two different ways. Some of them represent the objects of the primitive, and so inputs and outputs of the algorithms. But there are also data used as "intermediate values" within a sequence of tasks of a given algorithm. The former data are called *intrinsic*, while the latter data are called *ephemeral*.

Definition of an instance. To sum up, an instance π of a primitive Π is defined by a set $\Theta = \bigcup_i^v \Theta_i$ of tasks, a set $\Phi = \bigcup_i^v \Phi_i$ of *intrinsic data* and a set $\Psi = \bigcup_i^v \Psi_i$ of *ephemeral data*, where each subset is related to one specific algorithm Alg_i. In the following, $\text{Alg}_i(\mathcal{E}, \mathsf{inp}) = \mathsf{out}$ denotes that the (intrinsic) data contained in $\mathsf{out} \in \Phi$ have been obtained by the execution of the algorithm Alg_i by the entity $\mathcal{E}$ using the data contained in $\mathsf{inp} \in \Phi$ as inputs.

2.2 Our Running Example: Group Signatures

In the group signature primitive [12], the set of entities is composed of one group manager (sometimes called the issuer), several group members, one opener and several verifiers. In this primitive, any member of a group can sign messages on behalf of the group. Such signatures remain anonymous and unlinkable for

anyone except a designated authority, the opener, who has the ability to identify the signer. Anyone, called in this case a verifier, can check the validity of a group signature. The objects related to this primitive are the issuer, members, and opener keys as well as all the possible messages (generally defined by a message space). Following [6], such primitive is composed of 7 procedures called SETUP (to compute secret keys and public parameters), USERKG (for users), JOIN (a protocol for users to become group members), GSIGN (for the production of a group signature), GVERIFY (for the verification step), OPEN (for anoymity revocation) and JUDGE (for the public verification of an anonymity revocation). An interested reader may refer to *e.g.* [6] for details.

2.3 Our Method in a Nutshell

Let π_0 be a particular instance (*e.g.* XSGS [17]) of a given primitive Π_0 (*e.g.* group signature scheme). Let Θ_i be an instance of one particular algorithm ALG_0 (*e.g.* the GSIGN algorithm) which is executed by the entity $\mathcal{E}_0$ (*e.g.* a group member). This task description is called the *initial version* of the algorithm. Our aim is to improve its time complexity without compromising the security of π_0, or in a controlled way.

In the literature, the notion of server-aided cryptography is most of the time related to the split of an entity $\mathcal{E}_0$ into two components, namely a trusted entity, denoted[2] $\mathcal{T}$, which manages all inputs of the algorithm, and an intermediary[3], denoted $\mathcal{I}$ with which $\mathcal{T}$ can interact and delegate some of his workload. We will speak in this case of a *server-aided version* of the algorithm. From one initial version of the instance π_0, it is possible to design several secure server-aided versions $\overline{\pi}_0^{(1)}, \overline{\pi}_0^{(2)}, \ldots$.

To decide which one is the most efficient, we first need to decide which data (intrinsic or ephemeral) can be given to the intermediary. This is done in accordance with the security properties of the studied primitive Π_0 (see Section 3). Then, depending on this result, we focus (in Section 4) on each task of ALG_0 and try to say whether it can be executed by $\mathcal{T}$ alone, $\mathcal{I}$ alone, and/or cooperatively (both $\mathcal{T}$ and $\mathcal{I}$ participate in its execution). This leads to a bunch of different secure repartitions. Our method finally outputs the most efficient one according to the performances of $\mathcal{T}$ and $\mathcal{I}$ (see Section 5).

3 Status of the Data

Our methodology is in particular based on the definition of a status for each manipulated data. In this section, we define the status of a data *w.r.t.* an entity of an instance. Then, we adapt the adversary against the server-aided instance in the security properties.

[2] We should have used the notation $\mathcal{T}_{\mathcal{E}_0}$ but, as it is not ambiguous, we simplify it.

[3] $\mathcal{I}$ is not necessarily a new entity in the system but can be in fact seen as a new role played by one existing entity.

3.1 Data Status and Intermediary

Traditionally in cryptography, some data are said *secret*, and some others are said *public*. This can be formalized by the notion of *known* data *w.r.t.* a specific entity, and we argue that this is enough to handle all possible cases. On the one hand, the status of a data is secret if an entity is the only one to *know* it. On the other hand, its status is public if all involved entities know such data. It also permits to formalize intermediate cases where a data is known by several entities, but not all. By the way, the only relevant status in our case is that of *known* data.

Let $\Pi = (\Xi, \Omega, \Lambda, \Sigma)$ be a primitive and let $\pi = (\Theta, \Phi, \Psi)$ be an instance of Π. The security properties verified by π determine the status of each intrinsic data *w.r.t.* the different entities. The status of an ephemeral data $\mathsf{ed} \in \Psi$ is known by an entity $\mathcal{E}$, denoted $\mathtt{st}_{\mathcal{E}}[\mathsf{ed}] = \mathsf{kn}$, if ed is an output of an elementary task $\mathtt{t}$ run by $\mathcal{E}$ for which the inputs are all known by $\mathcal{E}$. Otherwise the status of the data is unknown (noted ukn).

Now let us consider an algorithm ALG_0, executed by an entity $\mathcal{E}_0$, the status of intrinsic (Φ_0) and ephemeral (Ψ_0) data are thus known *w.r.t.* this entity. Now, if $\mathcal{E}_0$ is divided into the two entities $\mathcal{T}$ and $\mathcal{I}$, we should focus on the status of intrinsic data *w.r.t.* $\mathcal{I}$ since it follows from our above choices that the status of these data *w.r.t.* $\mathcal{T}$ is necessarily *known*.

Security Experiment and Adversaries. Let us consider a security property, denoted $\mathtt{secu}$, expected by the primitive Π. We assume that the studied instance π verifies this security property. We thus need to clearly describe the server-aided security property $\overline{\mathtt{secu}}$ expected by a server-aided execution of the primitive Π. We recall that $\mathcal{I}$ is only implicated in the execution of the ALG_0 procedure.

Since $\mathcal{T}$ and $\mathcal{I}$ are two distinct entities, possible corruption of $\mathcal{I}$ by an adversary must be taken into account, while we assume all along this paper that $\mathcal{T}$ is never corrupted (if $\mathcal{E}_0$ is not). We should then modify the related security experiments accordingly. First, we give the following definition which states the possible strategies given to the adversary against a server-aided version of π.

Definition 1 (server-aided adversary). *Let $\overline{\mathtt{secu}}$ be a security property related to a server-aided instance $\overline{\pi}$. Let $\mathcal{A}$ be an adversary against $\overline{\mathtt{secu}}$ in the server-aided setting. For the related experiment, an adversary is said to have the power of a*

- listener-receiver *if $\mathcal{A}$ can obtain all the communication between $\mathcal{T}$ and $\mathcal{I}$ and is given access to all the data known by $\mathcal{I}$.*
- controller *if $\mathcal{A}$ totally controls $\mathcal{I}$.*

It is obvious that a *controller* is necessarily *listener-receiver*. Then, there are three different types of adversary to study: *standard* (with no extra power), *listener-receiver* and *controller*.

Dealing with several security properties. In most of the complex cryptographic primitives, several security properties are required at the same time

(see the example of group signatures in Section 2.2). In order to build a server-aided instance with an improved efficiency it can make sense to relax some of them in regards of the intermediary, while the others are preserved[4].

Several server-aided instances of the initial instance π can thus be generated depending of the combinations of desired properties. For example: the adversary may *e.g.* be standard for the server-aided security property $\overline{\texttt{secu}}_1$ but controller for $\overline{\texttt{secu}}_2$.

3.2 Filling the Data Status Table

We should now make the link between the security properties and the status of all the data manipulated in ALG_0.

GENERIC OR NOT GENERIC. Again, our aim is to design a generic autonomous (as possible) method to obtain the best secure server-aided variant of a secure instance. In fact, from the security point of view, it seems hard to completely automate our work.

One solution is to make use of some formal analysis dedicated to cryptographic protocols, such as the (non exhaustive) work given in [7,1,3,4]. For this purpose, we first need to precisely formalize the operations available to the adversary. We then make use of the description of the experiment, depending on the power of the adversary (see above) and execute formal methods to find a way for the adversary to break the security, using its available operations as defined above. This execution is done several times by having the status of all the data vary. Finally, each set of data status which leads to "no attack" by formal analysis can be given to the next step of the procedure (with the deletion of some redundant choices). Such work requires a complete and deep study and it is not our aim in this paper to study this independent research topic.

Another possibility is to ask an operator to perform such choice(s). We give him the new experiment and ask him on output the status of all data. We then assume that such output is good regarding security.

In the following, we have chosen a compromise between the two. We have succeeded, using some results given below, in simplifying the work of this operator by automatically treating some cases. The way we assign a status to each task depends on their nature (intrinsic or ephemeral) and their type.

STATUS OF INTRINSIC DATA. Informally, we do not want that a non-standard adversary uses his extra power to get access to more data than the adversary in the original security experiment. We then use the following result.

Definition 2. *Let* $\Sigma = \{\texttt{secu}[1], ..., \texttt{secu}[t]\}$ *be the security properties ensured by* π*. For all* $i \in I = [1, t]$*,* Δ_i *is the set of all intrinsic data known by* $\mathcal{A}_i$*, the*

[4] In [9], the authors argue that the anonymity property may be relaxed *w.r.t.* the intermediary as this latter may already know the identity of $\mathcal{T}$, while it should not be able to produce a group signature without the help of $\mathcal{T}$, and thus break the traceability property.

adversary of the security experiment defining $\texttt{secu}[i]$. *Let* $\overline{\mathcal{A}_i}$ *be the adversary against* $\overline{\texttt{secu}}[i]$ *in the server-aided setting. We define* $\Delta = \bigcap_{i \in I} \Delta_i$ *where* $i \in I$ *if* $\overline{\mathcal{A}_i}$ *has extra power (*i.e. *has the power of a listener-receiver or a controller).*

We then consider that an intrinsic data d is stated as known to $\mathcal{I}$ *iff* $d \in \Delta$. We illustrate this method in Section 3.3 using the XSGS group signature scheme given in [17].

Status of ephemeral data. A server-aided version should not compromise the security of the instance π. One way is to make use of a strong notion, related to the zero-knowledge property used for proofs of knowledge.

Definition 3. *: A server-aided version* $\overline{\pi}$ *of an instance* π *is said to be*

- listener-receiver secure iff *there exists a simulator* $\mathcal{S}$*, whose inputs are known intrinsic data, such that the output of* $\mathcal{S}$ *is computationally indistinguishable from the view of the real communications between* $\mathcal{T}$ *and (a non-controller)* $\mathcal{I}$.
- controller secure iff*, for any intermediary* $\mathcal{I}^*$*, there exists a simulator* $\mathcal{S}_{\mathcal{I}^*}$*, whose inputs are known intrinsic data, such that the output of* $\mathcal{S}_{\mathcal{I}^*}$ *is computationally indistinguishable from the view of the real communications between* $\mathcal{T}$ *and* $\mathcal{I}^*$.

Definition 4. *Let* Π *be a primitive and* π *an initial instance of* Π *ensuring* $\Sigma = \{\texttt{secu}[1], ..., \texttt{secu}[t]\}$. *Let* secuvec *be a vector of length* t*, defining the class of each adversary* $\overline{\mathcal{A}_i}$*, such that for all* $i \in [1, t]$, $\texttt{secuvec}[i] \in \{$*standard, listener-receiver, controller*$\}$. *Let* $\overline{\pi}$ *be a server-aided version of the instance* π. *The server-aided instance* $\overline{\pi}$ *is a* secuvec *secure server-aided version of* π *if for all* $i \in [1, t]$ *such that* $\texttt{secuvec}[i] \neq$ *standard,* $\overline{\pi}$ *is* $\texttt{secuvec}[i]$ *secure.*

The next result helps for a partial automation of the filling of the data status table.

Lemma 1. *Let* Π *be a primitive and* π *an initial instance of* Π *ensuring* $\Sigma = \{\texttt{secu}[1], ..., \texttt{secu}[t]\}$. *If* $\overline{\pi}$ *is a* secuvec *secure server-aided version of the instance* π *then* $\overline{\pi}$ *ensures* $\bar{\Sigma} = \{\overline{\texttt{secu}}[1], ..., \overline{\texttt{secu}}[t]\}$.

Proof. First, let $\overline{\mathcal{A}_i}$ be a standard adversary against $\overline{\texttt{secu}}[i]$. Since it does not have access to the data given to $\mathcal{I}$ and cannot listen to the communication between $\mathcal{T}$ and $\mathcal{I}$, it is equivalent to an adversary against the initial security property $\texttt{secu}[i]$. Thus the security of the initial instance implies the security of the server-aided instance $\overline{\pi}$.

Now we consider $\overline{\mathcal{A}_i}$, a non-standard adversary against $\overline{\texttt{secu}}[i]$. We recall that the status of intrinsic data are known to $\mathcal{I}$ *iff* they are known by any adversary $\mathcal{A}$ of the security properties verified by π. Since we assume the existence of $\mathcal{S}$ which is able, using these known intrinsic data, to simulate the communications between $\mathcal{T}$ and $\mathcal{I}$, we do not give more information to the adversary than in the original experiment. Thus, the security of the initial instance implies the one of the server-aided version. □

One can argue that we could reach the security property $\overline{\texttt{secu}}$ without this strong "zero-knowledge" requirement. Indeed, in some experiments, if we allow the adversary to get access to some additional information, we can get a more efficient repartition of the tasks without endangering the security of $\overline{\pi}$. Yet it then seems hard to guarantee the security of $\overline{\pi}$ without asking the operator which ephemeral data can be stated as known.

In practice, the way our algorithm assigns status to each ephemeral data depends on the type of task in which it is involved. For example, if a secret data is involved in a multi-linear combination then another data involved in the same task has to be set as *unknown w.r.t.* $\mathcal{I}$. Now, if we consider a hash function evaluation we could consider the output as *known*, regardless of the status of the inputs. We still refer to Section 3.3 for a more explicit example.

Conclusions. The filling of the data status table by the operator is done by first choosing the security vector `secuvec`, enabling him to determine Δ, the set of intrinsic data known by $\mathcal{I}$. Then he runs our algorithm that will automatically assign a status to most of the ephemeral data. Finally, the operator just has to indicate the status of the non treated ephemeral data, using the security notions given in Definition 3.

3.3 Example of Group Signatures

Let us consider the XSGS group signature scheme given in [17,9] and we focus on the data. The whole GSign algorithm is given in Figure 1.

Group signature scheme [17] - GSign$(\mathsf{m}, \mathsf{gsk}[i] = (A, x, y))$					
t_1.	$\alpha_1 \in_R \mathbb{Z}_p^*$	t_{15}.	$\mathsf{r}_1 \in_R \mathbb{Z}_p^*$	t_{29}.	$\mathsf{t}''_5 = \mathsf{y}_2^{-\mathsf{r}_8}$
t_2.	$\beta_1 \in_R \mathbb{Z}_p^*$	t_{16}.	$\mathsf{r}_2 \in_R \mathbb{Z}_p^*$	t_{30}.	$\mathsf{t}_5 = \mathsf{t}'_5 \cdot \mathsf{t}''_5$
t_3.	$\alpha_2 \in_R \mathbb{Z}_p^*$	t_{17}.	$\mathsf{r}_3 \in_R \mathbb{Z}_p^*$	t_{31}.	$\mathsf{t}_6 = \mathsf{T}_3^{r_5}$
t_4.	$\beta_2 \in_R \mathbb{Z}_p^*$	t_{18}.	$\mathsf{r}_4 \in_R \mathbb{Z}_p^*$	t_{32}.	$\mathsf{e}_1 = e(\mathsf{t}_6, \mathsf{g}_2)$
t_5.	$\gamma_1 = \alpha_1 + \beta_1$	t_{19}.	$\mathsf{r}_5 \in_R \mathbb{Z}_p^*$	t_{33}.	$\mathsf{t}_7 = e(\mathsf{y}_1, \Gamma)^{-r_7}$
t_6.	$\gamma_2 = \alpha_2 + \beta_2$	t_{20}.	$\mathsf{r}_6 \in_R \mathbb{Z}_p^*$	t_{34}.	$\mathsf{t}'_7 = e(\mathsf{y}_1, \mathsf{g}_2)^{-r_6}$
t_7.	$\mathsf{T}_1 = \mathsf{g}^{\alpha_1}$	t_{21}.	$\mathsf{z} = \gamma_1 \cdot \mathsf{x} + \mathsf{y}$	t_{35}.	$\mathsf{t}_8 = \mathsf{e}_1 \cdot \mathsf{t}_7 \cdot \mathsf{t}'_7$
t_8.	$\mathsf{T}_2 = \mathsf{g}'^{\beta_1}$	t_{22}.	$\mathsf{r}_7 = \mathsf{r}_1 + \mathsf{r}_2$	t_{36}.	$c = \mathcal{H}(\mathsf{m} \Vert \mathsf{T}_1 \Vert \ldots \Vert \mathsf{T}_6 \Vert \mathsf{t}_1 \Vert \ldots \Vert \mathsf{t}_8)$
t_9.	$\mathsf{T}'_3 = \mathsf{y}_1^{\gamma_1}$	t_{23}.	$\mathsf{r}_8 = \mathsf{r}_3 + \mathsf{r}_4$	t_{37}.	$\mathsf{s}_1 = \mathsf{r}_1 + \mathsf{c} \cdot \alpha_1 \pmod p$
t_{10}.	$\mathsf{T}_3 = \mathsf{A} \cdot \mathsf{T}'_3$	t_{24}.	$\mathsf{t}_1 = \mathsf{g}^{\mathsf{r}_1}$	t_{38}.	$\mathsf{s}_2 = \mathsf{r}_2 + \mathsf{c} \cdot \beta_1 \pmod p$
t_{11}.	$\mathsf{T}_4 = \mathsf{g}^{\alpha_2}$	t_{25}.	$\mathsf{t}_2 = \mathsf{g}'^{\mathsf{r}_2}$	t_{39}.	$\mathsf{s}_3 = \mathsf{r}_3 + \mathsf{c} \cdot \alpha_2 \pmod p$
t_{12}.	$\mathsf{T}_5 = g'^{\beta_2}$	t_{26}.	$\mathsf{t}_3 = \mathsf{g}^{\mathsf{r}_3}$	t_{40}.	$\mathsf{s}_4 = \mathsf{r}_4 + \mathsf{c} \cdot \beta_2 \pmod p$
t_{13}.	$\mathsf{T}'_6 = \mathsf{y}_2^{\gamma_2}$	t_{27}.	$\mathsf{t}_4 = \mathsf{g}'^{\mathsf{r}_4}$	t_{41}.	$\mathsf{s}_5 = \mathsf{r}_5 + \mathsf{c} \cdot \mathsf{x} \pmod p$
t_{14}.	$\mathsf{T}_6 = \mathsf{A} \cdot \mathsf{T}'_6$	t_{28}.	$\mathsf{t}'_5 = \mathsf{y}_1^{\mathsf{r}_7}$	t_{42}.	$\mathsf{s}_6 = \mathsf{r}_6 + \mathsf{c} \cdot \mathsf{z} \pmod p$
Output: $\sigma = (\mathsf{T}_1, \mathsf{T}_2, \mathsf{T}_3, \mathsf{T}_4, \mathsf{T}_5, \mathsf{T}_6, \mathsf{c}, \mathsf{s}_1, \mathsf{s}_2, \mathsf{s}_3, \mathsf{s}_4, \mathsf{s}_5, \mathsf{s}_6)$					

Fig. 1. The XSGS GSign algorithm [17,9]

Dealing with security properties. The security properties required from a group signature scheme [6] are: `correctness`, `traceability`, `anonymity` and `non-frameability`. We consider a standard adversary against the correctness and listener-receiver against traceability and non-frameability. We use, as in [9], the relaxed anonymity property *w.r.t.* the intermediary since it has other ways to identify the signature issued by $\mathcal{E}_0$, and then consider a standard adversary against the anonymity. Using the above order for security properties we then set the corresponding security vector to $\mathtt{secuvec} = [\mathsf{standard}, \mathsf{listener\text{-}receiver}, \mathsf{standard}, \mathsf{listener\text{-}receiver}]$.

Status of intrinsic data. We now have to determine Δ using Definition 2. The status of most of the intrinsic data is obvious since the adversaries of the different security experiments have access to the public parameters $(\mathsf{g}, \mathsf{g}', ...)$, the message (m) and the group signature. We thus have:

$$\{\mathsf{g}, \mathsf{g}', \mathsf{y}_1, \mathsf{y}_2, \Gamma, \mathsf{g}_2, e(\mathsf{y}_1, \Gamma), e(\mathsf{y}_1, \mathsf{g}_2), \mathsf{m}, \mathsf{T}_1, \mathsf{T}_2, \mathsf{T}_3, \mathsf{T}_4, \mathsf{T}_5, \mathsf{T}_6, \mathsf{c}, \mathsf{s}_1, \mathsf{s}_2, \mathsf{s}_3, \mathsf{s}_4, \mathsf{s}_5, \mathsf{s}_6\} \subset \Delta.$$

Now let consider the user's key: $\mathcal{A}, \mathsf{x}, \mathsf{y}$. We have chosen non-standard adversaries for the traceability and non-frameability, the status of the user's key only depends on the knowledge of the adversaries of these experiments. Since the issuer is adversary-controlled in both experiments, the adversaries know A and x and thus we have: $\{A, \mathsf{x}\} \subset \Delta$. The only intrinsic data that has to remain unknown to $\mathcal{I}$ is then y.

Status of ephemeral data. Once the status of each intrinsic data is set, the operator runs the algorithm SetStatusData, described in Section 5, which outputs $\mathtt{st}_{\mathcal{I}}[\{\mathsf{r}_6, \mathsf{z}\}] = \mathsf{ukn}$ and $\mathtt{st}_{\mathcal{I}}[\Psi_0 \setminus \{\mathsf{r}_6, \mathsf{z}\}] = \mathsf{kn}$.

4 Status of a Task

We will now focus on each task of the studied algorithm Alg_0 and decide which entity(ies) can execute it. For this purpose, each task of the algorithm is characterized by a status. More precisely, let us consider independently each task $\mathsf{t}_i \in \Theta$ of the procedure Alg_0. We focus on the inputs and outputs of t_i and make use of the data status which has been stated as explained in the previous section.

4.1 Execution of a Task by the Intermediary

Based on the data status table, there are initially two cases that can be seen as trivial.

1. All input and output data of t_i are known to $\mathcal{I}$: for obvious reasons, this task may be executed by either $\mathcal{T}$ or the intermediary $\mathcal{I}$. We denote such case $\mathtt{st}(\mathsf{t}_i) = 1$.

2. At least one output data of $\mathtt{t}_i$ is not known to $\mathcal{I}$: we first consider that $\mathtt{t}_i$ is executed by $\mathcal{T}$ and we say that the status of $\mathtt{t}_i$ is 0, which is denoted $\mathtt{st}(\mathtt{t}_i) = 0$. However, depending on the existence of a cooperative version of this task, this status could be changed (see Section 4.2).

Multiplicity of the choices. One important thing is that a task with status 1 will not necessarily be executed by $\mathcal{I}$. Indeed, we consider that it can be either executed by $\mathcal{T}$ or by $\mathcal{I}$. Our main objective is to determine the best possible server-aided instance, which may include a task that can be executed by $\mathcal{I}$ will possibly be executed by $\mathcal{T}$ (if the latter has nothing more to do during enough time for example). This will be taken into account in our main method below.

We now try to do better by searching in the literature some tasks where $\mathcal{I}$ can help $\mathcal{T}$ even if some of the manipulated data are secret.

4.2 Server-Aided Execution of a Task

There are numerous papers about outsourcing some specific cryptographic tasks (for example [25,5,20,31,2]). We here talk about a "cooperative" execution of a task.

Cooperative execution of a task. We have previously made some choices regarding the set Θ of tasks. Then, for each type of task in Θ, we can say whether it exists in the literature a way to execute such task cooperatively or not, depending on the status of the different inputs and outputs. For example, the authors of [15,14] provide a method to compute $e(A, B)$ when A and B are secret. Appendix A lists some existing cooperative methods for the tasks in Θ. In the following, the database `CoopMeth` contains the status of each task depending on the status of the inputted and outputted data.

4.3 Status of a Task

Using the above results, we can now formally define the status of a task.

Definition 5 (status of a task). *Let $\Pi = (\Xi, \Omega, \Lambda, \Sigma)$ be a primitive and let $\pi = (\Theta, \Phi, \Psi)$ be an instance of Π. The status of a task $\mathtt{t} \in \Theta$ is defined as $\mathtt{st}[\mathtt{t}] = 0$ if $\mathtt{t}$ has to be executed by $\mathcal{T}$; $\mathtt{st}[\mathtt{t}] = 1$ if $\mathtt{t}$ can be executed by either $\mathcal{T}$ or $\mathcal{I}$; $\mathtt{st}[\mathtt{t}] = \star$ if $\mathtt{t}$ can be executed cooperatively, or by $\mathcal{T}$ alone; $\mathtt{st}[\mathtt{t}] = 1\star$ if $\mathtt{t}$ can be executed cooperatively, or by $\mathcal{I}$ or $\mathcal{T}$ alone; $\mathtt{st}[\mathtt{t}] = 2$ if $\mathtt{t}$ should be entirely executed by $\mathcal{I}$.*

The last item ($\mathtt{st}[\mathtt{t}] = 2$) corresponds to the case where the trusted entity $\mathcal{T}$ is not able to perform a task. For example, most of today's smart cards do not implement a bilinear pairing (even cooperatively), which makes all the other statuses impossible. However, depending on the chosen security vector, the introduction of such status may imply that no possible secure server-aided version of a given instance can be designed (the result is that all the tasks should be performed by $\mathcal{T}$ sole). As it may occur in practice, we prefer to keep it.

TABLE OF STATUS TASK. Each task $\mathtt{t}_i$ of ALG_0 (in Θ_0) has to be associated to one of these status in order to design the server-aided version, which is done by the SETSTATUSTASK algorithm. In order to ease the execution of this algorithm, we list in a database the existing possibilities of execution for all elementary tasks depending on the status of their inputs and outputs.

- If a task cannot be executed by $\mathcal{T}$, then the status is necessarily stated to 2.
- If all the inputs and outputs of $\mathtt{t}_i$ are known to $\mathcal{I}$ (except when the adversary is a controller *w.r.t.* the correctness), then it can be totally executed by $\mathcal{I}$ and the status includes a 1.
- We then focus on existing server-aided executions of the task. Regarding the status of all inputs and outputs, we are able to say whether such cooperative method exists or not (see Appendix A for some examples). If one exists, then we can introduce the $\star$ in the task status (which implies the possibility to obtain a status $1\star$ with the previous case). In addition, the cooperative method is inserted in the database.
- Otherwise, the status of $\mathtt{t}_i$ is set to 0.

Now the task status table is filled, we can explain how we determine the best variant, which will be done in the next section. We first illustrate our purpose on task status with our example of group signatures.

4.4 Example of Group Signatures

When considering the elementary tasks of the XSGS group signature we find two types of them which can be executed cooperatively, namely the pairing and the exponentiation. However, cooperative executions of the these tasks remain inefficient (for example the delegation of pairing provided by [15] and [14] requires respectively 10 and 7 exponentiations in $\mathbb{G}_T$ which is costlier than computing the pairing) or insecure [5,28]. We then do not consider in our example the status "$\star$" and get the following repartition:

$$\mathtt{st}[\{\mathtt{t}_{20}, \mathtt{t}_{21}, \mathtt{t}_{42}\}] = 0 \text{ and } \mathtt{st}[\Theta_0 \setminus \{\mathtt{t}_{20}, \mathtt{t}_{21}, \mathtt{t}_{42}\}] = 1.$$

5 Producing the Most Efficient Server-Aided Variant

Before formally describing the algorithms that construct the most efficient server-aided version, we introduce some useful notations for this section. The best server-aided version of an algorithm mainly depends on the efficiency of both actors, namely the trusted entity $\mathcal{T}$ and the intermediary $\mathcal{I}$. Consequently, a database denoted `Perf` containing the performances of both entities for all types of tasks must be set. $\mathtt{Perf}[\mathcal{X}][\mathtt{t}]$ returns the time taken by the entity $\mathcal{X}$ (either $\mathcal{T}$ or $\mathcal{I}$) to perform the task $\mathtt{t}$. If it is not able to perform this task, the associated time is arbitrarily fixed to ∞.

Each task is defined by its identifier, its type (exponentiation, pairing,...), its inputs and its outputs.

5.1 Description of the Global Method

We here make a high level description of our algorithms by taking again our running example of the XSGS group signature phase. The more formal description of the main algorithms is further given by Algorithms 1, 2 and 3.

We first run the algorithm SetStatusData to determine the status of each epheme-ral data depending on the status of the intrinsic data and on rules defined by the operator. For example, if a secret data is involved in a multi-linear combination over $\mathbb{Z}_p$, then another data involved in the same task has to be kept secret from $\mathcal{I}$. Considering the task $\mathsf{t}_{21} : z = \gamma_1 \mathsf{x} + \mathsf{y}$, this means that the algorithm will have to set γ_1 or z as *unknown* because y is secret and $\mathtt{st}_{\mathcal{I}}[\mathsf{x}] = kn$. Using the same methodology for each task, the algorithm will finally output the status of each data. The resulting status of ephemeral data will be *known*, except for r_6 and z. We are then able to determine the status of each task. As already explained, we will consider that a task can be executed by $\mathcal{I}$ if the status of all inputs and outputs are known to $\mathcal{I}$ because of the lack of efficient cooperative protocol for the considered tasks.

Before allocating each task to $\mathcal{T}$ or $\mathcal{I}$ we first have to ensure that their order is respected. Indeed, some tasks take as inputs the output of other ones and thus have to be computed after. For example tasks $\mathsf{t}_{37}, \mathsf{t}_{38}, \mathsf{t}_{39}, \mathsf{t}_{40}, \mathsf{t}_{41}, \mathsf{t}_{42}$ require the data c and thus have to be computed after the task t_{36}. We then assign to each task, using the algorithm RepRound described below, a *round* number such that every task of a same round only takes as input intrinsic data or ephemeral data produced during previous rounds. We may thus look for the best repartition of the tasks in the round without caring for their order of execution.

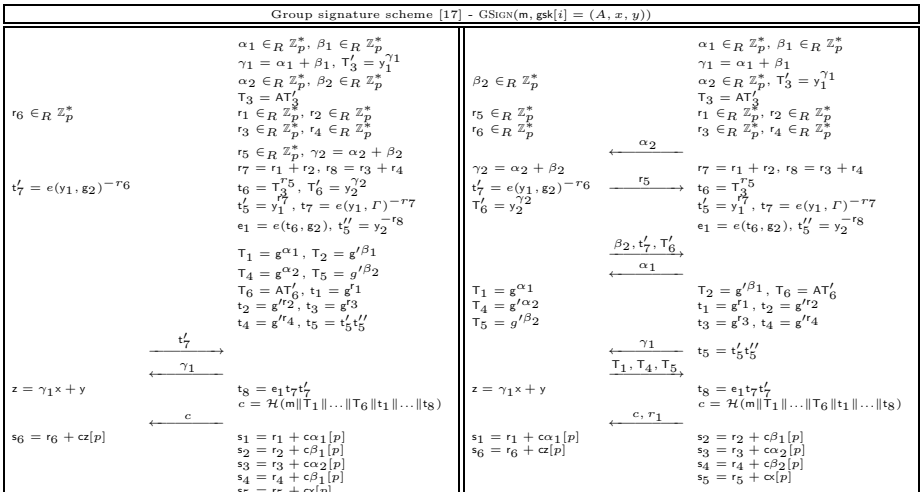

Fig. 2. The XSGS GSign algorithm [17,9]

We get this best repartition using the algorithm Rep which focus on the different types of tasks of the round rather than on the tasks themselves. This algorithm will determine how many tasks of each type have to be executed by $\mathcal{T}$ and by $\mathcal{I}$. For example, the round 7 is composed of the tasks $\mathsf{t}_7, \mathsf{t}_8, \mathsf{t}_{11}, \mathsf{t}_{12}, \mathsf{t}_{14}, \mathsf{t}_{24}, \mathsf{t}_{25}, \mathsf{t}_{26}, \mathsf{t}_{27}, \mathsf{t}_{30}$ and t_{35}, *i.e.* 8 exponentiations and 3 group operations. The algorithm Rep will then decide how many exponentiations and group operations have to be computed by $\mathcal{T}$ in order to minimize the execution time of this round. For a ratio of 2 between $\mathcal{T}$ and $\mathcal{I}$ we get the following repartitions: 3 exponentiations for $\mathcal{T}$ and the rest for $\mathcal{I}$. Now we must choose the 3 exponentiations among the 8 that will require the less communication time. This is done by the AttributeTask algorithm which is described in appendix B. In a nutshell, it mainly depends on the number of successors of each task. Indeed, since $\mathcal{T}$ handles less tasks than $\mathcal{I}$, the goal is to assign the tasks with the fewest successors to $\mathcal{T}$ since the probability that their output will be required by $\mathcal{I}$ is smaller.

Using the same methodology for each round finally gives us the repartitions described in Figure 2 for 2 different ratios (1000 and 2).

Considering the intermediary $\mathcal{I}$ as far more powerful than $\mathcal{T}$ is a typical approach in cryptography [9,15,22], the resulting protocols trying to delegate as many tasks as possible to the delegatee. The left side of figure 2 describes the result of our algorithm in such case, the only tasks handled by $\mathcal{T}$ being those who require knowledge of secret data $(\mathsf{y}, \mathsf{r}_6, \mathsf{z})$ and thus cannot be delegate to $\mathcal{I}$. One may note that we exactly find the same repartition as the one from [9]. Yet, the gap between $\mathcal{T}$ and $\mathcal{I}$ may not be so important, the right side of the figure describes a different repartition for a smaller ratio.

5.2 Round Attribution

The RepRound algorithm takes as inputs the number of tasks, $taskNumber$, an array $RepRound$ and a matrix $SuccNumber$ such that $SuccNumber[i, j] = 1$ if t_j takes as input the output of t_j. It assigns the current round to every tasks t with no successor and then removes t from the successor lists of all the other tasks.

5.3 Rep Algorithm

The Rep algorithm takes as inputs five arrays $TaskRoundT$, $TaskRoundI$, $TaskT$, $TaskI$, and $Index$ and an integer $typeNumber$ that is the number of different types of tasks. $TaskRoundT$ stores, for each type, the number of tasks that have to be executed by $\mathcal{T}$ while $TaskRoundI$ stores, for each type, the number of tasks that can be executed by $\mathcal{I}$. $Index$ ensures that the while loop tests all possible combinations. Finally, the best repartition is stored in $TaskT$ and $TaskI$.

Algorithm 1. REPROUND($RepRound, taskNumber, SuccNumber$)

```
round = 1; count = 0;
while count != taskNumber do
    for i ∈ [0; taskNumber[ do
        if RepRound[i] == 0 then
            /*If no round has been assigned to t_i */
            succNumber = 0;
            for j ∈ [0; taskNumber[ do
                succNumber+ = Succ[i * taskNumber + j];
            if succNumber == 0 then
                RepRound[i] = round; count + +;
    /* We remove each task assigned in this round from the successor list */
    for i ∈ [0; taskNumber[ do
        if RepRound[i] == round then
            for j ∈ [0; taskNumber[ do
                Succ[i * taskNumber + j] = 0;
    round + +;
```

Algorithm 2. REP($TaskRoundT, TaskRoundI, TaskT, TaskI, typeNumber, Index$)

```
bestTime = +∞; consTimeT = 0;
for i ∈ [0; typeNumber[ do
    /*We first compute the computation time of the tasks that have to be
    executed by T*/
    constTimeT+ = TaskRoundT[i] * Perf[T][i];
i = 0
while i! = typeNumber do
    timeT = constTimeT;timeI = 0;
    for j ∈ [0; typeNumber[ do
        timeT+ = Index[j] * Perf[T][j];
        timeI+ = (TaskRoundI[j] − Index[j]) * Perf[I][j];
    if timeT > timeI then
        timeMax = timeT
    else
        timeMax = timeI
    if timeMax < bestTime then
        bestTime = timeMax
        for j ∈ [0; typeNumber[ do
            TaskT[round * typeNumber + j] = TaskRoundT[j] + Index[j];
            TaskI[round * typeNumber + j] = TaskRoundI[j] − Index[j];
    Index[0] + +;i = 0;
    while Index[i] > TaskRoundI[i] and i < typeNumber do
        Index[i] = 0; Index[i + 1] + +;
        i + +;
```

6 Conclusion

In this paper, we have proposed an almost generic method to simplify and precise the work of an operator wanting to construct the most possible efficient secure server-aided instance of a cryptographic primitive. Our work can easily be applied or adapted to any instance of any primitive.

This is obviously a first step and it remains a lot of work to do to improve the final result. For example, regarding cooperative execution of elementary tasks such as modular exponentiations or pairings, the related work clearly lacks of efficient and secure dedicated solutions. Regarding our main methodology, we also need to work on a true operator-free solution, especially regarding the security part. As said before, one option seems to work with formal methods, but it needs to be confirmed by additional work.

Acknowledgments. The work of the first, third and sixth author has been partially supported by the French ANR-11-INS-0013 LYRICS Project. The fifth author was supported by the Camus Walloon Region project. We are grateful to Nicolas Desmoulins for helpful discussions on the implementation aspects, and to anonymous referees for their valuable comments.

References

1. Abadi, M., Blanchet, B., Comon-Lundh, H.: Models and proofs of protocol security: A progress report. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 35–49. Springer, Heidelberg (2009)
2. Atallah, M.J., Frikken, K.B.: Securely outsourcing linear algebra computations. In: ASIACCS, pp. 48–59 (2010)
3. Barthe, G., Daubignard, M., Kapron, B., Lakhnech, Y.: Computational indistinguishability logic. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, pp. 375–386. ACM (2010)
4. Barthe, G., Grégoire, B., Heraud, S., Béguelin, S.Z.: Computer-aided security proofs for the working cryptographer. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 71–90. Springer, Heidelberg (2011)
5. Béguin, P., Quisquater, J.-J.: Fast server-aided RSA signatures secure against active attacks. In: Coppersmith, D. (ed.) CRYPTO 1995. LNCS, vol. 963, pp. 57–69. Springer, Heidelberg (1995)
6. Bellare, M., Shi, H., Zhang, C.: Foundations of group signatures: the case of dynamic groups. In: Menezes, A. (ed.) CT-RSA 2005. LNCS, vol. 3376, pp. 136–153. Springer, Heidelberg (2005)
7. Blanchet, B., Pointcheval, D.: Automated security proofs with sequences of games. In: Dwork, C. (ed.) CRYPTO 2006. LNCS, vol. 4117, pp. 537–554. Springer, Heidelberg (2006)
8. Brickell, E.F., Camenisch, J., Chen, L.: Direct anonymous attestation. In: ACM Conference on Computer and Communications Security 2004, pp. 132–145. ACM (2004)
9. Canard, S., Coisel, I., De Meulenaer, G., Pereira, O.: Group signatures are suitable for constrained devices. In: Rhee, K.-H., Nyang, D. (eds.) ICISC 2010. LNCS, vol. 6829, pp. 133–150. Springer, Heidelberg (2011)

10. Canard, S., Girault, M.: Implementing group signature schemes with smart cards. In: CARDIS 2002, pp. 1–10. USENIX (2002)
11. Chaum, D., Pedersen, T.P.: Wallet databases with observers. In: Brickell, E.F. (ed.) CRYPTO 1992. LNCS, vol. 740, pp. 89–105. Springer, Heidelberg (1993)
12. Chaum, D., van Heyst, E.: Group signatures. In: Davies, D.W. (ed.) EUROCRYPT 1991. LNCS, vol. 547, pp. 257–265. Springer, Heidelberg (1991)
13. Chen, L.: A daa scheme requiring less tpm resources. In: Bao, F., Yung, M., Lin, D., Jing, J. (eds.) Inscrypt 2009. LNCS, vol. 6151, pp. 350–365. Springer, Heidelberg (2010)
14. Chevallier-Mames, B., Coron, J.-S., McCullagh, N., Naccache, D., Scott, M.: Secure delegation of elliptic-curve pairing. Cryptology ePrint Archive, Report 2005/150 (2005), http://eprint.iacr.org/
15. Chevallier-Mames, B., Coron, J.-S., McCullagh, N., Naccache, D., Scott, M.: Secure delegation of elliptic-curve pairing. In: Gollmann, D., Lanet, J.-L., Iguchi-Cartigny, J. (eds.) CARDIS 2010. LNCS, vol. 6035, pp. 24–35. Springer, Heidelberg (2010)
16. Cramer, R., Pedersen, T.P.: Improved privacy in wallets with observers (extended abstract). In: Helleseth, T. (ed.) EUROCRYPT 1993. LNCS, vol. 765, pp. 329–343. Springer, Heidelberg (1994)
17. Delerablée, C., Pointcheval, D.: Dynamic fully anonymous short group signatures. In: Nguyên, P.Q. (ed.) VIETCRYPT 2006. LNCS, vol. 4341, pp. 193–210. Springer, Heidelberg (2006)
18. Girault, M., Lefranc, D.: Public key authentication with one (online) single addition. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 413–427. Springer, Heidelberg (2004)
19. Girault, M., Lefranc, D.: Server-aided verification: theory and practice. In: Roy, B. (ed.) ASIACRYPT 2005. LNCS, vol. 3788, pp. 605–623. Springer, Heidelberg (2005)
20. Hohenberger, S., Lysyanskaya, A.: How to securely outsource cryptographic computations. In: Kilian, J. (ed.) TCC 2005. LNCS, vol. 3378, pp. 264–282. Springer, Heidelberg (2005)
21. Kawamura, S.I., Shimbo, A.: Fast server-aided secret computation protocols for modular exponentiation. IEEE Journal on Selected Areas in Communications 11(5), 778–784 (1993)
22. Kang, B.G., Lee, M.S., Park, J.H.: Efficient delegation of pairing computation. IACR Cryptology ePrint Archive, 2005:259 (2005)
23. Lim, C.H., Lee, P.J.: Server (prover/signer)-aided verification of identity proofs and signatures. In: Guillou, L.C., Quisquater, J.-J. (eds.) EUROCRYPT 1995. LNCS, vol. 921, pp. 64–78. Springer, Heidelberg (1995)
24. Maitland, G., Boyd, C.: Co-operatively formed group signatures. In: Preneel, B. (ed.) CT-RSA 2002. LNCS, vol. 2271, pp. 218–235. Springer, Heidelberg (2002)
25. Matsumoto, T., Kato, K., Imai, H.: Speeding up secret computations with insecure auxiliary devices. In: Goldwasser, S. (ed.) CRYPTO 1988. LNCS, vol. 403, pp. 497–506. Springer, Heidelberg (1990)
26. Moradi, A., Poschmann, A., Ling, S., Paar, C., Wang, H.: Pushing the limits: A very compact and a threshold implementation of aes. In: Paterson, K.G. (ed.) EUROCRYPT 2011. LNCS, vol. 6632, pp. 69–88. Springer, Heidelberg (2011)
27. Nguyên, P.Q., Shparlinski, I.E.: On the insecurity of a server-aided RSA protocol. In: Boyd, C. (ed.) ASIACRYPT 2001. LNCS, vol. 2248, pp. 21–35. Springer, Heidelberg (2001)

28. Nguyên, P.Q., Stern, J.: The béguin-quisquater server-aided RSA protocol from crypto '95 is not secure. In: Ohta, K., Pei, D. (eds.) ASIACRYPT 1998. LNCS, vol. 1514, pp. 372–379. Springer, Heidelberg (1998)
29. Pfitzmann, B., Waidner, M.: Attacks on protocols for server-aided RSA computation. In: Rueppel, R.A. (ed.) EUROCRYPT 1992. LNCS, vol. 658, pp. 153–162. Springer, Heidelberg (1993)
30. Schnorr, C.-P.: Efficient identification and signatures for smart cards. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 239–252. Springer, Heidelberg (1990)
31. van Dijk, M., Clarke, D.E., Gassend, B., Edward Suh, G., Devadas, S.: Speeding up exponentiation using an untrusted computational resource. Des. Codes Cryptography 39(2), 253–273 (2006)
32. Xu, S., Yung, M.: Accountable ring signatures: a smart card approach. In: CARDIS 2004, pp. 271–286. Kluwer (2004)
33. Yao, A.C.-C.: Protocols for secure computations (extended abstract). In: FOCS, pp. 160–164. IEEE Computer Society (1982)

A Cooperative Execution of Elementary Tasks

We here provide some cooperative protocols one can find in the literature. Our goal is not to be exhaustive but to show that they are relevant to our methodology. To the best of our knowledge there is no general method to cooperatively perform a pseudo random generation or a hash computation. We focus on the costliest types of tasks introduced in section 2.1.

A.1 Exponentiation

Let g be an element of a group $\mathbb{G}$ and $a \in \mathbb{Z}_p$. The cooperative execution of g^a depends on the status of g and a. Since our method only considers one intermediary $\mathcal{I}$, we do not use the method proposed by [20], secure under the strong assumption that $\mathcal{T}$ has access to two intermediaries that cannot communicate with each other.

The method of [31] describes the way to outsource the computation of an exponentiation with verifiability of the result (*i.e.* the intermediary cannot convince $\mathcal{T}$ to accept a false value for g^a). Nevertheless, this method requires that g and a are both public.

Several papers [25,21,5] provide protocols for secret data, however, they were later proven insecure [29,5]. We then do not consider cooperative execution of exponentiation when secret data are involved.

A.2 Bilinear Map

In [19], Girault and Lefranc have proposed a way to compute $e(A, B)$ for secret A or B. Their solution works as follows. First, $\mathcal{T}$ chooses at random u and v in $\mathbb{Z}_p$, computes $X = A^u$ and $Y = B^v$ and sends theses values to $\mathcal{I}$. Then, $\mathcal{I}$ computes $z = e(X, Y)$, sends it to $\mathcal{T}$ which recovers $e(A, B)$ by computing $z^{(uv)^{-1}}$. Since X and Y are random elements of $\mathbb{G}_1$ and $\mathbb{G}_2$ we are able to simulate the communication between $\mathcal{T}$ and $\mathcal{I}$ without knowledge of A and B. The above cooperative execution of a pairing is then listener-receiver secure.

However, this protocol does not ensure verifiability of the result. Indeed, if $\mathcal{I}$ returns a random value from $\mathbb{G}_T$ instead of $e(X, Y)$, then $\mathcal{T}$ is unable to detect it. In [15] and [22], the authors provide verifiability but their protocol remain inefficient since they require respectively 10 and 7 exponentiations in $\mathbb{G}_T$ to check the validity of the result.

B AttributeTask Algorithm

The algorithm AttributeTask takes as input $TaskT$, the output of the Rep algorithm, $AssignedTaskT$, an array storing for each round and each type the number of tasks with status 0, $StatusTask$, $TypeTask$ and $RepRound$, arrays indicating the status, the type and the round of each task and three integers $typeNumber$, $roundNumber$ and $taskNumber$. Recall that the Rep algorithm

has chosen, for each type of tasks, how many of them $\mathcal{T}$ must compute to get the best repartition. However, it remains to choose which ones will be computed by this entity to minimize communication time. Since some tasks, involving secret data, are already assigned to $\mathcal{T}$, the algorithm only has to find n (see algorithm 3) other ones. It proceeds as follows. It counts, for each task of this type, the number of successors *succNumber* and stores the n of them with the fewest number in *BestRep*. Once this is done, we get the best repartition and it only remains to add the communication time.

Algorithm 3. AttributeTask($TaskT$, $AssignedTaskT$, $StatusTask$, $TypeTask$, $RepRound$, $typeNumber$, $roundNumber$, $taskNumber$)

```
for i ∈ [1, roundNumber] do
    for j ∈ [0, typeNumber[ do
        /* n is the number of tasks with status 1 that T has to compute */
        n = TaskT[(i − 1) ∗ typeNumber + j] − AssignedTaskT[(i − 1) ∗
        typeNumber + j];
        if n != 0 then
            repNumber = 0;
            minIndex = 0;
            for k ∈ [0, taskNumber[ do
                succNumber = 0;
                if RepRound[k] == i and StatusTask[k] == 1 and
                TypeTask[k] == j then
                    for l ∈ [0, taskNumber[ do
                        succNumber+ = Succ[k ∗ taskNumber + j];
                    if taskNumber < n then
                        /*BestRep stores the best repartition at this stage */
                        BestRep[repNumber] = k;
                        BestSucc[repNumber] = succNumber;
                        repNumber + +;
                        if BestSucc[minIndex] < succNumber then
                            minIndex = repNumber;
                    else
                        if BestSucc[minIndex] > succNumber then
                            BestRep[minIndex] = k;
                            BestSucc[minIndex] = succNumber;
                            for l ∈ [0; n[ do
                                if Bestsucc[l] > succNumber then
                                    minIndex = l;
```