Delegating a Product of Group Exponentiations with Application to Signature Schemes

Giovanni Di Crescenzo\textsuperscript{1}, Matluba Khodjaeva\textsuperscript{2}, Delaram Kahrobaei\textsuperscript{3}, Vladimir Shpilrain\textsuperscript{4}

\textsuperscript{1} Perspecta Labs E-mail: gdicrescenzo@perspectalabs.com
\textsuperscript{2} CUNY John Jay College of Criminal Justice. E-mail: mkhodjaeva@jjay.cuny.edu
\textsuperscript{3} Department of Computer Science, University of York (UK). E-mail: delaram.kahrobaei@york.ac.uk
\textsuperscript{4} City University of New York. E-mail: shpil@groups.sci.ccny.cuny.edu

Abstract. Group exponentiations are important primitive operations used in many public-key cryptosystems and, more generally, cryptographic protocols. To expand the applicability of these solutions to computationally weaker devices, it has been advocated since [29] that such primitive operations are delegated from a computationally weaker client (i.e., capable of performing a small number of modular multiplications) to a computationally stronger server, while maintaining the privacy of the client’s input exponent and detecting any malicious server’s attempt to convince the client of an incorrect exponentiation result. Very few results are known in this area, and only recently, a protocol to provably delegate a fixed-based exponentiation has been presented in [22]. In this paper we show that a product of many fixed-based exponentiations, over a cyclic group with certain properties, can be privately and securely delegated by keeping the client’s online number of modular multiplications about the same as for delegating a single exponentiation. We use this result to delegate the first cryptographic schemes: the well-known digital signature schemes by El-Gamal [1], Schnoor [2] and Okamoto [3] over the $q$-order subgroup in $\mathbb{Z}_p$, as well as their variants based on elliptic curves.

Keywords: Secure Delegation, Modular Exponentiations, Discrete Logarithms, Cryptography, Group Theory, Elliptic Curves

1 Introduction

Server-aided cryptography is an active research direction addressing the problem of computationally weak clients delegating cryptographic computations to computationally powerful servers. Ideas related to this area have circulated in the literature already many years ago (see, e.g., [17], which introduced ‘wallets with observers’ where a third party, such as a bank, installs hardware on a user’s computer to facilitate its future computations). Recently, this area is
seeing an increased interest because of application scenario like cloud computing and computations with low-power devices, such as RFIDs.

The first formal model for outsourcing of cryptographic operations was introduced in [29], where the authors especially studied outsourcing of modular exponentiation, as this operation is a cornerstone of so many cryptographic protocols. In this model, we have a client, with an input $x$, who outsources to one or more servers the computation of a function $F$ on the client’s input, and the main challenges are:

1. **privacy**: only minimal or no information about $x$ should be revealed to the servers;
2. **security**: the servers should not be able, except possibly with very small probability, to convince the client to accept a result different than $F(x)$; and
3. **efficiency**: the client’s computation time should be much smaller than computing $F(x)$ without outsourcing the computation.

Moreover, in all previous work in the area, relatively expensive offline computation can be performed and stored on the client’s device, and a client can perform in the online phase (a not large number of) less expensive operations like multiplications, an assumption that might be reasonable even in computationally challenged devices, in light of recent advances (see, for instance, [5], showing how to practically implement group multiplication, for a specific group, and a related public-key cryptosystem, using RFID tags).

In [29], the authors studied secure delegation of exponentiation to 2 servers of which at most one was malicious, and to 1 server, who was honest on almost all inputs. Recently, in [22], we showed how to delegate a single fixed-based exponentiation in cyclic groups to a single, possibly malicious, server.

**Our Contributions.** In this paper we show that a product of many fixed-base exponentiations, over a cyclic group with certain properties, can be privately and securely delegated to a single, possibly malicious, server, by keeping the client’s online number of modular multiplications about the same as for delegating a single exponentiation. As an example of the efficiency achieved, a product of $m$ exponentiations with 2048-bit exponents can be delegated by a client that only uses less than 150 modular multiplications.

We use this result to delegate the first cryptographic schemes: the well-known digital signature schemes by El-Gamal [1], Schnoor [2] and Okamoto [3] over the $q$-order subgroup in $\mathbb{Z}_p$, as well as their variants based on elliptic curves. Previously, only primitive operations like group exponentiations or inverses were delegated, and no complete cryptosystem was delegated to a single, possibly malicious, server. As an example of the efficiency achieved here, Okamoto’s scheme normally requiring a verification of 3 exponentiations with 2048-bit exponents can now be delegated by a client that only uses less than 150 modular multiplications.

In the process, we formally define delegation of digital signature schemes, and provide a conversion theorem showing that a non-delegated signature scheme can be converted into a delegated signature scheme using a suitable delega-
tion protocol for a desired primitive (e.g., group exponentiation, product of exponents, etc.). This suggests a more appropriate, simulation-based, definition of privacy in delegation protocols (many previous definitions targeted indistinguishability-based definitions of privacy).

**Related Work.** The first formal model for secure delegation protocols was presented in [29]. There, a secure delegation protocol is formally defined as essentially a secure function evaluation (in the sense of the notion first proposed in [37]) of the client’s function delegated to the server. Follow-up models from [26] and [16, 22] define separate requirements of correctness, (input) privacy and (result) security. There, privacy is defined in the sense of the adversary’s indistinguishability of two different inputs from the client, even after corrupting the server; and security is defined in the sense of the adversary’s inability to convince the client of an incorrect function output, even after corrupting the server. Our paper’s protocols follow the latter definitions but can be shown to satisfy the former.

Previous work in [16] is the only work we are aware of proposing secure delegation protocols for exponentiation, as well as inverse computation, over general groups. There, our main problem is solved with a protocol that bounds the probability of adversary’s success in violating security by a constant. A simplified version of this protocol is used as a starting point for our paper’s protocols.

We can partition all other (single-server) secure delegation protocols we are aware of in 3 main classes, depending on whether they delegate (a) an arbitrary polynomial-time function; (b) exponentiation in a specific group; or (c) other specific computations (e.g., linear algebra).

With respect to (a), [26] proposed a protocol using garbled circuits [37] and fully homomorphic encryption [33]. This protocol delegates functions in settings where the client is powerful enough to run encryption and decryption algorithms of a fully homomorphic encryption scheme, but not enough to homomorphically evaluate a circuit that computes decryption steps in the garbling scheme for the function. Different protocols, not using garbled circuits, were later proposed in [13]. These protocols delegate function the client is assumed to be powerful enough to run encryption and decryption algorithms of a fully homomorphic encryption scheme, but not enough to homomorphically evaluate the delegated function.

With respect to (b), protocols were proposed mainly for specific groups related to discrete logarithm problems and factoring problems (see, e.g., [29, 18, 23, 36, 32, 22] and references therein).

With respect to (c), protocols for linear algebra and/or scientific computation were proposed in, e.g., [10, 12, 11].

Finally, results in [35] include reducing the security probability to a lower constant in secure delegation protocols for exponentiation in specific groups.
2 Definitions: Groups with Efficient Membership Proofs

In this section we formally define group notations and definitions that will be used in the rest of the paper.

**Group notations and definitions.** Let \((G, \times)\) be a group, let \(\sigma\) be its computational security parameter, and let \(L\) denote the length of the binary representation of elements in \(G\). Typically, in cryptographic applications we set \(L\) as about equal to \(\sigma\).

We also assume that \((G, \times)\) is cyclic, has order \(q\), and denote as \(g_i\) one of its generators and \(g_1, \ldots, g_m \in G\). By \(y = g_1^{x_1} \cdots g_m^{x_m} = \prod_{i=1}^{m} g_i^{x_i}\) we denote the product of \(m\) exponentiations (in \(G\)). Let \(Z_q = \{0, 1, \ldots, q - 1\}\), and let \(F_{q, g_i} : (Z_q \times \ldots \times Z_q) \to G\) denote the function that maps every \((x_1, \ldots, x_m) \in Z_q \times \ldots \times Z_q\) to the product of \(m\) exponentiations (in \(G\)) of base \(g_i\) to the \(x_i\)th power. By \(\text{desc}(F_{q, g_i})\) we denote a conventional description of the function \(F_{q, g_i}\) that includes its semantic meaning as well as generator \(g_i\), order \(q\) and the efficient algorithms computing multiplication and inverses in \(G\). By \(t_\text{exp}(\ell)\) we denote a parameter denoting the number of multiplications in \(G\) used to compute an exponentiation (in \(G\)) of a group value to an arbitrary \(\ell\)-bit exponent. By \(t_\text{m,exp}(\ell)\) we denote a parameter denoting the number of multiplications in \(G\) used to compute \(m\) exponentiations (in \(G\)) of the same group value to \(m\) arbitrary \(\ell\)-bit exponents. By \(t_\text{prod,m,exp}(\ell)\) we denote the max number of group multiplications used to compute a product of \(m\) exponentiations of (possibly different) group elements to \(m\) arbitrary \(\ell\)-bit exponents.

In [8], based on [4], the authors present the algorithm, called the ‘FastMult’ for the function computing product of \(m\) exponentiations (i.e., \(F(x_1, \ldots, x_m) = y = \prod_{i=1}^{m} g_i^{x_i}\)). Naive way to evaluate \(y\) is to compute \(y_i = g_i^{x_i}\) for \(i = 1, \ldots, m\) and then compute \(y = \prod_{i=1}^{m} y_i\). This takes \(m \cdot t_\text{exp}(\ell) + m - 1\) multiplications. Using square and multiply algorithm this may take in average \(\frac{3m\ell}{2} + m - 1\) multiplications. However using ‘FastMult’ algorithm to evaluate \(y\) requires in average \(\frac{m\ell}{2} + \ell\) multiplications.

We define an **efficiently verifiable membership protocol** for \(G\) as a one-message protocol, denoted as the pair \((\text{mProve}, \text{mVerify})\) of algorithms, satisfying

1. **completeness:** for any \(w \in G\), \(\text{mVerify}(w, \text{mProve}(w)) = 1\);
2. **soundness:** for any \(w \not\in G\), and any \(\text{mProve}'\), \(\text{mVerify}(w, \text{mProve'}(w)) = 0\);
3. **efficient verifiability:** the number of multiplications \(t_\text{m,verify}(\sigma)\) in \(G\) executed by \(\text{mVerify}\) is \(o(t_\text{exp})\);
4. **efficient provability:** the number of multiplications \(t_\text{m,prove}(\sigma)\) in \(G\) executed by \(\text{mProve}\) is not significantly larger than \(t_\text{exp}\).

We say that a group is **efficient** if its description is short (i.e., has length polynomial in \(\sigma\)), its associated operation \(\times\) and the inverse operation are efficient (i.e., they can be executed in time polynomial in \(\sigma\)), and it has an efficiently verifiable membership protocol. Note that for essentially all cyclic groups frequently used in cryptography, the description is short and its associated \(\times\) and inverse
operations can be run in time polynomial in \(\sigma\). The only non-trivial property to establish is whether the group has an efficiently verifiable membership protocol. In the rest of the paper we present our results for any arbitrary efficient cyclic group, of which we now show two examples that are often used in cryptography and that do have efficiently verifiable membership protocols.

Example 1: \((G, \times) = (G_q, \cdot \mod p)\), for large primes \(p, q\) such that \(p = kq + 1\), where \(k \neq q\) is another prime and \(G_q\) is the \(q\)-order subgroup of \(\mathbb{Z}_p^*\). This group is one of the most recommended for cryptographic schemes like the Diffie-Hellman key exchange protocol [24], El-Gamal encryption [25], Cramer-Shoup encryption [20], DSA etc. It is known that by Sylow’s theorem, \(G_q\) in this case is the only subgroup of order \(q\) in the group \(\mathbb{Z}_p^*\) (i.e. \(g^q = 1 \mod p\) if and only if \(g \in G_q\)). Also, the set of elements of \(G_q\) is precisely the set of \(k\)-th powers of elements of \(\mathbb{Z}_p^*\). Thus, an efficiently verifiable membership protocol can be built as follows:

1. on input \(w\), \(\text{mProve}\) computes \(r = w^{(q+1)/k} \mod p\) and returns \(r\);
2. on input \(w, r\), \(\text{mVerify}\) returns 1 if \(w = r^k \mod p\) and 0 otherwise.

The completeness and soundness properties of this protocol are easily seen to hold. The efficient provability follows by noting that \(\text{mProve}\) only performs 1 exponentiation \(\mod p\). The efficient verifiability property follows by noting that \(\text{mVerify}\) requires one exponentiation \(\mod p\) to the \(k\)-th power. We note that the case when \(k\) is small, where \(\text{mVerify}\) is very efficient, is a very common group setting for discrete logarithm based cryptographic protocols.

Example 2: \((G, +) = (E(F_p), \text{point addition})\), for a large prime \(p > 3\): an elliptic curve \(E\) over a field \(F_p\), is the set of pairs \((x, y) \in F_p\) that satisfy the Weierstrass equation

\[ y^2 = x^3 + ax + b \mod p, \]

together with the imaginary point at infinity \(\mathcal{O}\), where \(a, b \in F_p\) and \(4a^3+27b^2 \neq 0 \mod p\). The elliptic curve defined above is denoted by \(E(F_p)\). This group is one of the most recommended for cryptographic schemes like Elliptic-curve Diffie-Hellman key exchange protocol, Elliptic-curve ElGamal encryption, etc. Moreover, many discrete logarithm based cryptographic protocols defined over the set \(\mathbb{Z}_p\) in Example 1 can be rewritten as defined over \(E(F_p)\). For those protocols it is necessary to evaluate scalar multiplication (i.e. for \(a \in F_p\) and \(P \in E(F_p)\) compute \(aP\), meaning add \(P\) to itself \(a\) times) in the group \(E(F_p)\). One way to evaluate it is by using the textbook “double-and-add” algorithm. Another way to evaluate is when client is computationally weak then it can delegate to a computationally more powerful device satisfying all four requirements described above with the first protocol described in [22]. An efficiently verifiable membership protocol can be built as follows:

1. on input \((x, y)\), \(\text{mProve}\) does nothing;
2. on input \((x, y)\), \(\text{mVerify}\) returns 1 if \(y^2 = x^3 + ax + b \mod p\) and 0 otherwise.

The completeness, soundness, efficient provability properties of this protocol are easily seen to hold. The efficient verifiability property follows by noting that \(\text{mVerify}\) performs only 4 multiplications \(\mod p\).
3 Definitions: Delegated Protocols

In this section we formally define delegation protocols, and their correctness, security, privacy and efficiency requirements, mainly relying on the definition approach from [22], which in turn builds on those from [26, 29]. One new aspect in our definition (important for our results in later sections) is that we use a simulation-based definition of privacy instead of the indistinguishability-based definition in [22].

Basic notations. The expression \( y \leftarrow T \) denotes the probabilistic process of randomly and independently choosing \( y \) from set \( T \). The expression \( y \leftarrow A(x_1, x_2, \ldots) \) denotes the (possibly probabilistic) process of running algorithm \( A \) on input \( x_1, x_2, \ldots \) and any necessary random coins, and obtaining \( y \) as output. The expression \( (z_A, z_B, tr) \leftarrow (A(x_1, x_2, \ldots), B(y_1, y_2, \ldots)) \) denotes the (possibly probabilistic) process of running an interactive protocol between \( A \), taking as input \( x_1, x_2, \ldots \) and any necessary random coins, and \( B \), taking as input \( y_1, y_2, \ldots \) and any necessary random coins, where \( z_A, z_B \) are \( A \) and \( B \)'s final outputs, respectively, at the end of this protocol’s execution, and \( tr \) denotes the tuple of messages exchanged between \( A \) and \( B \). We denote a distribution as a sequence \( \{R_1; \ldots; R_n : x\} \), where \( R_1, \ldots, R_n \) are random processes and \( x \) denotes a variable set as a result of their execution.

System scenario, entities, and protocol. We consider a system with a single client, denoted as \( C \), and a single server, denoted as \( S \). As a client’s computational resources are expected to be more limited than a server’s ones, \( C \) is interested in delegating the computation of specific functions to \( S \). We assume that the communication link between \( C \) and \( S \) is private or not subject to confidentiality, integrity, or replay attacks, and note that such attacks can be separately addressed using known techniques in cryptography and security. As in all previous work in the area, we consider a model with an offline phase, where, say, exponentiations to random exponents can be precomputed and made somehow available onto \( C \)'s device. This model has been justified in several ways, all appealing to different application settings. In the presence of a trusted party (say, setting up \( C \)'s device), the trusted party can simply perform the precomputed exponentiations and store them on \( C \)'s device. If no trusted party is available, in the presence of a pre-processing phase where \( C \)'s device does not have significant computation constraints, \( C \) can itself perform the precomputed exponentiations and store them on its own device.

Let \( \sigma \) denote the computational security parameter (i.e., the parameter derived from hardness considerations on the underlying computational problem), and let \( \lambda \) denote the statistical security parameter (i.e., a parameter such that events with probability \( 2^{-\lambda} \) are extremely rare). Both parameters are expressed in unary notation (i.e., \( 1^\sigma, 1^\lambda \)).

Let \( F : \text{Dom}(F) \to \text{CoDom}(F) \) be a function, where \( \text{Dom}(F) \) denotes \( F \)'s domain, \( \text{CoDom}(F) \) denotes \( F \)'s co-domain, and \( \text{desc}(F) \) denotes \( F \)'s description. Assuming \( \text{desc}(F) \) is known to both \( C \) and \( S \), and input \( x \) is known only to \( C \), we define a client-server protocol for the outsourced computation of \( F \) in
the presence of an offline phase as a 2-party, 2-phase, communication protocol between $C$ and $S$, denoted as $(C(1^\sigma, 1^\lambda, \text{desc}(F), x), S(1^\sigma, 1^\lambda, \text{desc}(F)))$, and consisting of the following steps:

1. $pp \leftarrow \text{Offline}(1^\sigma, 1^\lambda, \text{desc}(F))$
2. $(y_C, y_S, tr) \leftarrow (C(1^\sigma, 1^\lambda, \text{desc}(F), pp, x), S(1^\sigma, 1^\lambda, \text{desc}(F)))$

As discussed above, Step 1 is executed in an offline phase, when the input $x$ to the function $F$ is not yet available. Step 2 is executed in the online phase, when the input $x$ to the function $F$ is available to $C$. At the end of both phases, $C$ learns $y_C$ (intended to be $= y$) and $S$ learns $y_S$ (usually an empty string in this paper). We will often omit $\text{desc}(F), 1^\sigma, 1^\lambda$ for brevity of description. Executions of outsourced computation protocols can happen sequentially (each execution starting after the previous one is finished), or concurrently ($S$ runs at the same time one execution with each one of many clients).

**Correctness Requirement.** Informally, the (natural) correctness requirement states that if both parties follow the protocol, $C$ obtains some output at the end of the protocol, and this output is, with high probability, equal to the value obtained by evaluating function $F$ on $C$’s input. A formal definition follows.

**Definition 1.** Let $\sigma, \lambda$ be the security parameters, let $F$ be a function, and let $(C, S)$ be a client-server protocol for the outsourced computation of $F$. We say that $(C, S)$ satisfies $\delta_c$-correctness if for any $x$ in $F$’s domain, it holds that

$$\Pr\left[ \text{out} \leftarrow \text{CorrExp}_F(1^\sigma, 1^\lambda) : \text{out} = 1 \right] \geq \delta_c,$$

for some $\delta_c$ close to 1, where experiment CorrExp is detailed below:

**CorrExp$_F(1^\sigma, 1^\lambda)$**

1. $pp \leftarrow \text{Offline}(\text{desc}(F))$
2. $(y_C, y_S, tr) \leftarrow (C(pp, x), S)$
3. if $y_C = F(x)$ then \textbf{return}: 1
   
   else \textbf{return}: 0

**Security Requirement.** Informally, the most basic security requirement would state the following: if $C$ follows the protocol, a malicious adversary corrupting $S$ cannot convince $C$ to obtain, at the end of the protocol, some output $y'$ different from the value $y$ obtained by evaluating function $F$ on $C$’s input $x$. To define a stronger and more realistic security requirement, we augment the adversary’s power so that the adversary can even choose $C$’s input $x$, before attempting to convince $C$ of an incorrect output. We also do not restrict the adversary to run in polynomial time. A formal definition follows.

**Definition 2.** Let $\sigma, \lambda$ be the security parameters, let $F$ be a function, and let $(C, S)$ be a client-server protocol for the outsourced computation of $F$. We say that $(C, S)$ satisfies $\epsilon_s$-security against a malicious adversary if for any algorithm $A$, it holds that

$$\Pr\left[ \text{out} \leftarrow \text{SecExp}_{F,A}(1^\sigma, 1^\lambda) : \text{out} = 1 \right] \leq \epsilon_s,$$
for some $\epsilon_s$ close to 0, where experiment SecExp is detailed below:

\[ \text{SecExp}_{F,A}(1^\sigma, 1^\lambda) \]
1. $pp \leftarrow \text{Offline}(\text{desc}(F))$
2. $(x, aux) \leftarrow A(\text{desc}(F))$
3. $(y', aux, tr) \leftarrow (C(pp, x), A(aux))$
4. if $y' = \perp$ or $y' = F(x)$ then return: 0
   else return: 1.

**Privacy Requirement.** Informally, the privacy requirement should guarantee the following: if $C$ follows the protocol, a malicious adversary corrupting $S$ cannot obtain any information about $C$’s input $x$ from a protocol execution. This is formalized here by extending the simulation-based approach typically used in various formal definitions for cryptographic primitives. That is, there exists an efficient algorithm, called the simulator, that generates a tuple of messages distributed exactly like those in a random execution of the protocol. A formal definition follows.

**Definition 3.** Let $\sigma, \lambda$ be the security parameters, let $F$ be a function, and let $(C,S)$ be a client-server protocol for the outsourced computation of $F$. We say that $(C,S)$ satisfies privacy (in the sense of simulation) against a malicious adversary if there exists an efficient algorithm $\text{Sim}$ such that for any efficient adversary $A$ and any input $x$ to $C$, the following two distributions are equal:

\[
D_{\text{sim}} = \{tr \leftarrow \text{Sim}(\text{desc}(F), 1^\sigma, 1^\lambda) : tr\}
\]
\[
D_{\text{prot}} = \{pp \leftarrow \text{Offline}(\text{desc}(F)); (y_C, y_A, tr_x) \leftarrow (C(pp, x), A(aux)) : tr_x\}
\]

**Efficiency Metrics and Requirements.** Let $(C,S)$ be a client-server protocol for the outsourced computation of function $F$. We say that $(C,S)$ has efficiency parameters $(t_F, t_F, t_C, t_S, cc, mc)$, if $F$ can be computed (without outsourcing) using $t_F(\sigma, \lambda)$ atomic operations, $C$ can be run in the offline phase using $t_C(\sigma, \lambda)$ atomic operations and in the online phase using $t_S(\sigma, \lambda)$ atomic operations, $S$ can be run using $t_S(\sigma, \lambda)$ atomic operations, $C$ and $S$ exchange a total of at most $mc$ messages, of total length at most $cc$. In our analysis, we only consider the most expensive group operations as atomic operations (e.g., group multiplications and/or exponentiation), and neglect lower-order operations (e.g., equality testing, additions and subtractions between group elements). While we naturally try to minimize all these protocol efficiency metrics, our main goal is to design protocols where

1. $t_C(\sigma, \lambda) << t_F(\sigma, \lambda)$, and
2. $t_S(\sigma, \lambda)$ is not significantly larger than $t_F(\sigma, \lambda),$

based on the underlying assumption, consistent with the state of the art in cryptographic implementations, for many group types, that group multiplication requires significantly less computing resources than group exponentiation.
4 Delegating a Product of Exponentiations

In this section we present our protocol for delegation of a product of (fixed-base) exponentiations in a large class of groups used in cryptographic protocols, which provably satisfies correctness, simulation-based privacy, security with exponentially small probability, and various desirable efficiency properties (most notably, the client’s online complexity is dominated by a single exponentiation to a smaller exponent).

We first formally state our result, then describe the protocol, and finally prove its correctness, security, privacy and efficiency properties.

Formal theorem statement. We obtain the following

**Theorem 1.** Let \((G, \times)\) be an efficient cyclic group, let \(\sigma\) be its computational security parameter, and let \(\lambda\) be a statistical security parameter. There exists (constructively) a client-server protocol \((C,S)\) for delegating the computation of function \(F_{g_i,q}\), which satisfies

1. \(\delta_c\)-correctness, for \(\delta_c = 1\);
2. \(\epsilon_s\)-security, for \(\epsilon_s \leq \frac{1}{2^\lambda}\);
3. \(\epsilon_p\)-privacy, for \(\epsilon_p = 0\);
4. efficiency with parameters \((t_F, t_S, t_P, t_C, cc, mc)\), where
   - \(t_F\) is \(t_{\text{prod},m,\exp}(\sigma)\);
   - \(t_S\) is \(2 t_{\text{prod},m,\exp}(\sigma) + 2 t_{\text{mProve}}(\sigma)\);
   - \(t_P\) is \(2 t_{\text{prod},m,\exp}(\sigma)\), with random exponents from \(\mathbb{Z}_q\);
   - \(t_C\) is \(t_{\text{exp}}(\lambda) + 2 t_{\text{mVerify}}(\sigma) + 2\) multiplications in \(G\) and 1 multiplication in \(\mathbb{Z}_q\);
   - \(cc = 4\) elements in \(G\) and 2\(m\) in \(\mathbb{Z}_q\)
   - \(mc = 2\).

The main takeaway from Theorem 1 is that \(C\) outsources the computation of product of multiple (i.e. \(m\)) exponentiations with a \(\sigma\)-bit exponents to \(S\) while \(C\) only performs an exponentiation with a \(\lambda\)-bit exponent, 2 group membership verifications in \(G\), 2 multiplications in \(G\) and 1 modular multiplication in \(\mathbb{Z}_q\). In other words, \(C\)’s online complexity is essentially the same as that in a delegation protocol for a single exponentiation (as in the protocol from [22]). Instead, a direct use of the protocol to delegate a single exponentiation from [22] would have a worse performance by a multiplicative factor of \(m\).

Also remarkable are the running time of \(S\), who only performs 2 product of \(m\) exponentiations and 2 group membership proof generations in \(G\). In other words, \(S\)’s complexity is only about twice as that in a non-delegated computation of the same function.

Even in the offline phase, only 2 fixed-base exponentiations with random exponents are needed by the client to later compute a product of \(m\) exponentiations. Finally, the protocol only requires 2 messages, which is clearly minimal in this model, and only requires the communication of 4 elements in \(G\) and 2\(m\) elements in \(\mathbb{Z}_q\).
The group membership test is realized via the assumed efficiently verifiable group membership protocol. While we do not know of such a protocol for any arbitrary cyclic group, we showed in Section 2 that groups commonly used in cryptography have one.

**Formal description of protocol** \((C,S)\). Let \(G\) be an efficient cyclic group, and let \((\text{mProve}, \text{mVerify})\) denote its efficiently verifiable membership protocol.

**Input to** \(C\) and \(S\): \(1^{\sigma}, 1^{\lambda}, \text{desc}(F_{g_1,\ldots,g_m,q})\)

**Input to** \(C\): \(x_1,\ldots,x_m \in \mathbb{Z}_q\)

**Offline phase instructions:**
1. Randomly choose \(u_{i,j} \in \mathbb{Z}_q\) for \(i = 1,\ldots,m\) and \(j = 0,1\)
2. Set \(v_j = \prod_{i=1}^m g_i^{u_{i,j}}\) and store \((u_{1,j},\ldots,u_{m,j},v_j)\) on \(C\) for \(j = 0,1\)

**Online phase instructions:**
1. \(C\) randomly chooses \(b \in \{1,\ldots,2^\lambda\}\)
   \(C\) sets \(z_{i,0} := (x_i - u_{i,0}) \mod q\), \(z_{i,1} := (b \cdot x_i + u_{i,1}) \mod q\) for \(i = 1,\ldots,m\)
   \(C\) sends \(z_{i,j}\) to \(S\) for \(i = 1,\ldots,m, j = 0,1\)
2. \(S\) computes \(w_j = \prod_{i=1}^m g_i^{z_{i,j}}\) and \(\pi_j := \text{mProve}(w_j)\), for \(j = 0,1\)
   \(S\) sends \(w_0, w_1, \pi_0, \pi_1\) to \(C\)
3. If \(\text{mVerify}(w_j, \pi_j) = 0\) for some \(j \in \{0,1\}\), then
   \(C\) **returns:** \(\bot\) and the protocol halts
   \(C\) computes \(y := w_0 \cdot v_0\)
   \(C\) checks that
   \(w_1 = y^b \cdot v_1\), also called the ‘probabilistic test’
   \(\text{mVerify}(w_0, \pi_0) = \text{mVerify}(w_1, \pi_1) = 1\),
   also called the ‘membership test’
   if any one of these tests is not satisfied then
   \(C\) **returns:** \(\bot\) and the protocol halts
   \(C\) **returns:** \(y\)

**Properties of protocol** \((C,S)\): The efficiency properties are verified by protocol inspection.

- **Round complexity:** the protocol only requires one round, consisting of one message from \(C\) to \(S\) followed by one message from \(S\) to \(C\).
- **Communication complexity:** the protocol requires the transfer of 2 elements in \(G\) and 2 proofs of group membership from \(S\) to \(C\), and 2m elements in \(\mathbb{Z}_q\) from \(C\) to \(S\).
- **Runtime complexity:** During the offline phase, 2 product of \(m\) exponentiations in bases \(g_1,\ldots,g_m\) and with random \(\sigma\)-bit exponents are performed. This product of \(m\) exponentiations can be evaluated by the the algorithm \(\text{FastMult}\) in [8]. During the online phase, \(S\) computes 2 product of \(m\) exponentiations to \(\sigma\)-bit exponents in \(G\) and 2 group membership proofs; and \(C\) verifies 2 group membership proofs and computes 2 multiplications in \(G\), 1 modular multiplication in \(\mathbb{Z}_q\), and 1 exponentiation in \(G\) to a random exponent that is much smaller \((\leq 2^\lambda)\) than \(2^\sigma\).
The correctness property follows by showing that if $C$ and $S$ follow the protocol, $C$ always output $y = \prod_{i=1}^{m} g_i^{z_i}$. We show that the 2 tests performed by $C$ are always passed. The membership test is always passed since $w_j$ is computed by $S$ as $\prod_{i=1}^{m} g_i^{z_i}$, for $j = 0, 1$, and $g_1, \ldots, g_m$ are generators of group $G$; the probabilistic test is always passed since

$$w_1 = \prod_{i=1}^{m} g_i^{z_i,1} = \prod_{i=1}^{m} g_i^{x_i + u_i,1} = \left(\prod_{i=1}^{m} g_i^{x_i}\right)^b \prod_{i=1}^{m} g_i^{u_i,1} = y^b v_1.$$ 

This implies that $C$ never returns $\perp$, and thus returns $y$. To see that this returned value $y$ is the correct output, note that

$$y = w_0 * v_0 = \prod_{i=1}^{m} g_i^{z_i,0} * \prod_{i=1}^{m} g_i^{u_i,0} = \prod_{i=1}^{m} g_i^{x_i - u_i,0} * \prod_{i=1}^{m} g_i^{u_i,0} = \prod_{i=1}^{m} g_i^{x_i}.$$ 

The privacy property of the protocol against any arbitrary malicious $S$ follows by observing that the distribution of $C$’s only message to $S$ does not depend on values $x_1, \ldots, x_m$. This message is $(z_1,0, \ldots, z_m,0, z_1,1, \ldots, z_m,1)$ where $z_{i,0} = (x_i - u_{i,0})$ mod $q$, $z_{i,1} = (bx_i + u_{i,1})$ mod $q$, and $z_{i,0}$ and $z_{i,1}$ are uniformly and independently distributed in $Z_q$, as so are $u_{i,0}$ and $u_{i,1}$ for all $i = 1, \ldots, m$. Thus, a simulator $\text{Sim}$ can be defined by generating a tuple $(z_{1,0}, \ldots, z_{m,0}, z_{1,1}, \ldots, z_{m,1})$ of random and independent values in $Z_q$, and then generating the message $(w_0, w_1, \pi_0, \pi_1)$ by simply running the same instructions run by $S$ on input $(z_{1,0}, \ldots, z_{m,0}, z_{1,1}, \ldots, z_{m,1})$. We obtain that distribution $D_{\text{Sim}}$ and distribution $D_{\text{prot}}$ are identical since in both distribution $C$’s message contains $2m$ random and independent values in $Z_q$ and a message by $S$ computed in exactly the same way starting from $C$’s message.

To prove the security property against any malicious $S$ we need to compute an upper bound $\epsilon_\delta$ on the security probability that $S$ convinces $C$ to output a $y$ such that $y \neq F_{y,q}(x_1, \ldots, x_m)$. We start by defining the following events with respect to a random execution of $(C, S)$ where $C$ uses $x$ as input:

- $e_{y,\neq}$, defined as ‘$C$ outputs $y$ such that $y \neq F_{y,q}(x_1, \ldots, x_m)$’
- $e_\perp$, defined as ‘$C$ outputs $\perp$’

By inspection of $(C, S)$, we directly obtain the following fact.

Fact 3.1. If event $e_{y,\neq}$ happens then event $(\neg e_\perp)$ happens.

With respect to a random execution of $(C, S)$ where $C$ uses $x_1, \ldots, x_m$ as input, we now define the following events:

- $e_{1,b}$, defined as ‘$\exists$ exactly one $b$ such that $S$’s message $(w_0, w_1)$ satisfies $w_1 = (w_0 * v_0)^b * v_1$’
- $e_{>1,b}$, defined as ‘$\exists$ more than one $b$ such that $S$’s message $(w_0, w_1)$ satisfies $w_1 = (w_0 * v_0)^b * v_1$’.
By definition, events $e_{1,b}, e_{1,b}$ are each other’s complement event.

Now, let $i \in \{1, \ldots, m\}$. We observe that no information is leaked by $z_0, z_1$ about $x_i$ as: (a) for any $x_i \in \mathbb{Z}_q$, there is exactly one $u_0, u_1$ corresponding to $z_i$; that is, $u_0 = x_i - z_i$ mod $q$; (b) for any $x_i \in \mathbb{Z}_q$, for any $b \in \{1, \ldots, 2^\lambda\}$ chosen by $C$, there is exactly one $u_i$ corresponding to $z_i$; that is, $u_i = z_i - bx_i$ mod $q$ for all $i = 1, \ldots, m$. This implies that, since $u_0, u_1$ are uniformly and independently distributed in $\mathbb{Z}_q$, the distribution of $x_i$ conditioned on $z_0, z_1$ is also uniform in $\mathbb{Z}_q$. Furthermore, by essentially the same proof, protocol $(C, S)$ satisfies the following property: for any $x_i, z_0$ and $z_1$ do not leak any information about $b$ for $i = 1, \ldots, m$. This implies that all values in $\{1, \ldots, 2^\lambda\}$ are still equally likely even when conditioning over message $(z_0, z_1)$. Then, if event $e_{1,b}$ is true, the probability that $S$’s message $(w_0, w_1)$ satisfies the probabilistic test, is $\frac{1}{2^\lambda}$ divided by the number $2^\lambda$ of values of $b$ that are still equally likely even when conditioning over message $(z_0, z_1)$. We obtain the following

**Fact 3.2.** $\Pr[-e_{1,b} | e_{1,b}] \leq 1/2^\lambda$

We now show the main technical claim, saying that if $S$ is malicious then it cannot produce in step 2 of the protocol values $w_0', w_1'$ satisfying both of $C$’s 2 tests relatively to two distinct values $b_1, b_2 \in \{1, \ldots, 2^\lambda\}$:

Since $S$ can be malicious, in step 2 it can send arbitrary values to $C$. Differently saying, $C$ can send $w_j'$ for $j = 0, 1$ for $w_j' = w_j$ or $w_j' \neq w_j$, where $w_j = \prod_{i=1}^{m} g_i^{z_i}$. Since the group $G$ is cyclic, $g_i$ is generator of $G$ we consider $g_1$ is generator of the group $G$ and $C$ uses $\pi_0, \pi_1$ to check in step 3 that $w_j' \in G$, we can write

$w_j' = g_1^u \ast w_0$ and $w_j' = g_1^v \ast w_1$ for some $u, v \in \mathbb{Z}_q$

then $y = w_0' \ast v_0 = g_1^u \ast w_0 \ast v_0 = g_1^u \ast \prod_{i=1}^{m} g_i^{z_i}$. Now, recall that the goal of a malicious $S$ is to pass $C$’s two verification tests and force $C$’s output to be $y \neq \prod_{i=1}^{m} g_i^{z_i}$; then, assume that $u \neq 0$ mod $q$. Now, consider the following equivalent rewriting of $C$’s probabilistic test, obtained by variable substitutions and simplifications:

$w_1' = y^b \ast v_1$

$g_1^v \ast w_1 = \left( g_1^u \ast \prod_{i=1}^{m} g_i^{z_i} \right) \ast \prod_{i=1}^{m} g_i^{u_i}$

$g_1^v \ast \prod_{i=1}^{m} g_i^{z_i} = g_1^u \ast \prod_{i=1}^{m} g_i^{x_i+u_i}$

$g_1^v \ast \prod_{i=1}^{m} g_i^{x_i+u_i} = g_1^u \ast \prod_{i=1}^{m} g_i^{x_i+u_i}$

$g_1^v = g_1^u$

$v = ub \mod q.$
Notice that if \( u = 0 \mod q \) then the above calculation implies that \( v = 0 \mod q \), and thus \( S \) is honest, from which we derive that \( \epsilon_s = 0 \). Now consider the case \( S \) is dishonest, in which case we have that \( u \neq 0 \mod q \). We want to show that \( b \) is unique in this case. If there exist two distinct \( b_1 \) and \( b_2 \) such that

\[
ub_1 = v \mod q \quad \text{and} \quad ub_2 = v \mod q
\]

then \( u(b_1 - b_2) = 0 \mod q \) then \( b_1 - b_2 = 0 \mod q \) (i.e \( b_1 = b_2 \)) because \( u \neq 0 \mod q \). This shows that \( b \) is unique and we obtain the following fact.

**Fact 3.3.** \( \operatorname{Prob} [e_{>1,b}] = 0 \)

The rest of the proof consists of computing an upper bound \( \epsilon_s \) on the probability of event \( e_{y,\neq} \). We have the following

\[
\operatorname{Prob} [e_{y,\neq}] \leq \operatorname{Prob} [\neg e_\perp] = \operatorname{Prob} [e_{1,b}] \cdot \operatorname{Prob} [\neg e_\perp | e_{1,b}] + \operatorname{Prob} [e_{>1,b}] \cdot \operatorname{Prob} [\neg e_\perp | e_{>1,b}] \leq \operatorname{Prob} [e_{1,b}] \cdot \frac{1}{2^\lambda} \leq \frac{1}{2^\lambda},
\]

where the first inequality follows from Fact 3.1, the first equality follows from the definition of events \( e_{1,b}, e_{>1,b} \) and the conditioning rule, the second equality follows from Fact 3.3, and the second inequality follows from Fact 3.2.

We finally obtain that \( \epsilon_s = \operatorname{Prob} [e_{y,\neq}] = 2^{-\lambda} \), which concludes the proof for the security property for \((C,S)\). \(\square\)

**Remark 1.** Although we have only analyzed our protocol \((C,S)\) with respect to a single execution, we note that the proofs of its properties naturally extend to multiple sequential, parallel or concurrent executions of the same protocol (both offline and online phases).

## 5 Delegating Signature Schemes

In this section we show efficient, private and secure delegation schemes for well-known (i.e., ElGamal, Schoor and Okamoto’s) signature schemes using the delegation of a product of (fixed-base) exponentiation for cyclic groups from Section 4. We start the presentation by recalling in Section 5.1 the definition of signature schemes in the standard (i.e., non-delegated) model. In Section 5.2 we augment this definition so to additionally take into account eavesdropping attacks in the delegated model. Then, in Section 5.3 we show a general results that shows how to convert signature schemes in the non-delegated model into signature schemes in the delegated model by using a suitable delegation protocol. Finally, in Section 5.4 we show delegated ElGamal, Schoor and Okamoto’s signature schemes.
5.1 Definitions: Signature Schemes in the standard model

We now recall the definition of digital signature schemes in the standard (i.e., non-delegated) model.

**Notations and algorithm syntax.** An oracle, denoted as $O(\cdot)$, is a function. An oracle algorithm, denoted as $A^{O(\cdot)}$, is an algorithm that during its computation can repeatedly make a query to the oracle and obtain the corresponding oracle’s output.

In a signature scheme $SS$, we consider two types of parties: signers and verifiers, and three algorithms: a key-generation algorithm $KG$, a signing algorithm $Sign$, and a verification algorithm $Ver$, satisfying the following syntax and requirements.

On input a security parameter $1^\sigma$, algorithm $KG$ returns a public key $pk$ and a matching secret key $sk$. On input a message $m$ of arbitrary length, algorithm $Sign$ returns a signature $sig$. On input a putative message $m'$, and a putative signature $sig'$, algorithm $Ver$ returns a bit = 1 (resp., 0) to denote that $sig'$ is a valid (resp., not valid) signature of $m'$.

**Requirements: Correctness and Unforgeability.** Informally speaking, the correctness requirement states that if both signer and verifier correctly run the algorithms, the verifier can recognize the signer’s signature as valid; and the unforgeability requirement states that no efficient algorithm querying the signature oracle can produce a message with a valid signature. Formal definitions follow.

**Definition 4.** We say that $SS=(KG,Sign,Ver)$ satisfies $\delta$-correctness if for any message $m \in \{0,1\}^*$, it holds that

$$\text{Prob} \left[ (pk,sk) \leftarrow KG(1^\sigma); sig \leftarrow Sign(pk,sk,m) \right] : Ver(pk,m,sig) = 1 \geq \delta,$$

for some $\delta$ close to 1.

**Definition 5.** We say that the signature scheme $SS=(KG,Sign,Ver)$ satisfies existential $\epsilon$-unforgeability under chosen message attack (briefly, $\epsilon$-cma-EU) if for any efficient oracle algorithm $A$, it holds that

$$\text{Prob} \left[ out \leftarrow \text{SecExp}_{SS,A}(1^\sigma) \right] : out = 1 \leq \epsilon,$$

for some $\epsilon$ close to 0, where experiment $\text{SecExp}$ is detailed below:

$\text{SecExp}_{SS,A}(1^\sigma)$

1. $(pk,sk) \leftarrow KG(1^\sigma)$
2. $(m',sig') \leftarrow A^{Sign(pk,sk,\cdot)}(pk)$
3. Let $Q$ be the set of message queries made by $A$ to oracle $Sign(pk,sk,\cdot)$
4. if $m \in Q$ or $Ver(pk,sk,m',sig') = 0$ then return: 0
   else return: 1.
5.2 Definitions: Delegated Signature Schemes

We now define delegated signature schemes by suitably augmenting the definition of an associated non-delegated signature schemes from Section 3, based on an associated delegation protocol.

Notations and algorithm syntax. In a delegated signature scheme dSS, we consider three parties: a signer, a verifier, and a server, where during their computations the signer and/or the verifier may act as clients interacting with the server. Since we only consider one-round client-server delegation protocols, we model an execution of such protocols as a query performed by a client to a server oracle $S(desc(F),1^\sigma,1^\lambda,\cdot)$, taking as input $C$’s query message. Accordingly, we consider a delegated signature scheme dSS as a tuple containing oracle $S$ and 3 oracle algorithms: a key-generation algorithm $KG$, an oracle signing algorithm $Sign^S$, and an oracle verification algorithm $Verify^S$, where $S$ is a server oracle, and satisfying the following syntax and requirements. Note that we intentionally do not perform delegation during key generation as we think of delegation as most critically useful for online operations (such as signing a message and verifying a signature) and not so for offline operations (such as key generation). Then, the syntax of algorithms $KG$, $Sign^S$ and $Verify^S$ is derived by the syntax of algorithm $KG$, $Sign$ and $Ver$, as defined in Section 5.1, and the above defined syntax for oracle $S$. In our formal description, we will separate algorithms $Sign^S$ and $Verify^S$ into an offline-phase and online-phase version, for the purpose of minimizing the online complexity. However, to reduce notation in the model description, in this subsection we keep both offline and online version as a single algorithm.

Requirements: Correctness and Unforgeability. The requirements of correctness and unforgeability for dSS are also obtained by suitably augmenting those for SS. In the case of correctness, the extension is immediate. In the case of unforgeability, we replace the adversary $A$’s oracle $Sign$ with two oracles:

1. an augmented oracle $dSign(pk,sk,\cdot)$ which, on input message $m$, returns a signature $sig$ as well as the transcript of any queries to the server oracle $S$ performed by $KG$, $Sign$, and $Ver$ during the generation of $pk, sk, sig$;
2. the server oracle $S(desc(F),1^\sigma,1^\lambda,\cdot)$, which, on input $C$’s query message $qmes_C$, returns $S$’s response to $qmes_C$ in an execution of protocol $(C,S)$.

Note that by giving the adversary access to oracle $dSign$, we model the adversary’s eavesdropping attacks on executions of the delegation protocol between a signer (acting as client) and the server, as well as between a verifier (acting as client) and the server. Moreover, by giving the adversary access to oracle $S$, we model the adversary’s impersonation of a signer or verifier.

Formal definitions of correctness and unforgeability requirements for dSS follow.

Definition 6. Let $S$ be a server oracle. We say that the delegated signature scheme dSS=$\langle KG, Sign^S, Ver^S \rangle$ satisfies $\delta$-correctness if for any message $m \in$
\{0, 1\}^*$, it holds that

\[
\text{Prob}\left[ (\text{pk}, \text{sk}) \leftarrow \text{KG}(1^\sigma); \text{sig} \leftarrow \text{Sign}^S(\text{pk}, \text{sk}, \text{m}) \right] : \text{Ver}^S(\text{pk}, \text{m}, \text{sig}) = 1 \geq \delta,
\]

for some $\delta$ close to 1.

**Definition 7.** Let $S$ be a server oracle. We say that the delegated signature scheme $\text{dSS}=(\text{KG}, \text{Sign}^S, \text{Ver}^S)$ satisfies **existential $\epsilon$-unforgeability under chosen message attack** (briefly, $\epsilon$-cma-EU) if for any efficient oracle algorithm $A$, it holds that

\[
\text{Prob}\left[ \text{out} \leftarrow \text{SecExp}_{\text{dSS},A}(1^\sigma) : \text{out} = 1 \right] \leq \epsilon,
\]

for some $\epsilon$ close to 0, where experiment $\text{SecExp}$ is detailed below:

\[
\text{SecExp}_{\text{dSS},A}(1^\sigma)
\]

1. $(\text{pk}, \text{sk}) \leftarrow \text{KG}(1^\sigma)$
2. $(\text{m}', \text{sig}') \leftarrow A(\text{dSign}(\text{pk}, \text{sk}, \cdot), S(\cdot)(\text{pk}))$
3. Let $Q$ be the set of message queries made by $A$ to oracle $\text{dSign}(\text{pk}, \text{sk}, \cdot)$
4. if $\text{m} \in Q$ or $\text{Ver}^S(\text{pk}, \text{sk}, \text{m}', \text{sig}') = 0$ then return: 0
   else return: 1.

### 5.3 Delegated Signature Schemes: a general result

We show the relationship between non-delegated signature schemes, delegation protocols and delegated signature schemes in the following theorem.

**Theorem 2.** Let $(C, S)$ be a client-server protocol for the delegation of function $F$ and let $\text{SS}=(\text{KG}, \text{Sign}, \text{Ver})$ be a (non-delegated) signature scheme. The associated triple $\text{dSS}=(\text{KG}, \text{Sign}^S, \text{Ver}^S)$ is a delegated signature scheme.

The main takeaway from Theorem 2 is to provide a shortcut to provably turn a conventional signature scheme into a delegated signature scheme, as defined in Section 5.2: just design a suitable delegation protocol, as defined in Section 3. In particular, the delegated signature scheme comes with protection of the original signature scheme against more powerful attacks such as eavesdropping on the delegation protocol messages, and querying the server oracle.

Critical to establish the relationship in the theorem is the delegation protocol's simulation-based privacy property. Indeed the correctness property of the delegated signature scheme directly follows from the analogue property of the original signature scheme. On the other hand, the unforgeability of the delegated signature scheme follows by the unforgeability of the non-delegated signature scheme and the delegation protocol's simulation-based privacy property. Specifically, assume an adversary $A$ is able to violate the unforgeability of the delegated signature scheme. One can construct an adversary $A'$ that violates the unforgeability of the non-delegated signature scheme, as follows:

1. $A'$ runs algorithm $A$ and processes $A$'s queries as follows
2. When $A$ queries $dSign(pk, sk, \cdot)$ with message $m$, $A'$ does the following:

   - $A'$ queries $Sign(pk, sk, \cdot)$ with message $m$, thus obtaining signature $sig$
   - $A'$ runs simulator $Sim$ to obtain the transcripts $\{tr\}$ containing queries to $S$ and replies from $S$ performed during
   - the executions of algorithms $Sign^S$ and $Ver^S$
   - $A'$ simulates the oracle $dSign(pk, sk, \cdot)$’s answer as $(sig, \{tr\})$

3. When $A$ queries $S(\cdot)$ with message $qmes$, $A'$ does the following:

   - $A'$ runs $S$ on input query message $qmes$ thus obtaining answer $qans$
   - $A'$ simulates the oracle $S(\cdot)$’s answer as $qans$

We note that the simulation-based privacy of the delegation protocol for $F$ implies that the success of $A'$ in breaking $SS$ is the same as the success of $A$ in breaking $dSS$. The theorem follows.

5.4 Delegating ElGamal, Schnoor and Okamoto’s Schemes

In this section we show delegated signature protocols for 3 well-known signature schemes: those by El Gamal [1], Schnoor [2] and Okamoto [3]. The delegated signature schemes $dSS$ are obtained by combining the non-delegated signature schemes $SS$, reviewed in Appendix A, with the delegation protocol $(C, S)$ for a product of exponentiations in the associated group, described in Section 4 and then applying Theorem 2. In the design of each $dSS$ scheme, we also carefully split the signature and verification computations between offline and online phases of the two algorithms. In what follows, we describe the non-delegated signature schemes.

**Delegated ElGamal Signature Scheme.** Our delegated version of the signature scheme in [1] uses a client-server protocol $(C, S)$ for the delegation of function $F_{g,y}$ and a cryptographic hash function $H$, and goes as follows.

1. **Key generation:** Let $g$ be a generator of $\mathbb{Z}_p^*$ where $p$ is large prime $p$. Randomly choose $x \in \{1, \ldots, p - 2\}$ and set $y := g^x \mod p$. The public key is $(p, g, y)$ and the private key is $x$.
2. **Offline Signing:** on input public key $(p, g, y)$ and private key $x$, choose random $k \in \{1, \ldots, p - 1\}$ such that $\gcd(k, p - 1) = 1$ and set $r := (g^k \mod p) \mod q$. Output offline signature $(r)$.
3. **Online Signing:** on input public key $(p, g, y)$, private key $x$, offline signature $(r)$ and a message $m$, compute $s := k^{-1}(H(m) - xr) \mod p - 1$ and output signature $(r, s)$ if $0 < r < p$ and $0 < s < p - 1$ or ⊥ otherwise.
4. **Offline Verifying:** on input a public key $(p, g, y)$, run the offline phase of the delegation protocol $(C, S)$ resulting in offline output $pp$.
5. **Online Verifying:** on input a public key $(p, g, y)$, offline output $pp$, a message $m$, and a signature $(r, s)$ with $0 < r < p$ and $0 < s < p - 1$, compute $x_1 = H(m)/s \mod (p - 1)$ and $x_2 = -r/s \mod (p - 1)$, query $S$ with inputs $g_1 = g, g_2 = y, x_1$ and $x_2$, and use $S$’s reply to compute the product $\pi$. Finally, check that $\pi = r \mod p$. 

Note that in the scheme the verification algorithm checks whether
\[ g^{H(m)s^{-1}y^{-rs^{-1}}} = r \mod p, \]
which is equivalent to the check
\[ g^{H(m)} = y^r r^s \mod p \]
in the original ElGamal’s scheme. We also note that contrarily to the original scheme, in the above there is a negligible probability (when \( r = 0 \) or \( s = 0 \)) that Sign does not compute a valid signature.

**The delegated Schnorr Signature Scheme.** Our delegated version of the signature scheme in [2] uses a client-server protocol \((C, S)\) for the delegation of function \( F_{g,q,p} \) and a cryptographic hash function \( H \), and goes as follows.

1. **Key generation:** Let \( g \) be a generator of group \( G \) of prime order, \( q \). Randomly choose \( x \in \mathbb{Z}_q \) and set \( y := g^x \). The public key is \((G, q, g, y)\) and the private key is \( x \).
2. **Offline Signing:** on input public key \((G, q, g, y)\) and private key \( x \), choose random \( k \in \mathbb{Z}_q \) and set \( I := g^k \). Output offline signature \( I \).
3. **Online Signing:** on input public key \((G, q, g, y)\), private key \( x \), offline signature \( I \) and a message \( m \), compute \( r := H(I, m) \) and \( s := rx + k \mod q \). Output signature \((r, s)\).
4. **Offline Verifying:** on input a public key \((G, q, g, y)\), run the offline phase of the delegation protocol \((C, S)\) resulting in offline output \( pp \).
5. **Online Verifying:** on input public key \((G, q, g, y)\), a message \( m \), offline output \( pp \) and signature \((r, s)\), set \( x_1 = s \) and \( x_2 = -r \mod q \), query \( S \) with inputs \( g_1 = g, g_2 = y, x_1 \) and \( x_2 \), and use \( S \)'s reply to compute the product \( \pi \). Finally, check that \( H(\pi, m) = r \mod p \).

### 5.5 The Okamoto Signature Scheme

The delegated Okamoto signature scheme uses a client-server protocol \((C, S)\) for the delegation of function \( F_{g,q,p} \) and a cryptographic hash function \( H \), and goes as follows.

1. **Key generation:** Let \( p = kq + 1 \) where \( p \) and \( q \) are primes and \( k \) be an integer (e.g., \( q \geq 2^{140} \) and \( p \geq 2^{512} \)). Choose \( g_1 \) and \( g_2 \) of order \( q \) in the group \( \mathbb{Z}_p^* \), and an integer \( t = \mathcal{O}(\sqrt{|p|}) \) (e.g., \( t \geq 20 \)). Randomly choose \( s_1, s_2 \in \mathbb{Z}_q \) and set \( v := g_1^{-s_1} \cdot g_2^{-s_2} \mod p \). The public key is \((p, q, g_1, g_2, t, v)\) and the private key is \((s_1, s_2)\).
2. **Offline Signing:** on input public key \((p, q, g_1, g_2, t, v)\) and private key \((s_1, s_2)\) and a message \( m \), let \( H \) be the hash function, choose random \( r_1, r_2 \in \mathbb{Z}_q \), set \( x := g_1^{r_1} \cdot g_2^{r_2} \mod p \) and output offline signature \( x \).
3. **Online Signing:** on input public key \((p, q, g_1, g_2, t, v)\) and private key \((s_1, s_2)\), offline signature \( x \) and a message \( m \), compute \( e := H(x, m) \in \mathbb{Z}_q^* \), followed by \((y_1, y_2)\) such that \( y_1 = r_1 + es_1 \mod q \) and \( y_2 = r_2 + es_2 \mod q \). Output signature \((e, y_1, y_2)\).
4. **Offline Verifying:** on input a public key \((p, q, g_1, g_2, t, v)\), run the offline phase of the delegation protocol \((C, S)\) resulting in offline output \(pp\).

5. **Online Verifying:** on input a public key \((p, q, g_1, g_2, t, v)\), offline output \(pp\), a message \(m\), and signature \((e, y_1, y_2)\), set \(x_1 = y_1, x_2 = y_2, x_3 = e\), query \(S\) with inputs \(g_1, g_2\) and \(g_3 = v\), and use \(S\)'s reply to compute the product \(\pi\). Finally, check that \(H(\pi, m) = e \mod p\).

6. **Conclusions**

We considered the problem of outsourcing a product of group exponentiation to a single, possibly malicious, server. We solved this problem by showing a protocol that provably satisfies formal correctness, privacy, security and efficiency requirements, in a large class of cyclic groups; specifically, cyclic groups whose multiplication and inverse operations can be efficiently computed, and which admit an efficiently verifiable protocol to prove that an element is in the group. The considered class of cyclic groups includes groups often discussed in cryptography literature, such as prime-order subgroups in \(\mathbb{Z}_p\) and analogue elliptic curve groups.

As an application, we showed delegated versions of well-known signature schemes including products of exponentiations over cyclic groups. Our methods provide hope towards delegating more expensive cryptographic protocols.

**Acknowledgement.** Research by Delaram Kahrobaei and Vladimir Shpilrain was supported by the ONR (Office of Naval Research) grant N000141210758.

**References**


A Signature Schemes

A.1 ElGamal Signature Scheme

The ElGamal signature scheme is as follows.

1. **Key generation:** Let \( g \) be a generator of \( \mathbb{Z}_p^* \) where \( p \) is a large prime \( p \). Randomly choose \( x \in \{1, \ldots, p-2\} \) and set \( y := g^x \mod p \). The public key is \((p, g, y)\) and the private key is \( x \).

2. **Signing:** on input private key \( x \) and a message \( m \), let \( H \) be the hash function, choose random \( k \in \{1, \ldots, p-1\} \) such that \( \gcd(k, p-1) = 1 \) and set \( r := g^k \mod p \) then compute \( s := k^{-1}(H(m) - xr) \mod p - 1 \). (If \( s = 0 \) then start again.) Output signature \((r, s)\).

3. **Verifying:** on input a public key \((p, g, y)\), a message \( m \), and signature \((r, s)\) with \( 0 < r < p \) and \( 0 < s < p - 1 \). The verification checks whether
\[
g^{H(m)} \equiv y^r r^s \mod p.
\]

A.2 DSA Signature Scheme

The DSA signature scheme is as follows.

1. **Key generation:** Let \( p = kq + 1 \) where \( p \) and \( q \) are primes and \( k \) be an integer. Choose \( g \), a number whose multiplicative order modulo \( p \) is \( q \) (i.e. \( q \) is smallest positive integer such that \( g^q = 1 \mod p \)). Choose random \( x \in \mathbb{Z}_q^* \) and set \( y := g^x \mod p \). The public key is \((p, q, g, y)\) and the private key is \( x \).

2. **Signing:** on input private key \( x \) and a message \( m \), let \( H \) be the hash function, choose random \( k \in \mathbb{Z}_q^* \) and set \( r := (g^k \mod p) \mod q \) then compute \( s := k^{-1}(H(m) + xr) \mod q \). (If \( r = 0 \) or \( s = 0 \) then start again with fresh choice of \( k \).) Output signature \((r, s)\).

3. **Verifying:** on input a public key \((p, q, g, y)\), a message \( m \), and signature \((r, s)\) with \( r, s \neq 0 \mod q \). Calculates \( u_1 = H(m) \cdot s^{-1} \mod q \) and \( u_2 = r \cdot s^{-1} \mod q \). The verification checks whether
\[
r \equiv (g^{u_1} y^{u_2} \mod p) \mod q
\]
A.3 The Schnorr Signature Scheme

The Schnorr signature scheme is as follows.

1. **Key generation:** Let $g$ be a generator of group $G$ of prime order, $q$. Randomly choose $x \in \mathbb{Z}_q$ and set $y := g^x$. The public key is $(G, q, g, y)$ and the private key is $x$.

2. **Signing:** on input private key $x$ and a message $m$, let $H$ be the hash function, choose random $k \in \mathbb{Z}_q$ and set $I := g^k$ then compute $r := H(I, m)$, followed by $s := rx + k \mod q$. Output signature $(r, s)$.

3. **Verifying:** on input a public key $(G, q, g, y)$, a message $m$, and signature $(r, s)$, compute $I := g^s \cdot y^{-r} \mod p$. The verification checks whether

$$H(I, m) \equiv r$$

A.4 The Okamoto Signature Scheme

The Okamoto signature scheme [3] is as follows.

1. **Key generation:** Let $p = kq + 1$ where $p$ and $q$ are primes and $k$ be an integer (e.g., $q \geq 2^{140}$ and $p \geq 2^{512}$.) Choose $g_1$ and $g_2$ of order $q$ in the group $\mathbb{Z}_p^*$, and an integer $t = \mathcal{O}(|p|)$. (e.g., $t \geq 20$.) Randomly choose $s_1, s_2 \in \mathbb{Z}_q$ and set $v := g_1^{-s_1} \cdot g_2^{-s_2} \mod p$. The public key is $(p, q, g_1, g_2, t, v)$ and the private key is $(s_1, s_2)$.

2. **Signing:** on input private key $(s_1, s_2)$ and a message $m$, let $H$ be the hash function, choose random $r_1, r_2 \in \mathbb{Z}_q$ and set $x := g_1^{r_1} \cdot g_2^{r_2} \mod p$ then compute $e := H(x, m) \in \mathbb{Z}_{2^t}$, followed by $(y_1, y_2)$ such that $y_1 = r_1 + es_1 \mod q$ and $y_2 = r_2 + es_2 \mod q$. Output signature $(e, y_1, y_2)$.

3. **Verifying:** on input a public key $(p, q, g_1, g_2, t, v)$, a message $m$, and signature $(e, y_1, y_2)$, compute $x := g_1^{y_1} \cdot g_2^{y_2} \cdot v^e \mod p$. The verification checks whether

$$H(x, m) \equiv e$$