On Non-Parallelizable Deterministic Client Puzzle Scheme with Batch Verification Modes

Qiang Tang and Arjan Jeckmans

DIES, Faculty of EEMCS
University of Twente, the Netherlands
{q.tang, a.j.p.jeckmans}@utwente.nl

Abstract. A (computational) client puzzle scheme enables a client to prove to a server that a certain amount of computing resources (CPU cycles and/or Memory look-ups) has been dedicated to solve a puzzle. Researchers have identified a number of potential applications, such as constructing timed cryptography, fighting junk emails, and protecting critical infrastructure from DoS attacks. In this paper, we first revisit this concept and formally define two properties, namely *deterministic computation* and *parallel computation resistance*. Our analysis show that both properties are crucial for the effectiveness of client puzzle schemes in most application scenarios. We prove that the RSW client puzzle scheme, which is based on the repeated squaring technique, achieves both properties. Secondly, we introduce two batch verification modes for the RSW client puzzle scheme in order to improve the verification efficiency of the server, and investigate three methods for handling errors in batch verifications. Lastly, we show that client puzzle schemes can be integrated with reputation systems to further improve the effectiveness in practice.

Keywords: Client puzzle, parallelization, batch verification, DoS attack

1 Introduction

Merkle [26] was the first to introduce the notion of *puzzle* which led to the invention of public key cryptography. In this context, the puzzle is required to be unsolvable by any polynomial-time entity. Dwork and Naor [12] proposed the concept of *pricing function* to combat junk emails. Rivest, Shamir, and Wagner [30] proposed the concept of *timed-lock puzzle*, which serves as a tool to realize the concept of *timed-release crypto*. Jakobsson and Juels [18] defined the notion of *proof of work*, and Juels and Brainard [20] coincided the concept *client puzzle* and suggested to use it to prevent Denial of Service (DoS) attacks. Regardless of the different notations, *pricing function*, *timed-lock puzzle*, *proof of work*, and *client puzzle* share the same characteristic: they can be regarded as another type

of *puzzle* (different that of Merkle [26]) which is moderately hard in the sense that a polynomial-time entity can successfully find a solution by spending a certain amount of resources in a reasonable period of time. Without loss of generality, we use the notation of client puzzle to refer to the second type of puzzle (*pricing function* [12], *timed-lock puzzle* [30], *proof of work* [18], and *client puzzle* [20]). Furthermore, we refer to the entity, which generates the puzzle, as the server and the entity, which solves the puzzle, as the client.

Roughly speaking, there are two types of client puzzle schemes. One type is CPU-bound, where the computation is measured by the amount of CPU cycles needed to solve the puzzle. Some examples are those in [3,8,12,18,20,30,34,38], which form the majority of the existing client puzzle schemes. Abadi *et al.* [1] first noticed the fact that CPU power varies a lot for different computers (such as PC, PDA, and Workstation), and introduced memory-bound puzzle schemes, where the computation is measured by the amount of memory look-ups needed to solve the puzzle. The schemes in [10,11] fall into this category.

1.1 Applications of Client Puzzles

In practice, a puzzle has (at least) two effects, namely *timing effect* and *computation effect*, which has motivated its potential applications.

- *Timing effect*. In order to solve the puzzle, the client needs to spend a certain amount of time in the computation. This effect is very intuitive since any computation takes some time.
- *Computation effect*. In order to solve the puzzle, the client needs to dedicate a certain amount of resources, such as CPU cycles and memory look-ups.

It is worth stressing that there are some subtle differences between these two effects although they are tightly related to each other. For example, based on whether or not a client puzzle achieves the *parallel computation resistance* property (defined in Section 2.2), the *timing effect* could vary a lot though the *computation effect* is the same. This is illustrated by the toy client puzzle scheme in Section 2.2.

Rivest, Shamir, and Wagner [30] made use of the *timing effect* to realize timed-encryption. Boneh and Noar [5] and Syverson [33] made use of the *timing effect* to realize timed bit commitment. Garay and Jakobsson [16] made use of the *timing effect* to realize timed release of digital signatures. Based on these two effects, Franklin and Malkhi [14] used client puzzle schemes to achieve trustworthy website usage metering. In their scheme, if a client visits the website, it will generate a solution to a puzzle, and the metering result is the total number of

solved puzzles. Similarly, Cai *et al.* [7] proposed using client puzzle schemes as uncheatable benchmarks. Goldschlag and Stubblebine [17] made use of the *timing effect* to generate delays in lottery applications.

In a denial-of-service (DoS) attack, an attacker attempts to prevent legitimate users from accessing information or services by sending a large number of fake requests; furthermore, in its distributed form (referred to as a DDoS attack [32]), an attacker may use the controlled zombie computers to simultaneously launch the attack as shown in Fig. 1.

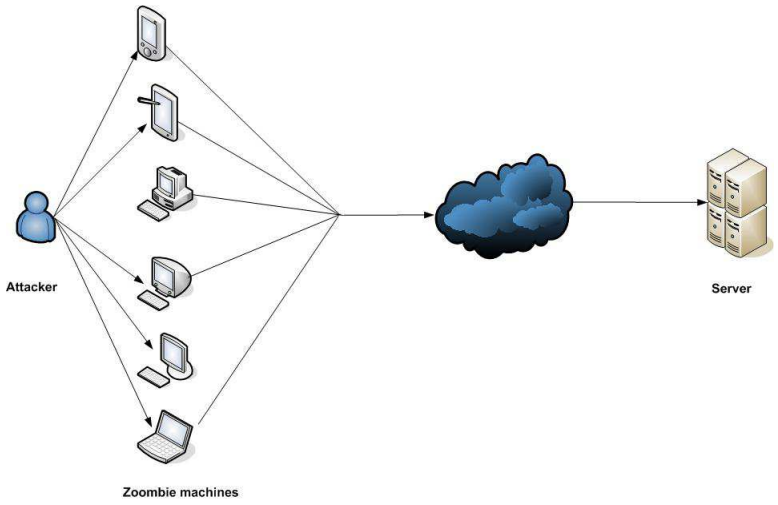

Fig. 1. DDoS Attack

The victim of a DoS attack could be high-profile web servers such as bank servers, DNS servers, cloud computing service providers, search engines (for example, Google), etc. In more details, there are two categories of DoS attacks.

- One is the exhaustion of *specific* types of very limited computer resources, such as TCP connections. For example, the SYN flood attack falls into this category [20].
- The other is the exhaustion of bandwidth or *general* CPU cycles or memory usages, for this purpose, the adversary just congests the communication links or sends nonsense messages to the victim. For example, the jamming attack in wireless sensor networks falls into this category [29].

For the first category, with a client puzzle scheme implemented, the server can mitigate an attack by asking every client to solve a puzzle before allocating any

resource. The rationale is that, the number of "valid" requests from a malicious client will drop to some extent because the client has only limited resources to find puzzle solutions. Note that a client puzzle scheme will incur extra computation and computation overhead. For the second category, with a client puzzle scheme implemented, if malicious clients send non-sense data as their puzzle solutions, the attack will become even worse because the server has to spend resources in verifying the fake puzzle solutions.

Juels and Brainard [20] first proposed using client puzzle schemes to mitigate DoS attacks such as the TCP SYN flooding attack. Aura, Nikander, and Leiwo [2] proposed using client puzzle schemes to prevent DoS attacks in authentication protocols. Dean and Stubblefield [9] suggested to use client puzzle schemes to protect TLS against DoS attacks. Wang and Reiter [35] introduced the concept of puzzle auction to solve the CPU power disparity problem when using (CPU-bounded) client puzzles. Lee and Fung [22] proposed using client puzzle schemes in the design of public-key based authentication and key establishment protocols. Martinovic *et al.* [25] introduced the concept of wireless client puzzle schemes for protecting access points against fake requests in wireless networks. Lei, Pierre, and Quintero [23] discussed how to combat DoS attacks using quasi partial collision based client puzzle schemes in UMTS networks. Dwork and Naor [12], and Dwork, Goldberg and Naor [11] suggested to use client puzzle schemes to combat junk emails. Fraser *et al.* [15] suggested to use client puzzle schemes to mitigate DoS attacks in the Tor networks. Borisov [6] proposed using client puzzle schemes for defending Sybil attacks in peer-to-peer networks. Ning, Liu, and Du [28] considered using client puzzle schemes to mitigate DoS attacks against broadcast authentication in wireless sensor networks. Feng *et al.* [13] suggested to implement client puzzle schemes in network protocols to prevent DoS attacks. Wang and Reiter proposed the concepts of *congestion puzzles* to defeat DoS attacks against IP networks [36,37]

In the literature, there has been some debate on whether client puzzle schemes are really helpful to defect DoS attacks. Based on the collected results from ISPs in UK, Laurie and Clayton [21] claimed that client puzzle schemes are hardly effective in combating junk emails in practice. While, Liu and Camp [24] argued that client puzzle schemes could be helpful if such schemes are used in combination with reputation systems.

1.2 Contribution and organization

The contribution of this paper lies in three aspects.

– Firstly, we present a new definition for client puzzle schemes. In contrast to that in [8], which is specifically for those schemes for defending DoS

attacks, our definition generally covers all client puzzle schemes. We describe the following properties: *puzzle hardness granularity*, *deterministic or probabilistic computation*, *parallel computation resistance*, *stateful or stateless*, and *communication complexity*. We show that *deterministic computation* and *parallel computation resistance* are the crucial properties accounting for the effectiveness of client puzzle schemes in most application scenarios. We propose the first formal definitions for these two properties.

– Secondly, we prove that the RSW client puzzle scheme in [30], which is based on the repeated squaring technique, achieves the *deterministic computation* and *parallel computation resistance* properties. To improve the efficiency of verifying multiple puzzle solutions (by the server), we propose two batch verification modes for the RSW scheme and analyze their security and efficiency implications. Moreover, we introduce three methods for handling errors in batch verifications. Our results have substantially improved that of Jeckmans [19], which has made similar attempts in this direction.

– Lastly, we show that client puzzle schemes can be integrated with reputation systems to further improve their effectiveness in defeating DoS attacks. Though similar concepts have been mentioned in [13,24], our proposal is more general and concrete. In addition, we provide more detailed analysis about the integration.

The rest of the paper is organized as follows. In Section 2 we present a new definition to client puzzle schemes, describe some relevant properties, and formally define *deterministic computation* and *parallel computation resistance*. In Section 3, we revisit the RSW client puzzle scheme and analyze its security. In Section 4, we introduce two batch verification modes for the RSW scheme and analyze their performances. In Section 5, we introduce three methods to handle errors in batch verifications. In Section 6 we provide some further remarks on the effectiveness of using client puzzle schemes to prevent DoS attacks, and show how to integrate them with reputation systems. In Section 7, we conclude the paper.

2 Properties and Formal Definitions

Similar to the definition in [19], a client puzzle scheme consists of four (probabilistic) polynomial-time algorithms (Setup, PuzzleGen, PuzzleSol, PuzzleVer).

– Setup(ℓ): Run by the server, this algorithm takes a security parameter ℓ as input, and outputs the public system parameter *params* and a private key *mk*. The public system parameter *params* is implicitly part of the input to

other algorithms. For reasons of simplicity, this system parameter has been omitted in the descriptions.

- PuzzleGen(mk, d, req): Run by the server, this algorithm takes the private key mk, a hardness parameter d, and some request information req as input, and outputs a puzzle puz and some relevant information $info$. The hardness parameter d is an integer which indicates the total amount of computation required. The server sends puz to the client, and keeps $info$ for verifying the solution.
- PuzzleSol(puz): Run by a client, this algorithm takes a puzzle puz as input and outputs a puzzle solution sol.
- PuzzleVer($mk, info, sol$): Run by the server, this algorithm takes the private key mk, the puzzle information $info$, and the solution sol as input, and outputs 1 if sol is correct or 0 otherwise.

Compared with the definition in [8] which aims at client puzzle schemes tailored for defeating DoS attacks, our definition aims at client puzzle schemes in general. Specifically, our definition does not contain a similar function to PuzzleAut [8], which is run by the server to check that all puzzles are indeed generated by itself.

2.1 Properties Outline

In most applications, the usefulness of client puzzle schemes is due to the computation disparity between the client and the server: it should take very little resource for the server to generate a puzzle and verify a solution, while it should take a certain amount of resource for a client to find a solution. Take the application of defeating DoS attacks as an example, suppose a client puzzle scheme requires a similar amount of resources for a client and the server to solve a puzzle and verify a solution respectively, then the server will not even be able to generate the necessary puzzles and verify the received solutions given that there will be a large volume of requests coming from clients to the server.

Besides the computation disparity requirement, the following properties are also of interest in practice.

- *Puzzle hardness granularity*: Recall that, for the CPU-bounded client puzzle schemes, the required resource in solving a puzzle is usually measured by the total number of some basic operations, such as hashing [20] and repeated squaring [30]. While, for the memory-bounded client puzzle schemes, the resource required is usually measured by the total number of memory look-ups [1]. Preferably, a client puzzle scheme should be able to generate puzzles

of fine-grained hardnesses. This will enable the server to set the puzzle hardness flexibly according to the threat level against the underlying application. Take the hash-based client puzzle scheme as an example [1]: a puzzle is in the form of (h, x_2), where $h = \mathsf{H}(x_1 || x_2)$, H is a collision-resistant hash function, and x_1 has bit-length k; a solution is a k bit x_1' such that $h = \mathsf{H}(x_1' || x_2)$. Clearly, the puzzle hardness can only be set exponentially with respect to k in this case. However, in the RSW client puzzle scheme [30], the puzzle hardness can be set linearly with respect to repeated squaring.

- *Deterministic computation*: For some client puzzle schemes, such as the RSW scheme in [30], the required resources from the client to solve a puzzle is deterministic. While, for other schemes, the required resource from the client is probabilistic. Take the above hash-based client puzzle scheme as an example, in the best case the client can find the solution by performing one hashing, while in the worst case the client needs to perform 2^k hashing operations. The *deterministic computation* property is desirable because the server can exactly determine the number of operations that the client must perform to solve a puzzle. Client puzzle schemes, with a probabilistic nature, may cause usability issues, such as the required resource is unpredictable to solve a puzzle, which may become the deterrent of their adoption.

- *Parallel computation resistance*: In practice, a client may have access to a number of computers as depicted in Fig. 2, so that the computing power of this client will be the sum of those of the control computers. Such a client may be able to solve a puzzle much faster than others, by dividing the workload and letting the computers work in parallel. Take the above hash-based client puzzle scheme as an example, suppose that the client has N computers, then on average each computer needs to perform $\frac{2^{k-1}}{N}$ hashing until one of them finds a solution.
  Client puzzle schemes, which allow parallel computation, will result in the huge disparity between a client, which has access to many computers, and another client, which has access to only one computer. Clearly, the *parallel computation resistance* property is crucial for achieving the *timing effect* required by the client puzzle schemes in [5,14,16,30,33]. For other applications such as defending DoS attacks, where an adversary may control a large number of Zombie computers, the *parallel computation resistance* property is also very important. Without this property, the disparity in computing power will make it impossible for the server to appropriately set puzzle hardness.

- *Stateful or stateless*: A client puzzle scheme is said to be stateful if the server needs to store the parameter $info$ associated with every puzzle. In practice, it is straightforward to turn a stateful scheme into a stateless one by sending both puz and $info$ to the client, which should send sol and $info$ back to the server for verification.

[1] This is a simplied version of the scheme in [20].

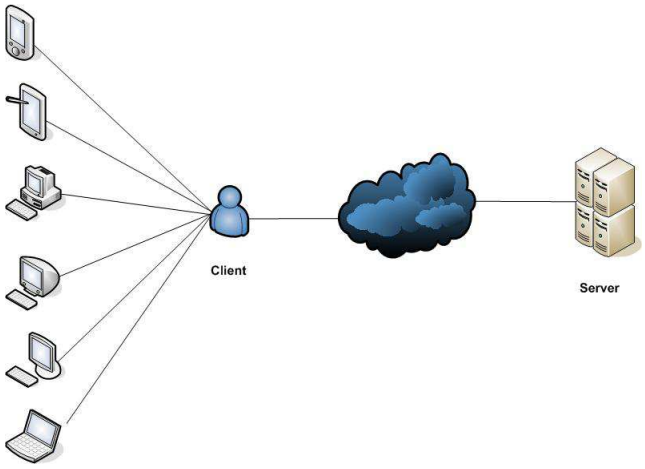

Fig. 2. Parallel Computation

– *Communication complexity*: The communication complexity is mainly concerned with the bandwidth needed to transfer puzzles and solutions between a client and the server. Another concern is the number of rounds needed. For example, in the model of Chen *et al.* [8], a client puzzle scheme has three rounds. From this perspective, it will be more efficient if a client puzzle scheme requires less rounds.

Remark 1. With respect to client puzzle schemes, especially the stateless ones, the forgeability of puzzles may be a serious security concern. If a malicious client can forge puzzles then it can pre-compute solutions, which will make client puzzle schemes ineffective in applications like defeating DoS attacks. Nonetheless, the unforgeability of puzzles can be achieved by some well-known techniques.

1. One technique is to use a secret key (for example, *mk*) to generate the puzzle. Alternatively, the server can attach a message authentication code or a digital signature to each puzzle. After receiving a puzzle solution, the server can check whether the related puzzle is generated by itself or not. In fact, the first technique has been used by many existing client puzzle schemes, although the formal definition of unforgeability has only been given very recently by Chen *et al.* in [8].
   Even if a client puzzle scheme achieves unforgeability, there may be replay attacks, in which a malicious client sends the same puzzle and solution multiple times to the server. To prevent replay attacks (in other words, to guarantee the freshness of received puzzles), synchronization techniques such as timestamps should be used in the generation of puzzles. Alternatively, the technique below can be adopted.

2. The other technique is for the server to keep a list of puzzles it has generated. Later on, the server can discard the puzzle solutions, for which the related puzzles are not on the list. With this solution, there are some management issues which need to be taken into account in practical deployment. For example, for the sake of storage efficiency, the server should decide how many puzzles it will keep on the list. To prevent replay attacks, the server also needs to remove a puzzle from the list once a correct solution has been received.

2.2 Formal Definitions

To facilitate the formal definitions, suppose that the client puzzle is denoted as Π and the total number of the Func operation is used as a metric for puzzle hardness. Formally, we say that the puzzle hardness is evaluated by the total number of queries to the Func oracle.

The properties of *puzzle hardness granularity*, *stateful or stateless*, and *communication complexity* don't need formal investigation, namely, they can be heuristically analyzed. Therefore, we only need to formally evaluate the properties *deterministic computation* and *parallel computation resistance*, respectively. In both cases, the adversary is a malicious client, and the attacks can be simulated through the following three-phrase game between the adversary and a challenger.

1. Setup phase: the challenger runs the Setup algorithm to generate the master key mk and the public system parameter $params$.
2. Challenge phase: The adversary issues queries to the PuzzleGen oracle with its chosen d and req, and obtains the reply $puz = \mathsf{PuzzleGen}(mk, d, req)$ from the challenger. At some point, the adversary sends d^* and req^* to the challenger, which sends $puz^* = \mathsf{PuzzleGen}(mk, d^*, req^*)$ back as a challenge.
3. Response phase: The adversary can issue queries to the PuzzleGen oracle as in the challenge phase. At some point, the adversary terminates by outputting a correct solution sol^* to the puzzle puz^*.

In practice, the only information to a client is the puzzles generated by the server and the public system parameter. It is straightforward to check that the adversary has all the same privileges as a malicious client in practice.

Definition of *deterministic computation*. Informally, the property *deterministic computation* requires that, after receiving a puzzle from the server, a client needs

to perform d times of Func operations (or, formally to issue d queries to the Func oracle), given that the puzzle hardness is d. Basically, this has two implications.

- One is that puzzles, generated by the server, are independent in the sense that solving one puzzle does not help solve another puzzle. In other words, any puzzle generated by the server will require the client to perform d times of Func operations.
- The other is that puzzles, generated by the server, are unpredictable in the sense that any puzzle generated by the server looks fresh so that the client is unable to pre-compute the solution.

An attack occurs if, after receiving a puzzle of hardness d, the adversary finds a solution using less than d queries to the Func oracle. Formally, this property is defined as follows.

Definition 1. *Suppose the adversary has issued $d^\dagger$ queries to the* Func *oracle in the Response phase. A client puzzle scheme Π achieves the* deterministic computation *property if the probability $d^\dagger < d^*$ is negligible.*

Definition of *parallel computation resistance*. If a client has access to more than one computers, then it is able to perform Func operations simultaneously. Let's take the following toy client puzzle scheme as an example.

- Setup(ℓ): This algorithm outputs a hash function H as the public system parameter *params*. The master secret key *mk* is set to be an empty value.
- PuzzleGen(mk, d, req): This algorithm outputs d random numbers r_i ($1 \leq i \leq d$) as the puzzle. The parameter $info$ is set to be r_i ($1 \leq i \leq d$).
- PuzzleSol(puz): This algorithm outputs $sol = \{\mathsf{H}(r_i) \ \ (1 \leq i \leq d)\}$ as the solution.
- PuzzleVer($mk, info, sol$): This algorithm computes d hash values and compare them with sol.

Clearly, suppose a client has access to d computers, the client will solve a puzzle of hardness d by asking every computer to compute only one hashing.

Informally, the property *parallel computation resistance* requires that, after receiving a puzzle of hardness d from the server, a client needs to *sequentially* perform d times of Func operations to find a solution. It means that the client is unable to speed up the process by letting more than one computers work

in parallel. This also implies that the best strategy for the client is to use its fastest computer to solve the puzzle. We first introduce the notion of *sequential computing time* as follows.

Definition 2. *Suppose that a client has access to a number of computers, which can simultaneously answer oracle queries from the client during a computation task. Within the occurrence of a computation task, consider all oracle query sequences of the following form*

$$Seq_x = \{q_1, q_2, \cdots, q_x\},$$

where, for all $1 \leq i \leq x-1$*, the query* q_{i+1} *is issued to a computer after the query* q_i *has been answered (by any computer). The* sequential computing time *of the computation task is the maximum value of the sequence indexes, namely* $\max_x$.

Formally, the property *parallel computation resistance* is defined as follows.

Definition 3. *Suppose the* sequential computing time *of the computation task of finding a correct solution* sol^* *in the Response phase is* $d^\dagger$*. A client puzzle scheme* Π *achieves the* parallel computation resistance *property if the probability* $d^\dagger < d^*$ *is negligible.*

Consider a computation task with the occurred oracle queries shown in Fig. 3, where the rectangles (either in red or blue) represent an oracle query. According to Definition 2, it is straightforward to count that the *sequential computing time* of this computation task is 11.

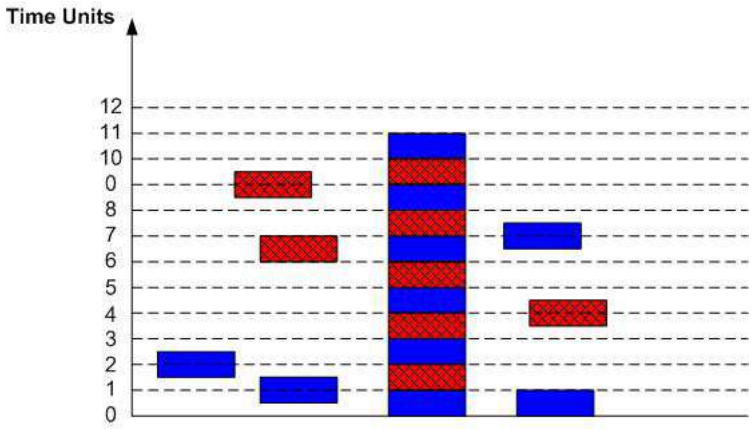

Fig. 3. The concept of *sequential computing time*

Remark 2. Clearly, according to our definition, the property *parallel computation resistance* implies the property *deterministic computation*. However, the reverse is untrue, as illustrated by the above toy client puzzle scheme.

3 Properties of the RSW Client Puzzle Scheme

The scheme, described below, is a slightly modified version of the RSW client puzzle scheme proposed by Rivest, Shamir, and Wagner [30]. For simplicity, we still refer to the described scheme as the RSW scheme.

- $\mathsf{Setup}(\ell)$: This algorithm selects two random large primes p, q and a hash function $\mathsf{H} : \{0,1\}^* \to \mathbb{Z}_{pq}^*$. The public parameter is pq, and the master key is $mk = (p, q)$.
- $\mathsf{PuzzleGen}(mk, d, req)$: This algorithm chooses $r \in_R \mathbb{Z}_{pq}^*$ and computes $g = \mathsf{H}(r||req)$, and outputs the puzzle $puz = (g, d)$. The related puzzle information is $info = (g, r, d, req)$.
- $\mathsf{PuzzleSol}(puz)$: This algorithm outputs $sol = g^{2^d} \mod pq$.
- $\mathsf{PuzzleVer}(mk, info, sol)$: this algorithm returns 1 if $sol \equiv g^{2^d \mod \phi(pq)} \pmod{pq}$, and returns 0 otherwise.

In the above scheme, g is computed as $g = \mathsf{H}(r||req)$, while g is randomly chosen generator in [30]. By doing so, if needed, the generator can be bound to situational information (such as the identity information of the client) contained in req.

With respect to computing the verification complexity of the server, we omit that of computing $2^d \mod \phi(pq)$ for two reasons. One is that it could be pre-computed and stored by the server. The other is that, in many cases, multiple puzzles might share the same hardness so that the computation only needs to be done once. As a consequence, it is straightforward to calculate that the average verification complexity for the server is approximately $\frac{3L}{2} - 2$ multiplications in $\mathbb{Z}_{pq}^*$, where L is the bit-length of $\phi(pq)$.

3.1 Preliminary

As a preliminary, we introduce a generic group model and an extended discrete logarithm assumption with respect to $\mathbb{Z}_{pq}^*$ where p, q are two prime numbers.

We first define a generic algorithm according to the multiplication operation in $\mathbb{Z}^*_{pq}$. Note that there are different ways of giving the definition (e.g. [27,31]), and we follow that of Shoup [31].

Let $g \in_R \mathbb{Z}^*_{pq}$ and σ be an encoding function of $\mathbb{G} = \{g^i \mid i \in \mathbb{N}\}$ on $\{0,1\}^{|pq|}$, where $\mathbb{N}$ is the set of integers and $|pq|$ means the bit-length of pq. Suppose O is a multiplication oracle, which, for any $r \in \mathbb{G}$, computes $\sigma(r)$ as follows: if r has been calculated before, then the same value of $\sigma(r)$ is returned; otherwise, it sets $\sigma(r)$ to be a random value from $\{0,1\}^{|pq|} \setminus \mathcal{S}$, where $\mathcal{S}$ is a set initialized to be $\{\sigma(g)\}$.

A generic algorithm $\mathcal{A}$ is a probabilistic algorithm, which takes $\mathcal{S}$ and pq as input, and behaves as follows. At any time, $\mathcal{A}$ can send a query with the input (x, y, b), where $x, y \in \mathcal{S}$ and $b \in \{1, -1\}$, to O, and will receive $\sigma(x \cdot y^b)$ as the reply. After every query, the result is added to the set $\mathcal{S}$.

Definition 4. *Let p, q be two prime numbers. The extended discrete logarithm assumption holds for $\mathbb{Z}^*_{pq}$ if the following event only occurs with a negligible probability with respect to a security parameter ℓ: Given $(pq, g, g_i\ (1 \le i \le L))$, where $g, g_i (1 \le i \le L) \in_R \mathbb{Z}^*_{pq}$ and L is any polynomial in the security parameter, a polynomial-time adversary finds $x \neq 0, x_i\ (1 \le i \le L) \in \mathbb{N}$ such that*

$$g^x \equiv \prod_{x_i \in \mathbb{N}}^{1 \le i \le L} g_i^{x_i} \pmod{pq}.$$

3.2 Proof of *parallel computation resistance*

Lemma 1. *If the adversary is modeled as a generic algorithm as defined above, then the RSW client puzzle scheme achieves* parallel computation resistance *based on the extended discrete logarithm assumption (given in Definition 4) in the random oracle model.*

Proof sketch. In order to prove the lemma, we need to show that the adversary's advantage in the attack game (described at the beginning of Section 2.2) is negligible according to Definition 3. Since the adversary is modeled as a generic algorithm, there are two types of oracle queries it may issue. One is the PuzzleGen query which can be issued to the challenger, and the other is multiplication oracle query which can be issued to the oracle O.

We first consider a simple situation where the adversary does not issue any PuzzleGen query in the game. In this situation, the best strategy for the adversary to issue oracle queries to O in the Response phase is the following.

1. At the beginning, the adversary issues two queries with the inputs $(\sigma(g^*), \sigma(g^*), 1)$ and $(\sigma(g^*), \sigma(g^*), -1)$. Clearly, until it receives the replies, namely $\sigma((g^*)^2)$ and $\sigma(1)$, from the oracle $\mathcal{O}$, it does not make sense for the adversary to send any other query.
2. After obtaining the replies, the set $\mathcal{S}$ becomes $\{\sigma(1), \sigma(g^*), \sigma((g^*)^2)\}$. Then the adversary issues queries with the following inputs

$$(\sigma(g^*), \sigma((g^*)^2), 1),\ (\sigma((g^*)^2), \sigma((g^*)^2), 1),$$

$$(\sigma(1), \sigma(g^*), -1),\ (\sigma(1), \sigma((g^*)^2), -1).$$

   Clearly, until it receives the replies, namely $\sigma((g^*)^3)$, $\sigma((g^*)^4)$, $\sigma((g^*)^{-1})$, and $\sigma((g^*)^{-2})$ from the oracle $\mathcal{O}$, it does not make sense for the adversary to send any other query.
3. After obtaining the replies, the set $\mathcal{S}$ becomes

$$\{\sigma(1), \sigma(g^*), \sigma((g^*)^2), \sigma((g^*)^3), \sigma((g^*)^4), \sigma((g^*)^{-1}), \sigma((g^*)^{-2})\}.$$

   Then the adversary issues queries with (x, y, b), where $x, y \in \mathcal{S}$, $b \in \{1, -1\}$, and $\sigma(x \cdot y^b) \notin \mathcal{S}$. Clearly, until it receives the replies, it does not make sense for the adversary to send any other query.
4. The adversary continues the above process until it sends the response to the challenger.

Note that, in our security analysis, we consider polynomial-time adversary, which can only issue a polynomial number of queries in the game. This means that, in each step, the adversary can only issue a polynomial number of oracle queries.

Suppose, at the end of the game, the adversary performs $d^* - 1$ steps as above (which also means the *sequential computing time* is $d^* - 1$), then the set $\mathcal{S}$ is a subset of

$$\{\sigma((g^*)^{-2^{d^*-1}}), \sigma((g^*)^{-2^{d^*-1}+1}), \cdots, \sigma(1), \cdots, \sigma((g^*)^{2^{d^*-1}-1}), \sigma((g^*)^{2^{d^*-1}})\}$$

In the game, the adversary could choose to submit a value $\sigma((g^*)^z)$ from $\mathcal{S}$ or a value r from $\{0,1\}^{|pq|} \setminus \mathcal{S}$ as the response to the challenger.

- In the first case, if $\sigma((g^*)^z) = \sigma((g^*)^{2^{d^*}})$, then the adversary has actually found $2^{d^*} - z$ such that $(g^*)^{2^{d^*}-z} = 1 \mod n$. Based on the extended discrete logarithm assumption, the probability is negligible.
- In the second case, if $(g^*)^{2^{d^*}}$ has not been queried to the oracle, the probability $r = \sigma((g^*)^{2^{d^*}})$ is $\frac{Q}{pq}$ where Q is the total number of queries issued to the oracle $\mathcal{O}$. Clearly, this probability is negligible. Otherwise, if $(g^*)^{2^{d^*}}$ has been queried to the oracle $\mathcal{O}$, the probability $r = \sigma((g^*)^{2^{d^*}})$ is 0.

To summarize, in this simple situation, the adversary's advantage is negligible.

Next, we evaluate a more complex situation where the adversary is free to issue PuzzleGen oracle queries. Let $\sigma(g_1), \sigma(g_2), \cdots, \sigma(g_L)$ be the elements resulted from the replies of PuzzleGen oracle queries, where $g_1, g_2, \cdots, g_L$ are random elements from $\mathbb{Z}^*_{pq}$. Certainly, L is a polynomial in the security parameter. Suppose, at the end of the game, the adversary performs $d^* - 1$ steps in the Response phase, then the set $\mathcal{S}$ is a subset of $\mathcal{S}_1 \bigcup \mathcal{S}_2 \bigcup \mathcal{S}_3$. The set $\mathcal{S}_1$ is the following.

$$\mathcal{S}_1 = \{\sigma((g^*)^{-2^{d^*-1}}), \sigma((g^*)^{-2^{d^*-1}+1}), \cdots, \sigma(1), \cdots, \sigma((g^*)^{2^{d^*-1}-1}), \sigma((g^*)^{2^{d^*-1}})\}$$

The set $\mathcal{S}_2$ contains a polynomial number of encodings of the form $\sigma(\prod_{x_i \in \mathbb{N}}^{1 \le i \le L} g_i^{x_i})$, and the set $\mathcal{S}_3$ contains a polynomial number of encodings of the form $\sigma(A \cdot B)$, where $\sigma(A) \in \mathcal{S}_1$ and $\sigma(B) \in \mathcal{S}_2$.

In the game, the adversary could choose to submit a value $\sigma((g^*)^z)$ from $\mathcal{S}_1$, $\mathcal{S}_2$, $\mathcal{S}_3$, or a value r from $\{0,1\}^{|pq|} \setminus \mathcal{S}$ as the response to the challenger.

- In the first case, from the analysis in the simple situation, the adversary's advantage is negligible.
- In the second and the third cases, if $\sigma((g^*)^z \cdot \prod_{x_i \in \mathbb{N}}^{1 \le i \le L} g_i^{x_i}) = \sigma((g^*)^{2^{d^*}})$, then the adversary has actually found $2^{d^*} - z$ such that $(g^*)^{2^{d^*}-z} \equiv \prod_{x_i \in \mathbb{N}}^{1 \le i \le L} g_i^{x_i} \pmod{pq}$. Based on the extended discrete logarithm assumption, the probability is negligible.
- In the forth case, from the analysis in the simple situation, the adversary's advantage is negligible.

As a result, in this situation, the adversary's advantage is negligible. The lemma follows. □

4 Proposals of Batch Verification Modes

Note the fact that a client puzzle scheme will trigger extra computation overhead to the server, which may be required to generate a large number of puzzles and verify the corresponding solutions. The situation becomes more serious when the computations need to be done in real-time. Similar to the case in signature schemes [4], given a client puzzle scheme, if it supports batch verification, then the verification efficiency will be dramatically improved. In the rest of this section, we propose two batch verification modes for the RSW client puzzle scheme.

4.1 A Batch Verification Mode - Attempt

As to the multiplication operation in $\mathbb{Z}_{pq}^*$, given that, for $1 \leq i \leq n$, $a_i \in \mathbb{Z}_{pq}^*$ and $b_i = a_i^r \mod pq$ for $r \in \mathbb{N}$, the following equality holds.

$$(\prod_{i=1}^{n} a_i)^r \equiv \prod_{i=1}^{n} b_i \pmod{pq}$$

Based on this observation, suppose that there are n puzzles $puz_i = (g_i, d)$ and solutions h_i from the RSW client puzzle scheme, we can verify the solutions using a batch verification mode, by checking the following equality.

$$(\prod_{i=1}^{n} g_i)^{2^d \mod \phi(pq)} \equiv \prod_{i=1}^{n} h_i \pmod{pq} \tag{1}$$

Note that we assume the puzzles share the same hardness granularity d.

Let L be the bit-length of $\phi(pq)$. The average batch verification complexity is $C_n = \frac{3L}{2} + 2n - 4$ multiplications in $\mathbb{Z}_{pq}^*$. If the server sequentially verifies the individual puzzle solutions, the complexity would be $(\frac{3L}{2} - 2) \cdot n$. With reasonable parameters (say, $L = 1024$ and $n = 100$), the batch verification is much more efficient, namely

$$C_n << (\frac{3L}{2} - 2) \cdot n.$$

Remark 3. Note that, without considering the constant $\frac{3L}{2} - 4$, the complexity is linear in the batch size n. It is also worth stressing that $\prod_{i=1}^{n} g_i$ can be pre-computed, which has the complexity of $n - 1$.

With respect to this batch verification mode, we have the following observations.

1. If the equality (1) does not hold, then at least one solution is incorrect, i.e. $h_j \neq g_j^r \pmod{pq}$ for some $1 \leq j \leq n$.
2. If all solutions are correct, i.e. $h_i \equiv g_i^r \pmod{pq}$ for all $1 \leq i \leq n$, then the equality (1) holds.
3. If the equality (1) holds, it does not imply that all solutions are correct. Clearly, if h_i $(1 \leq i \leq n)$ are replaced with any h_i' $(1 \leq i \leq n)$, where

   $$\prod_{i=1}^{n} h_i \equiv \prod_{i=1}^{n} h_i' \pmod{pq},$$

   the equality (1) still holds.

The third observation implies that there could be *false accept* if the server verifies the solutions simply by checking the equality (1). In fact, the client(s) only needs to perform d repeated squarings to compute H, where

$$H = (\prod_{i=1}^{n} g_i)^{2^d} \pmod{pq},$$

then it can randomly split H into h'_i $(1 \leq i \leq n)$ as the solutions.

4.2 A Batch Verification Mode - Improvement

Suppose that there are n puzzles $puz_i = (g_i, d)$ and solutions h_i from the RSW client puzzle scheme, the improved batch verification mode is as follows. Select $x_i \in \mathbb{Z}_N^*$, where N is an integer and smaller than pq, and check the following equality.

$$(\prod_{i=1}^{n} (g_i)^{x_i})^{2^d \mod \phi(pq)} \stackrel{?}{\equiv} \prod_{i=1}^{n} (h_i)^{x_i} \pmod{pq} \tag{2}$$

Let L be the bit-length of $\phi(pq)$. The average batch verification complexity is $\frac{3L}{2} + 2n - 4 + 2n \cdot (\frac{3L'}{2} - 2)$ multiplications in $\mathbb{Z}_{pq}^*$, where L' is the bit-length of N.

Remark 4. It is worth stressing that $\prod_{i=1}^{n} (g_i)^{x_i}$ can be pre-computed, which has the average complexity of $n \cdot (\frac{3L'}{2} - 2) + n - 1$.

With respect to this batch verification mode, the first and second observations in the previous subsection are still true. The third observation is also partially true, but the *false accept* probability can be reduced as low as possible by the following lemma.

Lemma 2. *If the equality (2) holds, the probability that there exist incorrect solutions (i.e.* $h_j \neq g_j^r \pmod{pq}$ *holds for some* $1 \leq j \leq n$*) is upper-bounded by* $\frac{1}{N}$.

Proof sketch. Note that, for any $(1 \leq j \leq n)$, the equality (2) can be rephrased as follows:

$$(\frac{(g_j)^{2^d \mod \phi(pq)}}{h_j})^{x_j} \equiv ((\prod_{i=1, i\neq j}^{n} (g_i)^{x_i})^{2^d \mod \phi(pq)})^{-1} \prod_{i=1, i\neq j}^{n} (h_i)^{x_i} \pmod{pq}. \tag{3}$$

Suppose that $h_j \neq (g_j)^{2^d \mod \phi(pq)}$ holds, then it is clear that the equality (3) holds at most with the probability $\frac{1}{N}$. The lemma follows. □

4.3 Further Improvement

Orthogonal to the improvement in Section 4.2, the *false accept* shortcoming may be mitigated by the following *divide-and-verify* strategy. Suppose that a server employs the RSW client puzzle scheme, and a dishonest client tries to use the following trick to cheat the server.

> **Attack assumption.** *As noted in Section 4.1, the client generates H first,*
>
> $$H = (\prod_{i=1}^{n} g_i)^{2^d} \pmod{pq},$$
>
> *and then randomly splits it into n individual solutions h'_i $(1 \le i \le n)$).*

With the *divide-and-verify* strategy, after receiving a certain number of puzzle solutions, the server first divides the received puzzle solutions into several subgroups, and then performs batch verification in each subgroup. With this strategy, the probability of *false accept* is determined by the following lemma.

Lemma 3. *Suppose that the server divides the received solutions into Y subgroups. The probability that a* false accept *occurs is* $(\frac{1}{Y})^{n-1}$.

Proof sketch. The verifications on h'_i $(1 \le i \le n)$ will pass only if they fall into the same subgroup. As a result, the probability is $(\frac{1}{Y})^{n-1}$. □

Clearly, when Y becomes larger (or, the size of subgroup become smaller), the *false accept* rate will drop much faster. On the other hand, when Y becomes larger, the server's computation complexity will increase. The extreme case in this direction is for the server to verify every puzzle solution independently. Clearly, there is a tradeoff between the *false accept* rate and the computation complexity of the server.

In practice, the *divide-and-verify* strategy and the improved batch verification mode (described in Section 4.2) can be integrated, namely the server first divides the received puzzle solutions into several subgroups, and then performs batch verification for each subgroup. The *false accept* rate is described by the following lemma.

Lemma 4. *Suppose that the server divides the received solutions into Y subgroups. With the above batch verification mode, the probability that a* false accept *occur is* $(\frac{1}{Y})^{n-1} \cdot \frac{1}{N}$.

Remark 5. It is worth stressing that a malicious client may not behave in the same way as stated in **Attack assumption**. Nonetheless, the *divide-and-verify* strategy will always help the server to detect fake puzzle solutions, though the successful detection probability will differ from the above.

5 Handling Errors in Batch Verification

With the batch verification modes described in Section 4, errors in the batch (referred to as $\mathcal{B} = (h_1, h_2, \cdots, h_n)$, will be detected when the following inequalities hold, respectively.

$$(\prod_{i=1}^{n} g_i)^{2^d} \mod \phi(pq) \neq \prod_{i=1}^{n} h_i \pmod{pq}$$

$$(\prod_{i=1}^{n} (g_i)^{x_i})^{2^d} \mod \phi(pq) \neq \prod_{i=1}^{n} (h_i)^{x_i} \pmod{pq}$$

Roughly, the server can deal with an erroneous batch in two ways. One solution is to treat all puzzle solutions to be false and reject them. As shown in Section 6.2, this could be a reasonable solution when combined with reputation systems in some application scenarios. However, generally, it is not a good choice because an adversary can pollute (multiple) puzzle batches by sending false solutions to the server and make the server reject the puzzle solutions from legitimate clients. An alternative solution is for the server to sort out the false solutions and reject them. Furthermore, the server may also enforce other punishment on the client(s) which have sent the false solutions.

Remark 6. If the improved batch verification mode has been used, the values of $(g_i)^{x_i}$ and $(h_i)^{x_i}$ for all $1 \leq i \leq n$ will have been computed in the batch verification process. As a result, the error-searching procedures will be exactly the same, regardless which batch verification has been used. Without loss of generality, the following discussion assumes the basic verification mode.

Next, we consider three different methods to figure out the incorrect solutions, namely *sequential searching*, *sequential searching with batch verification*, and *dividing-and-conquering*.

5.1 The Case of *sequential searching*

The strategy of *sequential searching* is straightforward: if errors are detected, the server verifies each puzzle solution in the batch and finds out all the incorrect ones. Clearly, the error-searching complexity is $n \cdot (\frac{3L}{2} - 2)$ multiplications in $\mathbb{Z}_{pq}^*$.

5.2 The Case of *sequential searching with batch verification*

Choose i as an index and initialize it to be 1, then the algorithm of *sequential searching with batch verification* works as follows.

1. Verify the solution h_i.
   (a) If the verification passes, set $i = i + 1$, re-execute this step if $i \leq n$ and stop otherwise.
   (b) Otherwise, h_i is incorrect, set $i = i + 1$. If $i > n$, stop; otherwise, go to step 2.
2. Verify the puzzle solutions h_j $(i \leq j \leq n)$ using the batch verification mode. If the verification passes, stop; otherwise, go to step 1.

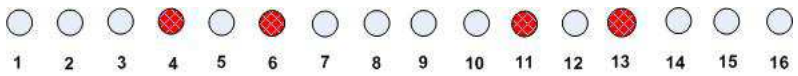

Fig. 4. Example

Suppose that there are $1 \leq t \leq n$ errors which are uniformly distributed in the batch. With respect to the computations in the above two steps, we have the following observations.

– In step 1, the server needs to perform puzzle verification on individual puzzle solutions. Let the average complexity be $\bar{U}$, which is determined by the average of the distribution of the highest index z of the incorrect solutions in the batch. Note that we suppose there are t errors in the batch, the average complexity is as follows.

$$\bar{U} = (\frac{3L}{2} - 2) \cdot \sum_{z=t}^{n} (z \cdot P_z), \quad P_z = \frac{t}{z} \cdot \prod_{i=0}^{n-z-1} \frac{n-t-i}{n-i}$$

– In step 2, the server needs to perform batch verification if h_j is incorrect, and the complexity is $\frac{3L}{2} - 4 + 2(n - j)$. Note that $n - j$ the distance from h_j to h_n. For $1 \leq k \leq \frac{t}{2}$, the following two averages are the same: the average of the distance l from k-th incorrect solution to h_1, and the average of the distance l' from $(t - k + 1)$-th incorrect solution to h_n. Based on the remark in Section 4.1, the average complexity of batch verifications following these two incorrect solutions is

$$2(\frac{3L}{2} - 4) + 2(n - l + l' - 1) = 3L + 2n - 10.$$

As a result, the average complexity of batch verifications is $\bar{V}$, where

$$\bar{V} = \frac{t}{2} \cdot (3L + 2n - 10).$$

In summary, the complexity of the whole error-searching process is $\bar{U} + \bar{V}$.

5.3 The Case of *dividing-and-conquering*

Generate a puzzle set list $\mathcal{L}$ and initialize it to be the batch $\mathcal{B}$, namely $\mathcal{L} = \{\mathcal{B}\}$. The algorithm of *dividing-and-conquering* works as follows.

1. If the list $\mathcal{L}$ is empty, stop. Otherwise, pick up the first puzzle set in the list, and go to Step 2.
2. Equally split the chosen puzzle set into two subsets, and verify one of them (randomly chosen) first using the basic batch verification mode. Note that, if the number of solutions in the set is odd, then it can allow one subset has one more member than the other subset. Based on the verification result, do the following.
   – If the verification passes, do the following. If the size of the other subset is larger than 1, then adds it to the list $\mathcal{L}$ and go to Step 1. Otherwise, output the other subset as an incorrect puzzle solution and go to Step 1.
   – If the verification fails, do the following. If the size of this subset is larger than 1, then adds it to the list $\mathcal{L}$, otherwise output this subset as an incorrect puzzle solution. Verifies the other subset and do the following.
     • If the verification passes, go to Step 1.
     • If the verification fails, do the following: If the size of the other subset is larger than 1, then adds it to the list $\mathcal{L}$ and go to Step 1. Otherwise, output the other subset as an incorrect puzzle solution and go to Step 1.

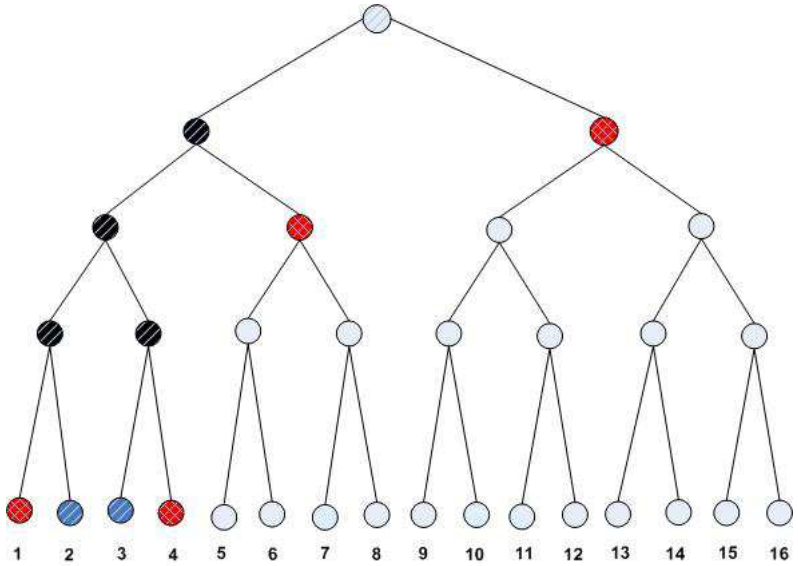

Fig. 5. Example of *dividing-and-conquering*

To illustrate the algorithm, suppose the batch size is 16 and the second and the third solutions are incorrect, as shown in Fig. 5.3. At the first level of the tree (when the batch is firstly split), the average number of multiplications is $\frac{3}{2} \cdot (\frac{3L}{2} - 4 + 16)$. At the second level of the tree, the average number of multiplications is $\frac{3}{2} \cdot (\frac{3L}{2} - 4 + 8)$. At the third level of the tree, the average number of multiplications is $2 \cdot (\frac{3L}{2} - 4 + 4)$. At the forth level (at the level of leaf nodes), the average number of multiplications is $3 \cdot (\frac{3L}{2} - 2)$. The total number of multiplication is $12L + 18$.

5.4 A Comparison of Different Methods

To compare the performances of different methods, we choose two cases with the batch sizes of 128 and 1024. In each case, we consider the subcases where there are 2, 10, and 50 incorrect solutions respectively. The results are summarized in the following table.

Complexity Comparison		
(n, t)	Searching Method	Number of Multiplications
(128,2)	*sequential searching*	-256+192L
	sequential searching with batch verification	74+132L
	dividing-and-conquering	431+26L
(128,10)	*sequential searching*	-256+192L
	sequential searching with batch verification	995+191L
	dividing-and-conquering	652+72L
(128,50)	*sequential searching*	-256+192L
	sequential searching with batch verification	5897 +257L
	dividing-and-conquering	795+200L
(1024,2)	*sequential searching*	-2048+1536L
	sequential searching with batch verification	671+1028L
	dividing-and-conquering	3879 + 39L
(1024,10)	*sequential searching*	-2048+1536L
	sequential searching with batch verification	8326+1413L
	dividing-and-conquering	6593+128L
(1024,50)	*sequential searching*	-2048+1536L
	sequential searching with batch verification	48940+1582L
	dividing-and-conquering	9208+374L

As to the methods *sequential searching* and *sequential searching with batch verification*, we have figured out the formulas for the verification complexities. In order to evaluate the complexity of the method *dividing-and-conquering*, we run a Mathematica program[2] 100 times to compute the average with respect to randomly chosen distributions of the t incorrect puzzle solutions.

From the figures, we can roughly draw the following conclusions. When the rate of incorrect solutions (namely $\frac{t}{n}$) is small, the method *dividing-and-conquering* is more efficient than the other two, and the method *sequential searching with batch verification* is also more efficient than the method *sequential searching*. When the rate increases, the advantage of the method *dividing-and-conquering* becomes less obvious, while *sequential searching with batch verification* may become less efficient than the method *sequential searching*. If the bit-length of ϕpq is 1024 (i.e. $L = 1024$); intuitively, the comparison is shown in Fig. 6.

[2] Available at `http://www.vf.utwente.nl/~tangq/new.nb`

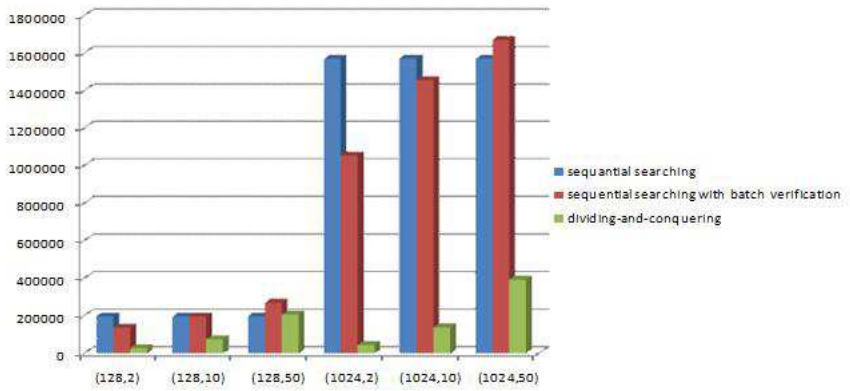

Fig. 6. Comparison Results

6 Further Remarks on Using Client Puzzle Schemes

6.1 On the Effectiveness in Mitigating DoS Attacks

As we have surveyed in Section 1.1, client puzzle schemes have been considered by many researchers as a solution to prevent DoS attacks in many scenarios. The main rationale behind these proposals is that the adversary needs to perform some computation before being allocated any resource from the server. In fact, client puzzle schemes have another useful effect, namely it may act as an indicator to identify Zombie machines. For example, if a computer performs intensively during some time periods, then the reason might be that, this computer has been captured as a Zombie and required to solve client puzzles when the adversary tries to mount a DoS attack. By examining the computation logs of the computer, the owner can potentially determine whether his machine has become a Zombie or not.

DoS attacks pose serious threat to today's ICT systems, and researchers have devised many different countermeasures, which are complementary to client puzzle schemes. Some examples are the following.

- Load balancing: The server's workload is distributed (logically and geographically) across two or more computers and network links. By doing so, the chance that all the computers and network links are flooded will be greatly reduced.
- Black-listing malicious clients: The server can block those clients which are suspicious for the attacks. Such measure can be regarded as a simple reputation system, and we give more details in the next subsection.

– Restricting resource usage of individual client: The server can have a strategy such as a client could only request a limited amount of resources in a certain period of time.
– Restricting the number of request of individual client: The server can have a strategy such as a client could only issue a limited number of request in a certain period of time.

Take black-listing malicious users as an example, it can prevent known Zombie machines from sending out service requests to the server. As a result, if a client puzzle scheme with *parallel computation resistance* is deployed, the adversary can not use these machines to solve any client puzzle in parallel either.

Another concern with using client puzzle schemes is the disparity of computing abilities of clients, which causes the concern of *fairness* with respect to the puzzle hardness.

– From the perspective of a resource-limited client, *fairness* may mean it should be served in the same way as other powerful clients. If they are required to solve puzzles of the same hardness, then the resource-limited client will be served with a much lower priority. What makes things worse is that, a malicious client may control a number of Zombie machines. As depicted in Fig. 2, the malicious client may be much more powerful than any legitimate clients.
– From the perspective of a powerful client, *fairness* may mean it should be served in a way according to its investment in its computing. For example, a client may buy a faster or buy more than one machine to increase its computing power, in order to obtain better services. To achieve this sort of fairness, Wang and Reiter introduced the concept of puzzle auction in [35].

In general, it is not possible to judge which definition of *fairness* is generally better than another one, because they may be reasonable in different application scenarios. It is worth stressing that, non-parallelable client puzzle schemes are an effective deterrent against the adversary which controls an army of Zombie machines.

There is another concern, namely an adversary may launch DoS attacks against a legitimate client in some situations. We refer this type of attack to be *reverse DoS attacks*. One example situation is when the adversary can spoof the server's address, then the adversary can send the hardest puzzles to the client to exhaust its resources. In order to mitigate this negative effect, we need to assume there is an integrity-protected link between any legitimate client and the server.

6.2 Integration with a Reputation System

In this subsection we first review some attempts in combining a client puzzle scheme with a reputation system to defeat DoS attacks, and then propose a more comprehensive solution for such combination. It is worth noting that this strategy may not be useful in other applications, such as those based on the *timing effect* [5,16,30].

Current Solutions The concept of utilizing a client puzzle scheme with a reputation system was mentioned as a future work by Feng *et al.* [13]. Their rough idea is that the server should base the puzzle hardness on the reputation of a client, but they did not give any further details. In an attempt to refute the claim of Laurie and Clayton [21] that client puzzle schemes are hardly effective in combating junk emails, Liu and Camp [24] suggested that a client puzzle scheme could be effective when being used in combination with a reputation system. Their idea is the following: the email server keeps a reputation value for each client; to send an email, a client needs to solve a puzzle, the hardness of which is a decreasing function with respect to the client's reputation value. Some more details about the solution in [24] are highlighted as follows.

- Clients are identified by their IP addresses.
- Initially, the reputation value of a new client is set to be very low so that the client needs to compute a hard puzzle to sends its first email. The reason is that, it would be difficult to judge that a new client is malicious or not, so that the puzzles should be hard enough to make the client not be able to send out a lot of junk emails.
- Over the time, the reputation of a legitimate client will increase while the puzzle hardness to send an email will decrease accordingly.
- If the total number of the emails reach one limit in one day, then the reputation of a client will begin to decrease.

To summarize, both proposals [13,24] attempt to implement the same strategy, which is to leverage the puzzle hardness based on the past behaviors of a client. This strategy is rational, and we will also use it in our proposal.

Our Proposal The logical structure of the proposed architecture is depicted in Fig. 7. In our proposal, the puzzle hardness according to a client's request is determined by two factors, namely the threat level and the reputation of the client.

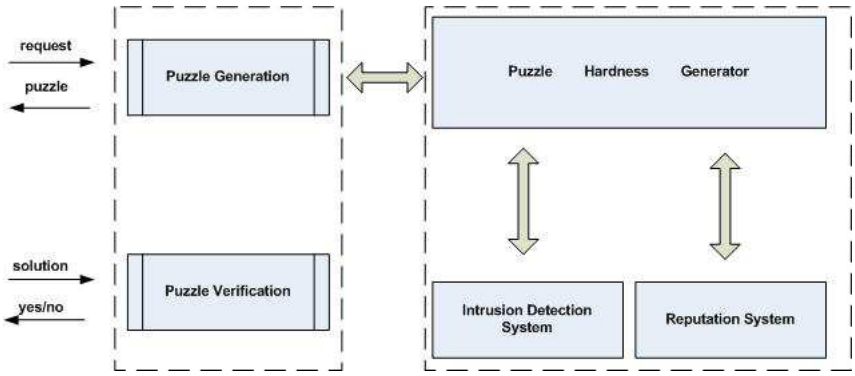

Fig. 7. Client Puzzle Application Architecture

- The threat level is an indicator about how severe the server is suffering from DoS attacks, which comes from the feedback from an intrusion detection system. Intuitively, if the threat level is high, then the server should increase the hardness of puzzles sent to the clients.
- In practice, there are three types of clients which may send requests to the server. The first type is legitimate clients who always honestly compute puzzle solutions. The second type is malicious clients which throw random bits to the server in order to deplete the resources of the server. The third type is curious clients which may submit either correct puzzle solutions or random bits. For example, this type of clients could be legitimate clients which, from time to time, are controlled by an adversary for a certain period. Just the same as in the case of [13,24], the integration of the reputation system is to distinguish these types of clients and be fair to the legitimate clients.

Formally, the puzzle hardness d is expressed in the following way

$$d = F_1(L) + F_2(L, R).$$

In this expression, F_1 is a function which takes a threat level L as input and returns the basic number of operations required. The function F_2 is a function which takes the threat level L and a reputation value R as input and returns the additional number of operations required. The detailed definitions of F_1 and F_2 rely on the applications of the specific application scenarios and the relevant security requirements, hence, we skip them in this paper. Nonetheless, these functions should satisfy the following properties.

- The function F_1 is increasing with respect to L, while the function F_2 is a non-decreasing function with respect to L. Moreover, the function F_2 is decreasing with respect to R.

– If there is no threat (or, equivalently $L = 0$), the equations $F_1(0) = 0$ should hold for any R. Regardless of the fact that, with a client puzzle scheme, the server needs to spend a certain amount of resources to generate and verify puzzles and solutions; even if there is no threat, the server may still set $F_2(0, R) \neq 0$. By doing so, the server can enable the client with higher reputation to perform less computation in order to receive the requested service, which is rational.

With respect to the implementation of a reputation system, instead of using the reputation value for each client, it may be more efficient for the server to use the aggregate reputation values for user groups. For example, aggregation can be done on the basis of organizations or IP rages instead of individual IP address. With this change, batch verification can be implemented for the puzzle solutions from the same group of clients. Moreover, the server may just reject all requests if there is any error in the batch, which is a severe punishment.

7 Conclusion

In this paper, we have revisited the concept of client puzzle schemes and presented formal definitions for the two important properties, namely *deterministic computation* and *parallel computation resistance*. We have proven that the RSW client puzzle scheme achieves both properties. To our knowledge, this is the first scheme which has been proven possessing the property *parallel computation resistance*. More interestingly, we have shown that the RSW scheme supports batch verification modes, which greatly improve the efficiency for the server. Client puzzle scheme is regarded as a useful tool in defeating DoS attacks, however, it is not a perfect solution by itself. Consequently, we proposed an integration between a client puzzle scheme and a reputation system. While our proposal is theoretical and abstract at the moment, an interesting future work is to instantiate the proposal in a real-world application, such as defeating junk emails, and to further investigate the effectiveness.

References

1. M. Abadi, M. Burrows, M. Manasse, and T. Wobber. Moderately hard, memory-bound functions. *ACM Transactions on Internet Technology*, 5(2):299–327, 2005.
2. T. Aura, P. Nikander, and J. Leiwo. DoS-resistant authentication with client puzzles. In *Security Protocols, 8th International Workshop*, pages 170–177, 2000.
3. A. Back. Hashcash — amortizable publicly auditable cost functions, 2002.
4. M. Bellare, J. Garay, and T. Rabin. Fast batch verification for modular exponentiation and digital signatures. In *Eurocrypt '98*, pages 236–250, 1998.

5. D. Boneh and M. Naor. Timed commitments. In *CRYPTO '00: Proceedings of the 20th Annual International Cryptology Conference on Advances in Cryptology*, pages 236–254. Springer, 2000.
6. N. Borisov. Computational puzzles as sybil defenses. In *P2P '06: Proceedings of the Sixth IEEE International Conference on Peer-to-Peer Computing*, pages 171–176. IEEE Computer Society, 2006.
7. J. Cai, R. J. Lipton, R. Sedgewick, and A. C. Yao. Towards uncheatable benchmarks. In *Structure in Complexity Theory Conference*, pages 2–11, 1993.
8. L. Chen, P. Morrissey, N. Smart, and B. Warinschi. Security notions and generic constructions for client puzzles. In *Advances in Cryptology — Asiacrypt 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 505–523. Springer, 2009.
9. D. Dean and A. Stubblefield. Using client puzzles to protect TLS. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, pages 1–1. USENIX Association, 2001.
10. S. Doshi, F. Monrose, and A. D. Rubin. Efficient memory bound puzzles using pattern databases. In *Applied Cryptography and Network Security, 4th International Conference, ACNS 2006*, pages 98–113, 2006.
11. C. Dwork, A. Goldberg, and M. Naor. On memory-bound functions for fighting spam. In Dan Boneh, editor, *Advances in Cryptology — CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 426–444. Springer, 2003.
12. C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In E. F. Brickell, editor, *Advances in Cryptology — CRYPTO '92*, volume 740 of *Lecture Notes in Computer Science*, pages 139–147. Springer, 1992.
13. W. Feng, E. Kaiser, W. Feng, and A. Luu. The design and implementation of network puzzles. In *Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2005)*, pages 2372–2382, 2005.
14. M. K. Franklin and D. Malkhi. Auditable metering with lightweight security. In R. Hirschfeld, editor, *Financial Cryptography, First International Conference, FC '97, Proceedings*, volume 1318 of *Lecture Notes in Computer Science*, pages 151–160. Springer, 1997.
15. N.A. Fraser, D.J. Kelly, R.A. Raines, R.O. Baldwin, and B.E. Mullins. Using client puzzles to mitigate distributed denial of service attacks in the tor anonymous routing environment. *Communications, 2007. ICC '07. IEEE International Conference on*, pages 1197–1202, 2007.
16. J. A. Garay and M. Jakobsson. Timed release of standard digital signatures. In M. Blaze, editor, *Financial Cryptography, 6th International Conference, FC 2002*, volume 2357 of *Lecture Notes in Computer Science*, pages 168–182. Springer, 2002.
17. D. Goldschlag and S. Stubblebine. Publicly verifiable lotteries: Applications of delaying functions. In *FC '98: Proceedings of the Second International Conference on Financial Cryptography*, pages 214–226. Springer-Verlag, 1998.
18. M. Jakobsson and A. Juels. Proofs of work and bread pudding protocols. In *CMS '99: Proceedings of the IFIP TC6/TC11 Joint Working Conference on Secure Information Networks*, pages 258–272. Kluwer, B.V., 1999.
19. A. Jeckmans. Practical client puzzle from repeated squaring. Technical report, Centre for Telematics and Information Technology, University of Twente, 2009. http://eprints.eemcs.utwente.nl/15951/.
20. A. Juels and J. G. Brainard. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 1999*, pages 151–165, 1999.
21. B. Laurie and R. Clayton. Proof-of-Work Proves not to Work. In *Third Annual Workshop on Economics of Information Security (WEIS04)*, 2004.

22. M. Lee and C. Fung. A public-key based authentication and key establishment protocol coupled with a client puzzle. *J. Am. Soc. Inf. Sci. Technol.*, 54(9):810–823, 2003.
23. Y. Lei, S. Pierre, and A. Quintero. Client puzzles based on quasi partial collisions against DoS attacks in UMTS. *Proceedings of the 64th IEEE Vehicular Technology Conference*, pages 1–5, 2006.
24. D. liu and L. Jean Camp. Proof of Work can Work. In *Fifth Workshop on the Economics of Information Security (WEIS06)*, 2006.
25. I. Martinovic, F. A. Zdarsky, M. Wilhelm, C. Wegmann, and J. B. Schmitt. Wireless client puzzles in IEEE 802.11 networks: Security by wireless. In *WiSec '08: Proceedings of the first ACM conference on Wireless network security*, pages 36–45. ACM, 2008.
26. R. C. Merkle. Secure communications over insecure channels. *Communications of the ACM*, pages 294–299, 1978.
27. V. I. Nechaev. Complexity of a determinate algorithm for the discrete logarithm. *Math. Notes*, 55(2):91–101, 1994.
28. P. Ning, A. Liu, and W. Du. Mitigating dos attacks against broadcast authentication in wireless sensor networks. *ACM Transactions on Sensor Networks*, 4(1), 2008.
29. D. R. Raymond and S. F. Midkiff. Denial-of-service in wireless sensor networks: Attacks and defenses. *IEEE Pervasive Computing*, 7(1):74–81, 2008.
30. R. L. Rivest, A. Shamir, and D. A. Wagner. Time-lock puzzles and timed-release crypto. Technical Report MIT/LCS/TR-684, Massachusetts Institute of Technology, 1996.
31. V. Shoup. Lower bounds for discrete logarithms and related problems. In *Advances in Cryptology — EUROCRYPT '97*, volume 1233 of *Lecture Notes in Computer Science*, pages 256–266. Springer, 1997.
32. S. M. Specht and R. B. Lee. Distributed denial of service: Taxonomies of attacks, tools, and countermeasures. In D. A. Bader and A. A. Khokhar, editors, *Proceedings of the ISCA 17th International Conference on Parallel and Distributed Computing Systems, September 15-17, 2004, The Canterbury Hotel, San Francisco, California, USA*, pages 543–550. ISCA, 2004.
33. P. Syverson. Weakly secret bit commitment: Applications to lotteries and fair exchange. In *CSFW '98: Proceedings of the 11th IEEE workshop on Computer Security Foundations*, page 2. IEEE Computer Society, 1998.
34. S. Tritilanunt, C. Boyd, E. Foo, and J. M. González Nieto. Toward non-parallelizable client puzzles. In *Cryptology and Network Security, 6th International Conference, CANS 2007*, pages 247–264, 2007.
35. X. Wang and M. K. Reiter. Defending against denial-of-service attacks with puzzle auctions. In *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy*, page 78, Washington, DC, USA, 2003. IEEE Computer Society.
36. X. Wang and M. K. Reiter. Mitigating bandwidth-exhaustion attacks using congestion puzzles. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 257–267, 2004.
37. X. Wang and M. K. Reiter. A multi-layer framework for puzzle-based denial-of-service defense. *International Journal of Information Security*, pages 243–263, 2008.
38. B. Waters, A. Juels, J. A. Halderman, and E. W. Felten. New client puzzle outsourcing techniques for DoS resistance. In *Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004*, pages 246–256, 2004.