Noname manuscript No.
(will be inserted by the editor)

Efficient and Verifiable Algorithms for Secure Outsourcing of Cryptographic Computations

Mehmet Sabır Kiraz · Osmanbey Uzunkol

Received: date / Accepted: date

Abstract Reducing computational cost of cryptographic computations for resource-constrained devices is an active research area. One of the practical solutions is to securely outsource the computations to an external and more powerful cloud server. Modular exponentiations are the most expensive computation from the cryptographic point of view. Therefore, outsourcing modular exponentiations to a single, external and potentially untrusted cloud server while ensuring the security and privacy provide an efficient solution. In this paper, we propose new efficient outsourcing algorithms for modular exponentiations using only one untrusted cloud server. These algorithms cover public-base & private-exponent, private-base & public-exponent, private-base & private-exponent, and more generally private-base & private-exponents simultaneous modular exponentiations. Our algorithms are the most efficient solutions utilizing only one single untrusted server with best checkability probabilities. Furthermore, unlike existing schemes, which have fixed checkability probability, our algorithms provide adjustable predetermined checkability parameters. Finally, we apply our algorithms to outsource Oblivious Transfer Protocols and Blind Signatures which are expensive primitives in modern cryptography.

Keywords Secure outsourcing algorithms · Modular exponentiation · Mobile computing · Secure cloud computing · Privacy

Mehmet Sabır Kiraz (Corresponding Author)
Mathematical and Computational Sciences Labs, TÜBİTAK BİLGEM UEKAE, National Research Institute of Electronics and Cryptology P. K. 74 41470 Gebze/Kocaeli, TURKEY
Tel.: +90 262 648 1945
Fax: +90 262 648 1100
E-mail: mehmet.kiraz@tubitak.gov.tr

Osmanbey Uzunkol
Mathematical and Computational Sciences Labs, TÜBİTAK BİLGEM UEKAE, National Research Institute of Electronics and Cryptology P. K. 74 41470 Gebze/Kocaeli, TURKEY
Tel.: +90 262 648 1782
Fax: +90 262 648 1100
E-mail: osmanbey.uzunkol@tubitak.gov.tr

1 Introduction

Cloud Computing is getting more and more attention in the scientific community due to its multiple benefits for real-world applications (e.g., on-demand self-service, ubiquitous network access, location independent resource pooling, pay per use, rapid elasticity, and outsourcing). Depending on demands, capabilities and resources, it is possible to efficiently outsource costly calculations to more powerful servers using cloud computing infrastructures.

Cryptographic key sizes have been steadily increasing due to mathematical and technological developments. This may lead to that existing resource-constrained devices may become incapable of assuring the desired level of security. In general, these devices are required to be replaced with more powerful new ones. However, this leads to highly impractical and costly solutions. The main motivation of outsourcing computation is to give a usable, secure and more practical solution. For example, modular exponentiation of the form u^a modulo a prime number p, where u, a, and p have minimum length of 2048 bits (in order to have a cryptographically secure algorithm) has a big computational obstacle for the computationally limited devices. To compute a single modular exponentiation for 2048-bits exponent a, more than 3000 modular multiplications must be performed in average (using square and multiply method). Therefore, it is usable to outsource the expensive computations to the cloud providers. Nevertheless, the outsourced computations may contain additional sensitive information that should not be leaked to the outsiders (e.g., personal, health or financial data). In order to prevent information leakage, the sensitive data have to be appropriately masked before outsourcing. On the one hand, the masking technique should be designed in such a way that the overall computational cost to the client is significantly reduced. Namely, reducing the cost of masking before outsourcing, and the cost of removing the mask after obtaining the result from the cloud provider are of utmost important, and the total cost should be less than performing the computation on the device itself. On the other hand, it is also essential to assure the client that the returned results are indeed correct. Namely, malicious servers or environmental attacks should not compromise the correctness without being detected with a non-negligible probability. Therefore, it is crucial to have an efficient outsourcing algorithm satisfying certain privacy preserving properties. This can be achieved by checking and verifying the correctness of the outcome.

One straightforward solution is to assume the existence of a fully-trusted or a semi-trusted cloud server. However, because of security and privacy concerns it is not that likely the case in real-life scenarios. For example, due to financial reasons, malicious cloud providers can insert a software bug that will fail after some particular steps of the algorithms and then return an incorrect result which is computationally indistinguishable from the correct output. By the checkability property, the client can easily detect these malicious behavior from the cloud side. The main question can be stated as follows:

How can security and privacy be guaranteed by using only a single untrusted server without revealing any information about the inputs and/or the outputs while assuring the correctness?

1.1 Related Work

Outsourcing secure computation allows parties to compute a functionality without leaking any information about their inputs and/or outputs. In general, it is expected to have no interactions between the parties, and computational cost and bandwidth of each user are expected to be independent of the functionality. However, general program obfuscation is impossible by utilizing only one cloud server [1]. This is one of the reasons that we mainly focus on expensive modular computations.

Many algorithms have been proposed for outsourcing computation [2–16]. They aim either to have a better outsourcing security model or to have more efficient constructions. However, these algorithms consider outsourcing of a public-base & private-exponent, or private-base & public-exponent, or they satisfy weaker security notions. For example, in [17], Clarke *et al.* propose protocols for speeding up exponentiation in a cyclic group using untrusted servers for public-base & private-exponent and private-base & public-exponent. They also extend these algorithms to compute an exponentiation modulo a composite integer.

Hohenberger and Lysyanskaya [3] presented the first outsource-secure algorithm for modular exponentiations for outsourcing cryptographic computations. This algorithm considers only the case private-base & private-exponent exponentiation modulo a prime number. With this algorithm, modular exponentiations can be computed by the client with $O(log^2(l))$ multiplications with error probability $\frac{1}{2}$, where l denotes the number of bits of the exponent. The main drawback of this solution is to use two non-colluding untrusted servers.

At ESORICS 2012, Chen *et al.* [2] propose a more efficient construction than Hohenberger-Lysyanskaya's algorithm, where the probability of detecting malicious behavior is improved to 2/3. However, modular exponentiations can be computed by the client with $O(log^2(l))$ multiplications. They also propose the first secure outsourcing algorithm for simultaneous modular exponentiations $u_1^{a_1} \cdot u_2^{a_2}$. Simultaneous modular exponentiations are also used in many cryptographic primitives such as commitments [18], zero-knowledge proofs [19] and additive variant of ElGamal encryptions [20]. Chen *et al.* apply their algorithms to outsource Cramer-Shoup encryptions and Schnorr signatures securely.

The authors in [21] proposed an algorithm using a single untrusted server for public-base & private-exponent and private-base & public-exponent cases. The algorithm is quite interesting since the privacy is guaranteed by the difficulty of solving the subset sum problem. Briefly, the client first randomizes the exponents and then puts a private pattern to the exponent values before they are sent to the server. After the server sends the computed values back

to the client, it can verify the correctness efficiently using the pattern. However, there is a checkability issue in their algorithm, where an untrusted server can easily manipulate the result. In particular, the client invokes the server $\mathsf{Exp}(a,g)$ to outsource the computation of g^a, and instead of $\mathsf{Exp}(a,g)$ the malicious server can compute $\mathsf{Exp}(a,gh)$ for some bogus value h without being detected. The checks will pass successfully and subsequently the result would become incorrect without being unnoticed.

The authors in [22] proposed the only existing algorithm for modular exponentiations modulo composite numbers. However, we also address an issue here that the checkability property of their scheme fails. By using their notation, the attack can be explained briefly as follows: A malicious server $\mathcal{S}$ uses the proposed values $\ell = \ell_1 = \ell_2 = 5$ in [22] (or any other case for which $\ell = \ell_1 = \ell_2$ holds), adds 1 to the values y_j, and outputs $x_i^{y_j+1}$ instead of $x_i^{y_j}$. This enables the server to manipulate the result u^a with $u^{a+\ell}$ without being detected by the client.

Another area of outsourcing computation is the use of homomorphic encryption techniques. Homomorphic encryption allows parties for processing computations on encrypted data without using any additional information like Yao's garbled circuits [23]. Conventional homomorphic encryption schemes are either additive or multiplicative (e.g., RSA is multiplicative, Paillier and modified version of ElGamal encryption are additive [20, 24], or the 2DNF protocol [25] which allows multiple additions up to only one exponentiation). These schemes allow to outsource secure function evaluation to a cloud server. Recent somewhat homomorphic and fully homomorphic schemes give a complete solution to the outsourcing problem [26]. However, these systems are not yet efficient enough to be applied in real-life scenarios.

1.2 Our Contributions

Our contributions are as follows:

1. We propose new, efficient and secure outsourcing algorithms of modular exponentiations using only one untrusted server. We consider the cases public-base & private-exponent, private-base & public-exponent, private-base & private-exponent and simultaneous modular exponentiations separately. This approach realizes privacy preserving and efficient outsourcing mechanisms, which are highly desirable and often inevitable for resource-constrained devices. Instead of having an adversary model, where distrustful servers are assumed not to collude, our algorithms borrow computing power from only a single untrusted cloud server. This constitutes a more realistic scenario when compared to the state-of-the-art algorithms in [2–4].
2. Our algorithms cover both modulo prime and modulo composite number cases. To the best of our knowledge, these algorithms make for the first time no distinction between prime and composite modulus by a unified modular exponentiation approach. Therefore, modular exponentiations in

cryptographic protocols based on both the Discrete Logarithm problem (DLP) and the RSA problem can be outsourced securely to an untrusted server.
3. In [5], the authors propose the first generic algorithm for a single untrusted server considering private-base & private-exponent. Furthermore, it has fixed 1/2 checkability probability. However, their scheme is quite inefficient since it requires approximately 2000 modular multiplications. In contrast to this scheme, we would like to highlight that our private-base & private-exponent algorithm $\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$ is the most efficient and verifiable solution with respect to the existing ones (e.g., we have $\approx 10,17$ times less MMs than the only existing algorithm [5]).
4. We emphasize further that although the existing solutions use two non-colluding malicious servers, they propose only 1/2, 2/3 or 3/4 probabilities for the checkability [2–4], respectively. Unlike all existing schemes, our algorithms have not only the best but also more importantly adjustable checkability. Also, any adversarial behavior can be detected by the client with the probability $1 - \frac{1}{c(c-1)}$, where c is a small integer used as a security parameter for checkability (e.g., for $c = 4$ and $c = 8$ we have 11/12 and 55/56, resp.).
5. Our algorithm for the case of simultaneous modular exponentiation is more efficient than the existing algorithms [2] and [5] (there is only a generalized result in [22], for which the checkability fails as explained above). The algorithm proposed in [2] only considers two simultaneous modular exponentiations. We generalize this by introducing the notion of *t-simultaneous modular exponentiation*, i.e., t modular exponentiations can be computed simultaneously in a single round. We also show that we gain linear complexity advantage in t for both the number of modular multiplications and modular inversions.
6. Lastly, we apply the proposed algorithms to outsource Oblivious Transfer (OT) and Blind Signature schemes securely. Note that OT is a powerful cryptographic primitive which is "complete" for secure multiparty computation [27,28]. It is also one of the major computational overhead for Yao's garbled circuit protocols [23,29]. OTs are also used in many applications like biometric authentication, e-auctions, private information retrieval, and private search [30–33]. Hence, the overall complexity for mobile environment and resource-constrained devices can be enhanced by outsourcing OT securely. Furthermore, blind signatures [34] are unforgeable, and can be verified by a public key like in conventional digital signatures. These signatures can be used in many applications like e-cash, e-voting and anonymous credentials [35]. Hence, outsourcing blind signatures can also be very beneficial for real-life applications.

1.3 Roadmap

In Section 2, we give our formal security and privacy model based on the model of [3] by simplifying their *two untrusted server model* to a more realistic and secure *one single untrusted server model.* Section 3 starts with basic mathematical background of outsourcing algorithms of modular exponentiation, and describes the main algorithm for private-base & private-exponent modular exponentiations. We also provide the proofs of correctness, security and checkability of our algorithms using security/privacy model in Section 2. In Section 4, we propose algorithms for all other relevant situations, i.e. public-base & private-exponent, private-base & public-exponent, and private-base & private-exponent, and simultaneous modular exponentiations. Section 5 gives the complexity of our algorithms, and compares the efficiency of the algorithms with the prior works. In Section 6, we apply our algorithms to Oblivious Transfer protocols and to Blind Signatures. Section 7 concludes the paper with highlighting future research directions on outsourcing cryptographic computations.

2 Security and Privacy Model

In this work, we follow the security model proposed by Hohenberger and Lysyanskaya [3] like the recent results in [2,5]. Assume that a client $\mathcal{C}$ would like to securely outsource an expensive cryptographic computation Alg to a cloud server $\mathcal{S}$. Our aim is to split the computation into two main procedures (1) $\mathcal{C}$ knows the input value to Alg, (2) $\mathcal{C}$ invokes $\mathcal{S}$ which is an untrusted server that can carry out expensive computation operations. Briefly, $\mathcal{C}$ securely outsources some computation if the following conditions hold:

1. $\mathcal{C}$ and $\mathcal{S}$ implement Alg, i.e., $\mathsf{Alg} = \mathcal{C}^{\mathcal{S}}$
2. Assume that $\mathcal{C}$ has oracle access to an adversary $\mathcal{S}'$ (instead of an honest $\mathcal{S}$) which stores its computational results during each run and behaves maliciously in order to learn extra information. $\mathcal{S}'$ is not able to retrieve any valuable information about the input-output pair of $\mathcal{C}^{\mathcal{S}'}$.

We are now ready to give the formal model for secure outsourced cryptographic algorithms, which is based on principally the model of [3].

Definition 1 [3] **(Algorithm with outsource-I/O)** An algorithm Alg obeys the outsource input/output specification if it takes five inputs, and produces three outputs. The first three inputs are generated by an honest party, and are classified by how much the adversary $\mathcal{A} = (\mathcal{E}, \mathcal{S}')$ knows about them, where $\mathcal{E}$ is the adversarial environment that submits maliciously chosen inputs to Alg, and $\mathcal{S}'$ is the adversarial software operating in place of oracle $\mathcal{S}$.

1. 1st is the honest secret input, which is unknown to both $\mathcal{E}$ and $\mathcal{S}'$,
2. 2nd is the honest protected input, which may be known by $\mathcal{E}$, but is protected from $\mathcal{S}'$,

3. 3rd is the honest unprotected input, which may be known by both $\mathcal{E}$ and $\mathcal{S}$,
4. 4th is the the adversarial protected input which is known to $\mathcal{E}$, but protected from $\mathcal{S}'$,
5. 5th is the the adversarial unprotected input, which may be known by $\mathcal{E}$ and $\mathcal{S}$,
6. 1st is the secret input which is unknown to both $\mathcal{E}$ and $\mathcal{S}'$,
7. 2nd is the protected input which may be known to $\mathcal{E}$, but not $\mathcal{S}'$,
8. 3rd is the unprotected input which may be known by both parties of $\mathcal{A}$.

Outsource-security means that if a malicious $\mathcal{S}'$ can obtain some information about the secret of $\mathcal{C}^{\mathcal{S}}$ by playing the role of $\mathcal{C}$ instead of $\mathcal{S}$, then $\mathcal{S}'$ can also obtain it without following this procedure. More concretely, when $\mathcal{C}^{\mathcal{S}}(x)$ is queried, a simulator $\mathsf{Sim}_{\mathcal{S}'}$ is constructed in such a way that without the knowledge of the secret or protected inputs of x, the view of $\mathcal{S}'$ can be simulated. In the following outsource-security definition, it is guaranteed that the malicious environment $\mathcal{E}$ cannot learn any valuable information about the secret inputs and outputs of $\mathcal{C}^{\mathcal{S}}$ (even in the case that $\mathcal{C}$ runs the malicious software $\mathcal{S}'$ developed by $\mathcal{E}$).

Definition 2 [3] **(Outsource security)** Let $\mathsf{Alg}(\cdot,\cdot,\cdot,\cdot,\cdot)$ be an algorithm with outsource-I/O. A pair of algorithms $(\mathcal{C},\mathcal{S})$ is said to be an outsource-secure implementation of Alg if:
Correctness: $\mathcal{C}^{\mathcal{S}}$ is a correct implementation of Alg.
Security: For all probabilistic polynomial-time adversaries $\mathcal{A}=(\mathcal{E},\mathcal{S}')$, there exist probabilistic expected polynomial-time simulators $(\mathsf{Sim}_{\mathcal{E}},\mathsf{Sim}_{\mathcal{S}'})$ such that the following pairs of random variables are computationally indistinguishable.

- Pair One. $\mathsf{EVIEW}_{\mathsf{real}} \sim \mathsf{EVIEW}_{\mathsf{ideal}}$
  - The real process:
    $\mathsf{EVIEW}^i_{\mathsf{real}} = \{(\mathsf{istate}^i, x^i_{hs}, x^i_{hp}, x^i_{hu}) \leftarrow I(1^k, \mathsf{istate}^{i-1});$
    $(\mathsf{estate}, j^i, x^i_{ap}, x^i_{au}, \mathsf{stop}^i) \leftarrow \mathcal{E}(1^k, \mathsf{EVIEW}^{i-1}_{\mathsf{real}}, x^i_{hp}, x^i_{hu});$ $(\mathsf{tstate}^i, \mathsf{ustate}^i,$
    $y^i_s,\ y^i_p,\ y^i_u) \leftarrow \mathcal{C}^{\mathcal{S}'(\mathsf{ustate}^{i-1})}(\mathsf{tstate}^{i-1}, x^{j^i}_{hs}, x^{j^i}_{hp}, x^{j^i}_{hu}, x^i_{ap}, x^i_{au}):$
    $(\mathsf{estate}^i, y^i_p, y^i_u)\}$
    $\mathsf{EVIEW}_{\mathsf{real}} = \mathsf{EVIEW}^i_{\mathsf{real}}$ if $\mathsf{stop}^i = \mathsf{TRUE}$.
    The real process proceeds in rounds. In round i, the honest (secret, protected, and unprotected) inputs $(x^i_{hs}, x^i_{hp}, x^i_{hu})$ are picked using an honest, stateful process I to which the environment $\mathcal{E}$ does not have access. Then $\mathcal{E}$, based on its view from the last round,
    1. chooses the value of its estate_i variable as a way of remembering what it did next time it is invoked;
    2. which previously generated honest inputs $(x^i_{hs}, x^i_{hp}, x^i_{hu})$ to give to $\mathcal{C}^{\mathcal{S}'}$ (note that $\mathcal{E}$ can specify the index j^i of these inputs, but not their values);
    3. the adversarial protected input x^i_{ap};

4. the adversarial unprotected input x^i_{au};
5. the Boolean variable stop^i that determines whether round i is the last round in this process.

Next, the algorithm $\mathcal{C}^{\mathcal{S}'}$ is run on the inputs (tstate^{i-1}, $x^{j^i}_{hs}$, $x^{j^i}_{hp}$, $x^{j^i}_{hu}$, x^i_{ap}, x^i_{au}), where tstate^{i-1} is $\mathcal{C}$'s previously saved state, and produces a new state tstate^i for $\mathcal{C}$, as well as the secret y^i_s, protected y^i_p and unprotected y^i_u outputs. The oracle $\mathcal{S}'$ is given its previously saved state, ustate^{i-1}, as input, and the current state of $\mathcal{S}'$ is saved in the variable ustate^i. The view of the real process in round i consists of estate^i, and the values y^i_p and y^i_u. The overall view of $\mathcal{E}$ in the real process is just its view in the last round (i.e., i for which $\mathsf{stop}^i = \mathsf{TRUE}$.).

- The ideal process:
  $\mathsf{EVIEW}^i_{\mathsf{ideal}} = \{(\mathsf{istate}^i, x^i_{hs}, x^i_{hp}, x^i_{hu}) \leftarrow I(1^k, \mathsf{istate}^{i-1});$
  $(\mathsf{estate}, j^i, x^i_{ap}, x^i_{au}, \mathsf{stop}^i) \leftarrow E(1^k, \mathsf{EVIEW}^{i-1}_{\mathsf{ideal}}, x^i_{hp}, x^i_{hu});$
  $(\mathsf{astate}^i, y^i_s, y^i_p, y^i_u) \leftarrow \mathsf{Alg}(\mathsf{astate}^{i-1}, x^{j^i}_{hs}, x^{j^i}_{hp}, x^{j^i}_{hu}, x^i_{ap}, x^i_{au});$
  $(\mathsf{sstate}^i, \mathsf{ustate}^i, Y^i_p, Y^i_u, \mathsf{rep}^{\mathsf{i}}) \leftarrow$
  $\mathsf{Sim}^{\mathcal{S}'(\mathsf{ustate}^{i-1})}_{\mathcal{E}}$ $(\mathsf{sstate}^{i-1}, x^{j^i}_{hp}, x^{j^i}_{hu}, x^i_{ap}, x^i_{au}, y^i_p$
  $, y^i_u)$; $(z^i_p, z^i_u) = \mathsf{rep}^{\mathsf{i}}(Y^i_p, Y^i_u) + (1 - \mathsf{rep}^{\mathsf{i}})(y^i_p, y^i_u) : (\mathsf{estate}, z^i_p, z^i_u)\}$
  $\mathsf{EVIEW}_{\mathsf{ideal}} = \mathsf{EVIEW}^i_{\mathsf{ideal}}$ if $\mathsf{stop}^i = \mathsf{TRUE}$.
  The ideal process also proceeds in rounds. In the ideal process, we have a stateful simulator $\mathsf{Sim}_{\mathcal{E}}$ who, shielded from the secret input x^i_{hs}, but given the non-secret outputs that Alg produces when run all the inputs for round i, decides to either output the values (y^i_p, y^i_u) generated by Alg, or replace them with some other values (Y^i_p, Y^i_u). Note that this is captured by having the indicator variable rep^i be a bit that determines whether y^i_p will be replaced with Y^i_p. In doing so, it is allowed to query oracle $\mathcal{S}'$; moreover, $\mathcal{S}'$ saves its state as in the real experiment.

- Pair Two. $\mathsf{EVIEW}_{\mathsf{real}} \sim \mathsf{EVIEW}_{\mathsf{ideal}}$
  - The view that the untrusted software $\mathcal{S}'$ obtains by participating in the real process described in Pair One. $\mathsf{UVIEW}_{\mathsf{real}} = \mathsf{ustate}^i$ if $\mathsf{stop}^i = \mathsf{TRUE}$.
  - The ideal process:
    $\mathsf{UVIEW}^i_{ideal} = \{\ (\mathsf{istate}^i, x^i_{hs}, x^i_{hp}, x^i_{hu}) \leftarrow I(1^k, \mathsf{istate}^{i-1});$
    $(\mathsf{estate}^i, j^i, x^i_{ap}, x^i_{au}, \mathsf{stop}^i) \leftarrow \mathcal{E}(1^k, \mathsf{estate}^{i-1}, x^i_{hp}, x^i_{hu}, y^{i-1}_p,\ y^{i-1}_u;$
    $(\mathsf{astate}^i, y^i_s, y^i_p, y^i_u) \leftarrow \mathsf{Alg}(\mathsf{astate}^{i-1}, x^{j^i}_{hs}, x^{j^i}_{hp}, x^{j^i}_{hu}, x^i_{ap}, x^i_{au});$
    $(\mathsf{sstate}^i, \mathsf{ustate}^i) \leftarrow \mathsf{Sim}^{\mathcal{S}'(\mathsf{ustate}^{i-1})}_{\mathcal{S}'}$ $(\mathsf{sstate}^{i-1}, x^{j^i}_{hu}, x^i_{au}) : (\mathsf{ustate}^i)\ \}$

    $\mathsf{EVIEW}_{\mathsf{ideal}} = \mathsf{EVIEW}^i_{\mathsf{ideal}}$ if $\mathsf{stop}^i = \mathsf{TRUE}$.

In the ideal process, we have a stateful simulator $\mathsf{Sim}_{\mathcal{S}'}$ who, equipped with only the unprotected inputs (x^i_{hu}, x^i_{au}), queries $\mathcal{S}'$. As before, $\mathcal{S}'$ may maintain state.

Definition 3 [3] (**α-efficient, secure outsourcing**) A pair of algorithms $(\mathcal{C}, \mathcal{S})$ is said to be an α-efficient implementation of Alg if

1. $\mathcal{C}^{\mathcal{S}}$ is a correct implementation of Alg and
2. $\forall$ inputs x, the running time of $\mathcal{C}$ is no more than an α-multiplicative factor of the running time of Alg.

Definition 4 [3] (**β-checkable, secure outsourcing**) A pair of algorithms $(\mathcal{C}, \mathcal{S})$ is said to be an β-checkable implementation of Alg if

1. $\mathcal{C}^{\mathcal{S}}$ is a correct implementation of Alg and
2. $\forall$ inputs x, if $\mathcal{S}$ deviates from its advertised functionality during the execution of $\mathcal{C}^{\mathcal{S}'}(x)$, $\mathcal{C}$ will detect the error with probability no less than β.

Definition 5 [3] (**(α, β)-outsource-security**) A pair of algorithms $(\mathcal{C}, \mathcal{S})$ is said to be an (α, β)-outsource-secure implementation of Alg if it is both α-efficient and β-checkable.

3 Main Algorithm for Modular Exponentiation (Private-Base & Private-Exponent)

3.1 Preliminaries

There are basically two different settings for which modular exponentiations are the most expensive parts of the cryptographic computation: the discrete logarithm problem (DLP) and the RSA problem. In both cases, we summarize the following conditions to obtain mathematical problem instances which are intractable enough to obtain the desired level of security for the corresponding cryptographic schemes.

3.1.1 DLP case

Let p and q be prime numbers and $\mathbb{G} \subseteq \mathbb{F}_p^*$ be a subgroup generated by a primitive element g of order q. In order to have an intractable DLP on $\mathbb{G}$, we impose the usual conditions on the number of distinct cosets in $\mathbb{F}_p^*/\mathbb{G}$ being comparably small, i.e. we have a small cofactor $c = \frac{p-1}{q}$ (since otherwise by Chinese Remainder Theorem (Pohling-Hellman reduction) the complexity of DLP reduces to much smaller groups leading to less secure group based cryptographic systems [36]). This means that we need to hide the exponent of the exponentiation but not necessarily the base for the security of the encryption algorithms. On the other hand, hiding the base element in the modular exponentiation realizes the privacy preserved applications.

We restrict ourselves to the multiplicative subgroup of prime field case $\mathbb{G} \leq \mathbb{F}_p^*$, although it is also possible to use prime order multiplicative subgroups of the extension fields of $\mathbb{F}_p$. The main reason of our restriction is

that the recent quasi-polynomial attacks on DLP of certain extension fields of small characteristics suggests not to use non-prime finite fields in cryptographic setting [37]. We note that all secure outsourcing algorithms for modular exponentiation (including the algorithms proposed in this paper) can easily be adapted to secure outsourcing algorithms for scalar multiplication of elliptic curve cryptography (ECC) by using a prime order subgroup of $E(\mathbb{F}_p)$ instead of the group $\mathbb{G}$. Using these algorithms for scalar multiplications of elliptic curves, one can also obtain hybrid privacy preserving outsourcing algorithms for pairing-based cryptosystems by means of outsourcing private inputs of pairing functions, bilinearity property, and private exponentiations in finite fields for the realization of ID-based cryptography [36].

3.1.2 RSA case

In this case, we have the modulus $n = p \cdot q$, where p and q are distinct large prime numbers. Since RSA based systems rely on the arithmetic of $\mathbb{G} := (\mathbb{Z}/n\mathbb{Z})^*$, we have an exponent ranging 0 to $(p-1)(q-1)-1$. For public key encryptions, the message must be private but the public key can be disclosed to the server (similarly, for the signatures only the private key is kept private). However, similar to the DLP case, hiding exponents or base elements in the modular exponentiations enables to obtain privacy preserving outsourced schemes. Constructing such a system makes impossible for the server to distinguish between encryption/decryption/signature/verification processes which can be an important design criteria for privacy preserving infrastructures (e.g., attribute-based encryption schemes). To the best of our knowledge, there is only one algorithm proposed for RSA based modular exponentiation [22], for which the checkability fails as explained in Section 1.1. Hence, our algorithm unifies modular exponentiation modulo a prime or a composite number for the first time.

For real-life applications, the group order m is typically chosen as a 2048-bit number for RSA or DLP based systems and as a 384-bit number for ECC based systems.

3.2 The Main Algorithm

In this section, we propose our main algorithm for modular exponentiation modulo n with the underlying group $\mathbb{G}$, which is either the subgroup of $\mathbb{F}_n^*$ or $(\mathbb{Z}/n\mathbb{Z})^*$ of order m. Note that n can be either a prime number or an RSA modulus covering the both cases as described above.

Client is willing to compute $u^a \bmod n$ privately, where $u \in \mathbb{G}$ and $a \in \{0, \cdots, m-1\}$ private inputs and n is public. More precisely, the algorithm has inputs u, a and n and outputs $u^a \bmod n$ without explicitly giving the values of u, a, and u^a to the server.

Let now the blinding factors (x, g^x, g^{-x}), (t_1, g^{t_1}, g^{-t_1}), $(t_2, g^{t_2}, g^{-t_2}) \in \mathbb{Z}/m\mathbb{Z} \times \mathbb{G}^2$ and (y, g^y), $(s, g^s) \in \mathbb{Z}/m\mathbb{Z} \times \mathbb{G}$ be given. Note that except g^{-t_1}

and g^{-t_2} these values can be computed using a Rand Algorithm as defined in [3]. For taking the inverse of g^{t_1} and g^{t_2}, one can either extend the Rand algorithm or they can be computed offline. Similarly, in the case of RSA setting, all these values can be computed by extending the Rand algorithm or by computing them offline. Furthermore, these values are stored on the client side, and one can argue that this is dangerous for the RSA setting if the client is a lower power device because of low-power devices are not generally temper-resistant. However, these values are stored in the same protected area as all other private keys and any attack to the values can also apply to the private keys.

Note that these blinding factors are precomputed in order to speed up computations [2, 3]. The values x, y, t_1 and t_2 can be used several times for different exponents whereas the value s should be used only once.

Furthermore, we abbreviate by $\mathcal{C}$ the Client and by $\mathcal{S}$ the Server. We have also the assumption that $\mathcal{C}$ can run an algorithm to query u^a to $\mathcal{S}$. We denote the output of such a query by $\mathsf{Exp}(a, u)$. Before we explain our main algorithm we propose the following subalgorithm SubAlg for outsourcing g^z, where g is a generator of the group $\mathbb{G}$ and z is a random group element. Note that g and z are not necessarily private for SubAlg, and it is only crucial to check the correctness of the result g^z. The client $\mathcal{C}$ has inputs g, z and c, where c is a checkability parameter.

During the precomputation phase, the Rand algorithm prepares $(s, g^s) \in \mathbb{Z}/m\mathbb{Z} \times \mathbb{G}$, (t_1, t_1^{-1}, g^{t_1}), $(t_2, t_2^{-1}, g^{t_2}) \in_R (\mathbb{Z}/m\mathbb{Z})^2 \times \mathbb{G}$, and $I^{-1} = \{1^{-1}, \cdots, c^{-1}\} \subseteq \mathbb{Z}/m\mathbb{Z}$, where $I=\{1, \cdots, c\} \subseteq \mathbb{Z}/m\mathbb{Z}$, and sends these values to $\mathcal{C}$. At the first phase, $\mathcal{C}$ picks random elements c_1^{-1}, $c_2^{-1} \in_R I^{-1}$ with $c_1, c_2 \in I,\ c_1 \neq c_2$, and computes $z_1 = (z - s) \cdot c_1^{-1}$ and $z_2 = (-z + 2s) \cdot c_2^{-1}$. Next, $\mathcal{C}$ runs $Z_1 = \mathsf{Exp}(z_1 \cdot {t_1}^{-1}, g^{t_1})$ and $Z_2 = \mathsf{Exp}(z_2 \cdot {t_2}^{-1}, g^{t_2})$. Finally, $\mathcal{C}$ verifies $Z_1^{c_1} \cdot Z_2^{c_2} \stackrel{?}{=} g^s$, and returns $Z_1^{2c_1} \cdot Z_2^{c_2}$. We would like to highlight that computing t_1, t_1^{-1} and t_2, t_2^{-1} does not cause a security issue in the RSA setting [1] because these values are masked before they are sent to $\mathcal{S}$ and they are never revealed to $\mathcal{S}$.

We would like to emphasize that c is chosen to be a very small positive integer. For example, if c=4 then the checkability becomes 11/12 that is already sufficient to be able to compare with the existing schemes. In particular, since c_1 and c_2 are very small integers, the values $Z_1^{c_1}$ or $Z_2^{c_2}$ can be computed by only few multiplications (for $c = 4$ we have at most 2 multiplications instead of approximately 3000 multiplications for 2048 bit security level.). Therefore, it does not add any considerable complexity (computational, storage and communication) as demonstrated in Section 5.

SubAlgorithm (SubAlg): Outsourcing an auxiliary modular exponen-

[1] Note that disclosing both x and x^{-1} mod $\phi(n)$ for $x \in \mathbb{Z}/n\mathbb{Z}$ allows an attacker to recover the factorization of the RSA modulus n.

tiation

Input: (z, g, c) (where $z \in \mathbb{Z}/m\mathbb{Z}$ with $< g >= \mathbb{G} \leq \mathbb{F}_n^*$ for DLP or $g \in_R \mathbb{G} = (\mathbb{Z}/n\mathbb{Z})^*$ for RSA with $|\mathbb{G}| = m$, where $n, m \in \mathbb{N}$, and an arbitrary small $c \in \mathbb{N}$). (*Note that $c = 4$ is sufficient for our experiments to compare with the existing schemes.*)
Output: The value g^z in $\mathbb{G}$.
Precomputation: A Rand algorithm computes and stores the following values for $\mathcal{C}$:

- $(s, g^s) \in \mathbb{Z}/m\mathbb{Z} \times \mathbb{G}$,
- (t_1, t_1^{-1}, g^{t_1}), $(t_2, t_2^{-1}, g^{t_2}) \in_R (\mathbb{Z}/m\mathbb{Z})^2 \times \mathbb{G}$,
- $I=\{1, \cdots, c\} \subseteq \mathbb{Z}/m\mathbb{Z}$ with $I^{-1} = \{1^{-1}, \cdots, c^{-1}\} \subseteq \mathbb{Z}/m\mathbb{Z}$.

1. $\mathcal{C}$ picks random elements c_1^{-1}, $c_2^{-1} \in_R I^{-1}$ with $c_1, c_2 \in I$, $c_1 \neq c_2$, and computes $z_1 \leftarrow (z - s) \cdot c_1^{-1}$ and $z_2 \leftarrow (-z + 2s) \cdot c_2^{-1}$.
2. $\mathcal{C}$ runs
   (a) $Z_1 \leftarrow \mathsf{Exp}(z_1 \cdot {t_1}^{-1}, g^{t_1})$.
   (b) $Z_2 \leftarrow \mathsf{Exp}(z_2 \cdot {t_2}^{-1}, g^{t_2})$.
3. $\mathcal{C}$ verifies $Z_1^{c_1} \cdot Z_2^{c_2} \stackrel{?}{=} g^s$ and returns $Z_1^{2c_1} \cdot Z_2^{c_2}$.

Theorem 1 SubAlg *terminates and outputs correctly with probability* $\frac{1}{c(c-1)}$.

Proof The termination simply follows from the algorithm specification. Namely, $\mathcal{C}$ outsources the computations $\mathsf{Exp}(z_1 \cdot {t_1}^{-1}, g^{t_1})$ and $\mathsf{Exp}(z_2 \cdot {t_2}^{-1}, g^{t_2})$, and obtains Z_1 and Z_2, respectively. If $\mathcal{S}$ does not respond within specific time interval $\mathcal{C}$ aborts and outputs fail. Otherwise, $\mathcal{C}$ verifies and outputs the result. More precisely, $\mathcal{C}$ first computes $z_1 = (z - s) \cdot c_1^{-1}$ and $z_2 = (-z + 2s) \cdot c_2^{-1}$, where c_1^{-1}, $c_2^{-1} \in_R I^{-1}$ with $c_1, c_2 \in I$, $c_1 \neq c_2$.

$\mathcal{S}$ returns $Z_1 = g^{z_1} = g^{(z-s)\cdot c_1^{-1}}$ and $Z_2 = g^{z_2} = g^{(-z+2s)\cdot c_2^{-1}}$. Finally, $\mathcal{C}$ computes and verifies the following result.

$$\begin{aligned} Z_1^{c_1} \cdot Z_2^{c_2} &= (g^{(z-s)\cdot c_1^{-1}})^{c_1} \cdot (g^{(-z+2s)\cdot c_2^{-1}})^{c_2} \\ &= g^{(z-s)+(-z+2s)} = g^s \end{aligned}$$

If the equality does not hold then algorithm outputs checkability failure. Finally, $\mathcal{C}$ outputs

$$\begin{aligned} Z_1^{2c_1} \cdot Z_2^{c_2} &= (g^{(z-s)\cdot c_1^{-1}})^{2c_1} \cdot (g^{(-z+2s)\cdot c_2^{-1}})^{c_2} \\ &= g^{2(z-s)+(-z+2s)} = g^z \end{aligned}$$

To prove that a malicious $\mathcal{S}$ cannot maliciously behave without being detected with probability $\frac{1}{c(c-1)}$ because $\mathcal{S}$ does not have any knowledge about z, s, c_1 and z_1 (and c_2 and z_2). More concretely, in order to validate the correctness $\mathcal{C}$ verifies the equality $Z_1^{c_1} \cdot Z_2^{c_2} \stackrel{?}{=} g^s$. Since c_1, c_2 and g^s are unknown

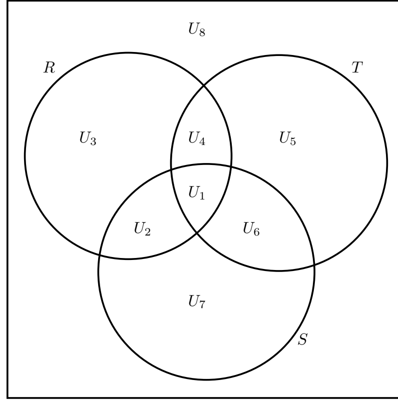

Fig. 1 The Partition of the Set $U = \cup_{i=1}^{8} U_i$

to $\mathcal{S}$, the only way for $\mathcal{S}$ to be successful is to guess c_1 and c_2 correctly. Note that the probability of guessing the correct value of c_1 is $1/c$, and once c_1 is chosen we have the probability $1/(c-1)$ for guessing the correct value of c_2. Hence, the overall probability becomes $\frac{1}{c(c-1)}$. □

We now propose our main algorithm Algorithm 1 ($\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$) for private-base and private-exponent. For completeness, we introduce the following notation: Let a finite set $M = \{m_1, \cdots, m_n\}$ be given. We denote by $\mathbb{S}_n(M)$ the group of permutations on M. Note that we can identify any permutation on $\mathbb{S}_n(M)$ with a permutation on $\mathbb{S}_n(\{1, \cdots, n\})$. By abuse of notation, we will write $\sigma(m_i) = \sigma(i)$ for any $\sigma \in \mathbb{S}_n(\{1, \cdots, n\})$.

Before we go into the details we give a brief summary of $\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$ as follows. The client $\mathcal{C}$ first masks the base u and the exponent a, respectively, and sends them to the server in a special form (based on the precomputed values). The server applies the algorithm specifications and returns the masked results. The client then removes the masks and verifies the correctness of the result. More precisely, for the goal of computing u^a we first precompute $v = g^x$, $w = ug^{-x}$, $\mu = g^y$, $Z = g^{ax-y}$, where x and y are randomly chosen. Then u^a is converted into $(vw)^a = g^{xa}\ w^a = \mu g^z w^a = \mu Z w^a$ such that $w = uv^{-1}$, $z = ax - y$, where v, w, Z looks random and is independent of u and a in the view of the attacker. Therefore, the algorithm has basically three computations in order to compute u^a, i.e., μ, Z and w^a.

- The first value $\mu = g^y$ is already precomputed and stored.
- The second value $Z = g^z$ is computed via the subalgorithm SubAlg for computing a modular exponentiation for a generator g and an exponent $z = ax-y$. We highlight that this subalgorithm only assures the correctness of the result rather than hiding the base g and the exponent z. Note that

z is already masked with x and y therefore does not leak any information to $\mathcal{S}$.

- Finally, w^a is outsourced securely which is the longest and the most complicated part. This value is outsourced by first dividing the private exponent a and a random value r into k and ℓ subcomponents such that $a = \sum_{i=1}^{k} a_i$ and $r = \sum_{i=1}^{\ell} r_i$, respectively. More precisely,
  - $\mathcal{C}$ computes $w \leftarrow uv^{-1}$, $z \leftarrow ax - y$, runs $Z = \mathsf{SubAlg}(z, g, c)$.
  - For the sign of the values $\mathcal{C}$ chooses further a random $\alpha = (\alpha_1, \cdots, \alpha_{\ell+k}) \in_R \{0,1\}^{\ell+k}$.
  - Using Figure 1, $\mathcal{C}$ first chooses random subsets $U_i, i = 1, \cdots, 8$ (with arbitrary length) such that
    - $S := U_1 \cup U_2 \cup U_6 \cup U_7$, $T := U_1 \cup U_4 \cup U_5 \cup U_6$, $U_i \neq \emptyset \ \forall\ i$, $U_i \cap U_j = \emptyset \ \forall\ i \neq j$, $U := A \cup R := \cup_{i=1}^{8} U_i := \{u_1, \cdots, u_{k+\ell}\}$,
    - $s = \sum_{s_i \in S} s_i$, $t = \sum_{u_i \in T} (-1)^{\alpha_i} \cdot u_i$, $s + c_1 t = c_2$, where $c_1, c_2 \in_R \{1, \cdots, c\}$ (the aim of this condition is to assure the checkability property of $\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$),
    - $R := U_1 \cup U_2 \cup U_3 \cup U_4 := \{r_1, \cdots, r_\ell\}$, $A := U_5 \cup U_6 \cup U_7 \cup U_8 := \{a_1, \cdots, a_k\}$,
    - $r = \sum_{i=1}^{\ell} r_i$, $a = \sum_{i=1}^{k} a_i$. Note that the elements of R are used to randomize the private exponent a.
  - $\mathcal{C}$ chooses a random permutation $\sigma \in_R \mathbb{S}_{\ell+k}(U)$ and sets the permuted elements $U = \sigma(U) := (\sigma_1, \cdots, \sigma_{k+\ell})$.
  - Note that we use a temporary value *temp* in our main algorithm for the following reason: First observe that the following values are computed during the main algorithm:
    - $\mathcal{U}_{\mp}$'s are computed for the sets $U = \cup_{i=1}^{8} U_i$
    - $\mathcal{R}_{\mp}$'s are computed for the sets $R = \cup_{i=1}^{4} U_i$
    - $\mathcal{S}_{\mp}$'s are computed for the sets $S = \cup_{i=1,2,6,7} U_i$,
    - $\mathcal{T}_{\mp}$'s are computed for the sets $T = \cup_{i=1,4,5,6} U_i$.

    Since the sets R, S, T have common and disjoint values *temp* is used to minimize modular multiplications by distributing the elements into these sets. Note that *temp* is not used

    - in the first round because $U_1 = U \cap R \cap S \cap T$. After assigning $\mathcal{U}_{\mp}$ to $\mathcal{T}_{\mp}$ *temp* will be used to compute the final $\mathcal{T}_{\mp}$.
    - in the second round because $U_2 = U \cap R \cap S$. After assigning $\mathcal{U}_{\mp}$ to $\mathcal{S}_{\mp}$ *temp* will be used to compute the final $\mathcal{S}_{\mp}$.
    - in the third round because $U_3 = R \subset U$.
  - Let $U := \{u_1, \cdots, u_{k+\ell}\}$. $\mathcal{C}$ chooses a random permutation $\sigma \in_R \mathbb{S}_{\ell+k}(U)$ and sets the permuted elements $U = \sigma(U) := (\sigma_1, \cdots, \sigma_{k+\ell})$. The permutation σ basically mixes the subcomponents of a and r to ensure the privacy of the exponent a. Moreover, the invocations take place with signed values of the subcomponents using $\alpha = (\alpha_1, \cdots, \alpha_{\ell+k}) \in_R \{0,1\}^{\ell+k}$ (i.e., $\mathcal{S}$ computes $w^{(-1)^{\alpha_{\sigma(i)}} \cdot \sigma(u_i)}$).

  - After $\mathcal{S}$ returns the computed values $\mathcal{C}$ basically computes w^{a+r}, w^r, w^s and w^t and verifies the correctness of the result w^a by checking $s+c_1t = c_2$ in the exponents.
  - If the verification is successful, $\mathcal{C}$ outputs w^a by removing w^r from w^{a+r}.
- $\mathcal{C}$ finally returns the expected outcome u^a by computing $\mu Z w^a$.

We give a toy example in Section 3.2.2 for better understanding of the algorithm. The algorithm is now given as follows.

Algorithm 1 ($\mathsf{Alg}^{\mathsf{pr}}_{\mathsf{pr}}$): Private-Base & Private-Exponent Modular Exponentiations

---

Input: (a, u, k, ℓ, c) (where $a \in \mathbb{Z}/m\mathbb{Z}$ with $u \in < g >= \mathbb{G} \leq \mathbb{F}_n^*$ for DLP or $u \in_R \mathbb{G} = (\mathbb{Z}/n\mathbb{Z})^*$ for RSA with $|\mathbb{G}| = m$, where $n, m \in \mathbb{N}$, and an arbitrary small $c \in \mathbb{N}$).
Output: The value u^a in $\mathbb{G}$.
Precomputation: A Rand algorithm computes $(y, g^y) \in_R \mathbb{Z}/m\mathbb{Z} \times \mathbb{G}$ and $(x, g^x, g^{-x}) \in \mathbb{Z}/m\mathbb{Z} \times \mathbb{G}^2$ for $\mathcal{C}$ with $v = g^x$ and $\mu = g^y$.

1. $\mathcal{C}$ computes $w \leftarrow uv^{-1}$, $z \leftarrow ax - y$, runs $Z = \mathsf{SubAlg}(z, g, c)$.
2. For the sign of the values $\mathcal{C}$ chooses further a random $\alpha = (\alpha_1, \cdots, \alpha_{\ell+k}) \in_R \{0,1\}^{\ell+k}$.
3. Using Figure 1, $\mathcal{C}$ first chooses random subsets $U_i, i = 1, \cdots, 8$ (with arbitrary length) such that
   - $S := U_1 \cup U_2 \cup U_6 \cup U_7$,
   - $T := U_1 \cup U_4 \cup U_5 \cup U_6$,
   - $U_i \neq \emptyset \ \forall\ i$, $U_i \cap U_j = \emptyset \ \forall\ i \neq j$,
   - $U := A \cup R := \cup_{i=1}^{8} U_i := \{u_1, \cdots, u_{k+\ell}\}$,
   - $s = \sum_{s_i \in S} s_i$,
   - $t = \sum_{u_i \in T} (-1)^{\alpha_i} \cdot u_i$
   - $s + c_1 t = c_2$, where $c_1, c_2 \in_R \{1, \cdots, c\}$,
   - $R := U_1 \cup U_2 \cup U_3 \cup U_4 := \{r_1, \cdots, r_\ell\}$,
   - $A := U_5 \cup U_6 \cup U_7 \cup U_8 := \{a_1, \cdots, a_k\}$,
   - $r = \sum_{i=1}^{\ell} r_i$, $a = \sum_{i=1}^{k} a_i$.
4. $\mathcal{C}$ chooses a random permutation $\sigma \in_R \mathbb{S}_{\ell+k}(U)$ and sets the permuted elements $U = \sigma(U) := (\sigma_1, \cdots, \sigma_{k+\ell})$.
5. $\mathcal{C}$ sets $\mathcal{U}_-$, $\mathcal{U}_+ \leftarrow 1$ and uses the partitions in Figure 1. Furthermore, $\mathcal{C}$ runs and computes in random order for $j \in \{1, \cdots, k + \ell\}$
   `(`$\mathcal{U}_-$`, `$\mathcal{U}_+$` are negative/positive parts of the exponents of `U`)`
   (a) If $\sigma_j \in U_1$: `(Computation of signed elements of `U_1`)`
      i. If $\alpha_{\sigma(j)} = 1$: $w_j \leftarrow \mathsf{Exp}(-\sigma_j, w)$
         A. $\mathcal{U}_- \leftarrow \mathcal{U}_- \cdot w_j$
      ii. If $\alpha_{\sigma(j)} = 0$: $w_j \leftarrow \mathsf{Exp}(\sigma_j, w)$
         A. $\mathcal{U}_+ \leftarrow \mathcal{U}_+ \cdot w_j$
      iii. $\mathcal{C}$ sets $\mathcal{T}_- \leftarrow \mathcal{U}_-$ and $\mathcal{T}_+ \leftarrow \mathcal{U}_+$

(b) If $\sigma_j \in U_2$: `(Computation of signed elements of` U_2`)`
  i. If $\alpha_{\sigma(j)} = 1$: $w_j \leftarrow \mathsf{Exp}(-\sigma_j, w)$
    A. $\mathcal{U}_- \leftarrow \mathcal{U}_- \cdot w_j$
  ii. If $\alpha_{\sigma(j)} = 0$: $w_j \leftarrow \mathsf{Exp}(\sigma_j, w)$
    A. $\mathcal{U}_+ \leftarrow \mathcal{U}_+ \cdot w_j$
  iii. $\mathcal{C}$ sets $\mathcal{S}_- \leftarrow \mathcal{U}_-$ and $\mathcal{S}_+ \leftarrow \mathcal{U}_+$

(c) If $\sigma_j \in U_3$: `(Computation of signed elements of` U_3`)`
  i. If $\alpha_{\sigma(j)} = 1$: $w_j \leftarrow \mathsf{Exp}(-\sigma_j, w)$
    A. $\mathcal{U}_- \leftarrow \mathcal{U}_- \cdot w_j$
  ii. If $\alpha_{\sigma(j)} = 0$: $w_j \leftarrow \mathsf{Exp}(\sigma_j, w)$
    A. $\mathcal{U}_+ \leftarrow \mathcal{U}_+ \cdot w_j$

(d) If $\sigma_j \in U_4$, $\mathcal{C}$ sets $temp_-$, $temp_+ \leftarrow 1$: `(Computation of signed elements of` U_4`)`
  i. If $\alpha_{\sigma(j)} = 1$: $w_j \leftarrow \mathsf{Exp}(-\sigma_j, w)$
    A. $temp_- \leftarrow temp_- \cdot w_j$
  ii. If $\alpha_{\sigma(j)} = 0$: $w_j \leftarrow \mathsf{Exp}(\sigma_j, w)$
    A. $temp_+ \leftarrow temp_+ \cdot w_j$
  iii. $\mathcal{C}$ sets $\mathcal{U}_-$, $\mathcal{R}_- \leftarrow \mathcal{U}_- \cdot temp_-$, $\mathcal{T}_- \leftarrow \mathcal{T}_- \cdot temp_-$, $\mathcal{U}_+$, $\mathcal{R}_+ \leftarrow \mathcal{U}_+ \cdot temp_+$ and $\mathcal{T}_+ \leftarrow \mathcal{T}_+ \cdot temp_+$

(e) If $\sigma_j \in U_5$, $\mathcal{C}$ sets $temp_-$, $temp_+ \leftarrow 1$: `(Computation of signed elements of` U_5`)`
  i. If $\alpha_{\sigma(j)} = 1$: $w_j \leftarrow \mathsf{Exp}(-\sigma_j, w)$
    A. $temp_- \leftarrow temp_- \cdot w_j$
  ii. If $\alpha_{\sigma(j)} = 0$: $w_j \leftarrow \mathsf{Exp}(\sigma_j, w)$
    A. $temp_+ \leftarrow temp_+ \cdot w_j$
  iii. $\mathcal{C}$ sets $\mathcal{U}_- \leftarrow \mathcal{U}_- \cdot temp_-$, $\mathcal{T}_- \leftarrow \mathcal{T}_- \cdot temp_-$, $\mathcal{U}_+ \leftarrow \mathcal{U}_+ \cdot temp_+$ and $\mathcal{T}_+ \leftarrow \mathcal{T}_+ \cdot temp_+$

(f) If $\sigma_j \in U_6$, $\mathcal{C}$ sets $temp_-$, $temp_+ \leftarrow 1$: `(Computation of signed elements of` U_6`)`
  i. If $\alpha_{\sigma(j)} = 1$: $w_j \leftarrow \mathsf{Exp}(-\sigma_j, w)$
    A. $temp_- \leftarrow temp_- \cdot w_j$
  ii. If $\alpha_{\sigma(j)} = 0$: $w_j \leftarrow \mathsf{Exp}(\sigma_j, w)$
    A. $temp_+ \leftarrow temp_+ \cdot w_j$
  iii. $\mathcal{C}$ sets $\mathcal{U}_- \leftarrow \mathcal{U}_- \cdot temp_-$, $\mathcal{T}_- \leftarrow \mathcal{T}_- \cdot temp_-$, $\mathcal{S}_- \leftarrow \mathcal{S}_- \cdot temp_-$, $\mathcal{U}_+ \leftarrow \mathcal{U}_+ \cdot temp_+$, $\mathcal{T}_+ \leftarrow \mathcal{T}_+ \cdot temp_+$ and $\mathcal{S}_+ \leftarrow \mathcal{S}_+ \cdot temp_+$.

(g) If $\sigma_j \in U_7$, $\mathcal{C}$ sets $temp_-$, $temp_+ \leftarrow 1$: `(Computation of signed elements of` U_7`)`
  i. If $\alpha_{\sigma(j)} = 1$: $w_j \leftarrow \mathsf{Exp}(-\sigma_j, w)$
    A. $temp_- \leftarrow temp_- \cdot w_j$
  ii. If $\alpha_{\sigma(j)} = 0$: $w_j \leftarrow \mathsf{Exp}(\sigma_j, w)$
    A. $temp_+ \leftarrow temp_+ \cdot w_j$
  iii. $\mathcal{C}$ sets $\mathcal{U}_- \leftarrow \mathcal{U}_- \cdot temp_-$, $\mathcal{S}_- \leftarrow \mathcal{S}_- \cdot temp_-$, $\mathcal{U}_+ \leftarrow \mathcal{U}_+ \cdot temp_+$ and $\mathcal{S}_+ \leftarrow \mathcal{S}_+ \cdot temp_+$

(h) If $\sigma_j \in U_8$: `(Computation of signed elements of` U_8`)`
  i. If $\alpha_{\sigma(j)} = 1$: $w_j \leftarrow \mathsf{Exp}(-\sigma_j, w)$

A. $\mathcal{U}_- \leftarrow \mathcal{U}_- \cdot w_j$
ii. If $\alpha_{\sigma(j)} = 0$: $w_j \leftarrow \mathsf{Exp}(\sigma_j, w)$
A. $\mathcal{U}_+ \leftarrow \mathcal{U}_+ \cdot w_j$

6. $\mathcal{C}$ verifies $\mathcal{S}_+ \cdot (\mathcal{T}_- \cdot \mathcal{T}_+)^{c_1} \stackrel{?}{=} w^{c_2} \cdot \mathcal{S}_-$
   `(Verification step by checking` $s + c_1 t = c_2$ `in the exponents)`
7. $\mathcal{C}$ returns $\mu \cdot Z \cdot (\mathcal{U}_- \cdot \mathcal{R}_+)^{-1} \cdot (\mathcal{R}_- \cdot \mathcal{U}_+)$
   `(This is the expected outcome` u^a`)`

3.2.1 Correctness and Termination.

Theorem 2 $\mathsf{Alg}^{\mathsf{pr}}_{\mathsf{pr}}$ *terminates and outputs correctly.*

Proof Precomputation and Step 1 of $\mathsf{Alg}^{\mathsf{pr}}_{\mathsf{pr}}$ imply that $u^a = (vw)^a = g^{xa} w^a = \mu g^z w^a = \mu Z w^a$, where $w = uv^{-1}$ and $z = ax - y$.

We set $\Theta = \{\alpha_{\sigma(j)} : \ \alpha_{\sigma(j)} = 1, j = 1, \cdots, k + \ell\}$. Let the u_i^- (reps. u_i^+) denotes the sum of the negatively (reps. positively) signed part of the elements of U_i for all $i \in \{1, \cdots, 8\}$. In Step 7 part (a) with using the query results of $\mathcal{S}$, $\mathcal{C}$ computes the negative part $w^{u_1^-}$ and the positive part $w^{u_1^+}$ as follows:

$$w^{u_1^-} = \prod_{\substack{\alpha_{\sigma(i)} \in \Theta, \\ \sigma_i \in U_1}} w^{-\sigma_i} \text{ and } w^{u_1^+} = \prod_{\substack{\alpha_{\sigma(i)} \notin \Theta, \\ \sigma_i \in U_1}} w^{\sigma_i}.$$

The output will be assigned to the negative part $\mathcal{T}_-$ and the positive part $\mathcal{T}_+$ in the exponent the elements of T. Analogously, $\mathcal{C}$ computes in steps (b) and (c) the corresponding negative parts and the positive parts, and assigns the output to the exponent elements of the contributed sets.

Different from the steps (a), (b) and (c), $\mathcal{C}$ computes in step (d) the negative part $w^{u_4^-}$ and the positive part $w^{u_4^+}$:

$$w^{u_4^-} = \prod_{\substack{\alpha_{\sigma(i)} \in \Theta, \\ \sigma_i \in U_4}} w^{-\sigma_i} \text{ and } w^{u_4^+} = \prod_{\substack{\alpha_{\sigma(i)} \notin \Theta, \\ \sigma_i \in U_4}} w^{\sigma_i}.$$

The output will be multiplied in this case with the negative parts $\mathcal{U}_-$, $\mathcal{T}_-$ and the positive parts $\mathcal{U}_+$, $\mathcal{T}_+$ in the exponent elements of U and T. Analogously, $\mathcal{C}$ computes $w^{u_j^-}$ and $w^{u_j^+}$ for $j = 5, 6, 7, 8$ and multiply with the corresponding positive and negative parts in the exponent elements of the contributed sets. As a result, one obtains:

$$w^r = \left(\prod_{\substack{\alpha_{\sigma(i)} \in \Theta, \\ \sigma_i \in R}} w^{-\sigma_i} \right)^{-1} \cdot \prod_{\substack{\alpha_{\sigma(i)} \notin \Theta, \\ \sigma_i \in R}} w^{\sigma_i} = \mathcal{R}_-^{-1} \cdot \mathcal{R}_+,$$

$$w^s = \left(\prod_{\substack{\alpha_{\sigma(i)} \in \Theta, \\ \sigma_i \in S}} w^{-\sigma_i} \right)^{-1} \cdot \prod_{\substack{\alpha_{\sigma(i)} \notin \Theta, \\ \sigma_i \in S}} w^{\sigma_i} = \mathcal{S}_-^{-1} \cdot \mathcal{S}_+.$$

Using the definition of t, we obtain also

$$w^t = \prod_{\sigma_i \in T} w^{(-1)^{\alpha_{\sigma(i)}} \sigma_i} = \prod_{\substack{\alpha_{\sigma(i)} \in \Theta, \\ \sigma_i \in T}} w^{-\sigma_i} \cdot \prod_{\substack{\alpha_{\sigma(i)} \notin \Theta, \\ \sigma_i \in T}} w^{\sigma_i} = \mathcal{T}_- \cdot \mathcal{T}_+,$$

Together with steps (a) to (g) and step (h), we obtain

$$w^{a+r} == \left(\prod_{\substack{\alpha_{\sigma(i)} \in \Theta, \\ \sigma_i \in A \cup R}} w^{-\sigma_i} \right)^{-1} \cdot \prod_{\substack{\alpha_{\sigma(i)} \notin \Theta, \\ \sigma_i \in A \cup R}} w^{\sigma_i} = \mathcal{U}_-^{-1} \cdot \mathcal{U}_+,$$

$$w^{c_2} = w^{s+c_1 t} = w^s \cdot (w^t)^{c_1} = \mathcal{S}_-^{-1} \cdot \mathcal{S}_+ \cdot (\mathcal{T}_- \cdot \mathcal{T}_+)^{c_1}, \text{ hence,}$$

$$\mathcal{S}_+ \cdot (\mathcal{T}_- \cdot \mathcal{T}_+)^{c_1} = w^{c_2} \cdot \mathcal{S}_-.$$

If the equality does not hold then the checkability fails. If $\mathcal{S}$ runs the query algorithm properly then the algorithm ends with Step 7 as follows:

$$\begin{aligned}
\mu \cdot Z \cdot (\mathcal{U}_- \cdot \mathcal{R}_+)^{-1} \cdot (\mathcal{R}_- \cdot \mathcal{U}_+) &= \mu \cdot Z \cdot \mathcal{U}_-^{-1} \cdot \mathcal{U}_+ \cdot \mathcal{R}_- \cdot \mathcal{R}_+^{-1} \\
&= \mu \cdot Z \cdot \mathcal{U}_-^{-1} \cdot \mathcal{U}_+ \cdot (\mathcal{R}_-^{-1} \cdot \mathcal{R}_+)^{-1} \\
&= g^y \cdot g^{ax-y} \cdot w^{r+a} \cdot (w^r)^{-1} \\
&= g^y \cdot g^{ax-y} \cdot w^{r+a} \cdot w^{-r} \\
&= g^{ax} \cdot w^a \\
&= (g^x \cdot w)^a = (v \cdot w)^a \\
&= u^a.
\end{aligned}$$

□

3.2.2 A Toy Example.

We illustrate with a toy example our main algorithm Alg_{pr}^{pr} for better understanding. Note that this example is just to explain our algorithm in a simple setting. For real-life applications, the group size should be at least 1024 bit for DLP or RSA and 160 bit for the elliptic curve DLP. In particular, the computational advantage of our algorithms increases with the key sizes (see Section 5). Let $p = 103$ be given with the primitive element $g = 3$. Hence, $\mathbb{G} =< 3 >$ with $|\mathbb{G}| = p - 1 = 102$. Furthermore, assume that $(a, u, k, \ell, c)=$ $(72, 37, 4, 4, 4)$ are given as an input of Alg_{pr}^{pr}. We want to compute $u^a \equiv 37^{72}$

mod 103.

Precomputation:: If we choose $x = 59, y = 23$ then we have $(x, g^x, g^{-x}) = (59, 31, 10)$ and (y, g^y)=$(23, 95)$. Moreover, we have $v = g^x = 31$ and $v^{-1} = g^{-x} = 10$. Note that $\mu = g^y = 95$.

First of all, $w \equiv u \cdot v^{-1} \equiv 37 \cdot 10 \mod 103 = 61$. We also have $z \equiv a \cdot x - y \equiv 72 \cdot 59 - 23 \equiv 43 \mod 102$. Next, run $\mathsf{SubAlg}(43, 3, 4)$ which outputs $3^{43} \equiv 10 \mod 103$ (i.e., we have $Z \equiv g^z \equiv 10 \mod 103$).

Hence, we have $37^{72} = \mu \cdot Z \cdot 61^{72} = 95 \cdot 10 \cdot 61^{72}$. We now need to outsource 61^{72} securely. Since $k = \ell = 4$ we have 8 non-empty sets that we choose randomly with the conditions of steps 2, 3, 4 and 5. $U_1 = \{u_1 = 13\}$, $U_2 = \{u_2 = 79\}$, $U_3 = \{u_3 = 19\}$, $U_4 = \{u_4 = 93\}$, $U_5 = \{u_5 = 82\}$, $U_6 = \{u_6 = 42\}$, $U_7 = \{u_7 = 57\}$, $U_8 = \{u_8 = 95\}$. Note that $U = \{u_1, u_2, u_3, u_4, u_5, u_6, u_7, u_8\}$.

We now choose $\alpha = (10011010)$ as in Step 4. Then, the signed U becomes U_{Signed}=$\{ -u_1, u_2, u_3, -u_4, -u_5, u_6, -u_7, u_8 \}$. Let also $s = u_1 + u_2 + u_6 + u_7 = 13+79+42+57 \equiv 89 \mod 102$ and $t \equiv -u_1 - u_4 - u_5 + u_6 = -13-93-82+42 \equiv 58 \mod 102$.

Now, for $c_1 = 2$ and $c_2 = 1$ we have $89 + 2 \cdot 58 \equiv 1 \mod 102$ (because $s + c_1 \cdot t \equiv c_2 \mod p - 1$).

Now, we have $U = \{13, 79, 19, 93, 82, 42, 57, 95\}$ and U_{Signed}= $\{-13,$ 79, 19, -93, -82, 42, -57, 95 $\}$ = $\{89, 79, 19, 9, 20, 42, 45, 95\}$ (modulo 102).

Furthermore, we choose $\sigma = \left(\begin{smallmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 4 & 5 & 2 & 7 & 1 & 8 & 6 & 3 \end{smallmatrix}\right)$.Then, $\sigma(U_{Signed}) = \{ 9, 20, 79, 45, 89, 95, 42, 19 \}$. Now calculating each single substep of Step 7 we obtain the following:

$$
\begin{aligned}
w^{-u_1} &\equiv 61^{89} \equiv 66 \mod 103 \\
w^{u_2} &\equiv 61^{79} \equiv 100 \mod 103 \\
w^{u_3} &\equiv 61^{19} \equiv 13 \mod 103 \\
w^{-u_4} &\equiv 61^{9} \equiv 79 \mod 103 \\
w^{-u_5} &\equiv 61^{20} \equiv 72 \mod 103 \\
w^{u_6} &\equiv 61^{42} \equiv 30 \mod 103 \\
w^{-u_7} &\equiv 61^{45} \equiv 100 \mod 103 \\
w^{u_8} &\equiv 61^{95} \equiv 81 \mod 103
\end{aligned}
$$

At Step 7, we obtain

$$\mathcal{R}_- \equiv w^{-u_1} \cdot w^{u_4} \equiv 66 \cdot 79 \equiv 64 \mod 103$$
$$\mathcal{R}_+ \equiv w^{u_2} \cdot w^{u_3} \equiv 100 \cdot 13 \equiv 64 \mod 103$$
$$\mathcal{S}_- \equiv w^{-u_1} \cdot w^{-u_7} \equiv 66 \cdot 100 \equiv 8 \mod 103$$
$$\mathcal{S}_+ \equiv w^{-u_2} \cdot w^{u_6} \equiv 100 \cdot 30 \equiv 13 \mod 103$$
$$\mathcal{T}_- \equiv w^{-u_1} \cdot w^{u_4} \cdot w^{u_5} \equiv 66 \cdot 79 \cdot 72 \equiv 76 \mod 103$$
$$\mathcal{T}_+ \equiv w^{u_6} \equiv 30 \mod 103.$$

Now at Step 8 we check whether $\mathcal{S}_+ \cdot (\mathcal{T}_- \cdot \mathcal{T}_+)^2 \stackrel{?}{=} w \cdot \mathcal{S}_-$. In fact, $\mathcal{S}_+ \cdot (\mathcal{T}_- \cdot \mathcal{T}_+)^2 \equiv 13 \cdot (76 \cdot 30)^2 \equiv 76 \mod 103$ and $w \cdot \mathcal{S}_- \equiv 61 \cdot 8 \equiv 76 \mod 103$.

Step 9 finally computes the outcome as follows: $\mathcal{U}_- \equiv w^{-u_1} \cdot w^{-u_4} \cdot w^{-u_5} \cdot w^{-u_7} \equiv 66 \cdot 79 \cdot 72 \cdot 100 \equiv 81 \mod 103$. Similarly, $\mathcal{U}_+ \equiv w^{u_2} \cdot w^{u_3} \cdot w^{u_6} \cdot w^{u_8} \equiv 100 \cdot 13 \cdot 30 \cdot 81 \equiv 93 \mod 103$.

$\mathcal{U}_- \cdot \mathcal{R}_+ \equiv 81 \cdot 64 \equiv 34 \mod 103$. Next, $(\mathcal{U}_- \cdot \mathcal{R}_+)^{-1} \equiv 34^{-1} \equiv 100 \mod 103$. Similarly, $\mathcal{R}_- \cdot \mathcal{U}_+ \equiv 64 \cdot 93 \equiv 81 \mod 103$. Hence, we have $\mu \cdot Z \cdot (\mathcal{U}_- \cdot \mathcal{R}_+)^{-1} \, (\mathcal{U}_+ \cdot \mathcal{R}_-) \equiv 95 \cdot 10 \cdot 100 \cdot 81 \equiv 76 \mod 103$.

The final outcome is $u^a \equiv 76 \mod 103$.

3.2.3 Security and Checkability.

In this part, we give the security analysis of $\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$ and show that a malicious server cannot be able to get any valuable information about u and a.

The next lemma gives the probability that a malicious server obtains the exponent.

Lemma 1 *A malicious server $\mathcal{S}'$ learns the exponent a with probability at most $\frac{\sqrt{\pi k}}{2^{3k}}$ for $k = \ell$.*

Proof The output will only be disclosed if $\mathcal{S}'$ obtains exactly the same position of a_i's with their signs. Hence, the probability of this event is $1/\left(\binom{2k}{k} \cdot 2^k\right)$.

Hence, $\mathcal{S}'$ cannot distinguish the two test queries from all of the $2k$ queries that $\mathcal{C}$ makes, and during any execution of $\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$ the server $\mathcal{S}'$ can successfully cheat without being detected with probability at most $\frac{\sqrt{\pi k}}{2^{3k}}$ by using the Stirling's approximation $\binom{2k}{k} \approx \frac{4^k}{\sqrt{\pi k}}$ [38]. Note that letting $k = \ell = 29$ the probability becomes negligible ($\approx 2^{-80}$). □

We are now ready to prove the security of $\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$. As explained above, outsource-security informally means that there exists a simulator which simulates the view of the adversary in a real algorithm run. This means that the adversary obtains no relevant information from the real run since it could output any result from what it knows by itself.

Theorem 3 *The algorithms $(\mathcal{C}, \mathcal{S})$ are an outsource-secure implementation of $\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$, where the input (a, u) may be honest secret; or honest protected; or adversarial protected.*

Proof We note that this proof is inspired from the proof of the security analysis of [3]. Let $\mathcal{A} = (\mathcal{E}, \mathcal{S}')$ be a probabilistic polynomial-time (PPT) adversary interacting with a PPT-based algorithm $\mathcal{C}$ in the outsource-security model.

Firstly, we prove $\mathsf{EVIEW}_{\mathsf{real}} \sim \mathsf{EVIEW}_{\mathsf{ideal}}$. (Pair One– The external adversary $\mathcal{E}$ learns nothing.)

Let (a, u) be a private input of an honest party. Assume that $\mathsf{Sim}_{\mathcal{E}}$ is a PPT simulator which acts as follows. $\mathsf{Sim}_{\mathcal{E}}$ ignores the the ith round when getting input, like using Figure 1 it chooses random sets $R := U_1 \cup U_2 \cup U_3 \cup U_4 := \{r_1, \cdots, r_\ell\}$ and $A := U_5 \cup U_6 \cup U_7 \cup U_8 := \{a_1, \cdots, a_k\}$ such that $r = \sum_{i=1}^{\ell} r_i$, $a = \sum_{i=1}^{k} a_i$. $\mathsf{Sim}_{\mathcal{E}}$ first forms random subsets U_i with arbitrary length such that $U := A \cup R = \cup_{i=1}^{8} U_i$, where $U_i \neq \emptyset$, $\forall\ i$ and $U_i \cap U_j = \emptyset$, $\forall\ i \neq j$. For the sign of the values $\mathsf{Sim}_{\mathcal{E}}$ chooses further a random $\alpha = (\alpha_1, \cdots, \alpha_{\ell+k})$ $\in_R \{0,1\}^{\ell+k}$. Next, $\mathsf{Sim}_{\mathcal{E}}$ forms random subsets $S := U_1 \cup U_2 \cup U_6 \cup U_7$ and $T := U_1 \cup U_4 \cup U_5 \cup U_6$ of U such that $s = \sum_{s_i \in S} s_i$ and $t = \sum_{t_i \in T} (-1)^{\alpha_i} \cdot t_i$ satisfying the condition that $s + c_1 t = c_2$, where $c_1, c_2 \in_R \{1, \cdots, c\}$. Let $U := \{u_1, \cdots, u_{k+\ell}\}$. $\mathcal{C}$ chooses a random permutation $\sigma \in_R \mathbb{S}_{\ell+k}(U)$ and sets the permuted elements $U = \sigma(U) := (\sigma_1, \cdots, \sigma_{k+\ell})$. $\mathsf{Sim}_{\mathcal{E}}$ sets $\mathcal{U}_-, \mathcal{U}_+ \leftarrow 1$ and uses the partitions in Figure 1.

If an error occurs, $\mathsf{Sim}_{\mathcal{E}}$ stores its own and $\mathcal{S}'$'s states and outputs $Y_p^i =$ "$error$", $Y_u^i = \emptyset, \mathsf{rep}^{\mathsf{i}} = 1$. If all checkability steps are valid, $\mathsf{Sim}_{\mathcal{E}}$ outputs $Y_p^i = \emptyset$, $Y_u^i = \emptyset$, $\mathsf{rep}^{\mathsf{i}} = 0$; otherwise, $\mathsf{Sim}_{\mathcal{E}}$ chooses a random group value $h \in_R \mathbb{G}$ and outputs $Y_p^i = h$, $Y_u^i = \emptyset$, $\mathsf{rep}^{\mathsf{i}} = 1$. Next, $\mathsf{Sim}_{\mathcal{E}}$ stores the corresponding states. The distributions in the real and ideal executions of the input to $\mathcal{S}'$ are computationally indistinguishable. In the ideal setting, the inputs are uniformly chosen random from $\mathbb{Z}/m\mathbb{Z} \times \mathbb{G}$. In the real setting, we follow Step 7 of $\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$ to assure that all parts of Exp $\mathcal{C}$ invokes is randomized independently using σ and α. Now, we consider all possible cases. If $\mathcal{S}'$ behaves in an honest manner in the ith round, then $\mathsf{EVIEW}_{\mathsf{real}}^{i} \sim \mathsf{EVIEW}_{\mathsf{ideal}}^{i}$, because in the real execution $\mathcal{C}^{\mathcal{S}'}$ perfectly runs $\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$ and in the ideal execution $\mathsf{Sim}_{\mathcal{E}}$ does not change the output of $\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$. If $\mathcal{S}'$ gives a incorrect output in the ith round, then the output will be detected by $\mathcal{C}$ and $\mathsf{Sim}_{\mathcal{E}}$ with probability at most $\frac{\sqrt{\pi k}}{2^{3k}}$ due to Lemma 1, resulting in an output of "$error$"; otherwise, the software will indeed be successful in manipulating the output of $\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$ (e.g., because each request is independent of each other, sending approximately 29 wrong results with their signs to the client $\mathcal{C}$ makes the probability of not being detected to negligibly small ($\approx 1/2^{80}$).).

In the real execution, the $k+\ell$ real outputs of $\mathcal{S}'$ are firstly grouped into two different parts corresponding to their signs (positive or negative). The negative and positive parts will be independently computed due to the checkability condition $s + c_1 t = c_2$. The result will be multiplied corresponding to their signs (7 and 8 of $\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$). At the last step, we multiply the overall result with the masking values of the base element generated at the first step according to their signs. Hence, a manipulated output of $\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$ will seem to be wrong, but random to $\mathcal{E}$.

We simulate this situation in the ideal execution by replacing the output of $\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$ with a random element in $\mathbb{G}$ when there is an attempt to behave maliciously by $\mathcal{S}'$ which would not be detected by $\mathcal{C}$ in the real execution. Hence, even if $\mathcal{S}'$ behaves maliciously in the ith round, $\mathsf{EVIEW}_{\mathsf{real}}^i \sim \mathsf{EVIEW}_{\mathsf{ideal}}^i$. By the hybrid argument, we can easily conclude that $\mathsf{EVIEW}_{\mathsf{real}} \sim \mathsf{EVIEW}_{\mathsf{ideal}}$.

Next, we prove $\mathsf{EVIEW}_{\mathsf{real}} \sim \mathsf{EVIEW}_{\mathsf{ideal}}$. (Pair Two– The untrusted server $\mathcal{S}'$ obtains no useful information).

We now consider the cases where (a, u) is honest secret/protected or adversarial protected. Let $\mathsf{Sim}_{\mathcal{S}'}$ be a PPT simulator that acts in the following manner. $\mathsf{Sim}_{\mathcal{S}'}$ ignores the ith round when getting input, and instead chooses a permutation $\sigma \in \mathbb{S}_{\ell+k}$ and prepares a signed permuted random query of the form $((-1)^{\alpha_{\sigma(j)}}\sigma_j) \in \mathbb{Z}/m\mathbb{Z} \times \mathbb{G}$ to $\mathcal{S}'$ using $\alpha_{\sigma(j)}$, where $j \in \{1, \cdots, k+\ell\}$. $\mathsf{Sim}_{\mathcal{E}}$ randomly checks $(k+\ell)$ outputs from each procedure using σ. Then, $\mathsf{Sim}_{\mathcal{S}'}$ stores its own and states of $\mathcal{S}'$. Note that these real and ideal executions are distinguishable by $\mathcal{E}$ but $\mathcal{E}$ cannot use this information to $\mathcal{S}'$ (e.g., the output of the ideal execution is never manipulated). During the ith round of the real execution, the inputs of $\mathcal{C}$) are always randomized to $2(k+\ell)$ utilizing σ, α (see steps 6 and 7 of $\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$). In the ideal execution, $\mathsf{Sim}_{\mathcal{S}'}$ always generates independently random queries for $\mathcal{S}'$. The view is consistent and indistinguishable from the server's view when there is an interaction with honest $\mathcal{C}$. Thus, for each round we have $\mathsf{EVIEW}_{\mathsf{real}} \sim \mathsf{EVIEW}_{\mathsf{ideal}}$, which by the hybrid argument yields $\mathsf{EVIEW}_{\mathsf{real}} \sim \mathsf{EVIEW}_{\mathsf{ideal}}$.

Consequently, we simulate every step of $\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$ for the simulator which completes the simulation for both malicious environment and server. □

Lemma 2 *The algorithm $(\mathcal{C}, \mathcal{S})$ is an $O(\log^2(l)/l)$-efficient implementation of $\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$, where l denotes the number of bits of the exponent a.*

Proof We use the same approach of the proof of the algorithm in [3]. The algorithm SubAlg makes 3 calls to Rand and $4\log c+8$ modular multiplications. The proposed algorithm $\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$ makes 2 further calls to Rand and together with SubAlg, $k+\ell+4\log c+30$ modular multiplications (MMs) and only 1 modular inversion (MInv) in order to compute $u^a \mod n$ (other operations like modular additions, doubling or multiplication with very small numbers like c are omitted). Also, a server aided exponentiation takes $O(\log^2(l))$ MMs using the number theoretic complexity analysis of Nguyen, Shparlinski, and Stern [14], or $O(1)$ MMs if a table-lookup method is used. On the other hand, it takes in average $1.5l$ MMs to compute $u^a \mod n$ by the classical square-and-multiply method. Thus, the algorithm $(\mathcal{C}, \mathcal{S})$ is an $O(\log^2 l/l)$-efficient implementation of $\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$. □

Lemma 3 *The algorithm $(\mathcal{C}, \mathcal{S})$ is an $(1-\frac{1}{c(c-1)})$-checkable implementation of $\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$.*

Proof By Alg_{pr}^{pr}, a malicious server $\mathcal{S}$ gives a incorrect result without being detected if it can find either

1. the correct values c_1 and c_2 in SubAlg, or

2. the correct value a or r, or
3. the correct value s or t, or
4. the position of a value s_i, where $S{=}\{s_1, \cdots, s_{k'}\}$ with $s = \sum_{j=1}^{k'} s_i$, or
5. the position of a value t_i, where $T{=}\{t_1, \cdots, t_{k''}\}$ with $t = \sum_{j=1}^{k''} t_i$

For the first case, $\mathcal{S}$ finds the correct values c_1 and c_2 in SubAlg with probability $\frac{1}{c(c-1)}$ (see Theorem 1 for details).

For the second case, finding either the exact value of a or r has negligibly probability (see Lemma 1).

For the third case, to be able to find the correct values of s, the server $\mathcal{S}$ first needs to find out the subset S from the power set $\mathcal{P}(U)$ such that $s{=}\sum_{i=1}^{k'} s_i$. The value t can subsequently be obtained by solving the subset sum problem for $s{+}c_1 t = c_2$, where c_1 and c_2 are small integers. Similarly, one can start with t to find s. The complexity of finding such (s,t) pairs from the power set $\mathcal{P}(U)$ is $2^{k+\ell} \cdot 2^{(k+\ell)/2} = 2^{3/2(k+\ell)}$ (note that $|\mathcal{P}(U)| = 2^{k+\ell}$). The reason is that the best generic algorithms to solve the subset sum problem are lattice-based methods which require $2^{n/2}$ for any set of cardinality n [11, 39–42].

For the last two cases, $\mathcal{S}$ can attack the checkability of the system if it can find a value s_i (or t_i) with its sign. Namely, the checkability follows from $\sum_{i=1}^{k'} s_i + c_1 \sum_{i=1}^{k''} t_i = c_2$ and with the knowledge of s_i (or t_i) and its sign and the knowledge of c_1 and c_2. Finding a value s_i has probability at least 1/2 and with probability at least 1/2 to decide whether it has negative or positive sign. Therefore, the overall probability of this event is $\frac{1}{4c^2}$.

Hence, the overall probability for a malicious server $\mathcal{S}$ to declare a incorrect value without being detected is

$$1 - \frac{1}{c(c-1)} = \min\{1 - \frac{1}{4c^2}, 1 - \frac{1}{c(c-1)}\}.$$

□

Now the security and the checkability of $\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$ follow obviously from the following corollary.

Corollary 1 *The algorithm $(\mathcal{C}, \mathcal{S})$ is an* $(O(\log^2(l)/l)$, $(1 - \frac{1}{c(c-1)})$*-outsource-secure implementation of* $\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$.

Remark 1 Letting $c = 4$ gives us the probability 11/12 by Lemma 3 which is the best checkability result compared to previous works [2–5].

Note that in outsourced computation model the malicious server $\mathcal{S}$ can be seen as a covert adversary [43], which may arbitrarily behave to cheat depending on whether being detected with reasonable probability (not necessarily with very high probability) by an honest party. In [43], covert adversaries are described for many real-life scenarios where they are always eager to cheat but only if they are not detected. Therefore, cloud servers can be seen as covert adversaries in outsourced computation setting because their financial interests and their reputation deter them from cheating.

4 Other Relevant Algorithms

In this section, we simplify $\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$ for Public-Base & Private-Exponent and Private-Base & Public-Exponent cases, and modify it to obtain a more efficient simultaneous modular exponentiations algorithm.

4.1 Public-Base & Private-Exponent

In this part, we modify $\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$ for the case of public-base & private-exponent. The modified method is especially designed to outsource the cryptographic outsourced computation for the cases in which there is no need to hide the base element if it is not required in the cryptographic setting (e.g., signatures). The first precomputation of $\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$ is unnecessary in this case since we are not forced to hide our base element u. The new algorithm $\mathsf{Alg}_{\mathsf{pb}}^{\mathsf{pr}}$ for public-base & private-exponent is a special case of $\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$ by setting the values $x = y = 0$ in the precomputation step.

Theorem 4 $\mathsf{Alg}_{\mathsf{pb}}^{\mathsf{pr}}$ *terminates and outputs correctly. Furthermore, there exists an algorithm which is an* $\left(O(\log^2(l)/l), \frac{1}{4c^2}\right)$*-outsource-secure implementation of* $\mathsf{Alg}_{\mathsf{pb}}^{\mathsf{pr}}$.

Proof Correctness, termination and security of the algorithm follow easily as a corollary of the results for $\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$ by excluding the subalgorithm SubAlg. Because SubAlg is not used for $\mathsf{Alg}_{\mathsf{pb}}^{\mathsf{pr}}$, the checkability property becomes $1 - \frac{1}{4c^2}$. □

4.2 Private-Base & Public-Exponent

In this part, we give another algorithm for private-base & public-exponent cryptographic computation by modifying $\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$. Note that especially for public-key encryption or signature verification based systems it could be desirable to have private-base & public-exponent. This algorithm is denoted by $\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pb}}$ which works in detail as follows.

Algorithm 3 ($\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pb}}$): Private-Base & Public-Exponent Modular Exponentiations

---

Input: (u, a, c) (where $a \in \mathbb{Z}/m\mathbb{Z}$, $u \in \mathbb{G}$ with $< g >= \mathbb{G} \leq \mathbb{F}_n^*$ for DLP or $g \in_R \mathbb{G} = (\mathbb{Z}/n\mathbb{Z})^*$ for RSA with $|\mathbb{G}| = m$, where $n, m \in \mathbb{N}$, and an arbitrary $i \in \mathbb{N}$).
Output: The value u^a in $\mathbb{G}$.
Precomputation: A Rand algorithm computes and stores the following values for $\mathcal{C}$:

- $(s_i, g^{s_i}, g^{-s_i}), (t_i, g^{t_i}, g^{-t_i}) \in_R \mathbb{Z}/m\mathbb{Z} \times \mathbb{G}^2$ for $i = 1, 2$.
- I=$\{1, \cdots, c\} \subseteq \mathbb{Z}/m\mathbb{Z}$ with $I^{-1} = \{1^{-1}, \cdots, c^{-1}\} \subseteq \mathbb{Z}/m\mathbb{Z}$.

1. $\mathcal{C}$ picks random elements c_1, $c_2 \in_R I$, where $\gcd(c_1, c_2) = 1$, and computes $u_1 \leftarrow u^{c_1} \cdot g^{b_1 s_1}$ and $u_2 \leftarrow u^{c_2} \cdot g^{b_2 s_2}$, where $b_1, b_2 \in_R \{1, -1\}$.
2. $\mathcal{C}$ runs
   (a) $U_1 \leftarrow \mathsf{Exp}(a, u_1)$.
   (b) $U_2 \leftarrow \mathsf{Exp}(a, u_2)$.
   (c) $T_1 \leftarrow \mathsf{Exp}((b_1' a \cdot s_1 + b_3 t_1) \cdot c_3^{-1}, g)$, where $b_1', b_3 \in_R I$ and $c_3 \in I^{-1}$.
   (d) $T_2 \leftarrow \mathsf{Exp}((b_2' a \cdot s_2 + b_4 t_2) \cdot c_4^{-1}, g)$, where $b_2', b_4 \in_R I$ and $c_4^{-1} \in I^{-1}$.
3. For verification, $\mathcal{C}$ does the following computations:
   (a) $T_1' = T_1^{c_3} \cdot g^{-b_3 t_1}$ and $T_2' = T_2^{c_4} \cdot g^{-b_4 t_2}$ `(masking removal)`
   (b) if $b_1 \neq b_1'$ computes $U_1' \leftarrow U_1 \cdot T_1'$ else computes $U_1' \leftarrow U_1 \cdot (T_1')^{-1}$
   (c) if $b_2 \neq b_2'$ computes $U_2' \leftarrow U_2 \cdot T_2'$ else computes $U_2' \leftarrow U_2 \cdot (T_2')^{-1}$
   (d) verifies $(U_1')^{c_2} \stackrel{?}{=} (U_2')^{c_1}$, where $k = c_1 \cdot c_2$ and returns u^a `(easily computable since` $\gcd(c_1, c_2) = 1$`)`

Theorem 5 $\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pb}}$ *terminates and outputs correctly. Furthermore, there exists an algorithm which is an* $(O(\log^2(l)/l), 1 - \frac{1}{c(c-1)})$*-outsource-secure implementation of* $\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pb}}$.

Proof Precomputation and Step 1 of $\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pb}}$ imply that

$$u_1 = u^{c_1} \cdot g^{b_1 s_1},$$
$$u_2 = u^{c_2} \cdot g^{b_2 s_2},$$

where $b_1, b_2 \in_R \{1, -1\}$, c_1, $c_2 \in_R I$ and $\gcd(c_1, c_2) = 1$.

At Step 2 $\mathcal{S}$ returns the query results

$$U_1 = u_1^a = u^{ac_1} \cdot g^{b_1 a s_1},$$
$$U_2 = u_2^a = u^{ac_2} \cdot g^{b_2 a s_2},$$

and

$$T_1 = g^{(b_1' a s_1 + b_3 t_1) \cdot c_3^{-1}} \text{ and } T_2 = g^{(b_2' a s_2 + b_4 t_2) \cdot c_4^{-1}}.$$

Then $\mathcal{C}$ computes the following to verify the result.

$\mathcal{C}$ first removes the masking values t_1 and t_2 using c_3 and c_4 as

$$T_1' = T_1^{c_3} \cdot g^{-b_3 t_1} = g^{b_1' a s_1},$$
$$T_2' = T_2^{c_4} \cdot g^{-b_4 t_2} = g^{b_2' a s_2}.$$

Next, the masking values from U_1 and U_2 will be removed, i.e. $U_1' = u^{ac_1}$ and $U_2' = u^{ac_2}$. In order to avoid inversion, we basically compare b_1 with b_1' and b_2 with b_2' ($U_1' = U_1 \cdot T_1'$ or $U_2' = U_2 \cdot (T_2')^{-1}$, respectively).

$\mathcal{C}$ verifies $(U_1')^{c_2} \stackrel{?}{=} (U_2')^{c_1}$, where $c_1, c_2 \in I$ and $k = c_1 \cdot c_2$. If the equality does not hold then algorithm outputs checkability failure. Finally, because $\gcd(c_1, c_2) = 1$ and c_1, c_2 are very small $\mathcal{C}$ efficiently computes u^a.

A malicious server cannot learn the private base u because it is randomized with g^{s_1} and g^{s_2}. Furthermore, a malicious server cannot also change the outcome unless it finds either c_1, c_2 or c_3, c_4 and the probability of this event is $1 - \frac{1}{c(c-1)}$. □

4.3 t-Simultaneous Modular Exponentiations

We now generalize the notion of simultaneous modular exponentiation method of [2] to the notion of *t-simultaneous modular exponentiations* $u_1^{a_1} \cdots u_t^{a_t}$ in the group $\mathbb{G}$ for $t \in \mathbb{N}$. t-simultaneous modular exponentiations are extensively used in many real-life cryptographic schemes including [19, 35, 44–47]. As described in [2], computing 2-simultaneous modular exponentiations is trivial by simply invoking $\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$ twice. Here, we show that it is possible to reduce the computation cost significantly for a generalized t-simultaneous setting by improving the method of [2] and utilizing only one untrusted server (instead of two non-colluding malicious servers). We denote by t-Sim-$\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$ for t-simultaneous modular exponentiation algorithm.

The scheme of Chen *et al.* [2] has probability 2/3 for checkability in modular exponentiation utilizing and has probability 1/2 for 2-Sim-$\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$ using two non-colluding servers. They simply add a one more variable on the exponentiation at the expense of reducing the probability from 2/3 to 1/2. Our solution has a scalable probability $1 - \frac{1}{c(c-1)}$ for checkability and utilizes only one single untrusted server.

We further emphasize that the natural generalization for 2-simultaneous modular exponentiation method in [2] reduces the checkability probability from $\frac{1}{2}$ of single exponentiation case to $\frac{2}{t+2}$ for t-simultaneous modular exponentiations. However, the use of t-simultaneous modular exponentiation in real-life protocols, like anonymous credentials [35], causes significant complexity overhead. Hence, this reduction hinders the use of this generalization from 2-simultaneous to t-simultaneous modular exponentiation. Unlike the scheme in [2], our scheme has an adjustable probability $1 - \frac{1}{c(c-1)}$ which is independent of t. More concretely, the algorithm works as follows:

$\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$ first runs Rand to compute the blinding pairs $(x, g^x), (y, g^y)$ and (k, g^k). Denote $v = g^x$ and $\mu = g^y$. Now, we have

$$u_1^{a_1} \cdots u_t^{a_t} = (vw_1)^{a_1} \cdots (vw_t)^{a_t} = \mu Z g^z w_1^{a_1} \cdots w_t^{a_t},$$

where $w_i = u_i v^{-1}$ and $Z = g^z$ with $z = x \sum_{i=1}^{t} a_i - y$ for $1 \leq i \leq t$. First, Z is computed by invoking $Z = \mathsf{SubAlg}(z, g, c)$ to $\mathcal{S}$.

Note that w_i's are completely random and therefore, can be revealed to $\mathcal{S}$. Hence, instead of invoking $\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$ t times, it is now possible to invoke more efficient algorithm $\mathsf{Alg}_{\mathsf{pb}}^{\mathsf{pr}}$ t times. In particular, we gain a linear factor for the number of total multiplication in the number t. More precisely, a t-simultaneous modular exponentiation requires $t(\ell + k + 4\log c + 28) + 10 + 4\log c$ modular multiplications and t modular inversions instead of invoking $\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$ t-times

which requires $t(\ell + k + 8\log c + 38)$ modular multiplications and t modular inversions. Hence, we save $10t + 4\log ct$ modular multiplications by using our t-simultaneous modular exponentiation technique.

For instance, the complexity of 2-simultaneous modular exponentiations running $2\text{-}\mathsf{Sim}\text{-}\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$ for $c = 4$ requires 184 MMs and 2 MInvs (instead of 200 MMs and 2 MInvs by running $\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$ twice).

By utilizing t calls of $\mathsf{Alg}_{\mathsf{pb}}^{\mathsf{pr}}$ and Theorem 4 the following holds.

Theorem 6 *There exists an algorithm $(\mathcal{C}, \mathcal{S})$ which is an $(O(t\log^2(l)/l), 1 - (\frac{1}{c(c-1)}))$-outsource-secure implementation of $t\text{-}\mathsf{Sim}\text{-}\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$.*

Table 1 Computation Complexity of the Proposed Algorithms Using Single Server

	Exp ($\mathcal{S}$)	MM	MInv	Rand	Check
SubAlg	2	$4\log c + 8$	0	3	$1 - \frac{1}{c(c-1)}$
Alg_{pr}^{pr}	$\ell + k + 2$	$\ell + k + 8\log c + 38$	1	5	$1 - \frac{1}{c(c-1)}$
Alg_{pb}^{pr}	$\ell + k$	$\ell + k + 4\log c + 28$	1	0	$1 - \frac{1}{4c^2}$
Alg_{pr}^{pb}	4	$16\log c + 16$	2	8	$1 - \frac{1}{c(c-1)}$
$t\text{-}\mathsf{Sim}\text{-}\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$	$t(\ell + k) + 1$	$t(\ell + k + 4\log c +$ $28) + 10 + 4\log c$	t	5	$1 - \frac{1}{c(c-1)}$

5 Complexity Analysis of the Proposed Algorithms

In this section, we first illustrate the complexity of our proposed algorithms using Table 1. In this table, we give the complexity results by counting the number of modular exponentiations for the server side; and for the client side the number of modular multiplications (MMs), the number of modular inversions (MInvs), the number of Rands and checkability probabilities. Note that we count the number of multiplication in the worst case by using classical double and algorithm, i.e., for an $l-$bit exponent we require $2l + 1$ MMs.

In Table 2, we give the complexity of the proposed algorithms by setting $\ell = k = 29$ and $c = 4$. We note that by Lemma 3 letting $\ell = k = 29$ reduces the probability of privacy leakage to negligible levels.

In order to compare $\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$ with the previous results properly, we need to equate the checkability probabilities of all algorithms and count the number of all operations in terms of modular multiplications. For this purpose, we use the fact that in a real-life hardware setting a modular inversion is about 100 times slower than a modular multiplication [48]. In order to have the same checkability probability 11/12, we have to run the algorithm [3] $\log_2 12 \approx 3,58$ times, and the algorithm [2] $\log_3 12 \approx 2,26$ times. The comparison will now be as follows:

Table 2 Computation Complexity for Proposed Algorithms for $k = \ell = 29$, $c = 4$

	Exp ($\mathcal{S}$)	MMs	MInvs	Rand	Check
SubAlg	2	10	0	3	11/12
$\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$	60	100	1	5	11/12
$\mathsf{Alg}_{\mathsf{pb}}^{\mathsf{pr}}$	58	86	1	0	63/64
$\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pb}}$	4	48	2	8	11/12
2-Sim-$\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$	117	184	2	5	11/12

In [3], we have 9 MMs and 5 MInvs in one round. Hence, in $\log_2 12$ rounds we obtain $9 \cdot \log_2 12$ MMs and $5 \cdot \log_2 12$ MInvs. Hence, we have a total number of $9 \cdot \log_2 12 + 100 \cdot 5 \cdot \log_2 12 \approx 1825$ MMs for [3].

In [2], we have 7 MMs and 3 MInvs in one round. Hence, in $\log_3 12$ rounds we obtain $7 \cdot \log_3 12$ MMs and$3 \cdot \log_3 12$ MInvs. Hence, we have a total number of $7 \cdot \log_3 12 + 100 \cdot 3 \cdot \log_3 12 \approx 694$ MMs for [2].

In [5], the goal is to outsource u^a, where $c = a - b\xi$ with b and c are known by the server with probability 1/6. Therefore, ξ must be large enough to prevent the brute-force attack. Hence, to have a negligible level, one has to choose $\xi \approx 2^{77}$. There are 167 MMs and 4 MInvs for the checkability of 1/2. Hence, in $\log_2 12$ rounds we obtain $167 \cdot \log_2 12$ MMs and $4 \cdot \log_2 12$ MInvs. Hence, we have a total number of $167 \cdot \log_2 12 + 100 \cdot 4 \cdot \log_2 12$ MMs for [5].

The algorithm Alg_{pr}^{pr} has 100 MMs and only 1 MInv. Hence, there is a total number of approximately $100 + 1 \cdot 100 = 200$ MMs.

Table 3 Comparison is shown for the client side. Total MMs are counted after equating the checkability to 11/12 for all schemes.

	MM	MInv	Single Server	Check	Total MMs
[3] TC05	9	5	✗	1/2	$\log_2 12 \cdot 509$ ≈ 1825
[2] ESORICS12	7	3	✗	2/3	$\log_3 12 \cdot 307$ ≈ 694
[5] ESORICS14	167	4	✓	1/2	$\log_2 12 \cdot 567$ ≈ 2033
Ours	100	1	✓	11/12	$100 + 100$ ≈ 200

In Table 3, we compare our algorithm Alg_{pb}^{pr} with the results of [3], [2] and [5]. In the last column of Table 3, we give the total number of MMs which shows that our algorithm Alg_{pb}^{pr} is the most efficient algorithm using only one single untrusted server $\mathcal{S}$ with the best checkability.

Remark 2 Although the number of MMs of Alg_{pr}^{pr} is slightly better than the number of MMs in the algorithm of [2] for only one outsourced modular exponentiation, using t-Sim-$\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$ we gain a linear factor in t which gives significantly better complexity results for the number of MMs.

Furthermore, Alg_{pr}^{pr} has better checkability probability (11/12 versus 2/3). We highlight that our checkability probability increases with the value of c at the expense of increasing the number of modular multiplication logarithmically. In particular, our approach enables the designer to obtain privacy preserving outsourcing algorithms with scalable checkability.

Memory complexity is also an important criteria especially for resource-constrained devices. The schemes [3] and [2], which use two-non colluding servers, need to store 22 and 17 group elements, respectively. In [5], which uses only one single untrusted server, $6k\ell + 5\ell + 16$ group elements are stored, where r, s are security parameters. By equating the checkability to 11/12 in all these schemes [2, 3, 5], the required memory becomes $22 \cdot \log_2 12 \approx 78.87$, $17 \cdot \log_3 12 \approx 38.45$, and $6k\ell + 5\ell + 16 \cdot \log_2 12$, respectively. Our main algorithm only stores $k + \ell + c + 27$ group elements, where k, ℓ are security parameters and c is a small integer. When compared to the only existing scheme using one untrusted server [5] we gain a linear factor in ℓ for the the memory requirement.

We remark that one demerit of Algorithm 1 ($\mathsf{Alg}_{\mathsf{pr}}^{\mathsf{pr}}$) in comparison with other schemes may be the communication overhead. More concretely, [3] requires $8 \log m\ +16 \cdot \mathfrak{g}$, [2] requires $6 \log m\ +\ 12 \cdot \mathfrak{g}$, and [5] requires $4 \log m$ $+8 \cdot \mathfrak{g}$ bits of transmission, where $\mathfrak{g}$ denotes the group size and m is the order of the group. For the same checkability, [3] requires $\log_2 12 \cdot (8 \log m\ +16 \cdot \mathfrak{g})$, [2] requires $\log_3 12 \cdot (6 \log m\ +\ 12 \cdot \mathfrak{g})$, and [5] requires $\log_2 12 \cdot 4(\log m\ +8 \cdot \mathfrak{g})$ bits of transmission. Our main algorithm transmits $(k+\ell+2) \cdot \log m + (k+\ell+2) \cdot \mathfrak{g}$ group elements to the server. We remark that the communication overhead of our algorithm for $k = \ell = 29$ (for security with negligible error probability) is just slightly larger than the others. In particular, this disadvantage is not noticeable from the practical point of view with the current computational power.

6 Applications: Outsourced Oblivious Transfer and Blind Signatures

6.1 Oblivious Transfer

Oblivious transfer is a powerful cryptographic primitive which is *complete* for secure multiparty computation [27, 28]. In an OT protocol, the sender has two private input bits (s_0, s_1) and the chooser has one private input bit b. At the end of the protocol, the chooser learns only the bit s_b, whereas the sender does not know any information which bit was selected by the chooser.

With the help of cloud providers it is possible to compute independently any outsourced functionality without disclosing the private input. Namely, clients only need to randomize/encrypt their data and de-randomize/decrypt

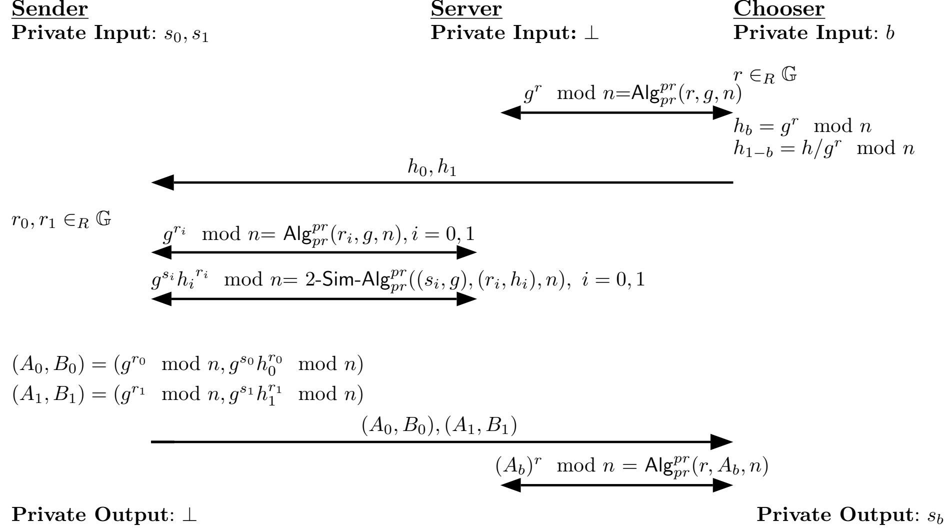

Fig. 2 Outsourcing Oblivious Transfer

the returned messages to get the desired results. OT is one of the major computational overhead for Yao's garbled circuit protocol [23,29], and used in several applications like biometric authentication, e-auctions, private information retrieval, private search [30–33]. Hence, running OT protocols for resource-constrained mobile environment may have substantial benefits.

In this section, we provide an example of outsourcing an OT protocol in a discrete log setting (see Figure 2). Assume that $\mathbb{G}$ is a group generated by g (i.e. $\mathbb{G} =< g >$) and $h \in \mathbb{G}$, where $\log_g h$ is unknown to any party. At the first step, the chooser chooses random $r \in_R \mathbb{G}$ and invokes the cloud server $\mathcal{S}$ to compute $\mathsf{Alg}_{pr}^{pr}(r, g, n)$ and computes $h_b = g^r \mod n$. Note that at this stage, cloud server and the environment do not learn any valuable information about the inputs or the outputs. The chooser then computes $h_{1-b} = h/g^r$. Next, the chooser sends (h_0, h_1) to the sender. The sender now invokes $\mathcal{S}$ to run $\mathsf{Alg}_{pr}^{pr}(r_0, g, n)$ and $2 - \mathsf{Sim}\text{-}\mathsf{Alg}_{pr}^{pr}((s_i, g), (r_i, h_i), n),\ i = 0, 1$ to compute and receive g^{r_i} and $h_i^{r_i} g^{s_i}$ for $i = 0, 1$, respectively. The sender then returns homomorphic ElGamal encryptions of s_0 and s_1 denoted as $(A_0, B_0) = (g^{r_0}, g^{s_0} h_0^{r_0})$ and $(A_1, B_1) = (g^{r_1}, g^{s_1} h_1^{r_1})$, respectively. Depending on his bit b, the chooser is able to decrypt one of these encryptions to learn either s_0 or s_1. Hence, if both parties follow the protocol specification, the chooser learns exactly one of the bits s_0 and s_1, and the sender does not know any information about what the chooser learns. The OT protocol used for outsourcing is secure in the semi-honest model but malicious versions of OT can be used analogously. We finally would like to highlight that in the original OT protocol, for each input bit, the Chooser computes 2 exponentiations, and the Server computes 2

exponentiations and 2 simultaneous exponentions. Therefore, our outsourcing algorithm gains higher computational efficiency in case the private inputs are longer. See Table 4 for the comparison of the standard OT protocol with the outsourced version.

Table 4 Comparison of Outsourced OT and BS with Classical Versions

	Standard	Outsourced
OT	4 MExp + 2 2-Sim-MExp + 1 MInv ≈ 24100 MMs	≈ 1668 MMs
BS	2 MExp + 1 MInv + 1 MM ≈ 6100 MMs	≈ 500 MMs

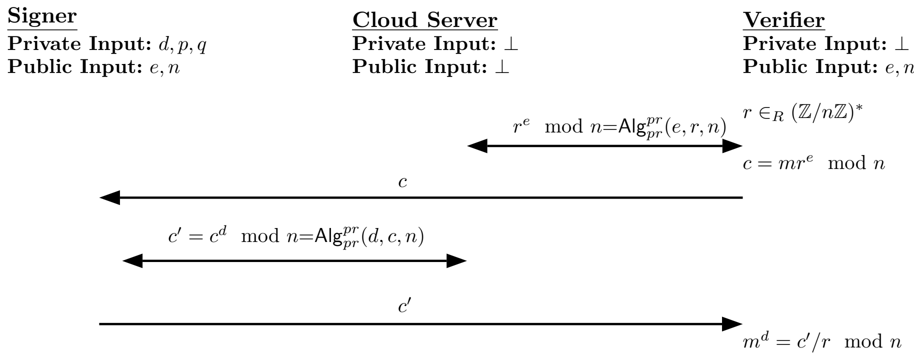

Fig. 3 Outsourcing Blind Signatures

6.2 Blind Signatures

Blind signatures have been suggested by Chaum [34]. Roughly speaking, it allows a signer interactively issue signatures and allows users to obtain them such that the signer does not see the resulting message, and the signature pair during the signing session. Like any conventional electronic signatures they are unforgeable and can be verified using a public key.

Blind signatures can be applied to privacy preserving protocols like e-cash, e-voting and anonymous credentials. For a e-cash scenario, a bank blindly signs coins withdrawn by the users. For an e-voting scenario, an authority blindly signs a vote for later to cast the signed votes. As for anonymous credentials which especially needs simultaneous exponentiations with expensive zero-knowledge proofs, the issuing authority blindly signs a key [35] for later to authenticate services anonymously. We would like to highlight that in the original blind signature protocol, the signer and the verifier computes exponentiation using private and public keys respectively. Hence, for mobile environment

and constrained-devices, outsourcing blind signatures gains higher computational efficiency, and therefore can be beneficial for real-life applications (see Figure 3). See Table 4 for the comparison of the standard OT protocol with the outsourced version.

7 Conclusion

In this paper, we propose new, scalable, secure and efficient algorithms for outsourcing modular exponentiations (i.e., public-base & private-exponent, private-base & public-exponent, private-base & private-exponent, and simultaneous modular exponentiations). Our algorithms are significantly more efficient than the previous algorithms. Moreover, the proposed algorithms are modeled, where only one single untrusted cloud server exists. Our algorithms also enjoy the predetermined checkability property which is a significant improvement compared to the prior works. The security of our algorithms is proven formally based on the model of [3]. We finally utilize our algorithms for outsourcing oblivious transfer protocols and blind signatures, which may be beneficial for resource-constrained mobile secure environments running on a client.

The algorithm for single server in [5] requires extremely large number of MMs whereas our algorithm needs comparably very small number of MMs ($\approx 10,17$ times less MMs). On the other hand, although the communication round of our algorithm is constant, the overhead of information exchange is still large. Therefore, it is an interesting open problem to find better constructions achieving smaller (possibly constant) communication overhead together with smaller number of modular multiplications without any modular inversions.

Acknowledgements

This work is partly supported by a joint research project funded by Bundesministerium für Bildung und Forschung (BMBF), Germany (01DL12038) and TÜBİTAK, Turkey (TBAG-112T011). It is also partially supported by the project (114C027) funded by EU FP7-The Marie Curie Action and TÜBİTAK (2236-CO-FUNDED Brain Circulation Scheme). It has also been partially supported by the COST Action CRYPTACUS (IC1403).

References

1. Marten Van Dijk and Ari Juels. On the impossibility of cryptography alone for privacy-preserving cloud computing. In *Proceedings of the 5th USENIX Conference on Hot Topics in Security*, HotSec'10, pages 1–8. USENIX Association, 2010.
2. Xiaofeng Chen, Jin Li, Jianfeng Ma, Qiang Tang, and Wenjing Lou. New algorithms for secure outsourcing of modular exponentiations. In Sara Foresti, Moti Yung, and Fabio Martinelli, editors, *Computer Security ESORICS 2012*, volume 7459 of *Lecture Notes in Computer Science*, pages 541–556. Springer Berlin Heidelberg, 2012.

3. Susan Hohenberger and Anna Lysyanskaya. How to securely outsource cryptographic computations. In Joe Kilian, editor, *Theory of Cryptography*, volume 3378 of *Lecture Notes in Computer Science*, pages 264–282. Springer Berlin Heidelberg, 2005.
4. Praveen Gauravaram Lakshmi Kuppusamy, Jothi Rangasamy. On secure outsourcing of cryptographic computations to cloud. In *ACM Symposium on Information, Computer and Communications Security ASIACCS*. ACM, 2014.
5. Yujue Wang, Qianhong Wu, Duncan S. Wong, Bo Qin, Sherman S.M. Chow, Zhen Liu, and Xiao Tan. Securely outsourcing exponentiations with single untrusted program for cloud storage. In Mirosaw Kutyowski and Jaideep Vaidya, editors, *Computer Security - ESORICS 2014*, volume 8712 of *Lecture Notes in Computer Science*, pages 326–343. Springer International Publishing, 2014.
6. Jingwei Li, Duncan Wong, Jin Li, Xinyi Huang, and Yang Xiang. Secure outsourced attribute-based signatures. *IEEE Transactions on Parallel and Distributed Systems*, 99(PrePrints), 2014.
7. Haixin Nie, Xiaofeng Chen, Jin Li, Josolph Liu, and Wenjing Lou. Efficient and verifiable algorithm for secure outsourcing of large-scale linear programming. In *Advanced Information Networking and Applications (AINA), 2014 IEEE 28th International Conference on*, pages 591–596, May 2014.
8. Donald Beaver and Joan Feigenbaum. Hiding instances in multioracle queries. In *STACS 90*, volume 415 of *Lecture Notes in Computer Science*, pages 37–48. Springer Berlin Heidelberg, 1990.
9. D. Beaver, J. Feigenbaum, J. Kilian, and P. Rogaway. Locally random reductions: Improvements and applications. *Journal of Cryptology*, 10(1):17–36, 1997.
10. M. Abadi, J. Feigenbaum, and J. Kilian. On hiding information from an oracle. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 195–203. ACM, 1987.
11. Victor Boyko, Marcus Peinado, and Ramarathnam Venkatesan. Speeding up discrete log and factoring based schemes via precomputations. In *Advances in Cryptology EUROCRYPT'98*, volume 1403 of *Lecture Notes in Computer Science*, pages 221–235. Springer Berlin Heidelberg, 1998.
12. Peter de Rooij. On schnorrs preprocessing for digital signature schemes. *Journal of Cryptology*, 10(1):1–16, 1997.
13. Tsutomu Matsumoto, Koki Kato, and Hideki Imai. Speeding up secret computations with insecure auxiliary devices. In *Advances in Cryptology CRYPTO 88*, volume 403 of *Lecture Notes in Computer Science*, pages 497–506. Springer New York, 1990.
14. Phong Q. Nguyen, Igor E. Shparlinski, and Jacques Stern. Distribution of modular sums and the security of the server aided exponentiation. In *Cryptography and Computational Number Theory*, volume 20 of *Progress in Computer Science and Applied Logic*, pages 331–342. Birkhauser Basel, 2001.
15. C.P. Schnorr. Efficient identification and signatures for smart cards. In Jean-Jacques Quisquater and Joos Vandewalle, editors, *Advances in Cryptology EUROCRYPT 89*, volume 434 of *Lecture Notes in Computer Science*, pages 688–689. Springer Berlin Heidelberg, 1990.
16. Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, 1991.
17. Marten Van Dijk, Dwaine Clarke, Blaise Gassend, G.Edward Suh, and Srinivas Devadas. Speeding up exponentiation using an untrusted computational resource. *Designs, Codes and Cryptography*, 39:253–273, 2006.
18. Marc Fischlin and Roger Fischlin. Efficient non-malleable commitment schemes. *Journal of Cryptology*, 22(4):530–571, 2009.
19. Ronald Cramer, Ivan Damgard, and Berry Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In *Advances in Cryptology CRYPTO 94*, volume 839 of *Lecture Notes in Computer Science*, pages 174–187. Springer Berlin Heidelberg, 1994.
20. Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Proceedings of CRYPTO 84 on Advances in Cryptology*, pages 10–18. Springer-Verlag New York, Inc., 1985.
21. Xu Ma, Jin Li, and Fangguo Zhang. Outsourcing computation of modular exponentiations in cloud computing. *Cluster Computing*, 16(4):787–796, 2013.

22. Jie Liu, Bo Yang, and Zhiguo Du. Outsourcing of verifiable composite modular exponentiations. In *INCoS*, pages 546–551, 2013.
23. Andrew C. Yao. Protocols for secure computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, SFCS '82, pages 160–164. IEEE Computer Society, 1982.
24. Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in Cryptology EUROCRYPT 99*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer Berlin Heidelberg, 1999.
25. Dan Boneh, Eu-Jin Goh, and Kobbi Nissim. Evaluating 2-dnf formulas on ciphertexts. In *Theory of Cryptography*, volume 3378 of *Lecture Notes in Computer Science*, pages 325–341. Springer Berlin Heidelberg, 2005.
26. Craig Gentry. *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford, CA, USA, 2009.
27. Joe Kilian. Founding crytpography on oblivious transfer. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, pages 20–31, New York, NY, USA, 1988. ACM.
28. Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, New York, NY, USA, 2004.
29. Yehuda Lindell and Benny Pinkas. A proof of security of yaos protocol for two-party computation. *Journal of Cryptology*, 22(2):161–188, 2009.
30. Giovanni Di Crescenzo, Tal Malkin, and Rafail Ostrovsky. Single database private information retrieval implies oblivious transfer. In *Advances in Cryptology EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 122–138. Springer Berlin Heidelberg, 2000.
31. Ari Juels and Michael Szydlo. A two-server, sealed-bid auction protocol. In Matt Blaze, editor, *Financial Cryptography*, volume 2357 of *Lecture Notes in Computer Science*, pages 72–86. Springer Berlin Heidelberg, 2003.
32. Helger Lipmaa. Verifiable homomorphic oblivious transfer and private equality test. In *Advances in Cryptology - ASIACRYPT 2003*, volume 2894 of *Lecture Notes in Computer Science*, pages 416–433. Springer Berlin Heidelberg, 2003.
33. Julien Bringer, Herv Chabanne, and Alain Patey. Shade: Secure hamming distance computation from oblivious transfer. In *Financial Cryptography and Data Security*, volume 7862 of *Lecture Notes in Computer Science*, pages 164–176. Springer Berlin Heidelberg, 2013.
34. David Chaum. Blind signatures for untraceable payments. In David Chaum, RonaldL. Rivest, and AlanT. Sherman, editors, *Advances in Cryptology, Proceedings of CRYPTO '82*, pages 199–203. Springer US, 1983.
35. Stefan A. Brands. *Rethinking Public Key Infrastructures and Digital Certificates: Building in Privacy*. MIT Press, Cambridge-London, 2000.
36. H. Cohen, G. Frey, R. Avanzi, C. Doche, T. Lange, K. Nguyen, and F. Vercauteren. *Handbook of elliptic and hyperelliptic curve cryptography*. Chapman & Hall, Boca Raton, FL, 1st edition, 2006.
37. Razvan Barbulescu, Pierrick Gaudry, Antoine Joux, and Emmanuel Thomé. A quasi-polynomial algorithm for discrete logarithm in finite fields of small characteristic. *CoRR*, abs/1306.4244, 2013.
38. Keith Conrad. Stirlings formula, 2011. Available at `http://www.math.uconn.edu/~kconrad/blurbs/analysis/stirling.pdf` (Retrieved in 28.05.2015).
39. Ellis Horowitz and Sartaj Sahni. Computing partitions with applications to the knapsack problem. *J. ACM*, 21(2):277–292, April 1974.
40. A.K. Lenstra, Jr. Lenstra, H.W., and L. Lovsz. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261(4):515–534, 1982.
41. MatthijsJ. Coster, Antoine Joux, BrianA. LaMacchia, AndrewM. Odlyzko, Claus-Peter Schnorr, and Jacques Stern. Improved low-density subset sum algorithms. *computational complexity*, 2(2):111–128, 1992.
42. C.P. Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Mathematical Programming*, 66(1-3):181–199, 1994.
43. Yonatan Aumann and Yehuda Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. *Journal of Cryptology*, 23(2):281–343, April 2010.

44. Giovanni Di Crescenzo and Rafail Ostrovsky. On concurrent zero-knowledge with preprocessing. In *Advances in Cryptology CRYPTO 99*, volume 1666 of *Lecture Notes in Computer Science*, pages 485–502. Springer Berlin Heidelberg, 1999.
45. TorbenPryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Advances in Cryptology CRYPTO 91*, volume 576 of *Lecture Notes in Computer Science*, pages 129–140. Springer Berlin Heidelberg, 1992.
46. Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers. A secure and optimally efficient multi-authority election scheme. In *Proceedings of the 16th Annual International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT'97, pages 103–118, Berlin, Heidelberg, 1997. Springer-Verlag.
47. Rosario Gennaro. Multi-trapdoor commitments and their applications to proofs of knowledge secure under concurrent man-in-the-middle attacks. In *Advances in Cryptology CRYPTO 2004*, volume 3152 of *Lecture Notes in Computer Science*, pages 220–236. Springer Berlin Heidelberg, 2004.
48. Martin Seysen. Using an rsa accelerator for modular inversion. In *Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings*, volume 3659 of *Lecture Notes in Computer Science*, pages 226–236. Springer, 2005.