Efficient Trapdoor-Based Client Puzzle Against DoS Attacks

Yi Gao, Willy Susilo, Yi Mu, and Jennifer Seberry

Contents

1 Introduction

It is well known that authentication, integrity, and confidentiality are the most important principles of network security. However, recent reports about a number of prominent Internet service providers that broke down because of malicious attacks [2, 3, 31, 32] urge people to realize that all security principles must be based on service availability. "Availability" in this context refers to a service that can be accessed within a reasonable amount of waiting time after a legitimate client sends a request.

Y. Gao (✉)
School of Information Technology and Computer Science, University of Wollongong, Australia
e-mail: yg70@uow.edu.au

S.C.-H. Huang et al. (eds.), *Network Security*, DOI 10.1007/978-0-387-73821-5_10,

The service availability of a network server can be destructed in a variety of ways, such as internal bugs within a system, hardware limits, or malicious attacks from outside. Network-based DoS attacks, in particular, denote malicious actions, which aim at shutting down a target server and destructing its service availability via the Internet. These attacks usually attempt to block or degrade service in a designated period temporarily, rather than intrude on the server directly or damage the data permanently.

One of the most popular DoS attacks, TCP SYN flooding attacks, was reported in 1996 [2, 3, 10, 32], which succeeded in crippling Panix, a major New York Internet service provider, in early September 1996, and created similar problems for the website of the New York Times a few days later. As a rule, an SYN flooding attacker exploits spoofed IP addresses to mount a large number of initial and unresolved connection requests to a victim server, depleting its resources and rendering it incapable of responding to legitimate clients.

Distributed Denial of Service (DDoS) is a new form of DoS attack, first reported in early 2000 [8, 28, 31]. In contrast to traditional DoS attacks, DDoS attackers, in particular, are armed with self-propagation worms which can be installed on a discretionary number of vulnerable computers on the Internet. An attacker is able to harness these compromised machines in order to mount a coordinated DoS attack. These infected machines are typically divided into two groups: "Masters" and "Zombies", which play different roles in a DDoS attack. "Masters" are more like intermediaries, while "Zombies" serve as attack platforms. Communication between an attacker and the "Zombies" is not direct, but depends on the "Masters". One "Master" may control and deliver the attacker's command to a number of "Zombies". By mounting such a coordinated DoS attack, the effectiveness of a DDoS can be multiplied by 10, 100, or even 10,000 times [13].

A typical DDoS attack process can be described as follows. An attacker first scans a large range of networks to find vulnerable hosts that have weak defences against a malicious intrusion. The number of these hosts is determined by the strength of the attack that an attacker intends to launch. Second, the attacker installs "Master" or "Agent" programs on these vulnerable hosts. A machine with an "Agent" program is called a "Zombie", which carries out the actual attack. A machine installed with a "Master" program is able to communicate with a number of "Zombies" and serves as a control-handler of the attacker. An attacker can command several "Masters" directly, and "Zombies" are activated by these "Masters" at the designated time for an attack. Figure 1 shows this three-layer control. The reason for using such an architecture is to keep the attacker safe and difficult to trace. Now, all the preparation has been accomplished. The attacker only needs to cross his fingers and wait for an appropriate time to launch his DDoS attack. When a defending server suspects that it is under a DoS attack, it can only find numerous legitimate connection requests received from a large number of legitimate IP addresses, consuming all the resources of the server. However, the real owners of these "Zombies" are unwitting accomplices [16], and do not know what has actually happened on their machines.

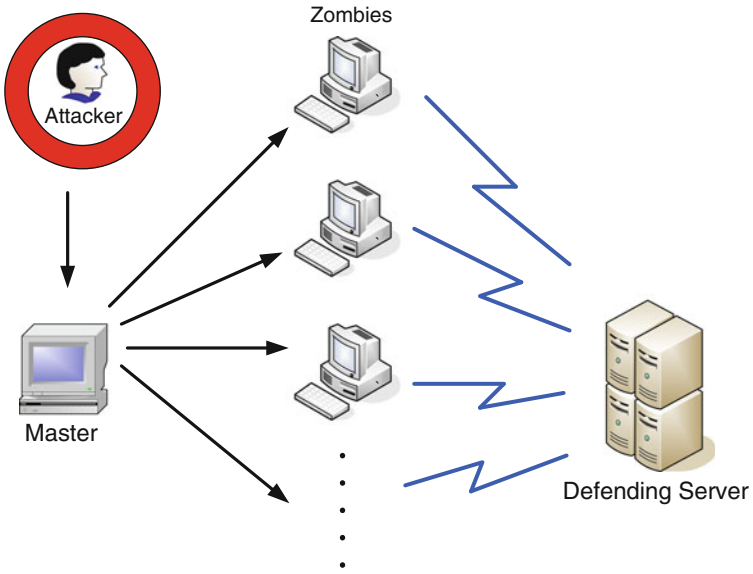

Fig. 1 Three-layer control for a DDoS attack

To prevent DoS attacks, many defence mechanisms have been proposed (e.g., [4,6,14,22,23,25]), among which client puzzle is one of the most notable and influential. Since DoS attacks mostly exploit defects in existing network protocols, the advantage of client puzzles over other proposed methods is that by improving protocols directly, it is feasible to confine DoS attacks in a harmless range for defending servers.

In a typical DoS attack, an adversary deploys unauthorized service requests to consume the limited resources of a target server. The aim of client puzzles is to impose a moderate authentication "cost"[1] on each client wishing to obtain service from a defending server. This can be achieved by asking clients to solve different cryptographic puzzles (here we call them *client puzzles*). The cost of computing client puzzles is negligible for legitimate clients, but unendurably expensive for DoS attackers who attempt to acquire considerable resources from the server. Because recent studies show that existing DDoS tools are designed carefully as to disturb "Zombie" computers and to avoid alerting their real owners. In other words, "Zombies" are unable to furtively compute puzzles for the adversary. In this way, the defending server can force the adversary to give up, because applying for more resources demands that the adversary must invest more resources himself/herself.

[1] In this context, "cost" means computational cost, such as CPU processing time or/and memory space.

On the other hand, client puzzles have one inadequacy. A client who requests for a service from a defending server has to install a small client-side program for the computation of puzzles. While most of the other proposed countermeasures against DoS attacks, such as Traceback IP [23], SYN cookies [6], and Ingress/Egress filtering [22] methods, merely demand a modification to a defending server or a fundamental network protocol, the potential disadvantages of these approaches are inevitable. For instance, SYN cookies are based on the assumption that an attacker launching a SYN flooding attack via spoofed IP cannot intercept all the SYN/ACK packets sent to the spoofed addresses, which is unfortunately not always true [15]. Similarly, the major idea of Traceback IP [21] is that a packet can be traced back to its source address by inserting traceback data into the packet when it passes through distinct routers to the destination, which obviously increases the traffic loads and raises much information redundancy. Compared with these, client puzzles are able to ensure service quality and protect servers against DoS attacks effectively, as long as clients install a puzzle-solving software. In today's network technology, such software can easily be implemented by a plug-in of a web browser or distributed by the servers. Hence, the requirements for this special software should not be a problem.

2 Related Work

Dwork and Naor introduced the original idea of cryptographic puzzles into junk mail defense [11], in which every successful delivery of a message requires the sender to solve a small cryptographic puzzle. By doing so, they successfully impose a large amount of computational costs on sending mass mails, while for legitimate clients, the costs to compute single puzzles are negligible.

In 1999, Juels and Brainard proposed a client puzzle protocol to defend against TCP SYN flooding attacks. The idea of the client puzzle protocol is very simple [15]. When there is no evidence of attack, a server accepts connection requests normally. If the defending server comes under a DoS attack, it distributes a unique client puzzle to each client who is applying for a connection. In order to obtain the server's resources for his/her connection, a client must compute the puzzle correctly and return the solution in time. This protocol is deployed in conjunction with the traditional time-out, which is used to control the time period for puzzle computation. Consequently, it is hard for an adversary to compute large numbers of puzzles in a short period, which can be used to differentiate legitimate requests from malicious half-open ones. In their paper, Juels and Brainard also presented a simple puzzle construction to implement their protocol although this seemed unsatisfactory and caused a lot of arguments in network forums [9, 24, 30].

In 2000, Aura et al. presented a more convincing scheme for client puzzles, which is based on Hash functions [4]. In their scheme, the server sends the parameters of a puzzle to the client, and the client performs a brute-force search and a number of computation to find the correct answer. The server distributes its resource to the client only after the solution is verified. However, the underlying problem of this

puzzle scheme is that a defending server must calculate a hash function, in order to verify each solution received from clients. As a result, a possible DoS attack may occur in which an adversary sends numerous bogus answers that the server has to process. In 2004, Waters et al. [5] suggested a new technique that permits the outsourcing of puzzles. Eventhough puzzles can be used by different servers, the solution of a puzzle still requires one modular exponentiation for every defending server.

2.1 Contribution

To overcome the underlying drawbacks of the existing puzzle systems, we propose a novel client puzzle scheme that is based on a trapdoor mechanism. Our scheme is expected to possess two prominent characteristics. One is that most of the computations for puzzle generation can be fulfilled in a preconstruction phase, independent of a puzzle construction. Pre-construction can be processed during idle time, and items calculated in this phase can be reused by combining them with time parameters. The other important feature is a quick verification. No computation occurs in the verification phase, which makes DoS attacks aiming at flooding bogus solutions to exhaust system resources impossible. We show that our puzzle is computationally efficient and applicable to the existing Internet protocols. A complete security proof is also included in this chapter.

2.2 Organization of the Chapter

The rest of the chapter is organized as follows. In Sect. 3, we provide the preliminary knowledge about trapdoor functions and the Discrete Logarithm Problem(DLP). In Sect. 4, we present a formal definition of trapdoor-based client puzzles. After that, we describe our trapdoor-based puzzle scheme in detail, along with the security proof and an analysis of system efficiency. We provide the system configuration in Sect. 6 and discuss several possible improvements in Sect. 7. Finally, we conclude this chapter in Sect. 8.

3 Preliminary

3.1 Trapdoor One-Way Function

Our proposed scheme utilizes a trapdoor one-way function. Informally, a one-way function is designed to provide an algorithm which is easy to compute, yet computationally infeasible to reverse [26]. For example, given x, it is easy to compute

the value of $f(x)$, but infeasible to obtain x if given $f(x)$ in polynomial time. A trapdoor one-way function is a special one-way function equipped with some secrets called "trapdoor". A trapdoor one-way function is uniformly easy to compute in one direction, but hard to reverse. Whereas in particular, it is also easy to compute x by reversing the function $f(x)$ as long as the trapdoor is known.

3.2 Security Assumption

It is well-known that the security of many famous public-key cryptosystems, such as RSA, Diffie–Hellman and DSS [17], is based on one assumption that there is a mathematically hard problem, which is difficult for cryptanalysts to solve in a polynomial time. The discrete logarithm is one of the prevalent hard problems [29].

The Discrete Logarithm Problem (DLP) is a problem that has been researched in [18, 20]: given a large prime p, a generator α of $\mathbb{Z}_p^*$, and a random element $\beta \in \mathbb{Z}_p^*$, it is hard to find the integer x $(0 \leq x \leq p-2)$ such that $\alpha^x \equiv \beta \pmod{p}$. Many popular public-key cryptosystems are based on this assumption, such as the ElGamal system and the DSS [1, 12].

We provide a formal presentation of the DLP, in which p is a large prime (e.g., a 1,024-bit number).

Assumption 1. *We say that the hardness of the DLP holds, if for all probabilistic polynomial time adversaries $\mathcal{A}$, the following probability*

$$Pr[\,\alpha \in \mathbb{Z}_p^*\,,\ \beta \in_R \mathbb{Z}_p^*\ :\ \mathcal{A}(p,\alpha,\beta) = x\ \ s.t.\ \ \alpha^x \equiv \beta \pmod{p}\,] < \varepsilon$$

4 Definition

The aim of our trapdoor-based client puzzle system is to protect the availability of the service transferred between a web server S and legitimate clients C against DoS attacks. The system exploits a specific trapdoor one-way function, where a puzzle creator (the server S) can efficiently compute the correct solution with the knowledge of the trapdoor, while other puzzle solvers (legitimate clients C and an adversary $\mathcal{A}$) must perform a brute-force search and a number of computation to obtain the answers. The scheme consists of three efficient algorithms for generating random puzzles, solving the puzzles, and verifying the solutions of these puzzles, respectively.

Definition 1. A trapdoor-based client puzzle scheme consists of a three-tuple of polynomial algorithms (*Gen*, *Solve*, *Verify*), where

- *Gen*: Puzzle generation algorithm used by a defending server S. It takes security parameter t and a difficulty level l as input, and outputs a random puzzle $\mathcal{P}$ along with a correct solution S_s to the puzzle $\mathcal{P}$. That is

$$(\mathcal{P}, S_s) \leftarrow \mathbf{Gen}(1^t, l).$$

  Indeed, a puzzle $\mathcal{P}$ comprises two elements denoted as $\mathcal{P} = (\sigma, [\alpha, \beta])$ where σ is a puzzle parameter, and $[\alpha, \beta]$ is a search range of length l. That is $l = |\beta - \alpha|$.
- *Solve*: Puzzle-solving algorithm. This is an efficient algorithm for a client C to solve the puzzles. On input $\mathcal{P} = (\sigma, [\alpha, \beta])$, it computes an answer S_c.

$$S_c \leftarrow \mathbf{Solve}(\mathcal{P})$$

- *Verify*: A deterministic algorithm performed by S. On input S_s and S_c, it outputs either one or zero.

$$\mathbf{Verify}(S_s, S_c) \in \{1, 0\}$$

Success: Define the success of a client solving a puzzle by:

$$\mathbf{Verify}(S_s, S_c) = 1, \ \ if \ S_s = S_c \ holds.$$

The diagram in Fig. 2 is a simple prototype for our trapdoor-based client puzzle scheme, in which the defending server takes charge of two positions: "Creator" and "Verifier", and a client is responsible for "Solver".

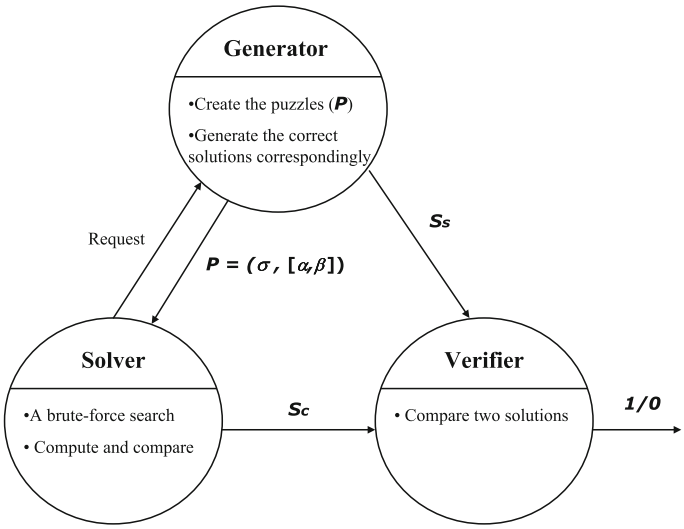

Fig. 2 A simple prototype

5 The DLP-Based Client Puzzle Scheme

5.1 Algorithm

The algorithm that we will deploy in our puzzle scheme is derived from Schnorr's Signature [7, 27], which was proposed in 1989 and could provide encryption and signature functions. Its security relies on the difficulty of computing discrete logarithms. First of all, we describe the algorithm as follows.

Let p be a large prime, and q be another prime, which is the prime factor of $p-1$. The size of p and q should be selected with care to ensure that DLP in $\mathbb{Z}_p^*$ is hard. g is a generator of the group G (order q). Denote $x \in \mathbb{Z}_q^*$ as a positive and secret integer. Let l be the difficulty lever of puzzles.

Our scheme is based on the following equations, in which random $a,\ b \in \mathbb{Z}_q^*$.

$$h = g^x \pmod{p} \tag{1}$$

$$C = g^a \cdot h^b \pmod{p} \tag{2}$$

$$W = a + b \cdot x \pmod{q} \tag{3}$$

Examining the above equations, we find a transformation by assembling them together as shown below.

- Put (1) into (2).

$$\begin{aligned} C = g^a \cdot h^b &= g^a \cdot (g^x)^b \\ &= g^a \cdot g^{bx} \\ &= g^{a+bx} \pmod{p} \end{aligned} \tag{4}$$

- Put (3) into (4).

$$C = g^{a+bx} = g^W \pmod{p} \tag{5}$$

  Note that g and p are constant in (5). Hence, if we can maintain the constancy of W, C should be a constant as well.

From (3), we can easily verify that a and b can be chosen arbitrarily, if the secret value x is known, to obtain W. Moreover, by a given random $a' \in \mathbb{Z}_q^*$, we can compute a unique $b' \in \mathbb{Z}_p^*$ by (6) such that W is kept constant.

$$b' = (W - a') \cdot x^{-1} \pmod{q} \tag{6}$$

Our puzzle scheme operates so that, using (3) and (5), the puzzle creator can easily compute the initial values of W and C from a pair (a, b). Then, keeping W and C constant, the creator changes the value of a' for each new puzzle and calculates a corresponding solution b' through (6). He also generates a search range associated with b', where the range length equals l (recall that l is the difficulty

level). The puzzle creator should broadcast p, h, and (2), but keep g, x, q, and W secret. In consequence, the only feasible method for a puzzle solver is to perform an exhaustive search in the candidate range to find a b' that makes (2) hold.

Since he knows the trapdoor – (6), the puzzle creator performs only one addition[2] and one modular multiplication to obtain a solution for each fresh a'. On the other hand, a solver must compute a sequence of modular exponentiations for testing each instance in the given range until he finds the correct answer. Meanwhile, the creator can adjust the size of the search range – l to control the computational costs consumed by the solvers.

5.2 System Description

Now, we look at how a system using our trapdoor-based puzzle scheme works for the defending server and legitimate clients. We split the scheme into four phases, use preprocessing to calculate, and store a number of constants for puzzle construction and verification, which are expected to relieve the computational overhead of the server.

1. **Preconstruction phase**

   (a) Determine a time interval T_d and the size of A denoted by n.
   There is a trade-off between T_d and n, which requires some careful thought. In fact, the server needs to find a minimum n and a maximum T_d that satisfy the following requirement: in each time period, n is by minimum greater than the maximum connection ability of the server. It means that no instance in set A can be chosen for puzzle construction more than once in same time period. For instance, if the hourly maximum connectivity of the server is 50,000 on average, n should be from 50,001 with $T_d = 1$.

   (b) Establish A.
   A is a random and nonrepeated integer set.

$$A = \{a_i |\, a_i \in \mathbb{Z}_q^*, |\, 1 \leq i \leq n\} \cap \{\forall a_m, a_n \in A |\, a_m \neq a_n\}$$

   (c) Compute G_A.
   G_A is computed by the server.

$$G_A = \{g^{a_i} \pmod{p} |\, 1 \leq i \leq n,\ a_i \in A\}$$

   (d) Generate and encrypt t.
   t is a time parameter which is in the form of

$$\{yymmddhh^1hh^2\} \cap \{hh^2 - hh^1 = T_d\}.$$

[2] A subtraction operation can be viewed as an addition in a computer system.

We can adjust the form of t according to the designated T_d. For instance, if $T_d = 30$ min, the form of t can be described as

$$\{yymmddhhmm^1 m^2\} \cap \{m^2 - m^1 = 30\}.$$

Then, the server computes $g^t \pmod p$ at the beginning of each new time period.

(e) Compute initial C, W, w_t and $x^{-1} \pmod q$.
Given a random pair $(a, b) \subset Z_q^*$, the server obtains

$$W = a + b \cdot x \pmod q$$

and

$$C = g^W \pmod p.$$

Compute

$$w_t = W - t \pmod q.$$

Using the extended Euclidean algorithm, the server easily computes $x^{-1} \pmod q$.

In the pre-construction phase, the server obtains constants $\{W, C, x^{-1}\ A, G_A\}$ and two periodic constants $\{g^t, w_t\}$.

2. **Construction phase**
When receiving a request from a client C, the server S generates a puzzle $\mathcal{P} = (g^{a'}, [\alpha, \beta])$ along with its correct solution b_s. $\mathcal{P}$ is essential for a client solving a puzzle. b_s is stored on the server for verification of the answer returned from the client. This stage is described as follows.

(a) Compute $g^{a'}$.

$$g^{a'} = g^t \cdot g^{a_i} \pmod p \tag{7}$$

where $g^{a_i} \in G_A$, t is the current time period, and g^t is a periodic constant calculated at the beginning of t. The server picks up random $g^{a_i} \in G_A$ to generate a fresh puzzle then marks this g^{a_i} to be unavailable until a new period comes. This means that when each period starts, all the elements in G_A are available. An element is marked unavailable throughout the same period, once it is chosen by the server to create a puzzle.

(b) Generate b_s.

$$\begin{aligned} b_s &= (W - a') \cdot x^{-1} && \pmod q \\ &= [W - (t + a_i)] \cdot x^{-1} && \pmod q \\ &= (w_t - a_i) \cdot x^{-1} && \pmod q \end{aligned} \tag{8}$$

Note that the values of i in (7) and (8) are uniform.

(c) Obtain a search range $[\alpha, \beta] \subset \mathbb{Z}_q^*$.

$$\alpha = b_s - c \pmod{q}$$

$$\beta = \alpha + l \pmod{q}$$

for random $c \in [0, l)$, where l is the current difficulty level of the puzzles.

3. **Puzzle-solving phase**
Unlike the other three, this phase is performed on the client's side. We assume that a client $\mathcal{C}$ has installed a specific piece of software which is distributed by the server. It solves puzzles received from the server with an equation

$$C = g^a \cdot h^b \pmod{p},$$

a triple of constants (C, h, p), and an interface for accepting puzzles from the server.
When a client receives a puzzle $\mathcal{P}$, he employs an exhaustive search (brute-force) to find the answer b_c which satisfies the equation. Due to the length of the search range $l = \beta - \alpha$, a client $\mathcal{C}$ needs to perform on average $l/2$ modular exponentiations to find the answer b_c.

4. **Verification phase**
Upon receiving the answer b_c from a client, the server compares it with the stored solution b_s that has been calculated in the construction phase. If they are equal, **Verify**$(\cdot)$ outputs 1, which means that authentication of the client is verified, and the server proceeds with the rest of the request. Otherwise, **Verify**$(\cdot)$ outputs 0, and the server drops the request.

$$\textbf{Verify}(b_s, b_c) = \begin{cases} 1 & \text{if } b_s = b_c \\ 0 & \text{otherwise} \end{cases}$$

5.3 Security Consideration

Theorem 1. *If (2) holds, then the solution b' computed by (6) can be verified.*

Proof: Given the initial values of (a, b), we can obtain the following result.

$$C = g^a \cdot h^b = g^a \cdot g^{bx} = g^{a+bx} = g^W \pmod{p}$$

To create a new puzzle, we choose a' and compute the solution $b' = (W - a) \cdot x^{-1} \pmod{q}$. Now, we verify it in (2).

$$\begin{aligned} C = g^W = g^{a'} \cdot g^{b'} &= g^{a'} \cdot h^{(W-a') \cdot x^{-1}} \pmod{p} \\ &= g^{a'} \cdot g^{(W-a') \cdot x^{-1} \cdot x} \pmod{p} \\ &= g^{a'} \cdot g^{W-a'} \pmod{p} \\ &= g^W \pmod{p} \end{aligned}$$

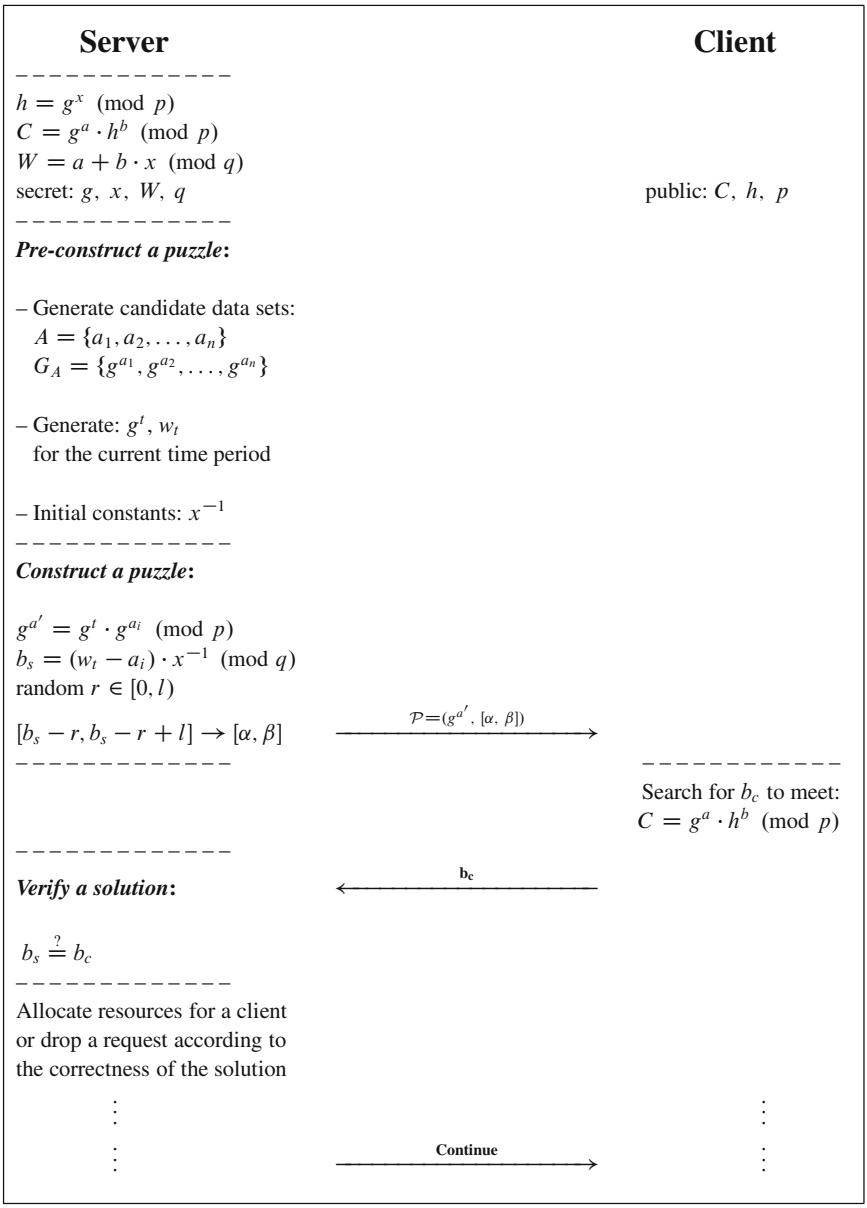

Fig. 3 A DLP-based puzzle scheme

Theorem 2. *Our scheme is secure assuming the hardness of the Discrete Logarithm Problem.*

Proof: To illustrate our proof, first let us recall the assumption made in the Discrete Logarithm Problem. Given two elements of a group, g^v and h, it is computationally infeasible to find a non-negative integer x such that

$$h = g^x \pmod{p}$$

holds, where the size of p is appropriately chosen (e.g. 1,024 bits).

We assume there is an algorithm A that, given $C, g, h \in Z_p{}^*$, outputs a, b such that

$$C = g^a \cdot h^b \pmod{p}$$

holds with a non-negligible probability.

We will show an algorithm B that uses A to solve an instance of a Discrete Logarithm Problem, which is assumed to be hard.

To be more precise, the task of algorithm B is to output

$$x = \log_g h \pmod{p}$$

given $g, h \in Z_p^*$ for a prime p. This simulation is as follows.

First, B provides g, h, p as the public parameters to A. Then, B performs the following:

- Select a random value $\theta \in Z_q^*$, where $q \mid p-1$.
- Compute: $C^* = g^\theta \bmod p$.

Finally, B provides C^* to A, and with a non-negligible probability, A outputs: (a, b) where:

$$C^* = g^a \cdot h^b \pmod{p}$$

holds. Now, the simulation is restarted. Again, first A is provided with the public parameters g, h, p, and finally given C^*. A will produce another forgery $(\hat{a}, \hat{b})$ where

$$C^* = g^{\hat{a}} \cdot h^{\hat{b}} \pmod{p}$$

holds. If $\hat{a} = a$ and $\hat{b} = b$ hold, then the last simulation needs to be repeated until $\hat{a} \neq a$ and $\hat{b} \neq b$. We note that A will output $\hat{a} = a$ and $\hat{b} = b$ with probability $1/q^2$.

After obtaining two values (a, b) and $(\hat{a}, \hat{b})$, B gathers the following equations:

$$C^* = g^a \cdot h^b \pmod{p}$$
$$C^* = g^{\hat{a}} \cdot h^{\hat{b}} \pmod{p}$$

Hence, B derives:

$$g^a \cdot h^b = g^{\hat{a}} \cdot h^{\hat{b}} \pmod{p}$$

which implies:

$$a + bx = \hat{a} + \hat{b}x \pmod{q}$$

or

$$x \cdot (b - \hat{b}) = \hat{a} - a \pmod{q}$$

and

$$x = (\hat{a} - a)(b - \hat{b})^{-1} \pmod{q}$$

Note that x is the solution to the DLP problem. The probability of success of B is lower bounded by $1/q^2$, which is non-negligible. Hence, we have provided the proof.

5.4 Remark

Remark 1. No client can precompute a puzzle or its solution in advance.

- In the preconstruction proces s, the defending server generates a random and nonrepeatable data set A ($A = \{a_i | 1 \leq i \leq n,\ a_i \in Z_q^*\}$). According to set A, the server obtains a multiplier factor set G_A ($G_A = \{g^{a_i} \pmod{p} \mid 1 \leq i \leq n,\ a_i \in A\}$), in which the values of g^{a_i} are consequently irregular.
- When the defending server constructs a puzzle, it performs the following computation:

  $$g^{a'} = g^t \cdot g^{a_i}$$

  where g^{a_i} is randomly selected from G_A. We notice that the probability of a client successfully predicting a puzzle equals $1/qn$. When q and n are big enough, precomputing a puzzle is computationally infeasible in the time period t.

Remark 2. The computational resources consumed by a client to solve a puzzle are greater than those used by the server to generate a puzzle and verify its solution.

A defending server performs two modular multiplications and three additions to create a puzzle and obtain the solution.

To produce $g^{a'}$, the server performs one modular multiplication:

$$g^{a'} = g^t \cdot g^{a_i} \pmod{p}$$

To obtain the solution, the server performs one modular multiplication and one addition:

$$b_s = (w_t - a_i) \cdot x^{-1} \pmod{q}$$

where w_t is computed at the beginning of the current time period.

To generate the search range $[\alpha, \beta]$, the server performs two additions:

$$\alpha = b_s - r$$

$$\beta = \alpha + l < N$$

To verify a solution, the server only needs to make a comparison.

$$b_s \stackrel{?}{=} b_c$$

Since we ensure that the sizes of p and q are large enough to resist any attacks on the DLP, the answer can only be obtained by exhaustively searching the seed range and performing modular exponentiations until the correct instance that satisfies the public equation is found. In consequence, to solve a puzzle, a client has to conduct on average $l/2$ modular exponentiations and comparisons, which are much more than the defending server does.

Remark 3. The DLP-based client puzzle scheme is better than the Hash function [4] and Diffie–Hellman [5] based puzzle schemes.

- The unique solution guarantees that the defending server can perform a simple comparison to verify puzzle solutions. This is more efficient and effective in protecting the verification phase against DoS attacks than the Hash function-based puzzle scheme, in which the defending server is required to perform a hash function for solution verifications.
- To adjust the difficulty of a puzzle, we exploit a non-negative integer l to control the length of a search range. The value of l varies according to the strength of a DoS attack. When the strength of an attack degree is low, l is correspondingly small and the cost of solving a puzzle is insignificant for a client. If an attack worsens and l enlarges, a client has to supply more resources to find a solution. On average, it requires $l/2$ modular exponentiations to solve a puzzle. Hence, our scheme is more easily measurable than the Hash function-based puzzle scheme.
- To avoid puzzle iteration, the Hash function-based scheme requires that a record of the used instances is kept for a long time, and it performs two comparisons in puzzle construction and verification respectively. However, relying on the time parameter t, our scheme can always obtain unique puzzles without wasting memory space and computational time. The items calculated in the preconstruction phase can also be reused in every new time interval.
- Apart from relying on the outsourcing for puzzle construction, the Diffie–Hellman based puzzle scheme requires a modular exponentiation to obtain a correct solution. Note that a modular exponentiation can be divided into many modular multiplications [19] and a large number of modular additions. Our DLP-based scheme, on the other hand, only needs two modular multiplications and three additions to obtain a puzzle and its corresponding solution, which is more efficient.
- Our scheme is resistant to eavesdropping attacks. As we claimed earlier, no puzzle construction information is revealed during communication between the defending server and its clients. In addition, each connection request has a unique solution, so that eavesdropping is rarely advantageous to an attacker.
- By varying the values of public instances (such as C, h, p) and constant instances (such as A, G_A), distinct web servers can obtain different puzzle schemes. This is more flexible for the current network servers that possess various capabilities.

6 System Configuration

In this section, we provide a general configuration for system parameters, which will enable the defending server to detect DoS attacks and start up the defence system in time. Our scheme begins with a regular examination to determine whether it is currently under a DoS attack. A standard parameter for an attack can be the status of buffer space consumed or the number of TCP connections. Here, we prefer the status of buffer space consumed as a standard; so let B_f denote the whole buffer space of a defending server and C_{sm} be the number of buffers consumed. We assume that there are five values of C_{sm} which reflect the strength of an attack (this can be adjusted flexibly according to the different web servers), and we call them 1–5. When less than half of B_f is consumed, the value of C_{sm} is zero, which means there is no attack alarm for a defending server. When the number of consumed buffers comes to half of B_f, the value of C_{sm} is increased to 1, indicating that the server is suspected of being under an attack. From now on, the defending server needs to construct and send out a puzzle for every client who is applying for a connection until the alarm is removed again. When the number of consumed buffers increases to 3/5 B_f, 7/10 B_f, 4/5 B_f, and 9/10 B_f, respectively, the values of C_{sm} are 2, 3, 4, and 5 correspondingly.

We have another parameter l – the difficulty level of a puzzle, which decides the length of the search range related to C_{sm}. When C_{sm} is zero, the value of l is also zero. When C_{sm} is increased to 1, l becomes 10 (the value of l can also be adjusted according to different web servers). As C_{sm} increases, the value of l appears to grow exponentially. We describe some variations below:

$$
\begin{aligned}
C_{sm} &= 0, \quad l = 0;\\
C_{sm} &= 1, \quad l = 10;\\
C_{sm} &= 2, \quad l = 100;\\
C_{sm} &= 3, \quad l = 1,000;\\
C_{sm} &= 4, \quad l = 10,000;\\
C_{sm} &= 5, \quad l = 100,000;
\end{aligned}
$$

We can see that the difficulty of a puzzle can be increased from 0 to a figure that is computationally infeasible as an attack becomes more severe.

We describe the details of our defending system as follows.

1. $C_{sm} = 0$**, which means there is no attack alarm for our defending server.** When a client sends an initial connection request and an enquiry as to whether the server is under a DoS attack, the server answers "No". Then, the client and the server should continue and finish their connection according to a standard connection establishment protocol such as TCP. The period of validation for this connection is within a certain time T_v.

2. $C_{sm} \geq 1$**, which means our defending server is suspected of being under a DoS attack.**
   When a client sends an initial connection request and an enquiry as to whether the server is under a DoS attack, the defending server constructs a puzzle according to the current value of l and sends back "Yes", together with a set of puzzle parameters, to the client. The server keeps a correct solution b'_{si} for every connection request i for a specified period T_w. A client uses the puzzle's parameters to solve the problem and sends the solution back. The defending server compares the two solutions and then decides whether to proceed or drop this request. A connection request should be dropped immediately if a solution received from a client is incorrect, or the time for sending back a solution is beyond T_w.

7 Discussion

In this section, we will discuss some possible attack scenarios and problems that occur when a defending server implements our trapdoor-based puzzle scheme.

- **Discussion 1**
  If a malicious client floods a defending server with initial connection requests by using spoofed IP addresses or his "Zombies" (see Fig. 4), how can a server protect itself? Wasting system resources to construct and send out a puzzle for every connection request is not an efficient approach for protecting a web server. The aim of such an attack is to make a defending server perform a large number of meaningless computations for the construction of puzzles, which may lead to the possibility of exhausting the server's resources. Recall that a basic principle of client puzzles in defending against DoS attacks is to force a client to consume a

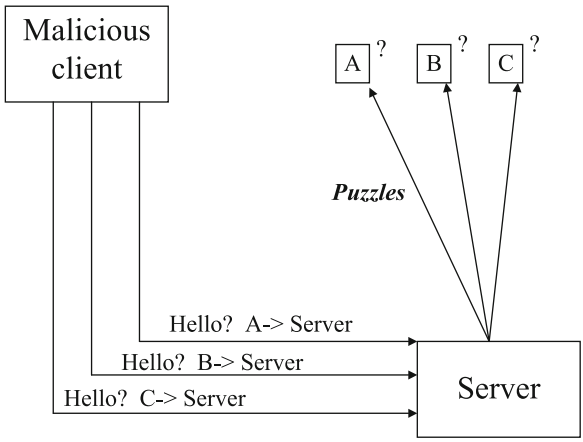

Fig. 4 A threat to our trapdoor-based puzzle scheme

large number of resources before obtaining resources from the defending server. If we do not comply with this rule, the threat of DoS attacks still exists.
We propose two improved solutions to defeat this threat.

Solution 1. We propose to calculate and obtain the puzzle solution pairs ($g^{a'}$, b'_s) in advance, and store them in a secret table on a defending server, before every new time period t starts. When there is a connection request, the server does not need to construct a puzzle by calculating the modular multiplication for $g^{a'}$ and b'_s. The server just fetches a pair of puzzle solutions ($g^{a'}$, b'_s) from the table, keeps b'_s, and sends the value of $g^{a'}$ to the client. Hence, the only thing that the server needs to do, when there is a connection request, is to select a random integer r to generate the seed range $[b'_s - r, b'_s - r + l]$ for b'_c. The size of the table can be determined by the capability of the connectivity of the defending server.

Solution 2. Before describing our second scheme, we make the following assumption for this solution. A malicious client who deploys IP spoofing cannot intercept all the packets sent from a defending server to the spoofed addresses. When a defending server receives a connection request, it sends back a sequence number to determine whether the address is bogus. The aim of this option is to alleviate an attack via the sequence number in this first step, and then ban it in the later puzzle verification.
When a client sends an initial connection request and an enquiry as to whether the server is under a DoS attack, the server returns "Yes" and a sequence number to the received IP address. If the client is willing to accept the puzzle, he should return a value that is equal to the sequence number plus one in a strict time period T_s. After receiving the correct sequence number in a valid time period, the server generates and sends a cryptographic puzzle to the client. Otherwise, the server will drop the service request immediately. The sequence number is a filter for spoofed IP addresses, which works under the above assumption. A legitimate client has to simply solve a puzzle and send the solution in a given time T_w to obtain the final service. If the solution is correct, the server will proceed with the rest of the connection request and distribute the system's resources to the client. On the other hand, if the answer is wrong or the time for computation is beyond the given time T_w, the server will drop this request. If a connection is established, the period of validation for this connection is within a specified time T_v.

- **Discussion 2**
  Compared with a hash-based puzzle, in which a defending server does not need to store any client information while a client is solving the puzzle, our scheme requires a defending server to store a solution b'_s. These are two totally different purposes. The hash approach sacrifices CPU time in the verification process to avoid any memory becoming exhausted. Our trapdoor-based puzzle conducts a reverse activity. We note that the final decision should be made by the server administrators, who are aware of which resources are more valuable for their own systems.

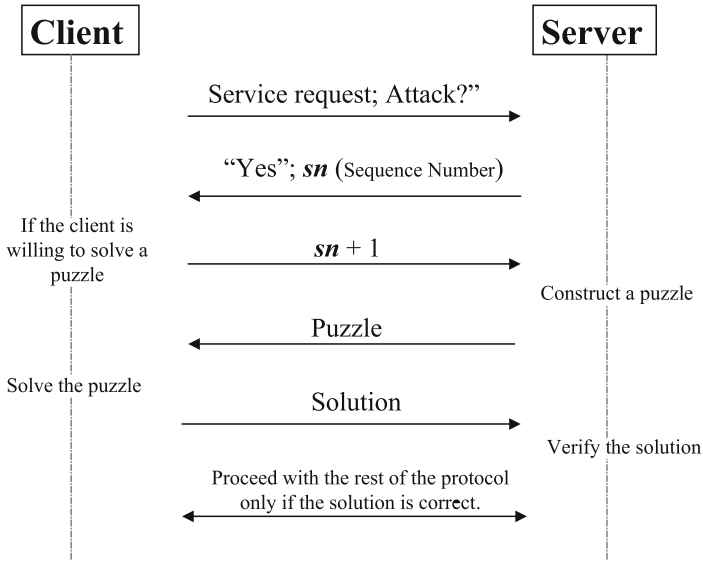

Fig. 5 Using a sequence number against IP Spoofing

- **Discussion 3**
  The main goals of our client puzzle scheme are to defend network servers against DoS attacks and ensure that legitimate clients gain qualified service as long as they can solve the puzzles. However, some DoS attacks may aim to take down clients' machines, where an attacker may broadcast false puzzles to consume a client's resources.
  In this case, the current solution (for example, the puzzle scheme proposed by Aura et al. [4]) is to make puzzles signed by the defending server. When receiving a puzzle, a client first verifies the signature by using the public key of the defending server and then computes it. Since the number of puzzles for a legitimate client is very small, the cost of public-key verification on the client's side is usually acceptable.

8 Conclusion

In this chapter, we have depicted a novel trapdoor-based client puzzle scheme. The puzzles generated by our scheme can conquer the potential problems of the traditional ones, such as the complexity of puzzle construction and verification. We have showed and explained that our scheme is efficient with a low computational cost and can be deployed in the current existing network protocols. Finally, we have presented that our scheme is provably secure assuming the hardness of the Discrete Logarithm Problem(DLP).

References

1. Digital signature standard (DSS). In Federal Information Processing Standards Publication 186. National Institute of Standards and Technology (NIST), 1994.
2. The New York Times, 12 September, 1996.
3. R. Aguilar, and J. Kornblum. New York Times site hacked. CNET NEWS.COM, 8 November, 1996.
4. T. Aura, P. Nikander, and J. Leiwo. Dos-resistant authentication with client puzzles. Security Protocols, 8th International Workshop, Cambridge, UK, April 3–5, 2000; revised papers, Vol. 2133 of Lecture Notes in Computer Science, pp. 170–177, Springer, 2001.
5. B. Waters, A. Juels, J. A. Halderman, and E. W. Felten. New client puzzle outsourcing techniques for dos resistance. In ACM Conference on Computer and Communications Security, pp. 246–256, 2004.
6. D. Bernstein. Syn floods - a solution. Available at http://www.op.net/jaw/syn-fix.html, 1996.
7. E. Brickell, and K. McCurley. An interactive identification scheme based on discrete logarithms and factoring. In Advances in Cryptology, Proceedings EUROCRYPT 90, LNCS 473, Vol. 5, pp. 23–29. Springer, 1991.
8. CNN. Cyber-attacks batter Web heavyweights. Available at http://www.cnn.com/2000/tech/computing/02/09/cyber.attacks.01/index.html, February 2002.
9. daN. Re: client puzzle protocol neohapsis archives. Available at http://archives.neohapsis.com/archives/nfr-wizards/2000-q1/0645.html, 2000.
10. C. Davidson. The "SYN flood" gates open for WebCom. iWorld Weekly, 16 December, 1996.
11. C. Dwork, and M. Naor. Pricing via processing or combatting junk mail. In Advances in Cryptology, Proceedings CRYPTO 92, LNCS 740, pp. 139–147, Santa Barbara, CA USA, Springer, August 1992.
12. T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. IEEE Transactions on Information Theory, 31:469–472, 1985.
13. J. Elliot. Distributed denial of service attacks and the zombie ant effect. IT Professional, pp. 55–57, March 2000.
14. P. Ferguson, and D. Senie. Network ingress filtering: Defeating denial of service attacks which employ ip source address spoofing. IETF, RFC 2267, January 1998.
15. A. Juels, and J. Brainard. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In S. Kent, (Ed.), Distributed Systems Security (SNDSS), pp. 151–165, 1999.
16. F. Kargl, J.Maier, and M. Weber. Protecting web servers from distributed denial of service attacks. In Proceedings of the 10th International WWW Conference, Hong Kong, May 1–5, 2001.
17. C. Kaufman, R. Perlman, and M. Speciner. Network Security: Private Communication in a Public World (2nd Edition). Prentice Hall PTR, 2002.
18. A.K. Lenstra, and H.W. Lenstra, Jr. Algorithms in number theory. In J. van Leeuwen, (Ed.), Handbook of Theoretical Computer Science, Vol. A, pp. 673–715, MIT/Elsevier, 1990.
19. C. Mclvor, M. Mcloone, and J. Mccanny. Modified montgomery modular multiplication and rsa exponentiation techniques. In IEE Proceedings - Computers & Digital Techniques, Vol. 151, pp. 402–408, November 2004.
20. A. Oldyzko. Discrete logarithms in finite fields and their cryptographic significance. In Advances in Cryptology, Proceedings EUROCRYPT 84, LNCS 209, pp. 224–314, Springer, 1984.
21. K. Park, and H. Lee. On the effectiveness of probabilistic packet marking for ip traceback under denial of service attack. IEEE INFOCOM 2001, pp. 338–347, 2001.
22. K. Park, and H. Lee. On the effectiveness of route-based packet filtering for distributed dos attack prevention in power-law internets. In Proceedings of ACM SIGCOMM'2001, August 2001.
23. K. Park, and H. Lee. Advanced packet marking mechanism with pushback for ip traceback. In ACNS04 PROGRAM - Academic Track, June 8–11, 2004.
24. M. B. Rash. client puzzle protocol. Available at http://honor.trusecure.com/pipermail/firewall-wizards/2000-february/007944.html, 2000.

25. L. Ricciulli, P. Lincoln, and P. Kakkar. TCP SYN flooding defense. In In Communication Networks and Distributed Systems Modeling and Simulation Conference (CNDS'99), 1999.
26. B. Schneier. Applied cryptography : protocols, algorithms, and source code in C. Wiley, 1996.
27. C. Schnorr. Efficient signature generation for smart cards. In Advances in Cryptology, Proceedings CRYPTO 89, LNCS 435, pp. 239–252, Springer, 1990.
28. L. Sherriff. Virus launches ddos for mobile phones. Available at http:// www.theregister.co.uk/ content/1/12394.html.
29. C. Wang, C. Lin, and C. Chang. Signature schemes based on two hard problems simultaneously. In the 17th International Conference on Advanced Information Networking and Applications, pp. 557–560, 2003.
30. G. Weijers. re:client puzzle protocol. Available at http://archives.neohapsis.com/archives/ nfr-wizards/2000-q1/0558. html, 2000.
31. M. Williams. Ebay, amazon, buy.com hit by attacks. IDG News Service, 9 February 2000.
32. B. Ziegler. Hacker tangles panix Web site. Wall Street Journal, 12 September 1996.