

Privately Outsourcing Exponentiation to a Single Server: Cryptanalysis and Optimal Constructions

Céline Chevalier*

Fabien Laguillaumie†

Damien Vergnaud‡

Abstract

We address the problem of speeding up group computations in cryptography using a single untrusted computational resource. We analyze the security of an efficient protocol for securely outsourcing multi-exponentiations proposed at ESORICS 2014. We show that this scheme does not achieve the claimed security guarantees and we present several practical polynomial-time attacks on the delegation protocol which allows the untrusted helper to recover part (or the whole) of the device secret inputs. We then provide simple constructions for outsourcing group exponentiations in different settings (*e.g.* public/secret, fixed/variable bases and public/secret exponents). Finally, we prove that our attacks on the ESORICS 2014 protocol are unavoidable if one wants to use a single untrusted computational resource and to limit the computational cost of the limited device to a constant number of (generic) group operations. In particular, we show that our constructions are actually optimal.

Keywords. Secure outsource computation, Cryptanalysis, Coppersmith methods, Protocols, Optimality results

1 Introduction

We address the problem of “outsourcing” computation from a (relatively) weak computational device to a more powerful entity. This problem has been considered in various settings since many years (distributed-computing projects – *e.g.*, Mersenne prime search – or cloud computing) but the proliferation of mobile devices, such as smart phones or RFID tags, provides yet another venue in which a computationally weak device would like to be able to outsource a costly operation to a third party helper. Low-cost RFID tags do not usually have the computational or memory resources to perform complex cryptographic operations and it is natural to outsource these operations to some helper. The Near Field Technology (NFC) is embedded in the current generation of cellphone and can be used for transport tickets, credit cards, transit pass, loyalty cards or access control badges. This contactless technology raises many questions of disclosure of sensitive personal information. To preserve privacy, complex anonymity-oriented cryptographic protocols should be used and it is mandatory to delegate some of the costly operations from the chip to the phone, because these protocols are highly resource-consuming. However, in this scenario, this helper (*e.g.* the phone) can, potentially, be operated by a malicious adversary and we usually need to ensure that it does not learn anything about what it is actually computing.

The wild and successful deployment of cloud storage services, like Google Drive, Dropbox, or Amazon Cloud Drive make users outsource their data, for a personal or commercial purpose. These users actually have to trust their storage providers concerning the availability of their data, and indeed outages happen regularly. That is why it has been proposed to audit online storage services [34]. Cryptographic primitives are needed to convince customers (or an external trusted auditor) that their platforms are reliable. Among such primitives, provable data possessions [2] and proofs of retrievability [23] allow the storage cloud to prove that a file uploaded by a client has not been deteriorated or that it can be entirely retrieved. The computation needed on the verification side by the client are highly “*exponentiation-consuming*”.

Indeed, the core operation of these cryptosystems is group exponentiation, *i.e.*, computing u^a from a group element u and an exponent a which is central to cryptography, and the main goal of this paper is to analyze new and existing protocols outsourcing group exponentiation to a *single* untrusted helper.

*Université Paris 2

†LIP (UCBL, U. Lyon, CNRS, ENS Lyon, INRIA)

‡DI/ENS (ENS, CNRS, INRIA, PSL)

1.1 Prior work

One can date back the first protocol for securely outsourcing group exponentiation to the pre-computation scheme proposed by Schnorr in his seminal paper on discrete-logarithm based signatures [32]. Schnorr proposed a scheme for fast generation of pairs (g^k, k) where g is a generator of a cyclic group $\mathbb{G} = \langle g \rangle$ of prime-order p and k is a (purported) random element in \mathbb{Z}_p . The scheme was broken by de Rooij for the small parameters suggested by Schnorr (see [16]) but new proposals with provable security were proposed subsequently (see [7] and references therein).

Even if the problem of outsourcing cryptographic operations is not new, it has known a revival of interest in the last ten years with the development of mobile technologies. In 2005, Hohenberger and Lysyanskaya [21] provided a formal security definition for securely outsourcing computations from a computationally limited device to an untrusted helper and they presented two practical schemes. Their first scheme shows how to securely outsource group exponentiation to two, possibly dishonest, servers that are physically separated (and do not communicate). Their protocol achieves security as long as one of them is honest (even if the computationally limited device does not know which one). In 2012, Chen, Li, Ma, Tang and Lou [11] presented a nice efficiency improvement to the protocol from [21], but the security of their scheme also relies on the assumption that the two servers cannot communicate.

Since this separation of the two servers is actually a strong assumption hard to be met in practice, at ESORICS 2014 [38], Wang, Wu, Wong, Qin, Chow, Liu and Tan proposed a protocol to outsource group exponentiations to a *single* untrusted server. Their generic algorithm is very efficient and allows to outsource multi-exponentiations with fixed or variable exponent and bases (that can be public or secret).

1.2 Contributions of the paper

Our contributions are both theoretical and practical.

Our first result is a practical attack on a protocol for outsourcing multi-exponentiation proposed at ESORICS 2014 by Wang, Wu, Wong, Qin, Chow, Liu and Tan [38]. Our attack allows to recover secret information in polynomial time using lattice reduction. It shows that their solution is completely insecure. We will show that what they expected to achieve is actually theoretically impossible.

Our second contribution is the proposal of a taxonomy of exponentiation delegation protocols and the associated simple yet precise and formal models of protocols that allow a client \mathcal{C} (or delegator) who wants to compute a multi-exponentiation (which is a computation of the form $\prod_{i=1}^n u_i^{a_i}$ for group elements u_i 's and exponents a_i 's) to delegate an intermediate exponentiation to a more powerful server \mathcal{S} (or delegatee). The client's contribution in the computation is then only few multiplications of group elements and arithmetic operations modulo the underlying group order. We consider in this work only prime-order groups.

Our taxonomy covers all the practical situations : the group elements can be secret or public, variable or fixed, the exponents can be secret or public, and the result of the multi-exponentiation can also be either public or secret. As an example, a BLS digital signature [6] is a group element $\sigma = h(m)^a$, where m is the signed message, h a hash function, and a the secret key. The signature computation can be delegated with our protocol for a public group element (the hashed value of the message), a secret exponent (the secret key), and a public output (the signature). During an ElGamal decryption of a ciphertext $(c_1, c_2) = (g^r, m \cdot y^r)$ (where m is the plaintext and $y = g^a$ is the public key), one may want to securely delegate the computation of c_1^a (to recover m as c_2/c_1^a). Such an exponentiation can be delegated with our protocol for known group element (c_1), secret exponent (a) and secret result (c_1^a , in order to keep the plaintext m secret).

We propose a delegation protocol for each of the previously mentioned scenarios. The latency of sending messages back and forth has been shown to often be the dominating factor in the running time of cryptographic protocols. Indeed, round complexity has been the subject of a great deal of research in cryptography. We thus focus on the problem of constructing one-round delegation protocols; i.e., we authorize the client to call only once the server \mathcal{S} , and give him access to some pre-computations (consisting of pairs of the form (k, g^k)). We then consider their complexity, in terms of the number of group operations needed by the client to eventually get the desired result securely. These algorithms are simple and we prove that they are essentially optimal.

Our third and main contribution is the computation of lower bounds on the number of group operations needed on the delegator's side to securely compute his exponentiation when it has access to a helper server. To give these lower bounds, we analyze the security of delegation protocols in the generic group model which considers that algorithms do not exploit any properties of the encodings of group element. This model is usually used to rule out classes of attacks by an adversary trying to break a cryptographic assumption. We use it only to prove our lower bounds but we do not assume that an adversary against our protocols is limited to generic operations in the underlying group. As mentioned above, these lower bounds

Constructions for Outsourcing Fixed Base Exp.							Constructions for Outsourcing Var. Base Exp.						
u	a	u^a	Nb of delegations	Lower bound for #operations in \mathbb{G}	Achieved Complexity	Optimality	u	a	u^a	Nb of delegations	Lower bound for #operations in \mathbb{G}	Achieved Complexity	Optimality
Pub.	Pub.	Pub.					Pub.	Pub.	Pub.				
Pub.	Pub.	Sec.					Pub.	Pub.	Sec.				
Pub.	Sec.	Pub.	1	0	0 (P. 2)	✓	Pub.	Sec.	Pub.	s	$\frac{\log p}{s+1}$	$\frac{\log p}{s+1}$ (P. 4 and 5)	✓
Pub.	Sec.	Sec.	1	1	1 (P. 3)	✓	Pub.	Sec.	Sec.	s	$\frac{\log p}{s+1}$	$\frac{\log p}{s+1}$ (P. 4 and 5)	✓
Sec.	Pub.	Pub.					Sec.	Pub.	Pub.				
Sec.	Pub.	Sec.	1	1	1 (P. 1)	✓	Sec.	Pub.	Sec.	1	$\frac{\log p}{\ell+3}$ *	$\frac{\log p}{\ell}$ (P. 8)	?
										2	≤ 3	3 (P. 7)	?
Sec.	Sec.	Pub.	1	0	0 (P. 2)	✓	Sec.	Sec.	Pub.	s	$\frac{\log p}{s+1}$	$\frac{\log p}{s}$ (P. 6)	?
Sec.	Sec.	Sec.	1	1	1 (P. 1)	✓	Sec.	Sec.	Sec.	s	$\frac{\log p}{s+1}$	$\frac{\log p}{s}$ (P. 6)	?

* ℓ is the number of available pairs (k, g^k) , p is the order of \mathbb{G} , and P. means "Protocol" (See p. 11).

Table 1: Outsourcing Protocols for Single Exponentiation (Summary)

tell us that our protocols cannot be significantly improved. A summary of our results is given in Table 1 (and all our results are collected in Table 2 in the core of the paper).

2 Exponentiation Delegation: Definitions.

The (multi-)exponentiations are computed in a group \mathbb{G} whose description is provided by an algorithm `GroupGen`, which takes as input a security parameter λ . It provides a set *params* which contains the group description, its prime¹ order, say p , and one or many generators. Let n be an integer, we denote by \mathbf{a} (*resp.* \mathbf{u}) a vector of n exponents $a_i \in \mathbb{Z}_p$ (*resp.* group elements $u_i \in \mathbb{G}$). The aim of the protocols that follows is to compute $\prod_{i=1}^n u_i^{a_i}$, denoted as $\mathbf{u}^{\mathbf{a}}$.

We consider a delegation of an exponentiation as a 2-party protocol between a client \mathcal{C} and a server \mathcal{S} . We denote as $(y_C, y_S, tr) \leftarrow (\mathcal{C}(1^\lambda, params, (\mathbf{a}, \mathbf{u})), \mathcal{S}(1^\lambda))$ the protocol at the end of which \mathcal{C} knows y_C and \mathcal{S} learns y_S (usually an empty string). The string tr is the transcript of the interaction. In all our protocols, the server will be very basic, since it will only perform exponentiations whose basis and exponent are sent to him by the client. In [10], Cavallo et al. emphasized the need for delegation of group inverses since almost all known protocols for delegated exponentiation do require inverse computations from the client. They presented an efficient and secure protocol for delegating group inverses. However, our protocols do not require such computations and our lower bounds hold even in groups in which inverse computation is efficient (and therefore does not need to be delegated, see Remark 6).

To model the security notions, and to simplify the exposition, we describe by a *computation code* β (which is a binary vector of length 4), the scenario of the computation. Indeed, according to the applications, some of the data on which the computations are performed may be either public or secret. In the computation of $\mathbf{u}^{\mathbf{a}}$, the vector of basis \mathbf{u} , the vector of exponents \mathbf{a} or the result $\mathbf{u}^{\mathbf{a}}$ may be unknown (and especially to the adversary). The three first entries of the code describe the secrecy of respectively \mathbf{u} , \mathbf{a} and $\mathbf{u}^{\mathbf{a}}$: a 0 means that the data is hidden to the adversary, and 1 means that the data is public. The last entry indicates whether the base is fixed (f) or variable (v). For instance, the code 101v means that \mathbf{u} is public, the exponent \mathbf{a} is secret, and the result $\mathbf{u}^{\mathbf{a}}$ is public, while the base is variable. Note that we consider the *whole* vectors (*i.e.*, all of its coordinates) to be either public or private, whereas we could imagine that, for a vector \mathbf{u} of exponents for instance, some of these could be public, and others could be kept secret. The following notions should then be declined according to these scenarios

¹In this paper, following prior works, we consider only prime order groups, but most of our results can be generalized to composite order groups.

2.1 Correctness

The correctness requirement for delegation of a (multi-)exponentiation means that when the server and the client follow honestly the protocol, the client's output is actually the expected (multi-)exponentiation.

Definition 1 (Correctness). *Let λ be a security parameter. We say that $(\mathcal{C}, \mathcal{S})$ satisfies correctness if*

$$\Pr[\text{params} \leftarrow \text{GroupGen}(1^\lambda), (y_C, y_S, tr) \leftarrow (\mathcal{C}(1^\lambda, \text{params}, \mathbf{a}, \mathbf{u}), \mathcal{S}(1^\lambda)) : y_C = \prod_{i=1}^n u_i^{a_i}] = 1$$

for all \mathbf{a} in the range \mathbb{Z}_p^n and all \mathbf{u} in \mathbb{G}^n .

2.2 One-wayness

The most natural security notion that a delegation protocol must fulfill is the *one-wayness* during the computation. It basically means that an attacker cannot compute any secret data involved during the computation. More precisely, Fig. 1 describes the one-wayness security experiment. The attack presented in Section 4 against Wang et al.'s protocol is against the one-wayness of their scheme. The attacker \mathcal{A} is initially fed with information that depends on the scenario. The role of the procedure \mathcal{I} is to set the initial information given to the attacker. It takes as input \mathbf{u}, \mathbf{a} and the computation code $\beta = (\beta_1, \beta_2, \beta_3, \beta_4)$ and outputs a subset $\mathcal{I}(\mathbf{u}, \mathbf{a}; \beta) \subseteq \{\mathbf{u}, \mathbf{a}, \mathbf{u}^{\mathbf{a}}\}$ such that

- $\mathbf{u} \in \mathcal{I}(\mathbf{u}, \mathbf{a}; \beta)$ if and only if $\beta_1 = 1$;
- $\mathbf{a} \in \mathcal{I}(\mathbf{u}, \mathbf{a}; \beta)$ if and only if $\beta_2 = 1$;
- $\mathbf{u}^{\mathbf{a}} \in \mathcal{I}(\mathbf{u}, \mathbf{a}; \beta)$ if and only if $\beta_3 = 1$.

The attacker then engages in a series of delegation protocols, where he can adaptively choose the secrets involved during the protocols, and he eventually outputs an answer $A^* = (\mathbf{u}^*, \mathbf{a}^*, v^*) \in \mathbb{G}^n \times \mathbb{Z}_p^n \times \mathbb{G}$. The attacker is said to win this experiment, if the predicate $P_1(A^*, \mathbf{u}, \mathbf{a})$ holds where $P_1(A^*, \mathbf{u}, \mathbf{a})$ is equal to 1 if and only if the three following equalities hold:

- $\mathbf{u}^* = \mathbf{u}$;
- $\mathbf{a}^* = \mathbf{a}$;
- $v^* = \mathbf{u}^{\mathbf{a}}$.

Note that if the computation code β contains some values equal to 1, it is actually trivial for the adversary to output an answer A^* that satisfies some of these equalities. In particular, for the case where $\mathbf{u}, \mathbf{a}, \mathbf{u}^{\mathbf{a}}$ are public (i.e., the computation code $\beta = (1, 1, 1, \beta_4)$), the security notion cannot be achieved. Similarly, the security notion for the cases with the computation code $\beta = (1, 1, 0, \beta_4)$ and $\beta = (0, 1, 1, \beta_4)$ cannot be achieved (but the latter case may have some interest for composite order groups).

Definition 2 (One-wayness). *Let λ be a security parameter, GroupGen be a group generator, and $(\mathcal{C}, \mathcal{S})$ be a client-server protocol for the server-aided computation of the multi-exponentiation $\mathbf{u}^{\mathbf{a}}$. Let $\beta \in \{0, 1\}^4$ be the computation code. We say that $(\mathcal{C}, \mathcal{S})$ satisfies (τ, ε) -one-wayness against a malicious adversary if, for any algorithm running in time τ , it holds that*

$$\Pr[\text{bit} \leftarrow \text{Exp}_{ow}(\mathcal{A}, \beta) : \text{bit} = 1] \leq \varepsilon$$

where $\text{Exp}_{ow}(\mathcal{A}, \beta)$ is the one-wayness experiment described in Figure 1.

2.3 Privacy

We describe now a notion of security that relaxes the usual simulation-based security from [21] and [38]. The simulation-based security notion captures in perhaps the most direct way the intuition of a good notion of privacy. Roughly, it says that “whatever can be efficiently computed about the secret inputs given the protocol's view can be computed without this view”. However, it is a relatively complex and subtle notion to formalize (see [21] or [38] for details).

In this paper we instead consider a simpler indistinguishability-based security notion, called *privacy* that captures that an untrusted helper cannot tell which inputs the other parties might have used. The formalization was provided in [10]. It is simple and easy to use: it says that if we take two secret inputs

Experiment $\mathbf{Exp}_{ow}(\mathcal{A}, \beta)$

```

params  $\leftarrow$  GroupGen( $1^\lambda$ )
 $(\mathbf{a}, \mathbf{u}) \leftarrow^R \mathbb{Z}_p^n \times \mathbb{G}^n$ 
init  $\leftarrow \mathcal{I}(\beta)$ 
 $i \leftarrow 1, tr_0 \leftarrow \emptyset$ 
 $(\alpha, aux) \leftarrow \mathcal{A}(1^\lambda, params, init)$ 
while  $\alpha \neq \mathbf{attack}$  do
     $(y_i, (\alpha, (\mathbf{a}_{i+1}, \mathbf{u}_{i+1}), aux), tr_i) \leftarrow (\mathcal{C}(1^\lambda, params, (\mathbf{a}_i, \mathbf{u}_i)), \mathcal{A}(aux, tr_{i-1}))$ 
     $i \leftarrow i + 1$ 
 $(y, A^*, tr) \leftarrow (\mathcal{C}(1^\lambda, params, (\mathbf{a}, \mathbf{u})), \mathcal{A}(aux))$ 
Return 1 if  $P_1(A^*, \mathbf{u}, \mathbf{a}) = 1$  and 0 otherwise

```

Figure 1: One-wayness

(even adversarially chosen), an “honest-but-curious” adversary running the outsource protocol with one input picked uniformly at random cannot tell which it was with a probability significantly better than that of guessing. This notion is similar to *Input-Indistinguishable Computation* introduced by Micali, Pass and Rosen in [27].

Note that this security notion is implied by the simulation-based one from [21, 38]. In particular, since we will prove that the protocol from [38] does not achieve our security notion, we obtain that it does not achieve the stronger simulation-based security notion from [21, 38] (contrary to what is claimed in [38]). In Section 6, we prove that it is impossible to design some secure outsourcing exponentiation protocols (for our security definition) for a single untrusted computational resource if one wants to limit the computational cost of the limited device to a constant number of (generic) group operations. This result readily implies that this task is also impossible for the stronger simulation-based security notion from [21] and [38].

Once again, the advantage of the adversary \mathcal{A} in the privacy experiment, depicted in Fig. 2, will be settled according to the context of the delegation. We use the predicate $P_2((\mathbf{a}_0^*, \mathbf{u}_0^*), (\mathbf{a}_1^*, \mathbf{u}_1^*), \beta)$ to tell that when the base, the exponent or the result of the exponentiation is known (according to the computation code), the base, the exponent chosen by the adversary or the corresponding result must be the same for both pairs (otherwise, the privacy would be trivially broken). The predicate $P_2((\mathbf{a}_0^*, \mathbf{u}_0^*), (\mathbf{a}_1^*, \mathbf{u}_1^*), \beta)$ is defined as the conjunction of the three following disjunctions:

- $\mathbf{u}_0^* = \mathbf{u}_1^*$ or $\beta_1 = 0$;
- $\mathbf{a}_0^* = \mathbf{a}_1^*$ or $\beta_2 = 0$;
- $(\mathbf{u}_0^*)^{\mathbf{a}_0^*} = (\mathbf{u}_1^*)^{\mathbf{a}_1^*}$ or $\beta_3 = 0$;

As above, the security definition cannot be achieved for some computation codes.

Definition 3 (Privacy). *Let λ be a security parameter, GroupGen be a group generator, and $(\mathcal{C}, \mathcal{S})$ be a client-server protocol for the server-aided computation of the multi-exponentiation $\mathbf{u}^{\mathbf{a}}$. Let $\beta \in \{0, 1\}^4$ be the computation code. We say that $(\mathcal{C}, \mathcal{S})$ satisfies (τ, ε) -privacy against a malicious adversary if, for any algorithm running in time τ , it holds that*

$$\left| \Pr[\text{bit} \leftarrow \mathbf{Exp}_{priv}(\mathcal{A}, \beta) : \text{bit} = 1] - \frac{1}{2} \right| \leq \varepsilon$$

where $\mathbf{Exp}_{priv}(\mathcal{A}, \beta)$ is the privacy experiment described in Figure 2.

Remark 1. *As mentioned in [12, 9], a delegation protocol that does not ensure verifiability may cause severe security problems (in particular if the delegated computation occurs in the verification algorithm of some authentication protocol). However, verifiability is not necessarily mandatory in scenarios where the delegated computation is used for instance in an encryption scheme as a session key. In this case, one can indeed use additional cryptographic techniques to ensure that the values returned by the powerful device are correct (e.g. by adding a MAC or other redundancy to the ciphertext).*

Experiment $\mathbf{Exp}_{priv}(\mathcal{A}, \beta)$
--

```

params ← GroupGen( $1^\lambda$ )
( $\alpha, (\mathbf{a}_1, \mathbf{u}_1), aux$ ) ←  $\mathcal{A}(1^\lambda, params)$ 
 $i \leftarrow 1, tr_0 \leftarrow \emptyset$ 
while  $\alpha \neq \mathbf{attack}$  do
    ( $y_i, (\alpha, (\mathbf{a}_{i+1}, \mathbf{u}_{i+1}), aux), tr_i$ ) ← ( $\mathcal{C}(1^\lambda, params, (\mathbf{a}_i, \mathbf{u}_i)), \mathcal{A}(aux, tr_{i-1})$ )
     $i \leftarrow i + 1$ 
    ( $(\mathbf{a}_0^*, \mathbf{u}_0^*), (\mathbf{a}_1^*, \mathbf{u}_1^*), aux$ ) ←  $\mathcal{A}(aux)$ 
     $b \xleftarrow{\$} \{0, 1\}$ 
    ( $y, b^*, tr$ ) ← ( $\mathcal{C}(1^\lambda, params, (\mathbf{a}_b^*, \mathbf{u}_b^*)), \mathcal{A}(aux)$ )
Return 1 if  $(b^* = b) \wedge P_2((\mathbf{a}_0^*, \mathbf{u}_0^*), (\mathbf{a}_1^*, \mathbf{u}_1^*), \beta)$ 
0 otherwise

```

Figure 2: Privacy

3 Underlying Tools

3.1 Generic Group Model

Let GroupGen be a group generator which takes as input a security parameter λ . It provides a set $params$ which contains a description of a (multiplicative) group (\mathbb{G}, \cdot) , the group order, say $p = |\mathbb{G}|$, and one generator g . As usual, the generic group model in \mathbb{G} is implemented by choosing a random encoding $\sigma : \mathbb{G} \rightarrow \{0, 1\}^m$ (with $2^m > p$). Instead of working directly with group elements, a generic algorithm \mathcal{A} takes as input (in addition to the group order p) their image under σ . This way, all \mathcal{A} can test is group elements equality (by encoding equality). \mathcal{A} is also given access to an oracle \mathcal{G} computing group multiplication: taking two encodings $\sigma(g_1)$ and $\sigma(g_2)$ of two group elements $g_1, g_2 \in \mathbb{G}$ as inputs and returning the encoding $\sigma(g_1 \cdot g_2)$ of the product $g_1 \cdot g_2 \in G$. We can assume that \mathcal{A} submits to the oracle only encodings of elements it had previously received. This is because we can choose m large enough so that the probability of choosing a string that is also in the image of σ is negligible (see [35] for details). In particular, in this paper², a generic algorithm \mathcal{A} cannot generate encodings of new group elements.

Usually, the generic group model is used to rule out classes of attacks by an adversary trying to break a cryptographic assumption. In contrast, in this paper, we use the generic group model to prove a lower bound on the complexity of the delegation protocol. In order to prove our complexity lower bounds, we make intensive use of the following simple lemma:

Lemma 1. *Let GroupGen be a group generator, let \mathbb{G} be a group of prime order p output by GroupGen and let \mathcal{A} be a generic algorithm in \mathbb{G} . If \mathcal{A} is given as inputs encodings $\sigma(g_1), \dots, \sigma(g_n)$ of groups elements $g_1, \dots, g_n \in \mathbb{G}$ (for $n \in \mathbb{N}$) and outputs the encoding $\sigma(h)$ of a group element $h \in \mathbb{G}$ in time τ , then there exists positive integers $\alpha_1, \dots, \alpha_n$ such that $h = g_1^{\alpha_1} \dots g_n^{\alpha_n}$ and $\max(\alpha_1, \dots, \alpha_n) \leq 2^\tau$.*

Proof. We can define a map $\pi : \{0, 1\}^m \rightarrow \mathbb{Z}^n$ which associates to each encoding obtained by \mathcal{A} during its execution an n -dimensional vector in \mathbb{Z}^n . For each input encoding $\sigma(g_i)$, $\pi(\sigma(g_i))$ is defined as the i -th vector from the \mathbb{Z}^n canonical basis (for $i \in \{1, \dots, n\}$) and for each encoding $\sigma(h_1)$ and $\sigma(h_2)$ queried to \mathcal{G} , $\pi(\sigma(h_1 \cdot h_2)) = \pi(\sigma(h_1)) + \pi(\sigma(h_2))$. By construction, during the whole execution of \mathcal{A} , we have $\pi(\sigma(h)) = (\alpha_1, \dots, \alpha_n)$ if and only if $h = g_1^{\alpha_1} \dots g_n^{\alpha_n}$ for all encodings $\sigma(h)$ and the ℓ_∞ -norm of $\pi(\sigma(h_1 \cdot h_2))$ is upper-bounded by $\ell_\infty(\pi(\sigma(h_1))) + \ell_\infty(\pi(\sigma(h_2)))$. Since the ℓ_∞ -norm of the input encodings $\pi(\sigma(g_i))$ is equal to 1 (for $i \in \{1, \dots, n\}$) and the ℓ_∞ -norm of encodings at most doubles for each query to \mathcal{G} , we obtained the claimed result. \square

All the exponentiation delegation protocols we present in Section 5 are generic (interactive) algorithms. However, we want to stress that their security analysis is provided in the standard security model (without any idealized assumption and in particular we do not assume that an adversary is limited to generic operations in the underlying group).

In some specific groups, it is possible to improve the efficiency of exponentiation algorithms by using non-generic operations (see [18, § 11.3], for instance):

²The lower bounds on the complexity of generic delegation protocols given in Section 6 and Section D do hold without this assumption but with unnecessarily complicated proofs.

- One may take advantage of an additional structure in subgroups of the multiplicative group of (non-prime) finite fields³ $\mathbb{F}_{q^n}^*$ with $n \geq 2$. Indeed, in this setting one can use a *normal basis* $\{\alpha, \alpha^q, \dots, \alpha^{q^{n-1}}\}$ of \mathbb{F}_{q^n} over \mathbb{F}_q to represent group elements in order to make the computation of the q -th power of an element as a simple (and almost free) cyclic shift of its representation.
- One may also take advantage of the fact that in certain algebraic groups, group inversion is sometimes much more efficient than a group multiplication (in particular in subgroups of elliptic curves over finite fields). In this case, one can use signed expansions of exponents when computing (multi-)exponentiation and in particular the *w-ary non-adjacent form* method which guarantees that on average there will be fewer group multiplications in Algorithm 1 for instance.
- One may use (more generally) groups equipped with efficient endomorphisms (e.g., Frobenius endomorphism, complex multiplication endomorphism). This method was originally proposed by Gallant, Lambert and Vanstone [19] to perform *group exponentiation with endomorphism decomposition*. In a cyclic group \mathbb{G} , any endomorphism is the group exponentiation by some integer (*eigenvalue* of the endomorphism) and for general group exponentiation, one can decompose the exponent as a weighted sums of these eigenvalues (with “small” weights) and then use a multi-exponentiation algorithm such as Algorithm 1.

It is sometimes possible to improve (but only by a constant factor) the efficiency of the exponentiation delegation protocols we present in Section 5 by using similar techniques. In these situations, the complexity lower bounds from Section 6 and Appendix D do no hold anymore but one can adapt our arguments (see Remark 6 for instance).

3.2 Multi-exponentiation by Simultaneous 2^w -ary method

Algorithm 1 computes the multi-exponentiation $\prod_{i=1}^t g_i^{x_i} \in \mathbb{G}$, for $g_1, \dots, g_t \in \mathbb{G}$ and $x_1, \dots, x_t \in \mathbb{N}$ by using the simultaneous 2^w -ary method introduced by Straus in 1964 [37]. The method looks at w bits of each of the exponents for each evaluation stage group multiplication (where w is a small positive integer), i.e. tw bits in total (see [28, 3] for details of different multi-exponentiation techniques).

Algorithm 1 Multi-Exponentiation by Simultaneous 2^w -ary method

Input: $g_1, \dots, g_t \in \mathbb{G}$, $x_1, \dots, x_t \in \mathbb{N}$ with $\ell = \max_{i \in \{1, \dots, t\}} \lceil \log x_i \rceil$ and

$$x_j = \sum_{i=0}^{\lfloor \ell/w \rfloor - 1} e_{i,j} 2^{wi} \in \mathbb{N} \text{ and } e_{i,j} \in \{0, 2^w - 1\} \text{ for } i \in \{0, \dots, \ell/w - 1\} \text{ and } j \in \{1, \dots, t\}$$

Output: $g_1^{x_1} \dots g_t^{x_t} \in \mathbb{G}$

for all non-zero t -tuples $E = (E_1, \dots, E_t) \in \{0, \dots, 2^w - 1\}^t$ **do**

$$g_E \leftarrow \prod_{1 \leq i \leq t} g_i^{E_i}$$

▷ Precomputation stage

end for

$$h \leftarrow 1_{\mathbb{G}}$$

for i from $\lfloor \ell/w \rfloor - 1$ to 0 **do**

for j from 1 to w **do**

$$h \leftarrow h^2$$

end for

$$E \leftarrow (e_{i,1}, e_{i,2}, \dots, e_{i,t})$$

$$h \leftarrow h \cdot g_E$$

▷ Multiply h by table entry $g_E = \prod_{1 \leq k \leq t} g_i^{e_{i,k}}$

end for

return h

Complexity: The precomputed table contains $2^{tw} - 1 - t$ non-trivial entries among which $2^{t(w-1)} - 1$ can be computed by squaring other table entries (all the E_i are even). The remaining $2^{tw} - 2^{t(w-1)} - t$ entries require one general multiplication each. The total cost is thus for the precomputation phase $2^{tw} - 2^{t(w-1)} - t$ multiplications and $2^{t(w-1)} - 1$ squarings and $\ell(2^{tw} - 1)/2^{tw} w \leq \ell/w$ multiplications on average and ℓ squarings. For $t = 2$, the cost is minimal for w around $1/2 \log \ell - \log \log \ell$ with $\ell(1 + 3/\log \ell) = \ell(1 + o(1))$ multiplications overall.

³The most studied case was $q = 2$ but it is not interesting anymore in cryptography due to the recent impressive progress on finite field discrete logarithms [5]. However, this technique may still found applications in pairing-based cryptography.

3.3 Exponents Decomposition

Let p be a prime number (in our protocols in the following, p will be the order of the underlying group \mathbb{G}). Let $s \geq 1$ be an integer and $\rho_1, \dots, \rho_s \in \mathbb{Z}_p$. Let $a \in \mathbb{Z}_p$, a s -dimensional decomposition of a with respect to $\boldsymbol{\rho} = (\rho_1, \dots, \rho_s)$ is an s -dimensional vector $\boldsymbol{\alpha} = (\alpha_1, \dots, \alpha_s) \in \mathbb{Z}_p^s$ such that

$$\langle \boldsymbol{\alpha}, \boldsymbol{\rho} \rangle := \alpha_1 \rho_1 + \dots + \alpha_s \rho_s = a \pmod{p}.$$

It is well-known that if the scalars ρ_i for $i \in \{1, \dots, s\}$ have pairwise differences of absolute value at least $p^{1/s}$, then there exists a polynomial-time algorithm which on inputs a and $\boldsymbol{\rho}$ outputs a s -dimensional decomposition $\boldsymbol{\alpha} \in \mathbb{Z}_p^s$ of a with respect to $\boldsymbol{\rho}$ such that $0 \leq \alpha_i \leq C \cdot p^{1/s}$ for $i \in \{1, \dots, s\}$ (for some small constant $C > 0$). To find this “small decomposition” of a , the algorithm applies a lattice reduction algorithm (such as the LLL-algorithm) to produce a short basis of the \mathbb{Z} -lattice of dimension $s+1$ spanned by the vectors $(p, 0, 0, \dots, 0), (\rho_1, 1, 0, \dots, 0), (\rho_2, 0, 1, \dots, 0), \dots, (\rho_s, 0, 0, \dots, 1)$ and applies Babai rounding algorithm [4] to find a nearby vector in this lattice from $(a, 0, \dots, 0)$ (see [36] for details). In the following, we will refer to this algorithm as the GLV Decomposition Algorithm (GLV-Dec for short) since the method was first introduced by Gallant, Lambert and Vanstone [19] to perform group exponentiations with endomorphism decomposition.

Many important problems in cryptanalysis amount to solving polynomial equations with partial information about the solutions. In 1996, Coppersmith introduced two celebrated lattice-based techniques [14, 13] for finding small roots of polynomial equations. In the following, we will consider settings in which there exists an s -dimensional decomposition of a scalar a that is shorter than the one produced by the GLV Decomposition Algorithm. Given some bounds X_1, \dots, X_s in \mathbb{N} such that $X_1 \cdots X_s < p$, for a random scalar $a \in \mathbb{Z}_p$ and a random vector $\boldsymbol{\rho} \in \mathbb{Z}_p^s$, we expect a unique vector $\boldsymbol{\alpha} = (\alpha_1, \dots, \alpha_s) \in \mathbb{Z}_p^s$ such that $\langle \boldsymbol{\alpha}, \boldsymbol{\rho} \rangle = a$ with $\alpha_i < X_i$ for all $i \in \{1, \dots, s\}$. In Section 4, we provide an algorithm that solves such a problem for $s = 2$. In his PhD thesis [20], Herrmann mentions a “folklore method” to solve this problem. He states the following theorem.

Theorem 1. [20, Theorem 6] *Let $p \in \mathbb{N}$ and $f(x_1, \dots, x_s) = \rho_1 x_1 + \dots + \rho_s x_s$ be a linear polynomial in s variables with $\gcd(\rho_i, p) = 1$ for at least one $i \in \{1, \dots, s\}$. Further let $X_i \in \mathbb{N}$ be upper-bounds on the solutions α_i (for $i \in \{1, \dots, s\}$) of the equation $f(\alpha_1, \dots, \alpha_s) = 0 \pmod{p}$. Heuristically, we can find the solution $(\alpha_1, \dots, \alpha_s)$ if $\prod_{i=1}^s X_i \leq p$ in time polynomial in $\log p$.*

This Coppersmith-like results holds for the homogeneous case, so our exponent decomposition problem can be solved by adding an extra variable α_{s+1} . The heuristic assumption in the proof of Theorem 1 comes from the fact that the lattice constructed by the algorithm may contain several solutions (bounded by the X_i 's) which satisfy $f(x_1, \dots, x_n) = 0 \pmod{p}$ (see [20] for details). In the context of our proof of optimality, this is not an issue, since finding any solution will allow to distinguish between the challenge cases.

3.4 Computations of pairs (g^k, k) .

To outsource the computation of an exponentiation in a group \mathbb{G} of prime order p , (pseudo-)random pairs of the form $(g^k, k) \in \mathbb{G} \times \mathbb{Z}_p$ are sometimes used to hide sensitive information to the untrusted server. This looks like a “chicken-and-egg problem” but there exist several techniques to make it possible for a computationally limited device to have such pairs at its disposal, at a low cost. A trivial method is to load its memory with many genuine (generated by a trusted party) random and independent couples. In other settings, a mobile device with limited computing capabilities can precompute “offline” such pairs at low speed and power. If the device can do a little more computation, there exist other preprocessing techniques, that may depend whether the base or the exponent varies.

We only mention here the main technique to produce these pairs (among many others [15, 8, 26]). The key ingredient is Boyko, Peinado and Venkatesan generator from [7]: the idea is to store a small number of precomputed pairs (g^{α_i}, α_i) , and when a fresh pair is needed, the device outputs a product $g^k = \prod_{i \in S} g^{\alpha_i}$ with $k = \sum_{i \in S} \alpha_i$ for a random set S . It has then been improved by Nguyen, Shparlinski and Stern [29] that allows to re-use some α_i in the product. This generator is secure against adaptive adversaries and performs $O(\log \log(p)^2)$ group operations. For some parameters, the generator from [29] is proved to have an output distribution statistically close to the uniform distribution. Obviously, these generators are of practical interest only if the base g is fixed and used multiple times.

In the sequel we will assume that the delegator may have access to some (pseudo-)random power generator $\mathcal{B}(\cdot)$ that at invocation (with no input) outputs a single (pseudo)-random pair $(g^k, k) \in \mathbb{G} \times \mathbb{Z}_p$ where k is uniformly distributed in \mathbb{Z}_p (or statistically close to the uniform distribution). If the generator $\mathcal{B}(\cdot)$ is invoked several times, we assume that the output pairs are independent. In order to evaluate the efficiency

of delegation protocols, we consider explicitly the query complexity to the generator $\mathcal{B}(\cdot)$ (and depending on the context, this can be interpreted as storage of precomputed values, offline computation or use of the generator from [29] and thus additional multiplications in \mathbb{G}).

4 Attack on Wang et al.’s algorithm from ESORICS 2014

Wang et al. proposed a generic algorithm to outsource the computation of several multi-exponentiations with variable exponents and variable bases. Their algorithm, called **GExp**, takes as input a list of tuples $((\{a_{i,j}\}_{j=1,s}; \{u_{i,j}\}_{j=1,s}))_{i=1,r}$ and computes the list of multi-exponentiations $(\prod_{j=1}^r u_{i,j}^{a_{i,j}})_{i=1,s}$. It is claimed that this algorithm is secure in a strong model where the computation is outsourced to a single untrusted server [38, Theorem 1]. We will show that the protocol can indeed be broken in polynomial time using lattice reduction if two (simple) exponentiations are outsourced with the *same* exponent, which is the case in the scenario of proof of data possession presented in [38, Section 4]. This means that their generic algorithm to outsource exponentiations does not achieve the claimed security.

Description of Wang et al.’s protocol. The setting of Wang et al.’s protocol is the following: \mathbb{G} is a cyclic group of prime order p , and g is a generator. For $1 \leq i \leq r$ and $1 \leq j \leq s$, $a_{i,j}$ are uniform and independent elements of \mathbb{Z}_p^* , and $u_{i,j}$ are random elements from \mathbb{G} . They assumed the $a_{i,j}$ ’s, the $u_{i,j}$ ’s and the result are secret (and the $u_{i,j}$ are variable, i.e. $\beta = 000v$). The protocol is divided into three steps:

- **Step 1.** The delegator \mathcal{C} generates four random pairs $(\alpha_k, \mu_k)_{k=1,4}$ where $\mu_k = g^{\alpha_k}$ (using a pseudo-random power generator). A Υ -bit element χ is randomly picked. Then, for all $1 \leq i \leq r$ and $1 \leq j \leq s$, the elements $b_{i,j}$ are randomly picked in \mathbb{Z}_p^* . It sets⁴

$$c_{i,j} = a_{i,j} - b_{i,j}\chi \pmod{p} \quad (1)$$

$$w_{i,j} = u_{i,j}/\mu_1 \quad (2)$$

$$h_{i,j} = u_{i,j}/\mu_3 \quad (3)$$

$$\theta_i = (\alpha_1 \sum_{j=1}^s b_{i,j} - \alpha_2) + (\alpha_3 \sum_{j=1}^s c_{i,j} - \alpha_4) \pmod{p}. \quad (4)$$

- **Step 2.** The second step consists in invoking the (untrusted) delegatee \mathcal{S} for some exponentiations. To do so, \mathcal{C} generates (using a (pseudo-)random power generator) $r + 2$ random pairs $(t_i, g^{t_i})_{i=1,r+2}$ and queries (in random order) \mathcal{S} on

- $(\theta_i/t_i, g^{t_i})$ to obtain $B_i = g^{\theta_i}$ for all $1 \leq i \leq r$,
- $(\theta/t_{r+1}, g^{t_{r+1}})$ to obtain $A = g^\theta$ with $\theta = t_{r+2} - \sum_{i=1}^r \theta_i \pmod{p}$,
- $\begin{cases} (b_{i,j}, w_{i,j}) & \text{to get } C_{i,j} = (u_{i,j}/\mu_1)^{b_{i,j}} \\ (c_{i,j}, h_{i,j}) & \text{to get } D_{i,j} = (u_{i,j}/\mu_3)^{c_{i,j}} \end{cases}$ for $1 \leq i \leq r$ and $1 \leq j \leq s$.

- **Step 3.** It consists in combining the different values obtained from \mathcal{S} to recover the desired multi-exponentiations. In particular, an exponentiation to the power χ is involved. The protocol to be efficient, needs χ not too large.

Simple attack. Suppose that a delegation of a single exponentiation u^a , for u and a secret, is performed using Wang et al.’s protocol. If a is a secret key, an element of the form h^a is likely to be known at one point to the adversary, together with h (one can think of a public key in a scenario of delegation of BLS signatures [6], for instance). In this case, as the attacker sees an element of the form $c = a - b\chi$ (see equation 1) and knows b (cf. Step 2), he can compute h^c which is equal to $h^a \cdot (h^\chi)^{-b}$, so that recovering χ can be done by computing the discrete logarithm of $(h^a/h^c)^{b^{-1}}$ in base h . Using a baby-step giant-step algorithm, this can be done in $2^{\Upsilon/2}$ operations, which contradicts [38, Theorem 1].

Main attack. The crucial weakness of this protocol is the use of this *small* element χ which hides the exponents. The authors suggest to take it of bit-size Υ , for $\Upsilon = 64$. We will show that it cannot be that small since it can be recovered in polynomial time if two exponentiations with the *same* exponent are outsourced to the server \mathcal{S} . The scenario of our attack is the following: two exponentiations of the form $\text{GExp}((a_{1,1}, \dots, a_{1,s}); (u_{1,1}, \dots, u_{1,s}))$ and $\text{GExp}((a_{1,1}, \dots, a_{1,s}); (u'_{1,1}, \dots, u'_{1,s}))$ are queried to \mathcal{S} . The

⁴Note that the protocol from [38] can also be described without inversion in the group \mathbb{G} but to help the reader familiar with this paper, we use the same description.

exponentiations are computed with the *same* exponents. This is typically the case in the first application proposed in [38, Section 4.1] to securely offload Shacham and Waters’s proofs of retrievability [33].

For the sake of clarity, it is sufficient to focus on the elements that masks the first exponent $a_{1,1}$. An attacker will obtain (see Step 2) $b_{1,1}$, $b'_{1,1}$, $c_{1,1}$ and $c'_{1,1}$ such that $c_{1,1} = a_{1,1} - b_{1,1}\chi \pmod p$ and $c'_{1,1} = a_{1,1} - b'_{1,1}\chi' \pmod p$. Subtracting these two equations gives a modular bi-variate polynomial equation:

$$b_{1,1}X - b'_{1,1}Y + c_{1,1} - c'_{1,1} = 0 \pmod p \quad (5)$$

which has χ and χ' as roots, satisfying $\chi \leq X$ and $\chi' \leq Y$, for some X and Y which will be larger than 2^Υ , say 2^{64} . In the following, we show that it is (heuristically) possible to recover in polynomial time any χ and χ' that are lower than \sqrt{p} .

Solving this bi-variate polynomial equation with small modular roots can be done using the well-known Coppersmith technique [14] (see also Section 3.3). Finding small roots of modular bi-variate polynomials was studied in [24], but his method is very general, whereas we consider here only simple linear polynomials. The following lemma, inspired by Howgrave-Graham’s lemma [22] suggests how to construct a particular lattice that will help to recover small modular roots of a linear polynomial in $\mathbb{Z}[x, y]$. We denote as $\|\cdot\|$ the Euclidean norm of polynomials.

Lemma 2. *Let $g(x, y) \in \mathbb{Z}[x, y]$ be a linear polynomial that satisfies*

- $g(x_0, y_0) = 0 \pmod p$ for some $|x_0| < X$ and $|y_0| < Y$,
- $\|g(xX, yY)\| < p/\sqrt{3}$.

Then $g(x_0, y_0) = 0$ holds over the integer.

Let us write a bi-variate linear polynomial as $P(x, y) = x + by + c$, with $b, c \in \mathbb{Z}_p$, which has a root (x_0, y_0) modulo p satisfying $|x_0| < X$ and $|y_0| < Y$. It suffices to divide by $b_{1,1}$ the polynomial from Equation (5) to make it unary in the first variable. Lemma 2 suggests to find a small-norm polynomial $h(x, y)$ that shares its root with the initial polynomial $P(x, y)$. To do so, we construct the matrix whose rows are formed by the coefficients of the polynomials p , pyY and $P(xX, yY)$ in the basis $(1, X, Y)$. Using the LLL algorithm [25], we can find a small linear combination of these polynomials that will satisfy Lemma 2. Indeed, this matrix has determinant p^2XY and an LLL reduction of the basis of the lattice spanned by the rows of M will output one vector of norm upper bounded by $2^{3/4}(\det(M))^{1/3}$. We expect the second vector to behave as the first, which is confirmed experimentally.

To obtain two polynomials which satisfy Lemma 2, we need $2^{3/4}(\det(M))^{1/3} < p/\sqrt{3}$, i.e. $XY < 3^{-3/2} \cdot 2^{-9/4}p$. If $g(x, y) = g_0 + g_1x + g_2y$ and $h(x, y) = h_0 + h_1x + h_2y$ are the polynomials corresponding to the shortest vectors output by LLL, we can recover (x_0, y_0) as

$$x_0 = \frac{X(h_0g_2 - g_0h_2)}{g_1h_2 - h_1g_2} \text{ and } y_0 = \frac{Y(h_0g_1 - h_1g_0)}{g_2h_1 - h_2g_1}.$$

As a consequence, this method makes it possible to recover in polynomial time any values χ and χ' that masks the secret value $a_{1,1}$ if they are both below \sqrt{p} . The complexity of Nguyen and Stehlé’s LLL is quadratic [30], in our case it is $O(d^5 \log(3/2 \log(p))^2)$, with $d = 3$. Then $a_{1,1}$ can be computed as $a_{1,1} = c_{1,1} + b_{1,1}\chi \pmod p$ (see Appendix A for a practical example of this attack). The scheme from [38] is therefore completely insecure.

Remark 2. *One could fix this issue in Wang et al.’s protocol by using a larger Υ (such that the value χ is actually uniformly distributed over \mathbb{Z}_p). This would make the protocol not more efficient for the delegator than the actual computation of a single exponentiation. However, even this inefficient protocol would not achieve the privacy security notion as explained in Appendix C.*

5 Generic Constructions for Privately Outsourcing Exponentiation

We focus in this section on protocols for outsourcing a single exponentiation $(u, a) \mapsto u^a$. Protocols for outsourcing multi-exponentiations are given in Appendix C. As mentioned in the introduction, round complexity is the main bottleneck in improving the efficiency of secure protocols due to latency, and we consider only 1-round delegation protocols.

We recall that each case is referred to as its computation code β (see Section 2). All these protocols are secure (in the (indistinguishability) privacy notion defined in Section 2 in the information-theoretic sense (see Theorem 2). Optimality results (in terms of computation in the group \mathbb{G}) are given in Section 6, and summed up in Table 2.

Table 2: Outsourcing protocols for single exponentiation

Constructions for Outsourcing Fixed Base Exponentiation (with a pseudo-random power generator of pairs (k, u^k) available)									
Code	u	a	u^a	Secure protocol [†]	Complexity		Complexity Lower Bound (for \mathcal{G})		Optimality
					Resources	Lower Bound	Proof	Optimality	
111f	Public	Public	Public	Trivial					
110f	Public	Public	Secret	Non-sense					
101f	Public	Secret	Public	Protocol 2	$1\mathcal{S} + 1\mathcal{B}$	$0\mathcal{G}$	From Protocol 2		✓
100f	Public	Secret	Secret	Protocol 3	$1\mathcal{S} + 1\mathcal{G} + 1\mathcal{B}$	$1\mathcal{G}$	From Protocol 3		✓
011f	Secret	Public	Public	Non-sense*					
010f	Secret	Public	Secret	Protocol 1	$1\mathcal{S} + 1\mathcal{G} + 2\mathcal{B}$	$1\mathcal{G}$	From Protocol 1		✓
001f	Secret	Secret	Public	Protocol 2	$1\mathcal{S} + 1\mathcal{B}$	$0\mathcal{G}$	From Protocol 2		✓
000f	Secret	Secret	Secret	Protocol 1	$1\mathcal{S} + 1\mathcal{G} + 2\mathcal{B}$	$1\mathcal{G}$	From Case 010f		✓
Constructions for Outsourcing Variable Base Exponentiation (with a pseudo-random power generator of pairs (k, g^k) available)									
Code	u	a	u^a	Secure protocol [‡]	Complexity		Complexity Lower Bound (for \mathcal{G})		Optimality
					Resources	Lower Bound	Proof	Optimality	
111v	Public	Public	Public	Trivial					
110v	Public	Public	Secret	Non-sense					
101v	Public	Secret	Public	Protocol 4	$1\mathcal{S} + L_p/2\mathcal{G}$	$L_p/2\mathcal{G}$	Th. 3, Sec. 6		✓
				Protocol 5	$s\mathcal{S} + L_p/(s+1)\mathcal{G}$	$L_p/(s+1)\mathcal{G}$	Th. 4, Sec. 6		✓
100v	Public	Secret	Secret	Protocol 4	$1\mathcal{S} + L_p/2\mathcal{G}$	$L_p/2\mathcal{G}$	Th. 3, Sec. 6		✓
				Protocol 5	$s\mathcal{S} + L_p/(s+1)\mathcal{G}$	$L_p/(s+1)\mathcal{G}$	Th. 4, Sec. 6		✓
011v	Secret	Public	Public	Non-sense*					
010v	Secret	Public	Secret	Protocol 8	$1\mathcal{S} + L_p/\ell\mathcal{G} + \ell\mathcal{B}$	$L_p/(\ell+3)\mathcal{G}$	Th. 5, Sec. 6		?
				Protocol 7	$2\mathcal{S} + 3\mathcal{G} + 3\mathcal{B}$	$t\mathcal{G}^\dagger$	From Protocol 7		?
001v	Secret	Secret	Public	?	$L_p/2\mathcal{G} + 2\mathcal{B}$	$L_p/2\mathcal{G}$	From Case 101v		✗
				6 (using 4)	$L_p/s\mathcal{G} + 2\mathcal{B}$	$L_p/3\mathcal{G}$	From Case 101v		?
				6 (using 5)	$L_p/s\mathcal{G} + 2\mathcal{B}$	$L_p/(s+1)\mathcal{G}$	From Case 101v		?
000v	Secret	Secret	Secret	?	$L_p/2\mathcal{G} + 2\mathcal{B}$	$L_p/2\mathcal{G}$	From Case 100v		✗
				6 (using 4)	$L_p/s\mathcal{G} + 2\mathcal{B}$	$L_p/3\mathcal{G}$	From Case 100v		?
				6 (using 5)	$L_p/s\mathcal{G} + 2\mathcal{B}$	$L_p/(s+1)\mathcal{G}$	From Case 100v		?

Notations: $\ell = O(1)$ and $L_p = \log(p)$.

[†] refers to our numbering of the protocols

* Prime order setting.

[‡] With $t \in \{0, 1, 2, 3\}$.

_____ **Protocol 1:** 000*f* (and 010*f*) _____

Input: $u \in \mathbb{G}, a \in \mathbb{Z}_p$
Output: $u^a \in \mathbb{G}$ \triangleright private result
 $(u^r, r) \leftarrow \mathcal{B}(\cdot)$
 $(u^s, s) \leftarrow \mathcal{B}(\cdot)$
 $t \leftarrow (a - s)/r \bmod p$
 $h \leftarrow \mathcal{S}(u^r, t \bmod p)$
return $h \cdot u^s$

_____ **Protocol 2:** 001*f* (and 101*f*) _____

Input: $u \in \mathbb{G}, a \in \mathbb{Z}_p$
Output: $u^a \in \mathbb{G}$ \triangleright public result
 $(u^k, k) \leftarrow \mathcal{B}(\cdot)$
 $h \leftarrow \mathcal{S}(u^k, a/k \bmod p)$
return h

_____ **Protocol 3:** 100*f* _____

Input: $u \in \mathbb{G}, a \in \mathbb{Z}_p$
Output: $u^a \in \mathbb{G}$ \triangleright private result
 $(u^k, k) \leftarrow \mathcal{B}(\cdot)$
 $h \leftarrow \mathcal{S}(u, a - k \bmod p)$
return $h \cdot g^k$

_____ **Protocol 4:** 100*v* (and 101*v*) _____

Input: $u \in \mathbb{G}, a \in \mathbb{Z}_p$
Output: $u^a \in \mathbb{G}$ \triangleright private result
 $T \leftarrow \lceil \sqrt{p} \rceil$
 $h \leftarrow \mathcal{S}(u, T)$
 $a_0 = a \bmod T$
 $a_1 = a \operatorname{div} T$ \triangleright Euclidean division s.t.
 $a = a_1 \cdot T + a_0$
return $u^{a_0} h^{a_1}$ \triangleright using Algorithm 1

_____ **Protocol 5:** 100*v* (and 101*v*) _____

Input: $u \in \mathbb{G}, a \in \mathbb{Z}_p$
Output: $u^a \in \mathbb{G}$ \triangleright private result
 $T \leftarrow \lceil p^{1/s+1} \rceil$
for i from 1 to s **do**
 $h_i \leftarrow \mathcal{S}(u, T^i)$
end for
 $\text{temp} \leftarrow a$
for i from s down to 0 **do**
 $a_i = \text{temp} \operatorname{div} T^i$
 $\text{temp} = \text{temp} - a_i \cdot T^i$
end for
 $\triangleright a = a_s \cdot T^s + \dots + a_1 T + a_0$
return $u^{a_0} \prod_{i=1}^s h_i^{a_i}$ \triangleright using Algo. 1

_____ **Protocol 6:** 000*v* (and 001*v*) _____

Input: $u \in \mathbb{G}, a \in \mathbb{Z}_p$
Output: $u^a \in \mathbb{G}$ \triangleright private result
 $(g^{k_1}, k_1) \leftarrow \mathcal{B}(\cdot)$
 $v \leftarrow u \cdot g^{k_1}$
 $h_1 \leftarrow v^a$ \triangleright delegated using Prot. 4 or 5
(public base): $h_1 = v^a = u^a \cdot g^{ak_1}$
 $(g^{k_2}, k_2) \leftarrow \mathcal{B}(\cdot)$
 $h_2 \leftarrow \mathcal{S}(g, -ak_1 - k_2 \bmod p)$
 $\triangleright h_2 = g^{-ak_1 - k_2}$
return $h_1 \cdot h_2 \cdot g^{k_2}$

_____ **Protocol 7:** 010*v* _____

Input: $u \in \mathbb{G}, a \in \mathbb{Z}_p$
Output: $u^a \in \mathbb{G}$ \triangleright private result
 $(g^r, r) \leftarrow \mathcal{B}(\cdot)$
 $(g^s, s) \leftarrow \mathcal{B}(\cdot)$
 $(g^t, t) \leftarrow \mathcal{B}(\cdot)$
 $k \leftarrow (t - ra)/s \bmod p$
 $h_1 \leftarrow \mathcal{S}(u \cdot g^r, a)$
 $h_2 \leftarrow \mathcal{S}(g^s, k)$
return $h_1 h_2 g^t$

_____ **Protocol 8:** 010*v* _____

Input: $u \in \mathbb{G}, a \in \mathbb{Z}_p$
Output: $u^a \in \mathbb{G}$ \triangleright private result
 $(g^r, r) \leftarrow \mathcal{B}(\cdot)$
for i from 1 to s **do**
 $(g^{t_i}, t_i) \leftarrow \mathcal{B}(\cdot)$
end for
 $(k_0, k_1, \dots, k_s) \leftarrow$
GLV-Dec($1, t_1, \dots, t_s, -ra \bmod p$) \triangleright with $k_i \leq p^{1/(s+1)}$
 $h_1 \leftarrow \mathcal{S}(u \cdot g^r, a)$
 $h_2 \leftarrow g^{k_0} (g^{t_1})^{k_1} \dots (g^{t_s})^{k_s}$ \triangleright using Algo. 1
return $h_1 h_2$

_____ **Protocol 9:** 010*v* from [10] _____

Input: $u \in \mathbb{G}, a \in \mathbb{Z}_p$
Output: $u^a \in \mathbb{G}$ \triangleright private result
for i from 1 to s **do**
 $g_i \xleftarrow{R} \mathbb{G}$
end for
 $\mathcal{I} \xleftarrow{R} \mathfrak{P}_m(\{1, \dots, s\})$ \triangleright random subset
of cardinal m of $\{1, \dots, s\}$
 $g_{s+1} \leftarrow u \cdot \prod_{i \in \mathcal{I}} g_i$
for i from 1 to s **do**
 $h_i \leftarrow \mathcal{S}(g_i, -a)$
end for
 $h_{s+1} \leftarrow \mathcal{S}(g_{s+1}, a)$
return $h_{s+1} \cdot \prod_{i \in \mathcal{I}} h_i$

5.1 Constructions for Outsourcing Fixed Base Exponentiation

When the base u is fixed, one can assume that \mathcal{C} can use a pseudo-random power generator for u . As described in Section 3.4, this generator \mathcal{B} is invoked with no input and outputs a single (pseudo)-random pair $(u^k, k) \in \mathbb{G} \times \mathbb{Z}_p$ where k is uniformly distributed in \mathbb{Z}_p (or statistically close to the uniform distribution). If the generator $\mathcal{B}(\cdot)$ is invoked several times, we assume that the output pairs are independent.

Trivial Cases. Obviously, the case 111*f* (everything public) is trivial (simply ask in clear to the delegatee \mathcal{S} the computation of u^a as $\mathcal{S}(u, a)$) and the case 110*f* does not make sense (public inputs and private output), as well as the case 011*f* (secret base) in the prime order setting (but the latter case may have some interest

for composite order groups).

Cases where the Base is Secret (0f).** If everything is secret (case 000f), it is rather easy to delegate the computation of u^a for any exponent a using Protocol 1. The delegator computation amounts to two invocations of the pseudo-random power generator, one inversion modulo p and one multiplication in \mathbb{G} , with only one exponentiation delegated to \mathcal{S} .

Even if the exponent is public (case 010f), Protocol 1 remains the best possible in terms of multiplications in \mathbb{G} (with only one invocation to \mathcal{S}) since there is only one multiplication and it is needed to hide the private result of the exponentiation.

If the result is public (case 001f), one can propose the improved Protocol 2, which needs only one invocation of the pseudo-random power generator, one inversion modulo p and no multiplication in \mathbb{G} , with only one exponentiation delegated to \mathcal{S} .

Cases where the Base is Public (1f).** If the result is public (case 101f), Protocol 2 remains the best possible in terms of multiplications in \mathbb{G} (with only one invocation to \mathcal{S}) since no multiplication is needed.

If the result is secret (case 100f), Protocol 3 is the best possible in terms of multiplications in \mathbb{G} since it only needs one invocation of the pseudo-random power generator and one multiplication in \mathbb{G} (needed to hide the private result of the exponentiation), with only one exponentiation delegated to \mathcal{S} .

5.2 Constructions for Outsourcing Variable Base Exponentiation

In this paragraph, we consider the case when \mathcal{C} wants to delegate the computation of u^a but with a variable u . In this setting, one cannot assume that \mathcal{C} can use a pseudo-random power generator for u but we can still suppose that it can use a pseudo-random power generator for a fixed generator g that we still call \mathcal{B} with the same properties as before.

Trivial Cases. As above, the case 111v (everything public) is trivial (simply ask in clear to the delegatee \mathcal{S} the computation of u^a as $\mathcal{S}(u, a)$) and the case 110v does not make sense (public inputs and private output), as well as the case 011v (secret base) in the prime order setting.

Cases where the Base is Public (1v).** We first consider the case where the variable base u can be made public but not the public exponent nor the result (case 100v). We propose a family of protocols depending on a parameter s that perform the computation of u^a by delegating s exponentiations to a delegator and $\log(p)/(s+1)$ operations in \mathbb{G} . This family of protocols are given in Protocol 5 and the specific case $s=1$ is Protocol 4. Note that these protocols do not make use of the pseudo-random power generator for g . Unfortunately, the efficiency gain is only a factor s and if the number of delegated exponentiations is constant the delegator still has to perform $O(\log p)$ operations in \mathbb{G} . These protocols are actually optimal in terms of operations in \mathbb{G} , as shown in Theorems 3 and 4.

Obviously, we can also use these protocols if we allow the result u^a to be public (case 101v) and the optimal result of Theorems 3 and 4 show that even in this easier setting, the protocol cannot be improved.

Cases where the Base is Private (0v).** We can use this protocol family to construct another delegation protocol for the corresponding cases where the base is kept secret (000v and 001v). We obtain Protocol 6 that makes two invocations of the pseudo-random generator for g and requires the delegation of one further exponentiation compared to Protocol 5 (and Protocol 4). We do not actually know if these protocols are optimal but the gap is rather tight (see Table 2). Constructing an outsourcing protocol in these cases with only one exponentiation delegation (or proving it is impossible) is left as an open problem.

We can also use this protocol if we allow the exponent a to be public (010v). However, in this case one can improve it with Protocol 7 where the delegator performs only a constant number of group operations in \mathbb{G} . In this case, one can also improve it with Protocol 8 where the delegator makes only one call to the delegatee, but at the price of a $O(\log(p))$ number of group operations in \mathbb{G} .

Remark 3. In [10], Cavallo et al. presented two other protocols for outsourcing private variable base and public exponent exponentiation. The first one [10, §4, p. 164], recalled in Protocol 9, achieves only the basic security requirement (i.e., in the sense of one-wayness instead of indistinguishability). It relies on a subset-sum in a group and in order to achieve a stronger privacy notion, the delegation scheme actually becomes less efficient for the delegator than performing the exponentiation on its own. The second scheme is much more efficient since the delegator computation is constant but it requires a stronger pseudo-random powers generator \mathcal{B} that outputs pseudo-random triples of the form (g^r, g^{ar}, r) . In particular, this second protocol can only be used for fixed values of the public exponent a .

Theorem 2. Let GroupGen be a group generator, let λ be a security parameter and let \mathbb{G} be a group of prime order p output by $\text{GroupGen}(\lambda)$. Let $(\mathcal{C}, \mathcal{S})$ be one client-server protocol for the delegated computation of the exponentiation u^a described in Protocols 1 – 8 (for the corresponding computation code $\beta \in \{0, 1\}^4$ given in their description). The protocol $(\mathcal{C}, \mathcal{S})$ is unconditionally $(\tau, 0)$ -private against an honest-but-curious adversary for any time τ .

Proof. Since the protocols (and thus the proofs) are all very similar, we focus on Protocol 1. The correctness follows from the equality

$$h \cdot u^s = [(u^r)^{t \bmod p}]u^s = [(u^r)^{(a-s)/r \bmod p}]u^s = u^a.$$

We now prove that there is no adversary \mathcal{A} (running in any time τ) for the privacy security notion from Definition 3. The adversary chooses a group element u and two scalars $(a_0, a_1) \in \mathbb{Z}_p^2$. The challenger picks uniformly at random a bit $b \in \{0, 1\}$ and sets $a = a_b$. The delegator runs the delegation protocol with inputs u and a and delegates one exponentiation to the adversary acting as the delegatee. The adversary has to guess the bit b .

Due to the properties of the pseudo-random power generator, r and s are uniformly distributed in \mathbb{Z}_p , so that t is also uniformly distributed in \mathbb{Z}_p and does not depend on the value a . The invocation $\mathcal{S}(u^r, t \bmod p)$ thus does not reveal anything on the value a (in an information-theoretic sense), meaning that the advantage of the adversary in guessing the bit b is 0. \square

Remark 4. *Theorem 2 asserts that our protocols achieves unconditionally the privacy experiment described in Fig. 2 (i.e., in the information theoretic sense).*

Remark 5. *In this paper, we do not consider the setting where the delegator can also store precomputed values (in addition to having access to a pseudo-random power generator \mathcal{B} and a delegatee oracle \mathcal{S}). For instance, in Protocol 8, we use the pseudo-random power generator s times in order to generate pairs (g^{t_i}, t_i) for $i \in \{1, \dots, s\}$ and then the GLV decomposition algorithm (GLV-Dec) in order to decompose the scalar $-ra \bmod p$ as $-ra = k_0 + k_1 t_1 + \dots + k_s t_s \bmod p$ with “small” scalars $k_i \leq p^{1/(s+1)}$. Actually, the pairs (g^{t_i}, t_i) for $i \in \{1, \dots, s\}$ can be re-used and they do not need to be pseudo-random to ensure privacy. We can thus consider a simpler variant of Protocol 8 in which the delegator stores precomputed values g^{t_i} with $t_i = T^i$ for $T = \lceil p^{1/(s+1)} \rceil$ for $i \in \{0, \dots, s\}$ and decomposes the scalar $-ra \bmod p$ in base T as $-ra = k_0 t_0 + k_1 t_1 + \dots + k_s t_s \bmod p$ with “small” scalars $k_i < T$ as in Protocol 5. The resulting protocol is then simpler and more efficient in practice. However, it has the same “oracle complexity” since it only replaces queries to the pseudo-random power generator \mathcal{B} by storage of precomputed values. It can be easily seen that the complexity lower bounds from Section 6 can be generalized to this setting.*

6 Complexity Lower Bound for One-Round Protocols

We focus on studying protocols with minimal interaction, namely the delegator is allowed to delegate the computation of several group exponentiations but it must send all of them to the delegatee in only one communication round. Indeed, interactions over computer networks are usually the most time consuming operations (due to lagging or network congestion) and it is very important to study protocols which require the minimal number of rounds to complete. In Appendix D, we present complexity lower bounds for multi-round protocols.

By “lower bounds”, we mean that the number of calls to the delegatee oracle \mathcal{S} and to the pseudo-random power generator \mathcal{B} are fixed, and that we consider the number of group operations. All the results concerning this section are summed up in the column “Complexity Lower Bound” of Table 2. The last column of Table 2 gives a hint for the proof of those lower bounds. Concerning the first part of the table, the bounds come from the protocols given in Section 5, since at least one call to the group oracle is mandatory when the result is private (the delegator \mathcal{C} needs to do at least one computation after having received a public result from the delegatee oracle \mathcal{S}). The cases 101v and 100v are then dealt with in Theorem 4. For all these cases, the protocols proposed in Section 5 are thus actually optimal. As for Case 010v, the lower bound for a unique call to \mathcal{S} is proven in Theorem 5, whereas Protocol 7 gives a (constant) upper bound in case we allow a second call to \mathcal{S} . Finally, the lower bounds for Cases 001v and 000v come from the equivalent bounds for Cases 101v and 100v, since the variable base is furthermore assumed to be secret.

In what follows, and as mentioned above, we use the generic group model to prove these lower bounds. We model the different operations as follows:

- The group oracle \mathcal{G} takes as inputs two encodings $\sigma_1 = \sigma(h_1)$ and $\sigma_2 = \sigma(h_2)$ and outputs the encoding σ_3 such $\sigma_3 = \sigma(h_1 h_2)$ (see Section 3.1).
- The pseudo-random power generator \mathcal{B} outputs pairs $(t, \sigma(g^t))$ where the scalar t is picked uniformly at random in \mathbb{Z}_p (independently for all queries).
- The delegatee oracle \mathcal{S} takes as inputs an encoding $\sigma = \sigma(h)$ and a scalar x and outputs the encoding $\sigma' = \sigma(h^x)$ (i.e. $\sigma^{-1}(\sigma') = \sigma^{-1}(\sigma)^x$).

The following theorems assert that for the cases $101v$ and $100v$, the protocols proposed in Section 5 are actually optimal in terms of calls to \mathcal{S} and \mathcal{G} .

For the ease of exposition, we first state our result and present a proof for the simple case where the delegator \mathcal{C} outsources only one exponentiation to the delegatee \mathcal{S} :

Theorem 3. *Let GroupGen be a group generator and let $(\mathcal{C}, \mathcal{S})$ be one client-server protocol for the delegated computation of the exponentiation u^a for the corresponding computation code $\beta = 101v$. We assume that the delegator \mathcal{C} is a generic group algorithm that uses*

- $c \log(p) + O(1)$ generic group operations (for all groups \mathbb{G} of primer order p output by $\text{GroupGen}(\lambda)$ for some constant c ,
- $\ell = O(1)$ queries to the (private) pseudo-random power generator \mathcal{B}
- and only 1 delegated exponentiation to the delegatee \mathcal{S}

If $c < 1/2$, then $(\mathcal{C}, \mathcal{S})$ is not private: there exists an algorithm running in polynomial-time such that

$$\Pr[\text{bit} \leftarrow \mathbf{Exp}_{\text{priv}}(\mathcal{A}) : \text{bit} = 1] \geq 1 - \lambda^{O(1)}.$$

Proof. We assume that \mathcal{C} gets as input two encodings $\sigma(u)$, $\sigma(g)$ of two group elements u and g and one scalar a in \mathbb{Z}_p and outputs the encoding $\sigma(u^a)$ of the group element u^a by making q queries to the group oracle \mathcal{G} , ℓ queries to the (private) pseudo-random power generator \mathcal{B} and 1 query to \mathcal{S} .

We assume that $q = c \log p + O(1)$ with $c < 1/2$ and we prove that it is not possible for \mathcal{C} to compute $\sigma(u^a)$ in such a way that the delegatee \mathcal{S} learns no information on a . More precisely, we construct a polynomial-time adversary \mathcal{A} for the privacy security notion from Definition 3 (page 5). The adversary chooses a group element u and two scalars $(a_0, a_1) \in \mathbb{Z}_p^2$. For the sake of simplicity, we assume that the adversary picks $(a_0, a_1) \in \mathbb{Z}_p^2$ uniformly at random among the scalars of bit-length $\log(p)$ and u uniformly at random in \mathbb{G} . The challenger picks uniformly at random a bit $b \in \{0, 1\}$ and sets $a = a_b$. The delegator runs the delegation protocol with inputs u and a and delegates one exponentiation to the adversary acting as the delegatee. The adversary has to guess the bit b .

Let us denote $(t_1, \sigma(g^{t_1}))$, $(t_2, \sigma(g^{t_2}))$, \dots , $(t_\ell, \sigma(g^{t_\ell}))$ the pairs obtained from the pseudo-random power generator \mathcal{B} by the delegator \mathcal{C} . Since \mathcal{B} takes no inputs and outputs independent pairs, we can assume without loss of generality that the delegator \mathcal{C} makes the ℓ queries to \mathcal{B} in a first phase of the delegation protocol. We denote $(\sigma(h), x)$ the unique pair encoding of group element/scalar made by \mathcal{C} to the delegatee \mathcal{S} (which is executed by the adversary \mathcal{A} in an “honest-but-curious” way). Using generic group operations, \mathcal{C} can only construct the corresponding group elements such that:

$$h = u^{\alpha'} \cdot g^{\kappa'} \cdot g^{t_1 \gamma'_1} \dots g^{t_\ell \gamma'_\ell} \quad (6)$$

for some scalars $(\alpha', \kappa', \gamma'_1, \dots, \gamma'_\ell)$. We denote $k = h^x$ the response of \mathcal{S} . Eventually, the delegator \mathcal{C} outputs the encoding $\sigma(u^a)$ of the group element u^a . Again, using generic group operations, it can only construct it as

$$u^a = u^\alpha g^\kappa \cdot g^{t_1 \gamma_1} \dots g^{t_\ell \gamma_\ell} k^\delta h^\varepsilon \quad (7)$$

for some scalars $(\alpha, \kappa, \gamma_1, \dots, \gamma_\ell, \delta, \varepsilon)$. If we assume that $q = c \log n + O(1)$ (and in particular $q = o(\sqrt{p})$), the delegator \mathcal{C} is not able to compute the discrete logarithm of u in base g . This means that necessarily the exponent of g in Equation (7) cancels out. Recall that $k = h^x$, h being constructed as in Equation (6). Thus, taking only the discrete logarithms of powers of u in base u of this equation, we obtain

$$a = \alpha + \varepsilon \alpha' + \delta \alpha' x \pmod{p} \quad (8)$$

We denote τ_1 the number of group operations performed by \mathcal{C} in the computation of h described in Equation (6) and τ_2 the number of operations in the computation of u^a described in Equation (7). By assumption, $\tau_1 + \tau_2 \leq c \log p + O(1)$. Furthermore, since \mathcal{C} only used generic group operations, we have (by Lemma 1 in Section 3.1) $\alpha' \leq 2^{\tau_1}$, $\alpha \leq 2^{\tau_2}$, $\delta \leq 2^{\tau_2}$ and $\varepsilon \leq 2^{\tau_2}$. If we note $\rho_1 = \alpha + \varepsilon \alpha'$ and $\rho_2 = \delta \alpha'$, Equation (8) becomes $a = \rho_1 + x \rho_2 \pmod{p}$, where x is known to the adversary, $\rho_2 = \delta \alpha' \leq 2^{\tau_1} 2^{\tau_2} = 2^{\tau_1 + \tau_2} \leq p^{c+o(1)}$ and $\rho_1 = \alpha + \varepsilon \alpha' \leq 2^{\tau_1} + 2^{\tau_1} 2^{\tau_2} \leq p^{c+o(1)}$.

The adversary \mathcal{A} can then try to decompose a_0 and a_1 as $a_i = \rho_{i,1} + x \rho_{i,2} \pmod{p}$, with $\rho_{i,1}, \rho_{i,2} \leq p^{c+o(1)}$. For $a_b = a$, the decomposition algorithm from Section 3.3 (which generalizes the main attack on Wang et al.’s protocol) will recover $\rho_{b,1}$ and $\rho_{b,2}$ in polynomial time. However, for a given x and a random a_{1-b} of bit-length $\log(p)$, there is only a negligible probability that such a decomposition exists (less than $p^{c+o(1)} \times p^{c+o(1)} = p^{2c+o(1)} = o(p)$ scalars can be written in this way). Thus, the adversary can simply run the decomposition algorithm from Section 3.3 on (a_0, x) on one hand and on (a_1, x) on the other hand and returns the bit b for which the algorithm returns a “small decomposition” on input (a_b, x) . By the previous analysis, its advantage is noticeable. \square

Theorem 4 generalizes Theorem 3 and consider the general case where the delegator \mathcal{C} outsources $s \geq 1$ exponentiations to the delegatee \mathcal{S} :

Theorem 4. *Let GroupGen be a group generator and let $(\mathcal{C}, \mathcal{S})$ be one client-server protocol for the delegated computation of one exponentiation for the computation code $\beta = 101v$. We assume that the delegator \mathcal{C} is a generic group algorithm that uses*

- $c \log(p) + O(1)$ generic group operations (for groups \mathbb{G} of order p output by $\text{GroupGen}(\lambda)$),
- $\ell = O(1)$ queries to the (private) pseudo-random power generator \mathcal{B}
- and s simultaneous delegated exponentiation to the delegatee \mathcal{S}

If the constant c satisfies $c < 1/(s+1)$, then $(\mathcal{C}, \mathcal{S})$ is not private: there exists an algorithm running in polynomial-time such that $\Pr[\text{bit} \leftarrow \mathbf{Exp}_{\text{priv}}(\mathcal{A}) : \text{bit} = 1] \geq 1 - \lambda^{O(1)}$.

Proof (Theorem 4). We assume that the delegator \mathcal{C} gets as input two encodings $\sigma(u), \sigma(g)$ of two group elements u and g picked uniformly at random in \mathbb{G} and one scalar a picked uniformly at random in \mathbb{Z}_p and outputs the encoding $\sigma(u^a)$ of the group element u^a by making only

- q queries to the group oracle \mathcal{G} ;
- ℓ queries to the pseudo-random power generator \mathcal{B} ;
- s simultaneous queries to the delegatee oracle \mathcal{S} .

We assume that $q = c \log p + O(1)$ with $c < 1/(s+1)$ and we prove that it is not possible for \mathcal{C} to compute $\sigma(u^a)$ in such a way that the delegatee \mathcal{S} learns no information on a .

More precisely, we construct a polynomial-time adversary \mathcal{A} for the privacy security notion from Definition 3. The adversary chooses a group element $u \in \mathbb{G}$ and two scalars $(a_0, a_1) \in \mathbb{Z}_p^2$. As above, for the sake of simplicity, we assume that the adversary picks $(a_0, a_1) \in \mathbb{Z}_p^2$ uniformly at random among the scalars of bit-length $\log(p)$ and u uniformly at random in \mathbb{G} . The challenger picks uniformly at random a bit $b \in \{0, 1\}$ and sets $a = a_b$. The delegator runs the delegation protocol with inputs u and a and delegates s exponentiations to the adversary acting as the delegatee. The adversary has to guess the bit b .

Let us denote $(t_1, \sigma(g^{t_1})), (t_2, \sigma(g^{t_2})), \dots, (t_\ell, \sigma(g^{t_\ell}))$ the pairs obtained from the pseudo-random power generator \mathcal{B} by the delegator \mathcal{C} . Since the pseudo-random power generator \mathcal{B} takes no inputs and outputs independent pairs, we can assume without loss of generality that the delegator \mathcal{C} makes the ℓ queries to \mathcal{B} in a first phase of the delegation protocol.

We denote $(\sigma(h_1), x_1), \dots, (\sigma(h_s), x_s)$ the pairs group element/scalar made by \mathcal{C} to the delegatee \mathcal{S} (which is executed by the adversary \mathcal{A} in an “honest-but-curious” way). Using generic group operations, \mathcal{C} can only construct the corresponding group elements such that:

$$\begin{aligned} h_1 &= u^{\alpha_1} \cdot g^{\kappa_1} \cdot g^{t_1 \gamma_{1,1}} \dots g^{t_\ell \gamma_{1,\ell}} \\ h_2 &= u^{\alpha_2} \cdot g^{\kappa_2} \cdot g^{t_1 \gamma_{2,1}} \dots g^{t_\ell \gamma_{2,\ell}} \\ h_3 &= u^{\alpha_3} \cdot g^{\kappa_3} \cdot g^{t_1 \gamma_{3,1}} \dots g^{t_\ell \gamma_{3,\ell}} \\ &\vdots \\ h_s &= u^{\alpha_s} \cdot g^{\kappa_s} \cdot g^{t_1 \gamma_{s,1}} \dots g^{t_\ell \gamma_{s,\ell}} \end{aligned} \tag{9}$$

for some scalars $(\alpha_1, \dots, \alpha_s)$, $(\kappa_1, \dots, \kappa_s)$ and $(\gamma_{i,j})_{i=1, s; j=1, \ell}$. We note $k_i = h_i^{x_i}$ the response of the delegatee \mathcal{S} to the i -th query. Eventually, the delegator \mathcal{C} outputs the encoding $\sigma(u^a)$ of the group element u^a and as above using generic group operations, it can only construct it as

$$u^a = u^\alpha g^\kappa \cdot g^{t_1 \gamma_1} \dots g^{t_\ell \gamma_\ell} k_1^{\delta_1} k_2^{\delta_2} \dots k_s^{\delta_s} h_1^{\varepsilon_1} h_2^{\varepsilon_2} \dots h_s^{\varepsilon_s} \tag{10}$$

for some scalars $(\alpha, \kappa, \gamma_1, \dots, \gamma_\ell, \delta_1, \dots, \delta_s, \varepsilon_1, \dots, \varepsilon_s)$. If we assume that $q = c \log n + O(1)$ (and in particular $q = o(\sqrt{p})$), the delegator \mathcal{C} is not able to compute the discrete logarithm of u in base g . This means that necessarily the exponents of g in Equation (10) cancel out. Recall that $k_i = h_i^{x_i}$ for all index i , h_i being constructed as in Equation (9). Thus, taking only the discrete logarithms of powers of u in base u of this equation, we obtain

$$a = \alpha + \sum_{i=1}^s (\delta_i \alpha_i x_i + \varepsilon_i \alpha_i) \pmod{p}. \tag{11}$$

We denote τ_i the number of group operations performed by \mathcal{C} in the computation of h_i described in Equation (9) and τ the number of group operations performed by \mathcal{C} in the computation of u^a described in Equation (10).

By assumption, $\tau_1 + \dots + \tau_s + \tau \leq c \log p + O(1)$. Furthermore, since \mathcal{C} only used generic group operations, we have (by Lemma 1) $\alpha_i \leq 2^{\tau_i}$, $\alpha \leq 2^\tau$, $\delta_i \leq 2^\tau$ and $\varepsilon_i \leq 2^\tau$ for $i \in \{1, \dots, s\}$. If we note $\mu_0 = \alpha + \sum_{i=1}^s \varepsilon_i \alpha_i$ and $\mu_i = \delta_i \alpha_i$ for $i \in \{1, \dots, s\}$, Equation (11) becomes

$$a = \mu_0 + \mu_1 x_1 + \mu_2 x_2 + \mu_3 x_3 + \dots + \mu_s x_s \pmod{p} \quad (12)$$

where x is known to the adversary, $\mu_i = \delta_i \alpha_i \leq 2^\tau 2^{\tau_i} \leq p^{c+o(1)}$ for $i \in \{1, \dots, s\}$ and $\mu_0 = \alpha + \sum_{i=1}^s \varepsilon_i \alpha_i \leq 2^\tau + \sum_{i=1}^s 2^{\tau+\tau_i} \leq p^{c+o(1)}$.

Therefore the delegatee \mathcal{S} knows that a satisfies the equation (12), in which it knows the value x_1, \dots, x_s and we have $\mu_i = o(p^{1/s})$. The adversary can then try to decompose a_0 and a_1 as

$$a_b = \mu_{b,0} + \mu_{b,1} x_1 + \mu_{b,2} x_2 + \mu_{b,3} x_3 + \dots + \mu_{b,s} x_s \pmod{p}$$

with $\mu_{0,i}, \mu_{1,i} \leq p^{c+o(1)}$ for $i \in \{0, \dots, s\}$. For $a_b = a$, the by using the decomposition algorithm from Section 3.3 (see Theorem 1) will recover the values $\mu_{i,b}$ in polynomial time (or a potentially even shorter decomposition). Once again, for a given x and a random $a_{1-b} = a^*$ of bit-length $\log(p)$, there is only a negligible probability that such a decomposition exists (less than $(p^{c+o(1)})^{s+1} = p^{(s+1)c+o(1)} = o(p)$ scalars a can be written in this way). Thus, the adversary can simply run the decomposition algorithm from Section 3.3 on (a_0, x) on one hand and on (a_1, x) on the other hand and returns the bit b for which the algorithm returns a “small decomposition” on input (a_b, x) . By the previous analysis, its advantage is noticeable. \square

Remark 6. *It is worth mentioning that even in (generic) groups where division is significantly less expensive than multiplication (such as elliptic curves or class groups of imaginary quadratic number fields), this lower bound (as well as the following ones) still holds (see Appendix B for details).*

Algorithm 7 shows that it is possible to delegate a secret base, public exponent exponentiation with only a constant number of operations if the delegator can delegate at least two exponentiations. Theorem 5 asserts that if the delegator is only allowed to delegate one exponentiation then Algorithm 8 is almost optimal in this setting. More precisely, we show that the delegator has to perform at least $O(\log(p))$ group operations if it delegates only one exponentiation and makes at most a constant number of queries to the pseudo-random power generator \mathcal{B} .

Theorem 5. *Let GroupGen be a group generator and let $(\mathcal{C}, \mathcal{S})$ be one client-server protocol for the delegated computation of one exponentiation for the computation code $\beta = 010v$. We assume that the delegator \mathcal{C} is a generic group algorithm that uses*

- $c \log(p) + O(1)$ generic group operations (for groups \mathbb{G} of order p output by $\text{GroupGen}(\lambda)$),
- $\ell = O(1)$ queries to the (private) pseudo-random power generator \mathcal{B}
- and only 1 delegated exponentiation to the delegatee \mathcal{S}

If the constant c satisfies $c < 1/(\ell + 2)$, then $(\mathcal{C}, \mathcal{S})$ is not private: there exists an algorithm running in time $O(p^{c/2+o(1)})$ such that $\Pr[\text{bit} \leftarrow \text{Exp}_{\text{priv}}(\mathcal{A}) : \text{bit} = 1] = 1$.

Proof. We assume that the delegator \mathcal{C} gets as input two encodings $\sigma(u), \sigma(g)$ of two group elements u and g picked uniformly at random in \mathbb{G} and one scalar a picked uniformly at random in \mathbb{Z}_p and outputs the encoding $\sigma(u^a)$ of the group element u^a by making only

- q queries to the group oracle \mathcal{G} ;
- ℓ queries to the pseudo-random power generator \mathcal{B} ;
- 1 query to the delegatee oracle \mathcal{S} .

We assume that ℓ is constant (with respect to the underlying group order) and we will show that $q = \log(p)/(\ell + 2) + \Omega(1)$ which proves indeed that Algorithm 8 is almost optimal since it requires $\log(p)/\ell + o(1)$ group operations.

We assume that $q = c \log p + O(1)$ with $c < 1/(\ell + 2)$ and we prove that it is not possible for \mathcal{C} to compute $\sigma(u^a)$ in such a way that the delegatee \mathcal{S} learns no information on u .

More precisely, we construct a polynomial-time adversary \mathcal{A} for the privacy security notion from Definition 3. The adversary chooses two scalars $(u_0, u_1) \in \mathbb{G}^2$ and a scalar a . For the sake of simplicity, we assume

that the adversary picks $(u_0, u_1) \in \mathbb{G}^2$ uniformly at random and picks a among the scalars of bit-length $\log(p)$. The challenger picks uniformly at random a bit $b \in \{0, 1\}$ and sets $u = u_b$. The delegator runs the delegation protocol with inputs u and a and delegates one exponentiation to the adversary acting as the delegatee. The adversary has to guess the bit b .

Let us denote $(t_1, \sigma(g^{t_1})), (t_2, \sigma(g^{t_2})), \dots, (t_\ell, \sigma(g^{t_\ell}))$ the pairs obtained from the pseudo-random power generator \mathcal{B} by the delegator \mathcal{C} . Since the pseudo-random power generator \mathcal{B} takes no inputs and outputs independent pairs, we can assume without loss of generality that the delegator \mathcal{C} makes the ℓ queries to \mathcal{B} in a first phase of the delegation protocol.

We denote $(\sigma(h), x)$ the unique pair group element/scalar queried by \mathcal{C} to the delegatee \mathcal{S} (which is executed by the adversary \mathcal{A} in an “honest-but-curious” way). Using generic group operations, \mathcal{C} can only construct the corresponding group elements such that:

$$h = u^{\gamma_1} \cdot g^{\gamma_2} (g^{t_1})^{\theta_1} \dots (g^{t_\ell})^{\theta_\ell} \quad (13)$$

for some scalars $(\gamma_1, \gamma_2, \theta_1, \dots, \theta_\ell) \in \mathbb{Z}_p^{\ell+2}$. We denote $h = u^{\gamma_1} g^r$ (with $r = \gamma_2 + t_1\theta_1 + \dots + t_\ell\theta_\ell$) and $k = h^x$ the response of the delegatee \mathcal{S} .

Eventually, the delegator \mathcal{C} outputs⁵ the encoding $\sigma(u^a)$ of the group element u^a and as above using generic group operations, it can only construct it as

$$u^a = u^{\alpha_1} h^{\alpha_2} k^{\alpha_3} g^{\alpha_4} \cdot g^{t_1\kappa_1} \dots g^{t_\ell\kappa_\ell} \quad (14)$$

for some scalars $(\alpha_1, \alpha_2, \alpha_3, \alpha_4, \kappa_1, \dots, \kappa_\ell) \in \mathbb{Z}_p^{\ell+4}$.

If we assume that $q = c \log n + O(1)$ (and in particular $q = o(\sqrt{p})$), the delegator \mathcal{C} is not able to compute the discrete logarithm of u in base g . This means that necessarily the exponent of g in Equation (14) cancels out. Recall that $k = h^x$, h being constructed as in Equation (13). Thus, taking only the discrete logarithms of powers of u in base u of this equation, we obtain

$$a = \alpha_1 + \alpha_2\gamma_1 + \alpha_3\gamma_1x \pmod{p}. \quad (15)$$

Similarly, taking only the discrete logarithm of powers of g of this equation, we obtain

$$0 = r\alpha_2 + r\alpha_3 + \kappa_1t_1 + \dots + \kappa_\ell t_\ell \pmod{p}. \quad (16)$$

We denote τ_1 the number of group operations performed by \mathcal{C} in the computation of h described in Equation (13) and τ_2 the number of group operations performed by \mathcal{C} in the computation of u^a described in Equation (14).

By assumption, $\tau_1 + \tau_2 \leq c \log p + O(1)$. Furthermore, since \mathcal{C} only used generic group operations, we have $\gamma_1 \leq 2^{\tau_1}$, $\alpha_i \leq 2^{\tau_2}$ for $i \in \{1, 2, 4\}$. In particular $\alpha_1 + \alpha_2\gamma_1 \leq 2^{\tau_1} + 2^{\tau_1+\tau_2} \leq p^{c+o(1)}$ and Equation (15) implies that $\alpha_3\gamma_1x \not\equiv 0 \pmod{p}$.

The delegation protocol must ensure the privacy of u therefore in Equation (13), the value r such that the group element g^r masks u^{γ_1} must be different from 0. Otherwise, the adversary can simply try to find the (small) discrete logarithm of h in base u_0 or u_1 using for instance Shanks “baby steps, giant steps” or Pollard λ algorithm in time $O(\sqrt{\gamma_1}) = O(p^{c/2+o(1)})$. Combining Equations (15) and (16), we have:

$$ra = r\alpha_1 + r\alpha_2\gamma_1 - (r\alpha_2\gamma_1 + \kappa_1\gamma_1t_1 + \dots + \kappa_\ell\gamma_1t_\ell) \pmod{p}.$$

with $r \neq 0$. Therefore, since $r = \gamma_2 + t_1\theta_1 + \dots + t_\ell\theta_\ell$, the random scalar a can be written as:

$$a = \frac{r\alpha_1 - (\kappa_1\gamma_1t_1 + \dots + \kappa_\ell\gamma_1t_\ell)}{\gamma_2 + t_1\theta_1 + \dots + t_\ell\theta_\ell} \pmod{p}$$

and

$$a = \frac{\gamma_2\alpha_1 + \sum_{i=1}^{\ell} t_i(\theta_i\alpha_1 - \kappa_i\gamma_1)}{\gamma_2 + t_1\theta_1 + \dots + t_\ell\theta_\ell} \pmod{p}. \quad (17)$$

For fixed values t_1, \dots, t_ℓ , the number of scalars a that can be written in this form is upper-bounded by the product of number of $\alpha_1, \gamma_1, \gamma_2, \theta_i$'s and κ_i 's. We have, by Lemma 1

$$\alpha_1 \leq 2^{\tau_2} \quad \gamma_1 \leq 2^{\tau_1} \quad \gamma_2 \leq 2^{\tau_1} \quad \theta_i \leq 2^{\tau_1} \quad \kappa_i \leq 2^{\tau_2}$$

for $i \in \{1, \dots, \ell\}$. Therefore, the number of scalars a that can be written as in Equation (17) is upper-bounded by

$$2^{\tau_2} \times 2^{\tau_1} \times 2^{\tau_1} \times (2^{\tau_1})^\ell \times (2^{\tau_2})^\ell \leq (2^{\tau_1+\tau_2})^{\ell+2}.$$

Since $2^{\tau_1+\tau_2} \leq p^{c+o(1)}$ with $c < 1/(\ell+2)$, we have shown that all scalars $a \in \mathbb{Z}_p$ cannot be written as in Equation (17) and therefore, the delegation protocol is not correct. \square

⁵We do not assume that the adversary learns this value but only that the delegator \mathcal{C} has to output it by the correctness property.

Remark 7. *It is worth noting that in the previous proof, we use only the fact that the scalar r (used in the exponent of the masking group element g^r) is not zero. It might be possible to improve our lower bound by using the much stronger privacy notion.*

7 Conclusion and Future Work

All our results on (one-round) secure delegation of group exponentiation are collected in Table 2. In addition, we also provide protocols and lower-bounds for multi-exponentiations in Appendix C and lower bounds for multi-rounds delegation of exponentiation protocols in Appendix D. As a future work, understanding the relationship between computational efficiency and memory usage is vital when implementing delegation protocols. In particular, it is interesting to propose efficient delegation protocols and to improve our lower bounds in settings where the memory complexity of the delegator is limited.

Acknowledgments

The authors are supported in part by the French ANR JCJC ROMAnTIC project (ANR-12-JS02-0004) and by ERC Starting Grant ERC-2013-StG-335086-LATTAC. The second author thanks Guillaume Hanrot and Damien Stehlé for helpful discussions.

References

- [1] Martin Albrecht, Shi Bai, David Cadé, Xavier Pujol, and Damien Stehlé. fpLLL-4.0, a floating-point LLL implementation. Available at <http://perso.ens-lyon.fr/damien.stehle>.
- [2] Giuseppe Ateniese, Randal C. Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary N. J. Peterson, and Dawn Xiaodong Song. Provable data possession at untrusted stores. In Ning et al. [31], pages 598–609.
- [3] Roberto Maria Avanzi. The complexity of certain multi-exponentiation techniques in cryptography. *Journal of Cryptology*, 18(4):357–373, September 2005.
- [4] László Babai. On Lovász’ lattice reduction and the nearest lattice point problem. *Combinatorica*, 6(1):1–13, 1986.
- [5] Razvan Barbulescu, Pierrick Gaudry, Antoine Joux, and Emmanuel Thomé. A heuristic quasi-polynomial algorithm for discrete logarithm in finite fields of small characteristic. In *EUROCRYPT 2014*, LNCS 8441, pages 1–16. Springer, May 2014.
- [6] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. In *ASIACRYPT 2001*, LNCS 2248, pages 514–532. Springer, December 2001.
- [7] Victor Boyko, Marcus Peinado, and Ramarathnam Venkatesan. Speeding up discrete log and factoring based schemes via precomputations. In *EUROCRYPT’98*, LNCS 1403, pages 221–235. Springer, May / June 1998.
- [8] Ernest F. Brickell, Daniel M. Gordon, Kevin S. McCurley, and David Bruce Wilson. Fast exponentiation with precomputation (extended abstract). In *EUROCRYPT’92*, LNCS 658, pages 200–207. Springer, May 1993.
- [9] Sébastien Canard, Julien Devigne, and Olivier Sanders. Delegating a pairing can be both secure and efficient. In *ACNS 14*, LNCS 8479, pages 549–565. Springer, June 2014.
- [10] Bren Cavallo, Giovanni Di Crescenzo, Delaram Kahrobaei, and Vladimir Shpilrain. Efficient and secure delegation of group exponentiation to a single server. In *Radio Frequency Identification. Security and Privacy Issues - 11th International Workshop, RFIDsec 2015, New York, NY, USA, June 23-24, 2015, Revised Selected Papers*, LNCS 9440, pages 156–173. Springer, 2015.
- [11] Xiaofeng Chen, Jin Li, Jianfeng Ma, Qiang Tang, and Wenjing Lou. New algorithms for secure outsourcing of modular exponentiations. In *ESORICS 2012*, LNCS 7459, pages 541–556. Springer, September 2012.

- [12] Benoît Chevallier-Mames, Jean-Sébastien Coron, Noel McCullagh, David Naccache, and Michael Scott. Secure delegation of elliptic-curve pairing. In *Smart Card Research and Advanced Application, 9th IFIP WG 8.8/11.2 International Conference, CARDIS 2010, Passau, Germany, April 14-16, 2010. Proceedings*, LNCS 6035, pages 24–35. Springer, 2010.
- [13] Don Coppersmith. Finding a small root of a bivariate integer equation; factoring with high bits known. In *EUROCRYPT'96*, LNCS 1070, pages 178–189. Springer, May 1996.
- [14] Don Coppersmith. Finding a small root of a univariate modular equation. In *EUROCRYPT'96*, LNCS 1070, pages 155–165. Springer, May 1996.
- [15] Peter de Rooij. Efficient exponentiation using precomputation and vector addition chains. In *EUROCRYPT'94*, LNCS 950, pages 389–399. Springer, May 1995.
- [16] Peter de Rooij. On Schnorr’s preprocessing for digital signature schemes. *Journal of Cryptology*, 10(1):1–16, 1997.
- [17] The Sage Developers. *Sage Mathematics Software (Version 5.3)*, 2012. <http://www.sagemath.org>.
- [18] Steven D. Galbraith. *Mathematics of public key cryptography*. Cambridge University Press, 2012.
- [19] Robert P. Gallant, Robert J. Lambert, and Scott A. Vanstone. Faster point multiplication on elliptic curves with efficient endomorphisms. In *CRYPTO 2001*, LNCS 2139, pages 190–200. Springer, August 2001.
- [20] Mathias Herrmann. *Lattice-based Cryptanalysis using Unravelling Linearization*. PhD thesis, Ruhr-Universität Bochum, 2011.
- [21] Susan Hohenberger and Anna Lysyanskaya. How to securely outsource cryptographic computations. In *TCC 2005*, LNCS 3378, pages 264–282. Springer, February 2005.
- [22] Nick Howgrave-Graham. Finding small roots of univariate modular equations revisited. In *6th IMA International Conference on Cryptography and Coding*, LNCS 1355, pages 131–142. Springer, December 1997.
- [23] Ari Juels and Burton S. Kaliski Jr. Pors: proofs of retrievability for large files. In Ning et al. [31], pages 584–597.
- [24] Charanjit S. Jutla. On finding small solutions of modular multivariate polynomial equations. In *EUROCRYPT'98*, LNCS 1403, pages 158–170. Springer, May / June 1998.
- [25] Arjen K. Lenstra, Hendrik W. Jr. Lenstra, and László Lovász. Factoring polynomials with rational coefficients. *Math. Ann.*, 261:515–534, 1982.
- [26] Chae Hoon Lim and Pil Joong Lee. More flexible exponentiation with precomputation. In *CRYPTO'94*, LNCS 839, pages 95–107. Springer, August 1994.
- [27] Silvio Micali, Rafael Pass, and Alon Rosen. Input-indistinguishable computation. In *47th FOCS*, pages 367–378. IEEE Computer Society Press, October 2006.
- [28] Bodo Möller. Algorithms for multi-exponentiation. In *SAC 2001*, LNCS 2259, pages 165–180. Springer, August 2001.
- [29] Phong Q. Nguyen, Igor E. Shparlinski, and Jacques Stern. Distribution of modular sums and the security of server aided exponentiation. In *Workshop on Comp. Number Theory and Crypt*, pages 1–16, 1999.
- [30] Phong Q. Nguyen and Damien Stehlé. Floating-point LLL revisited. In *EUROCRYPT 2005*, LNCS 3494, pages 215–233. Springer, May 2005.
- [31] Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors. *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*. ACM, 2007.
- [32] Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, 1991.

The two first vectors allow to recover

$$\begin{cases} x_0 = & 37921F2A & 890AA857 & DAC77BBF & 803B5D & \text{and} \\ y_0 = & 2379CD0E & 21A56BC1 & 33CAA48C & 43B4B2. \end{cases}$$

in less than 0.1 second on a standard laptop, using Sage math software [17] and fplll [1]. These vectors are of norms of size respectively 245 and 256 bits.

Remark 8. Note that an alternative attack can also be adapted from Yie’s paper [39], which aimed at recovering an ElGamal signature secret key when two signatures with small message nonces are available. Indeed, during the verification of an ElGamal signature, an equation of the form $h(m) = xr + ks \pmod q$ is verified, where x (the secret key) and k (the nonce) are unknown, and r and s are part of the signature of the message m . Two such equations make it possible to get rid of x , and we are back to an equation like Eq. 5. Yie adapted the algorithm of Gallant, Lambert and Vanstone GLV-Dec. This algorithm uses the extended Euclidean algorithm and has a complexity of $O(\log_2(q)^3)$.

B Outsourcing Exponentiations in Groups with Efficient Inverses

Let **GroupGen** be a group generator which takes as input a security parameter λ . It provides a set *params* which contains a description of a (multiplicative) group (\mathbb{G}, \cdot) , the group order, say $p = |\mathbb{G}|$, and one generator g . In this Appendix, we consider a variant of the generic group model in \mathbb{G} where the computation of group inverse is easy. This generic group is still implemented by choosing a random encoding $\sigma : \mathbb{G} \rightarrow \{0, 1\}^m$ (with $2^m > p$). As above, a generic algorithm \mathcal{A} takes as input (in addition to the group order p) their image under σ . This way, all \mathcal{A} can test is group elements equality (by encoding equality). \mathcal{A} is also given access to an oracle \mathcal{G} computing group multiplication: taking $\sigma(g_1)$ and $\sigma(g_2)$ encodings of two group elements $g_1, g_2 \in \mathbb{G}$ and a sign in $s \in \{-1, +1\}$ as inputs and returning $\sigma(g_1 \cdot g_2^s)$ the encoding of the product $g_1 \cdot g_2^s \in G$ (i.e., $g_1 \cdot g_2$ or g_1/g_2). We assume again that \mathcal{A} submits to the oracle only encodings of elements it had previously received. In this enhanced generic group model, we have the following lemma analogous to Lemma 1:

Lemma 3. *Considering this enhanced generic group model, let **GroupGen** be a group generator, let \mathbb{G} be a group of prime order p output by **GroupGen** and let \mathcal{A} be a generic algorithm in \mathbb{G} . If \mathcal{A} is given as inputs encodings $\sigma(g_1), \dots, \sigma(g_n)$ of groups elements $g_1, \dots, g_n \in \mathbb{G}$ (for $n \in \mathbb{N}$) and outputs the encoding $\sigma(h)$ of a group element $h \in \mathbb{G}$ in time τ , then there exists integers $\alpha_1, \dots, \alpha_n \in \mathbb{Z}$ such that $h = g_1^{\alpha_1} \dots g_n^{\alpha_n}$ and $\max(|\alpha_1|, \dots, |\alpha_n|) \leq 2^\tau$.*

Proof. We can – as in the proof of Lemma 1 – define a map $\pi : \{0, 1\}^m \rightarrow \mathbb{Z}^n$ which associates to each encoding obtained by \mathcal{A} during its execution an n -dimensional vector in \mathbb{Z}^n . For each input encoding $\sigma(g_i)$, $\pi(\sigma(g_i))$ is defined as the i -th vector from the \mathbb{Z}^n canonical basis (for $i \in \{1, \dots, n\}$) and for each encoding $\sigma(h_1)$ and $\sigma(h_2)$ and each sign $s \in \{-1, 1\}$ queried to \mathcal{G} , $\pi(\sigma(h_1 \cdot h_2^s)) = \pi(\sigma(h_1)) + s \cdot \pi(\sigma(h_2))$. By construction, during the whole execution of \mathcal{A} , we have $\pi(\sigma(h)) = (\alpha_1, \dots, \alpha_n)$ if and only if $h = g_1^{\alpha_1} \dots g_n^{\alpha_n}$ for all encodings $\sigma(h)$. As in the proof of Lemma 1, the ℓ_∞ -norm of $\pi(\sigma(h_1 \cdot h_2^s))$ is upper-bounded by $\ell_\infty(\pi(\sigma(h_1))) + \ell_\infty(\pi(\sigma(h_2)))$ Since the ℓ_∞ -norm of the input encodings $\pi(\sigma(g_i))$ is equal to 1 (for $i \in \{1, \dots, n\}$) and the ℓ_∞ -norm of encodings at most doubles for each query to \mathcal{G} , we obtained the claimed result. \square

In this setting, we can consider a variant of Algorithm 1 that computes the multi-exponentiation $\prod_{i=1}^t g_i^{x_i}$, for $g_1, \dots, g_t \in \mathbb{G}$ and $x_1, \dots, x_t \in \mathbb{N}$ by interleaving signed expansions of exponents. In particular, we can use the *w-ary non-adjacent form* method which guarantees that on average there will be fewer group multiplications for the same window size w (see [28, 3] for details). In this case, the precomputation stage generates $\prod_{1 \leq i \leq t} g_i^{E_i}$ (and the algorithm can use the values $\prod_{1 \leq i \leq t} g_i^{-E_i}$) for all non-zero t -tuples $(E_1, \dots, E_t) \in \{0, \dots, 2^w - 1\}^t$ at no extra storage-cost). The total cost is thus for the precomputation phase $t2^{tw-2} - t$ multiplications and t squarings and overall less than $\ell/(w + 1/t) \leq \ell/w$ multiplications on average and ℓ squarings. For $t = 2$, the cost is again minimal for w around $1/2 \log \ell - \log \log \ell$ with $\ell(1 + 3/\log \ell) = \ell(1 + o(1))$ multiplications overall. Therefore, the method does not improve the asymptotic complexity (at least when the precomputation stage and the storage are not strongly limited). We can replace the use of Algorithm 1 by this variant but this does not improve the asymptotic complexity of the delegation protocols in the number of generic group operations (even with efficient inverses).

Actually, this fact is not surprising, since we can replace the use of Lemma 1 in the proof of our lower bound complexities (Theorems 3 – 9) by the use of Lemma 3 to obtain the same lower bounds for delegation protocols in the enhanced generic group model with inverses.

C Generic Constructions for Outsourcing Multi-Exponentiations

As mentioned in Section 4, even if one can fix Wang *et al.*'s protocol by using a larger Υ (such that the value χ is actually uniformly distributed over \mathbb{Z}_p), the resulting inefficient protocol would still not achieve the privacy security notion. Indeed, in their protocol the μ_1 and μ_3 used to mask the secret bases $u_{i,j}$ in Equation (2) and Equation (3) are always the same for all bases. In particular, an adversary against the privacy of this protocol can simply pick bases $(u_{1,1}^0, u_{1,2}^0)$ and $(u_{1,1}^1, u_{1,2}^1)$ in $\mathbf{Exp}_{priv}(\mathcal{A})$ such that $u_{1,1}^0/u_{1,2}^0 \neq u_{1,1}^1/u_{1,2}^1$. Since, from Equation (3), we know that $w_{1,1}/w_{1,2} = u_{1,1}^b/u_{1,2}^b$ then it can determine the bit b used in the experiment with certainty.

————— **Protocol 10:** 100f (and 101f) —————

Input: $u_1, \dots, u_n \in \mathbb{G}$, $a_1, \dots, a_n \in \mathbb{Z}_p$

Output: $u_1^{a_1} \dots u_n^{a_n} \in \mathbb{G}$

for i from 1 to n do

$(u_i^{k_i}, k_i) \leftarrow \mathcal{B}(i)$

$h_i \leftarrow \mathcal{S}(u_i, a_i - k_i \bmod p)$

end for

return $\prod_{i=1}^n h_i u_i^{k_i}$

————— **Protocol 11:** 011v —————

Input: $u_1, \dots, u_n \in \mathbb{G}$, $a_1, \dots, a_n \in \mathbb{Z}_p$

Output: $u_1^{a_1} \dots u_n^{a_n} \in \mathbb{G}$

for j from 1 to n do

 for i from 1 to s do

$g_i \xleftarrow{R} \mathbb{G}$

 end for

$\mathcal{I} \xleftarrow{R} \mathfrak{P}_m(\{1, \dots, s\})$ ▷ random subset
of cardinal m of $\{1, \dots, s\}$

$g_{s+1} \leftarrow u_j \cdot \prod_{i \in \mathcal{I}} g_i$

 for i from 1 to $s+1$ do

$h_i \leftarrow \mathcal{S}(g_i, a_j)$

 end for

$v_j \leftarrow h_{s+1} / \prod_{i \in \mathcal{I}} h_i$

end for

return $v_1 \dots v_n$

————— **Protocol 12:** 101v —————

Input: $u_1, \dots, u_n \in \mathbb{G}$, $a_1, \dots, a_n \in \mathbb{Z}_p$

Output: $u_1^{a_1} \dots u_n^{a_n} \in \mathbb{G}$

$T \leftarrow [p^{1/s+1}]$

for i from 1 to n do

 for j from 1 to s do

$h_{i,j} \leftarrow \mathcal{S}(u_i, T^j)$

 end for

 temp $\leftarrow a_i$

 for j from s down to 0 do

$a_{i,j} = \text{temp} \bmod T^j$

 temp = temp - $a_{i,j} \cdot T^j$

 end for

▷ $a_i = a_{i,s} \cdot T^s + \dots + a_{i,1} T + a_{i,0}$

end for

return $\prod_{i=1}^n \left(u_i^{a_{i,0}} \prod_{j=1}^s h_{i,j}^{a_{i,j}} \right)$

▷ using Algorithm 1

————— **Protocol 13:** 100v (and 101v) —————

Input: $u_1, \dots, u_n \in \mathbb{G}$, $a_1, \dots, a_n \in \mathbb{Z}_p$

Output: $u_1^{a_1} \dots u_n^{a_n} \in \mathbb{G}$

$T \leftarrow [p^{1/s+1}]$

for j from 1 to n do

 for i from 1 to s do

$h_{i,j} \leftarrow \mathcal{S}(u_j, T^i)$

 end for

 temp $\leftarrow a_j$

 for i from s down to 0 do

$a_{i,j} = \text{temp} \bmod T^i$

 temp = temp - $a_{i,j} \cdot T^i$

 end for

▷ $a_j = a_{s,j} \cdot T^s + \dots + a_{1,j} T + a_{0,j}$

end for

return $\prod_{j=1}^n u_j^{a_{0,j}} \prod_{i=1}^s h_{i,j}^{a_{i,j}}$

▷ using Algorithm 1

————— **Protocol 14:** 010v —————

Input: $u_1, \dots, u_n \in \mathbb{G}$, $a_1, \dots, a_n \in \mathbb{Z}_p$

Output: $u_1^{a_1} \dots u_n^{a_n} \in \mathbb{G}$

for i from 1 to n do

$(g^{r_i}, r_i) \leftarrow \mathcal{B}(\cdot)$

$h_{1,i} \leftarrow \mathcal{S}(u_i \cdot g^{r_i}, a_i)$

end for

$(g^s, s) \leftarrow \mathcal{B}(\cdot)$

$(g^t, t) \leftarrow \mathcal{B}(\cdot)$

$k \leftarrow (t - \sum_{i=1}^s r_i a_i) / s \bmod p$

$h_2 \leftarrow \mathcal{S}(g^s, k)$

 return $\prod_{i=1}^s h_{1,i} h_2 g^t$

————— **Protocol 15:** 000v (and 001v) —————

Input: $u_1, \dots, u_n \in \mathbb{G}$, $a_1, \dots, a_n \in \mathbb{Z}_p$

Output: $u_1^{a_1} \dots u_n^{a_n} \in \mathbb{G}$

for j from 1 to n do

$(g^{k_{1,j}}, k_{1,j}) \leftarrow \mathcal{B}(\cdot)$

$v_j \leftarrow u_j \cdot g^{k_{1,j}}$

$h_{1,j} \leftarrow v_j^{a_j}$

▷ delegated using Prot. 13

▷ $h_{1,j} = v_j^{a_j} = u_j^{a_j} \cdot g^{a_j k_{1,j}}$

$(g^{k_{2,j}}, k_{2,j}) \leftarrow \mathcal{B}(\cdot)$

$h_{2,j} \leftarrow \mathcal{S}(g, -a_j k_{1,j} - k_{2,j} \bmod p)$

▷ $h_{2,j} = g^{-a_j k_{1,j} - k_{2,j}}$

end for

return $\prod_{j=1}^n h_{1,j} \cdot h_{2,j} \cdot g^{k_{2,j}}$

We thus give in this section several protocols for outsourcing multi-exponentiations $(u_1, \dots, u_n, a_1, \dots, a_n) \mapsto u_1^{a_1} \dots u_n^{a_n}$. Their security is stated in Theorem 6. The proof of this theorem as well as optimality results are similar to those given in the core of the paper.

Theorem 6. *Let GroupGen be a group generator, let λ be a security parameter and let \mathbb{G} be a group of primer order p output by GroupGen(λ). Let $(\mathcal{C}, \mathcal{S})$ be one client-server protocol for the delegated computation*

of the multi-exponentiation $u_1^{a_1} \cdots u_n^{a_n}$ described in Protocols 10 – 15 (for the corresponding computation code $\beta \in \{0, 1\}^4$ given in their description). The protocol $(\mathcal{C}, \mathcal{S})$ satisfies $(\tau, 0)$ -privacy against a malicious adversary for any time τ .

C.1 Construction for Outsourcing Fixed Based Multi-Exponentiation

When the bases (u_1, \dots, u_n) are fixed, one can assume that \mathcal{C} can use a pseudo-random power generator $\mathcal{B}(i)$ for each u_i . As for the single exponentiation case, the cases $111f$, $110f$ and $011f$ are trivial or do not make sense.

We give Protocol 10 in case $100f$ where the bases are public, the exponents private and the result private. This protocol obviously work in the cases where the exponents or the result become public (case $101f$), but could probably be improved in these latter cases.

This protocol does not apply when the bases are private and exponents public (case $010f$), but one can instead use Protocol 11.

C.2 Construction for Outsourcing Variable Base Multi-Exponentiation

Since multi-exponentiations are at least as difficult as single exponentiations, lower bounds obtained in Section 6 show that it is impossible to construct a protocol using a constant number of operations in \mathbb{G} when something is secret and the bases are variable. This gives further evidence that the protocols given in [38] cannot be private.

When the bases (u_1, \dots, u_n) are variable, one cannot assume that \mathcal{C} can use a pseudo-random power generator $\mathcal{B}(i)$ for each u_i , but he can still use one for the generator g , that we denote \mathcal{B} in the following constructions.

As for the single exponentiation case, the cases $111v$, $110v$ and $011v$ are trivial or do not make sense.

We give Protocol 11 in case $011v$ where the bases are private, the exponents public and the result public and Protocol 12 in case $101v$ where the bases are public, the exponents private and the result public.

Finally, we give three protocols for the cases $100v$, $010v$ and $000v$ (Protocols 13, 14 and 15, respectively) which are basically parallel repetitions of the protocols for single exponentiation for the same cases. One may be tempted to reuse masks generated by the pseudo-random power generator \mathcal{B} for several private bases u_i (for $i \in \{1, \dots, n\}$). However, one can prove using our techniques from Section 6 that this would result in insecure protocols.

D Complexity Lower Bound for Multi-Round Protocols

D.1 Complexity Lower Bound for Two-Round Protocols

We consider the delegation of the exponentiation u^a with variable and public base u and secret exponent a . One can easily adapt the proof of Theorem 3 to the case where the delegator is allowed to delegate two group exponentiations in an adaptive way (i.e., in two communication rounds). Informally, Theorem 7 asserts that in this case the delegator needs to perform at least $\log(p)/4$ group operations (even if it is allowed to make an arbitrary constant number of queries to a pseudo-random power generator for a generator $g \neq u$).

Theorem 7. *Let GroupGen be a group generator and let $(\mathcal{C}, \mathcal{S})$ be one client-server protocol for the delegated computation of the exponentiation u^a for the corresponding computation code $\beta = 101v$. We assume that the delegator \mathcal{C} is a generic group algorithm that uses*

- $c \log(p) + O(1)$ generic group operations (for all groups \mathbb{G} of primer order p output by $\text{GroupGen}(\lambda)$) for some constant c ,
- $\ell = O(1)$ queries to the (private) pseudo-random power generator \mathcal{B}
- and 2 adaptive delegated exponentiation to the delegatee \mathcal{S}

If $c < 1/(4)$, then $(\mathcal{C}, \mathcal{S})$ is not private: there exists an algorithm running in polynomial-time such that

$$\Pr[\text{bit} \leftarrow \mathbf{Exp}_{priv}(\mathcal{A}) : \text{bit} = 1] \geq 1 - \lambda^{O(1)}.$$

Proof. The proof is similar to the proof of Theorem 3.

We assume that the delegator \mathcal{C} gets as input two encodings $\sigma(u)$, $\sigma(g)$ of two group elements u and g picked uniformly at random in \mathbb{G} and one scalar a picked uniformly at random in \mathbb{Z}_p and outputs the encoding $\sigma(u^a)$ of the group element u^a by making only

- q queries to the group oracle \mathcal{G} ;
- ℓ queries to the (private) pseudo-random power generator \mathcal{B} ;
- 2 (adaptive) queries to the delegatee oracle \mathcal{S} .

We assume that $q = c \log p + O(1)$ with $c < 1/4$. and we prove that it is not possible for \mathcal{C} to compute $\sigma(u^a)$ in such a way that the delegatee \mathcal{S} learns no information on a . More precisely, the challenger picks uniformly at random a scalar $a^* \in \mathbb{Z}_p$ and a random bit b and sets $(a_b, a_{1-b}) = (a, a^*)$ (i.e. $\{a_0, a_1\} = \{a, a^*\}$ in a random order). The adversary (with the knowledge of the delegatee \mathcal{S} 's transcript) has to guess the bit b .

Let us denote $(t_1, \sigma(g^{t_1}))$, $(t_2, \sigma(g^{t_2}))$, \dots , $(t_\ell, \sigma(g^{t_\ell}))$ the pairs obtained from the pseudo-random power generator \mathcal{B} by the delegator \mathcal{C} . Since the pseudo-random power generator \mathcal{B} takes no inputs and outputs independent pairs, we can assume without loss of generality that the delegator \mathcal{C} makes the ℓ queries to \mathcal{B} in a first phase of the delegation protocol.

We denote $(\sigma(h_1), x_1)$ the first pair group element/scalar made by \mathcal{C} to the delegatee \mathcal{S} . Using generic group operations, \mathcal{C} can only construct the corresponding group elements such that:

$$h = u^{\alpha'} \cdot g^{\kappa'} \cdot g^{t_1 \gamma'_1} \dots g^{t_\ell \gamma'_\ell} \quad (18)$$

for some scalars $(\alpha', \kappa', \gamma'_1, \dots, \gamma'_\ell)$. We denote $k_1 = h_1^{x_1}$ the response of the \mathcal{S} .

We denote $(\sigma(h_2), x_2)$ the second pair group element/scalar made by \mathcal{C} to the delegatee \mathcal{S} . Using generic group operations, \mathcal{C} can only construct the corresponding group elements such that:

$$h_2 = u^{\alpha''} \cdot g^{\kappa''} \cdot g^{t_1 \gamma''_1} \dots g^{t_\ell \gamma''_\ell} k_1^{\delta''} h_1^{\varepsilon''} \quad (19)$$

for some scalars $(\alpha'', \kappa'', \gamma''_1, \dots, \gamma''_\ell, \delta'', \varepsilon'')$. We denote $k_2 = h_2^{x_2}$ the response of the \mathcal{S} .

Eventually, the delegator \mathcal{C} outputs the encoding $\sigma(u^a)$ of the group element u^a and as above using generic group operations, it can only construct it as

$$u^a = u^\alpha g^\kappa \cdot g^{t_1 \gamma_1} \dots g^{t_\ell \gamma_\ell} k_1^\delta h_1^\varepsilon k_2^\zeta h_2^\eta \quad (20)$$

for some scalars $(\alpha, \kappa, \gamma_1, \dots, \gamma_\ell, \delta, \varepsilon, \zeta, \eta)$. If we assume that $q = c \log n + O(1)$ (and in particular $q = o(\sqrt{p})$), the delegator \mathcal{C} is not able to compute the discrete logarithm of u in base g . This means that necessarily the exponent of g in Equation (20) cancel out. Recall that $k_1 = h_1^{x_1}$ and $k_2 = h_2^{x_2}$, h_1 and h_2 being constructed as in Equation (18) and Equation (19). Thus, taking only the discrete logarithms of powers of u in base u of this equation, we obtain

$$\begin{aligned} a &= (\alpha + \varepsilon \alpha' + \eta(\alpha'' + \varepsilon'' \alpha')) \\ &\quad + (\delta \alpha' + \eta \delta'' \alpha') x_1 \\ &\quad + (\alpha'' + \alpha'' + \varepsilon'' \alpha') \zeta x_2 \\ &\quad + (\zeta \delta'' \alpha') x_1 x_2 \pmod{p}. \end{aligned} \quad (21)$$

For a random choice of $a \in \mathbb{Z}_p$, we have $a = \Omega(p)$. We denote τ_1 the number of group operations performed by \mathcal{C} in the computation of h_1 described in Equation (18), τ_2 the number of group operations performed by \mathcal{C} in the computation of h_2 described in Equation (19) and τ_3 the number of group operations performed by \mathcal{C} in the computation of u^a described in Equation (20).

By assumption, $\tau_1 + \tau_2 + \tau_3 \leq c \log p + O(1)$. If we note $\rho_1 = \alpha + \varepsilon \alpha' + \eta(\alpha'' + \varepsilon'' \alpha')$, $\rho_2 = \delta \alpha' + \eta \delta'' \alpha'$, $\rho_3 = (\alpha'' + \alpha'' + \varepsilon'' \alpha') \zeta$ and $\rho_4 = \zeta \delta'' \alpha'$ Equation (21) becomes

$$a = \rho_1 + \rho_2 x_1 + \rho_3 x_2 + \rho_4 x_1 x_2 \pmod{p}$$

where x_1 and x_2 are known to the adversary. Furthermore, since \mathcal{C} only used generic group operations, we have as above $\rho_i \leq p^{c+o(1)}$ for $i \in \{1, 2, 3, 4\}$.

The adversary can then try to decompose a_0 and a_1 as

$$a_i = \rho_{i,1} + \rho_{i,2} x_1 + \rho_{i,3} x_2 + \rho_{i,4} x_1 x_2 \pmod{p}$$

with $\rho_{i,j} \leq p^{c+o(1)}$ for $i \in \{0, 1\}$ and $j \in \{1, 2, 3, 4\}$. For $a_b = a$, the algorithm from Section 3.3 will recover $\rho_{b,1}$, $\rho_{b,2}$, $\rho_{b,3}$ and $\rho_{b,4}$ in polynomial time. However, for a given pair (x_1, x_2) and a random $a_{1-b} = a^*$, there is only a negligible probability that such a decomposition exists (less than $(p^{c+o(1)})^4 = p^{4c+o(1)} = o(p)$ scalars can be written in this way). Thus, the adversary can simply run the Coppersmith-like algorithm on $(a_0, 1, x_1, x_2, x_1 x_2)$ on one hand and on $(a_1, 1, x_1, x_2, x_1 x_2)$ on the other hand and returns the bit b for which the algorithm returns a ‘‘small decomposition’’ on input $(a_b, 1, x_1, x_2, x_1 x_2)$. By the previous analysis, its advantage is noticeable. \square

In this setting, the best delegation protocol (to our knowledge) requires $\log(p)/3$ group operations for the delegator: it is Protocol 5 from Section 5 (with $s = 2$) that do not take advantage of the fact that the second delegated exponentiation may depend on the first one.

If there exists a way to express the exponent a as a weighted sum

$$a = \alpha_0 + \alpha_1 x_1 + \alpha_2 x_2 + \alpha_3 x_1 x_2 \pmod{p} \quad (22)$$

with $\alpha_i \leq p^{1/4}$ for $i \in \{0, 1, 2\}$ for some arbitrary scalars x_1 and x_2 that do not reveal information on a , then the delegator may query the delegatee the exponentiation $k_1 = u^{x_1}$ and subsequently $k_2 = (uk_1)^{x_2} = (u^{x_1+1})^{x_2}$ such that $u^a = u^{\alpha_0} k_1^{\alpha_1} k_2^{\alpha_2}$. Using Algorithm 1, this approach would make it possible for the delegator to compute u^a with roughly $\log(p)/4$ group operations by delegating two successive group exponentiations to the delegatee (and in this case Theorem 7 will prove the optimality of this algorithm).

Unfortunately, the GLV decomposition algorithm (GLV-Dec) does not permit to obtain such decomposition (but only a decomposition of the form $a = \alpha_0 + \alpha_1 x_1 + \alpha_2 x_2 + \alpha_3 x_1 x_2$ with $\alpha_i \leq p^{1/4}$ for $i \in \{0, 1, 2, 3\}$ where α_3 is very likely different from α_2). We do not know how to obtain a “small” decomposition of the form (22) (or a similar one). On the other hand, it seems impossible to rule out the existence of such decompositions by combinatorial arguments (such as the one used in the proof of Theorem 5). We run extensive experiments for small prime group orders p and we find out that in those case, all scalars $a \in \mathbb{Z}_p$ actually admits a large number of such decompositions.

Even if the computational improvement from $\log(p)/3$ to $\log(p)/4$ group operations would be marginal in practice compared to the increase of the round complexity (and thus the latency of the protocol), it is an interesting theoretical open problem to study the existence of such decompositions (and to provide an efficient algorithm to construct them).

We can also consider the case where the delegator is allowed to delegate several group exponentiations in an adaptive way but in only two rounds. We obtain the following theorem:

Theorem 8. *Let GroupGen be a group generator and let $(\mathcal{C}, \mathcal{S})$ be one client-server protocol for the delegated computation of the exponentiation u^a for the corresponding computation code $\beta = 101v$. We assume that the delegator \mathcal{C} is a generic group algorithm that uses*

- $c \log(p) + O(1)$ generic group operations (for all groups \mathbb{G} of primer order p output by $\text{GroupGen}(\lambda)$) for some constant c ,
- $\ell = O(1)$ queries to the (private) pseudo-random power generator \mathcal{B}
- and s simultaneous delegated exponentiation to the delegatee \mathcal{S} in two rounds

If $c < 4/(4 + (s + 1)^2)$, then $(\mathcal{C}, \mathcal{S})$ is not private: there exists an algorithm running in polynomial-time such that

$$\Pr[\text{bit} \leftarrow \mathbf{Exp}_{\text{priv}}(\mathcal{A}) : \text{bit} = 1] \geq 1 - \lambda^{O(1)}.$$

Proof. The proof is similar to the proof of Theorem 4. □

In particular, Theorem 8 asserts that in order to construct a delegation protocol in which the delegator performs only a constant number a group operations, then the round complexity of the protocol should be at least $\Omega(\sqrt{\log(p)})$. The proof of Theorem 8 actually shows the stronger result that even if the round complexity is $O(\sqrt{\log(p)})$, then the number of group operations for the delegator is also of order $\Omega(\sqrt{\log(p)})$ (and is therefore non-constant).

D.2 Complexity Lower Bound for Multiple-Round Protocols

For completeness, we mention that it is also possible to prove a lower bound on the efficiency of delegation protocols with any round complexity. For simplicity, we state only the complexity lower bound in the case of a delegation protocol that delegates the computation of s group exponentiations in s rounds (in an adaptive way). The lower bound is not as strong as the previous one since it decrease exponentially with s . Roughly speaking, Theorem 9 asserts that the best delegation protocol we can hope for requires $\Omega(\log \log(p))$ rounds in order to decrease the computational complexity of the delegator to only $O(\log \log(p))$ group operations.

Theorem 9. *Let GroupGen be a group generator and let $(\mathcal{C}, \mathcal{S})$ be one client-server protocol for the delegated computation of the exponentiation u^a for the corresponding computation code $\beta = 101v$. We assume that the delegator \mathcal{C} is a generic group algorithm that uses*

- $c \log(p) + O(1)$ generic group operations (for all groups \mathbb{G} of primer order p output by $\text{GroupGen}(\lambda)$) for some constant c ,
- $\ell = O(1)$ queries to the (private) pseudo-random power generator \mathcal{B}
- and s simultaneous delegated exponentiation to the delegatee \mathcal{S}

If $c < 2^{-s}$, then $(\mathcal{C}, \mathcal{S})$ is not private: there exists an algorithm running in polynomial-time such that

$$\Pr[\text{bit} \leftarrow \mathbf{Exp}_{\text{priv}}(\mathcal{A}) : \text{bit} = 1] \geq 1 - \lambda^{O(1)}.$$

Proof. The proof is again similar to the proof of Theorem 4. □