Improved Cryptographic Puzzle Based on Modular Exponentiation

Lakshmi Kuppusamy and Jothi Rangasamy

Abstract Cryptographic puzzles are moderately hard–neither easy nor hard to solve—computational problems. They have been identified to be useful in mitigating a type of resource exhaustion attacks on Internet protocols. Puzzles based on modular exponentiation are interesting as they possess some desirable properties such as deterministic solving time, sequential (non-parallelizable) solving process and linear granularity. We propose a cryptographic puzzle based on modular exponentiation. Our puzzle is as efficient as the state-of-art puzzle of its kind and also overcomes the major limitation of the previous schemes.

Keywords Cryptographic puzzle · Proof-of-work · Denial-f-service protection · Unforgeability · Difficulty

1 Introduction

A cryptographic puzzle is a *moderately difficult* computational problem in which a prover(client) must demonstrate to a puzzle generator (verifier) that it has performed the required computational task. Cryptographic puzzles were first introduced as *proof-of-work* systems by Dwork and Naor [8] in 1992 for combating junk emails [8]. Rivest et al. (1996) used puzzles for realizing *time-release cryptography* [19]. Juels and Brainard (1999) considered puzzles to mitigate Denial-of-Service (DoS) attacks in network protocols. In server-client scenario, they are known as *client puzzle* protocols.

DoS attack is one of the most common real-world network security attacks and presents a severe threat to the Internet and e-commerce. In DoS attack, the attacker

L. Kuppusamy · J. Rangasamy (✉)
Society For Electronic Transactions and Security, MGR Knowledge City,
CIT Campus, Taramani, Chennai 600113, Tamilnadu, India
e-mail: jothi.rangasamy@gmail.com

L. Kuppusamy
e-mail: lakshdev21@gmail.com

R.N. Mohapatra et al. (eds.), *Mathematics and Computing*,
Springer Proceedings in Mathematics & Statistics 139,
DOI 10.1007/978-81-322-2452-5_8

targets to drain out the service provider's resources such as bandwidth, memory, and computational time so that the resources will become unavailable to process legitimate clients' requests. In recent years, major e-commerce sites including eBay, Yahoo!, Amazon, and Microsoft's name server [17] have faced huge financial loss due to DoS attacks. Very recently, a DoS attack on several thousands of *time keeping servers* distributed across the world to keep the time in sync by running the network time protocol (NTP) has been mounted. Two vulnerabilities in the NTP were exploited by the attackers to mount the DoS attack using IP spoofing technique. This attack has been described as the world's largest DoS attack to date by security researchers due to its amplification factor of 206x.

Cryptographic puzzles have been shown to be a promising and effective mechanism to deter the effect of malicious requests. When the server is under DoS attack, it generates a (client) puzzle instance and sends it as a reponse to the client's connection request. The server processes the client's request only if the client proves its legitimate intentions of getting the request by sending the correct puzzle solution. Generating and verifying a client puzzle must be computationally easy for the server. That is, it must add a little computational and memory overhead to the server. Otherwise, the client puzzle may introduce a resource exhaustion attack where an attacker triggers puzzle generation and verification process by sending a large number of pretended requests or a large number of fake puzzle solutions respectively.

On the other hand, finding a correct solution to the client puzzle must be moderately hard for the client. This property is called *puzzle difficulty* which is a property that every good puzzle must satisfy. That is, for a legitimate client, the computational burden for solving a client puzzle is not high, whereas for an attacker who makes multiple connection requests, finding solution for many client puzzles received through multiple requests must be a huge resource-consuming process.

1.1 Modular Exponentiation-Based Puzzle

Client puzzles are mostly either hash based [3, 9, 11] or modular exponentiation [12, 19] based puzzles. Though it is essential that all the client puzzles must satify the puzzle difficulty property, exponentiation-based client puzzles are known to achieve additional properties such as non-parallelizability, deterministic solving time, and finer granularity. In a non-parallelizable client puzzle, the solution finding time remains constant even if the attacker/client uses multiple machines to solve a single client puzzle. Unlike in the hash-based puzzles where the running time to find a puzzle solution is probabilistic, the modular exponentiation-based puzzles have the property that the minimum amount of work required to solve a puzzle can be determined. Moreover, these puzzles support linear granularity; the puzzle generator (server) has the ability to increase the puzzle difficulty level linearly. This property is useful since the puzzle issuing server will have more options for the difficulty level and can choose one accordingly.

Rivest et al. [19] gave the first modular exponentiation-based puzzle which achieves non-parallelizability, deterministic solving time, and finer granularity. A problem with Rivest et al. puzzle construction is that the server has to perform modular exponentiation in order to verify the puzzle solution. Karame and Čapkun [12] proposed two puzzle constructions. First, one works for the fixed-difficulty level and reduces the running time of the puzzle verification by a factor of $\frac{|n|}{2k}$ for a given RSA modulus n, where k is the security parameter compared to Rivest et al.'s puzzle. In a puzzle with fixed difficulty, the busy server cannot adjust the difficulty levels of the puzzle based on its load. The second scheme of Karame and Čapkun supports various difficulty levels but it doubles the verification cost of their first scheme. Though Karame and Čapkun's puzzle is superior in efficiency compared to Rivestet al. puzzle, it still requires modular exponentiation for puzzle verification. To avoid mdoular exponentiation in the Karame-Čapkun puzzle verification, an alternative construction, namely RSApuz was proposed in [18], wherein the verification requires only few modular multiplications. However, the approach in [18] works only for the fixed difficulty level. In real-world attacks such as denial-of-service attacks, the target server is kept very busy in performing varoius computational tasks. Thus puzzles can be an effective countermeasure to DoS attacks when they support variable difficulty levels and avoid modular exponentiation cost on their side. The state-of-art puzzles, namely [12] and [18] fail to meet at least one of the above desirable properties as seen in Table 1.

1.2 Contributions

1. We give an efficient modular exponentiation-based puzzle which achieves non-parallelizability, deterministic solving time, and finer granularity. Our puzzle is superior in efficiency to Karame and Čapkun's variable puzzle difficulty level puzzle. Though our scheme is similar to [12] and [18], our puzzle does not involve any modular exponentiation during puzzle verification unlike [12] and does not require to repeat the pre-computation procedure to change the puzzle difficulty level unlike [18]. Our construction requires only a few modular multiplications

Table 1 Comparision of modular exponentiation-based puzzles

Puzzle	Difficulty level	Verification
RSWpuz [19]	variable	$\|n\|$-bit mod. exp.
KCpuz [12]	fixed	k-bit mod. exp.
KCpuz [12]	variable	$2k$-bit mod. exp.
RSApuz [18]	fixed	3 mod. mul.
Ours	variable	3 mod. mul.

Legend n is an RSA modulus, $k \ll n$ is a security parameter

to verify puzzle solutions. Table 1 compares our puzzle with other puzzles of the same kind.
2. We show that our puzzle is unforgeable and difficult in the puzzle security model proposed by Chen et al. [7]

Outline: We organize the rest of the paper as follows: Sect. 2 breifly presents the related work. Design and security analysis of our puzzle construction is described in Sect. 3. Finally Sect. 4 concludes our work.

2 Background on Modular Exponentiation-Based Puzzles

This section discusses the state-of-art puzzle schemes and identifies their limitations. Throughout the paper we use the following notations: Let n be an integer and $|n|$ be the length of the interger in bits; let $\phi(n)$ be the Euler phi function of n; the set of all integers $\{a, \ldots, b\}$ between and including a and b be denoted by $[a, b]$; denote by $x \leftarrow_r S$ to choose an element x uniformly at random from s set S; for an algorithm A to run on input y and produce an output x, we denote it by $x \leftarrow A(y)$; let $\text{negl}(k)$ denote a function which is neglible in k, where k is a security parameter; We denote p.p.t for a probabilistic polynomial time algorithm.

2.1 RSWpuz

Rivest et al. [19] proposed a puzzle scheme based on repeated squarings, which we call RSWPuz. In their puzzle construction, the puzzle generating server first chooses an RSA modulus $n = pq$ using two large primes p and q and then computes the Euler totient function $\phi(n) = (p-1)\cdot(q-1)$. Now the server sends a tuple (a, Q, n) as a puzzle instance to the client after selecting the difficulty level Q and an integer $a \leftarrow_r \mathbb{Z}_n^*$. Observe that the difficulty level determines the amount of work a client has to do. Now, the client performs Q repeated squarings to compute $b \leftarrow a^{2^Q} \bmod n$ and returns b to the server as a puzzle solution. After receivng the puzzle solution, the server checks whether $a^c \stackrel{?}{\equiv} b \bmod n$ where $c = 2^Q \bmod \phi(n)$. The server can reuse the computation of c as long as the puzzle difficutly value Q is fixed. Since the server knows the trapdoor information $\phi(n)$ the server can verify the solution in one $|n|$-bit exponentiation, whereas the client is forced to do Q repeated squarings for $Q \gg |n|$.

Note that the puzzle verification step is expensive in RSWPuz scheme as it involves the computation of full $|n|$-bit modular exponentiation on the server side. A malicious client can exploit this weakness to send a large number of fake puzzle solutions. The busy server now needs to engage in computationally expensive operation to verify all of them. Hence, the client puzzle construction itself introduces a new vulnerability to a resource exhaustion-based DoS attack.

2.2 KCpuz

Rivest et al.'s puzzle construction was improved by Karame and Čapkun [12]. The improvement in terms of computational efficiency is the significant reduction of puzzle verification cost from $|n|$-bit exponentiation (Rivest et al. puzzle verification cost) to $2k$-bit exponentiation modulo n for a security parameter k. That is, the burden for the server is reduced by a factor of $\frac{|n|}{2k}$. Their scheme with variable difficulty level, which we call KCPuz is illustrated in Fig. 1. Unlike [19], Karame and Čapkun analyzed their puzzle scheme under the security notions from [7] and showed that the puzzle satisfies both the unforgeability and difficulty notions.

Though KCPuz scheme requires less computation cost to verify each puzzle solution compared to RSWPuz, it still needs a $2k-$bit modular exponentiation. This could still be a burdensome computation for DoS defending servers. Also, KCPuz does not provide the property of finer granularity. That is, the gap between the two adjacent difficulty levels must be large for security reasons. In particular, the next difficulty level R' must satisfy $\frac{R'}{R} \geq n^2$ where R is the current difficulty level. This reduces the number of possible and acceptable difficulty levels to be chosen by the puzzle generator.

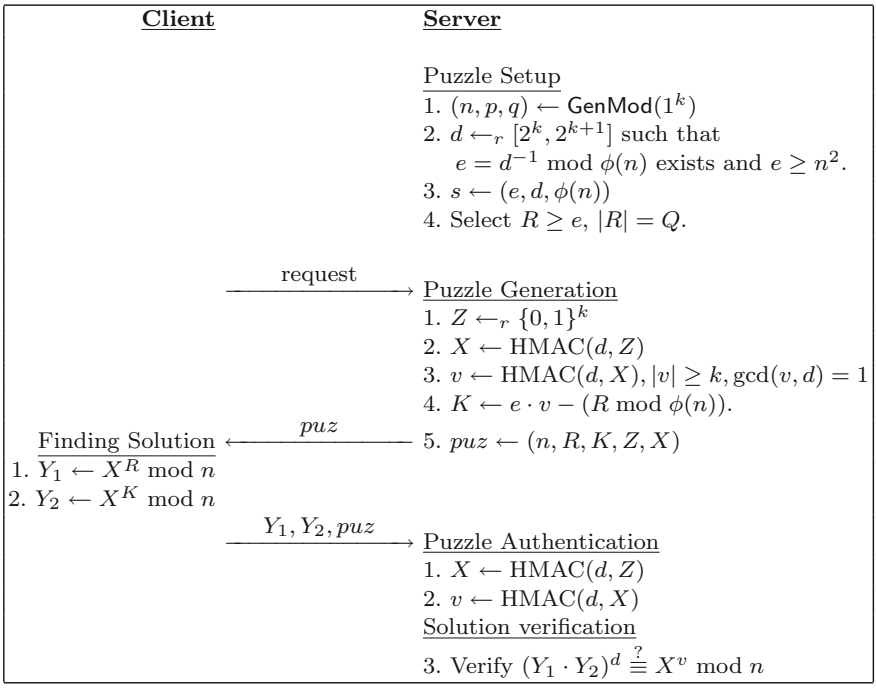

Fig. 1 The KCPuz Scheme [12]

The difficulty level for client puzzles employed in DoS scenarios is typically set between 0 to 2^{25} operations. Hence, the possible successive difficulty levels for KCPuz scheme are $R = 2^{512}$, $R' = 2^{1536}$ and $R'' = 2^{2560}$ for 512-bit moduli.

2.3 RSApuz

An alternative and more efficient vewrsion version of KCPuz was proposed in [18], which we call RSAPuz. In RSAPuz the puzzle issuing server does the most computation work offline so that it does not perform any modular exponentiation online during puzzle generation and solution verification. In fact, the solution verification requires only three bit modular multiplications and thus its efficiency is comparable with that of hash function-based puzzles [18]. RSAPuz is shown to meet the security notions of Chen et al. and additional desirable properties such as finer granularity, non-parallelizability, and deterministic solving time. The RSAPuz scheme is depicted in Fig. 2.

RSAPuz uses the (BPV) technique due to Boyko et al. [5] which reduces the online computation cost for the pair (x, X). The technique has two phases: BPV preprocessing phase, namely BPVPre and the BPV pair generation phase BPVGen. The pre-processing phase computes N pairs of the form (α_i, β_i) where $\alpha_i \leftarrow_r \mathbb{Z}_n^*$ and $\beta_i \leftarrow {\alpha_i}^u \bmod n$ for $i = 1, \ldots, N$ and stores them in a table. Whenever a new pair (x, X) is required to be computed online, the pair generation phase *randomly* chooses ℓ out of N pairs and computes the new pair as follows: $(x, X) \leftarrow (\prod_{j=1}^{\ell} \alpha_j, \prod_{j=1}^{\ell} \beta_j)$. Thus RSAPuz does not perform any computationally intensive operation online.

Though the puzzle verification requires only few modular multiplications, it works only for the fixed difficulty level. For changing one difficulty level to the other, the puzzle scheme needs to run the computationally demanding pre-computation again. That is, in the pre-computation phase (as seen in Fig. 2), the server first selects the difficulty parameter R of length Q, computes $u \leftarrow d - (2^Q \bmod \phi(n))$ and then runs the BPV pre-processing step with inputs (u, n, N) to obtain N pairs (α_i, β_i). Hence the server has to run the pre-computation phase every time the difficulty needs to be changed.

All the modular exponentiation-based puzzles in the related literature add computational burden, either offline (e.g., precomputation in Fig. 2) or online (e.g., solution verification in Fig. 1), to support change of difficulty. In the following section, we overcome the limitations in the above puzzle schemes by proposing a new modular exponentiation-based client puzzle which is as fast as RSAPuz and adds no cost to support variable difficulty. We then analyze its security properties.

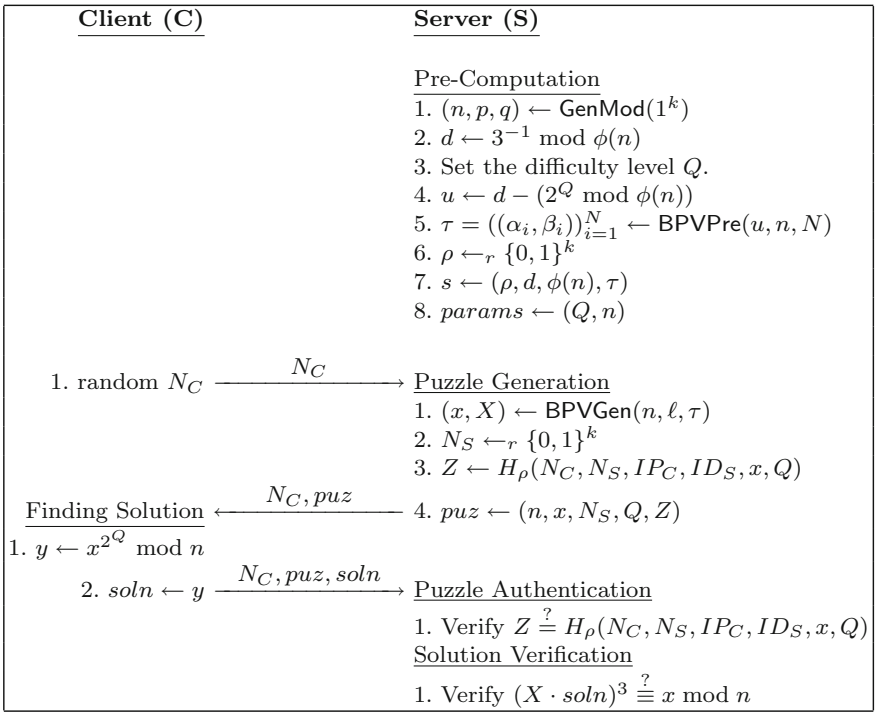

Fig. 2 The RSAPuz Scheme [18]

3 The Proposed Puzzle Scheme

Now we propose an efficient puzzle scheme requiring only few modular multiplications for puzzle generation and solution verification. Unlike in the existing puzzle schemes, our puzzle achieves both efficiency and the security properties such as unforgeability, puzzle difficulty, deterministic, non-parallelizability, and finer granularity. Our scheme uses all the algorithms used by RSAPuz in Fig. 2.

3.1 Definitions

We begin by defining an algorithm to generate a modulus $n = pq$ similar to the generation of RSA modulus as below:

Definition 1 (*Generating Modulus n*) For a security parameter k, the algorithm to generate a modulus n is a probabilistic polynomial time algorithm GenMod which accepts the input 1^k and produces (n, p, q) as output such that $n = pq$ where p and q are k-bit primes.

Like [18], our puzzle requires a server to generate a pair (x, X) for each puzzle which involves modular exponentiation. To avoid this exponentiation cost, we use the (BPV)technique proposed by Boyko et al. [5] which requires few modular multiplications and pre-computed values to generate the pairs of the form (x_i, X_i) where $X_i = x_i^u \bmod n$ for some predefined exponent u.

Definition 2 (*BPV Technique*) Suppose that $N \geq \ell \geq 1$ for the parameters N and ℓ. Let $n \leftarrow \mathsf{GenMod}(1^k)$ be an RSA modulus and u be an element in $\mathbb{Z}_{\phi(n)}$ of length m. The BPV technique has the following two phases:

- $\mathsf{BPVPre}(u, n, N)$: This pre-processing algorithm run once, generates N random integers $\alpha_1, \alpha_2, \ldots, \alpha_N \leftarrow_r \mathbb{Z}_n^*$ and computes $\beta_i \leftarrow {\alpha_i}^u \bmod n$ for each i. A table $\tau \leftarrow ((\alpha_i, \beta_i))_{i=1}^N$ consisting of pairs (α_i, β_i) is finally returned.
- $\mathsf{BPVGen}(n, \ell, \tau)$: Whenever a pair $(x, X \bmod n)$ is needed, the algorithm chooses a random set $S \subseteq_r \{1, \ldots, N\}$ of size ℓ and computes $x \leftarrow \prod_{j \in S} \alpha_j \bmod n$. If $x = 0$, then the algorithm stops and generates S again. Else, it computes $X \leftarrow \prod_{j \in S} \beta_j \bmod n$ and return (x, X). The indices S and the corresponding pairs $((\alpha_j, \beta_j))_{j \in S}$ are kept secret.

Security Analysis of BPV Technique. The results by Boyko and Goldwasser [4] and Shparlinski [20] show that the value x generated using the BPV technique are statistically close to the uniform distribution. In particular, the following theorem shows that with overwhelming probability on the choice of α_i's, the distribution of x is statistically close to the uniform distribution of a randomly chosen $x' \in \mathbb{Z}_n^*$.

Theorem 1 ([4], Chap. 2) *If $\alpha_1, \ldots, \alpha_N$ are chosen independently and uniformly from $\mathbb{Z}_n^*$ and if $x = \prod_{j \in S} \alpha_j \bmod n$ is computed from a random set $S \subseteq \{1, \ldots N\}$ of ℓ elements, then the statistical distance between the computed x and a randomly chosen $x' \in \mathbb{Z}_n^*$ is bounded by $2^{-\frac{1}{2}\left(\log \binom{N}{\ell}+1\right)}$. That is,*

$$\left| \Pr\left(\prod_{j \in S} \alpha_j = x \bmod n \right) - \frac{1}{\phi(n)} \right| \leq 2^{-\frac{1}{2}\left(\log \binom{N}{\ell}+1\right)} .$$

BPV Metrics. For defending against DoS attacks, the difficulty level Q can be set between 0 and 2^{25} operations. In [18] it is recommended to select N and ℓ such that $\binom{N}{\ell} > 2^{40}$. Instead of choosing $N = 512$ and $\ell = 6$ for the BPV generator as per Boyko et al. [4, 5], we can choose $N = 2500$ and $\ell = 4$ so as to reduce the number of online modular multiplications performed during the BPV pair generation process. We refer to [18] for more details about the choices of N and ℓ in a DoS scenario.

Making BPV pair generation process offline. At ESORICS 2014, Wang et al. [23] proposed that the BPV pair generation process can be executed offline. That is, the pairs generated during the BPV pre process $\mathsf{BPVPre}(u, n, N)$ are stored in the static table (ST) and the pairs generated during the BPV pair generation process $\mathsf{BPVGen}(n, \ell, \tau)$ are stored in the dynamic table (DT). Whenever a BPV pair is

required during puzzle generation, an entry from DT is selected and the table is updated with another BPV pair in idle time. The use of dynamic table allows us to completely avoid the number of modular multiplication computations required for the BPV pair generation process and thus make our puzzle more efficient.

3.2 The Construction

Our client puzzle illustrated in Fig. 3 is executed as a series of message exchanges between a client and a DoS defending server. The server generates a puzzle instance using BPV pairs computed offline and verifies the puzzle solution sent by the client as follows:

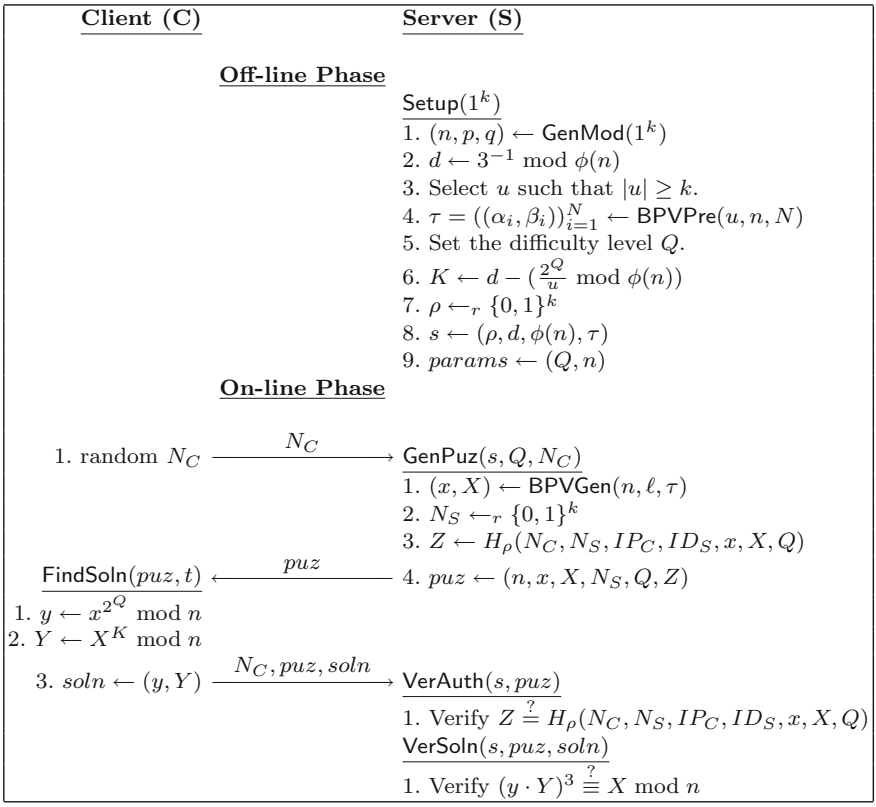

Fig. 3 Our modular exponentiation-based puzzle with variable difficulty

- PRE- COMPUTATION. In the pre-computation phase, the server generates (n, p, Q) using the modulus generation algorithm and generates d as the inverse of 3 mod $\phi(n)$. Then the server runs the BPVPre phase by selecting N and u such that $|u| \geq k$. Unlike [18], our algorithm requires to run the BPVPre phase (to obtain N pairs of (α_i, β_i)) only once as it does not depend on the difficulty level Q.

For a client puzzle to be effective against resource exhaustion-based DoS attacks, generation of puzzles and verification of their solutions should be very efficient for the busy server as can be seen in our scheme described below:

- PUZZLE GENERATION (GenPuz). The server has to spend its significant computational resource for generating the puzzle through BPV pair generation BPVGen process which requires $2(\ell - 1)$ modular multiplications. That is, it requires $\ell - 1$ modular multiplications to compute x and another $\ell - 1$ modular multiplications to compute X. The server runs pseudo-random function H_ρ to compute the puzzle-authentication tag Z after generating a nonce N_s at random. Note that ℓ could be set between 4 and 16 so that the puzzle requires only 8 modular multiplications for $\ell = 4$ [4, 18].
- PUZZLE AUTHENTICITY VERIFICATION (VerAuth). Verifying that the puzzle is originated from the server can be done using the pseudo-random function H_ρ again and comparing the result with the received Z.
- PUZZLE VERIFICATION (VerSoln). The puzzle solution is verified by performing only 3 modular multiplications.

Observe that our puzzle scheme does not require the server perform any modular exponentiation either to generate the puzzle or to verify its solution. On the other hand, the client has to perform modular exponentiations to find the solution to the puzzle as follows:

- PUZZLE SOLUTION (FindSoln). After receiving the puzzle from the server, the client computes the puzzle solution in the form of two modular exponentiations $x^{2^Q} \bmod n$ and $x^K \bmod n$. The client can opt either to factor n or to perform repeated squarings to solve the puzzle. Since factoring is hard, the best knowm method for the client to find the solution is to implement the square and multiply algorithm and perform repeated squarings, which is believed to be a highly sequential process [10, 12, 19]. Hence the client will be performing exactly Q sequential modular multiplications to find $x^{2^Q} \bmod n$ and $O(\log K)$ sequential modular multiplications to find $x^K \bmod n$, and hence the puzzle has deterministic solving time of Q repeated squarings and non-parallelizability properties.

3.3 Security Analysis

Client puzzles were analyzed in various security models proposed in [6, 7, 21]. In this section, we analyze our puzzle scheme using difficulty notions such as unforgeability

and difficulty proposed by Chen et al. [7] and prove that our puzzle is unforgeable and difficult. We refer [7] for a more formal treatment of these security difficulty notions.

Unforgeability. In our puzzle scheme, we use a pseudo-random function H_ρ in puzzle generation to generate Z. Thus showing that our puzzle meets the unforgeability notion is striaghforward following the same argument in [18] if H_ρ is a secure pseudo-random function. Hence we omit the unforgeability proof due to space constraints.

Difficulty. For proving the difficulty of our puzzle we again adapt the approach from [18] and show that our puzzle satisfies the difficulty notion of Chen et al. model as long as the KCPuz in Sect. 2 is difficult. In particular we relate the security of our puzzle to that of KCPuz with $R = 2^Q$ in Fig. 1. Note that our puzzle can be seen as the result of applying the precomputation approach in RSAPuz to KCPuz. The difficulty of our puzzle is proved in the following theorem.

Theorem 2 *Assume that k is a security parameter and Q is a difficulty parameter. If* KCPuz *with a modulus generation algorithm* GenMod *is $\epsilon_{k,Q}(t)$-difficult, then our puzzle, say puz, from Fig. 3 is $\epsilon'_{k,Q}(t)$-difficult for all probabilistic polynomial time $\mathcal{A}$ running in time at most t, where*

$$\epsilon'_{k,Q}(t) = 2 \cdot \epsilon_{k,Q}\left(t + (q_{\mathsf{C}} + 1)\,(2(\ell - 1)T_{\text{Mul}}) + c\right).$$

Here, q_{C} is the total number of CreatePuzSoln *queries issued in the experiment and T_{Mul} is the time complexity for computing a multiplication modulo n, and c is a constant.*

Proof We prove the theorem using the game hopping technique. Assume that $\mathcal{A}$ is a probabilistic algorithm running in time t and wins the puzzle difficulty experiment of puz. Using $\mathcal{A}$, we construct an algorithm $\mathcal{B}$ that solves KCPuz easily. Let the event E_i be such that $\mathcal{A}$ wins in game G_i.

Game G_0. Let G_0 be the original difficulty game $\mathsf{Exp}^{\mathsf{Diff}}_{\mathcal{A},puz}(k)$ defined as follows:

1. The challenger runs the Setup algorithm to generate $s \leftarrow (\rho, d, \phi(n), (\alpha_i, \beta_i)_{i=1}^N)$ and $params \leftarrow (Q, n)$. The challenger submits the parameters $params$ to $\mathcal{A}$ and keeps s.
2. Now, the challenger answers the CreatePuzSoln(N_C) query issued by $\mathcal{A}$ as follows:
   - The challenger runs the BPV pair generator BPVGen to obtain a pair (x, X) and computes Z, y and Y as per the protocol in Fig. 3.
   - The challenger submits $(puz, soln) \leftarrow ((N_S, Z, x, X), (y, Y))$ to $\mathcal{A}$.
3. At some time during the game, $\mathcal{A}$ is allowed to issue the Test(N^*_C) query to the challenger. The challenger answers the query with puz^* by generating a puzzle $puz^* = (N^*_S, Z^*, x^*, X^*)$ using GenPuz(s, Q, N_{C^*}) algorithm. The $\mathcal{A}$ may

continue to ask CreatePuzSoln(N_C) queries even after issuing the test query Test(N_C^*).
4. $\mathcal{A}$ outputs a valid solution $soln^* = (y^*, Y^*)$.
5. The challenger outputs 1 if VerSoln($puz^*, soln^*$) = true, otherwise the challenger outputs 0.

Then

$$\Pr\left(\mathsf{Exp}^{\mathsf{Diff}}_{\mathcal{A},puz}(k) = 1\right) = \Pr(\mathsf{E}_0)\ . \tag{1}$$

Game G_1. The difference between Game G_1 and Game G_0 is that the KCPuz challenger is used to answer the CreatePuzSoln queries issued by $\mathcal{A}$ and the KCPuz challenge is inserted in response to the Test query. Note that we assume that $R = 2^Q$ in KCPuz shown in Fig. 1 in order to be compatible with our puzzle scheme puz. The game is defined as folows:

1. The parameters $params \leftarrow (Q, n)$ are obtained from the KCPuz challenger.
2. Initiate the adversary $\mathcal{A}$ with $params$ as input.
   The adversary is allowed oracle access to CreatePuzSoln$(\cdot)$ and Test$(\cdot)$ oracles. That is, $\mathcal{B}$ interacts with KCPuz challenger and the adversary $\mathcal{A}$ individually. $\mathcal{B}$ acts as a puz challenger for $\mathcal{A}$. Whenever $\mathcal{A}$ issues CreatePuzSoln queries, $\mathcal{B}$ simply forwards the queries to KCPuz challenger and returns whatever it receives from KCPuz challenger with minor modifications to $\mathcal{A}$. We explain the interaction between $\mathcal{B}$ and KCPuz and between $\mathcal{B}$ and $\mathcal{A}$ in detail below:
   - CreatePuzSoln(str): Whenever $\mathcal{A}$ issues CreatePuzSoln(str) query, our challenger $\mathcal{B}$ forwards the same CreatePuzSoln query to the KCPuz challenger. The KCPuz challenger sends (puz $= (X, R = 2^Q, K, Z)$, soln $= (X^{2^Q}, X^K)$) to $\mathcal{B}$. Upon receiving a pair of the form (puz, soln) our challenger $\mathcal{B}$ acts as follows:
     – Assigns the puzzle values $x \leftarrow X$, $X_1 \leftarrow X^u$, for a fixed u of its choice and the solution values $y \leftarrow X^{2^Q}$ and $Y \leftarrow (X^K)^u$. Note that the value X received each time from KCPuz challenger is an output of the HMAC function, whereas in puz, (x, X) is an output of the BPV generator.
     – Return (puz, soln) $= ((x, X_1), (y, Y))$ to $\mathcal{A}$.
   - Test(str^*): When $\mathcal{A}$ issues a Test(str*) query, $\mathcal{B}$ simply passes the same query as its Test query to the KCPuz challenger that returns the challenge puzzle puz* $= (X^*, R^* = 2^Q, K^*, Z^*)$, where X^* is an output of HMAC. Then $\mathcal{B}$ sets $x^* \leftarrow X^*$, $X_1^* \leftarrow (X^*)^u$ and sends the target puzzle puz* $= (x^*, X_1^*, R^* = 2^Q, K^*, Z^*)$ to $\mathcal{A}$.
3. $\mathcal{A}$ may continue its CreatePuzSoln queries and $\mathcal{B}$ answers them as explained above.
4. When $\mathcal{A}$ outputs a potential solution soln* $= (y^* = (X^*)^{2^Q}, Y^* = ((X^*)^u)^K)$, $\mathcal{B}$ omits Y^*, computes $(X^*)^K$ and outputs its soln* as $(y^*, (X^*)^K)$.

We say that if the $\mathcal{A}$ wins game G_1, then the challenger $\mathcal{B}$ wins the puzzle difficulty experiment of KCPuz. Hence,

$$\Pr(\mathsf{E}_2) \leq \mathsf{Adv}^{\mathsf{Diff}}_{\mathcal{B},\mathsf{KCPuz},Q}(k) \leq \epsilon_{k,Q}(t) \ . \tag{2}$$

where $\mathcal{B}$ runs in time $t(B) = t(A) + (q_{\mathsf{C}} + 1)\left(T_{\mathrm{Exp}}\right)$ where q_{C} is the total number of CreatePuzSoln queries issued by $\mathcal{A}$ in G_0, and T_{Exp} is the total time needed to compute an exponentiation modulo n.

In the game G_0, a puzzle is of the form $(N_S, Z, x, X, R = 2^Q, K)$ where (x, X) is an output from the BPV generator BPVGen whereas in G_1, x is an output of HMAC run by the KCPuz challenger and $X = x^u$ is uniform at random.

Hence by Theorem 1, we get

$$|\Pr(\mathsf{E}_0) - \Pr(\mathsf{E}_1)| \leq 2^{-\frac{1}{2}\left(\log\binom{N}{\ell}+1\right)} \leq \epsilon_{k,Q}(t), \tag{3}$$

where the second inequality is due to the appropriate choices of N and ℓ.

Game G_2. The messages generated by the challenger in G_2 are identical to those in G_1 except for the following modification: The value X which is returned during the Test query: in G_1 it is a random integer from $[1, n]$ generated by the challenger whereas in G_2 it is the output of KCPuz challenger. This change is indistinguishable as we basically replace one random x with another. Hence

$$|\Pr(\mathsf{E}_1) - \Pr(\mathsf{E}_2)| = 0 \ . \tag{4}$$

Combining equations (1) through (3) yields the desired result. □

4 Conclusion

In this paper, we presented an efficient non-parallelizable puzzle based on modular exponentiation. Our puzzle can be viewed as a combination of two previously known puzzles, namely KCPuz and RSAPuz. However, our puzzle inherits all the advantages of these two puzzles and eludes their disadvantages. Our puzzle is the first modular exponentiation-based puzzle without computational burden, either offline (e.g., precomputation in RSAPuz) or online (e.g., solution verification in KCPuz), to support change of difficulty. Thus our puzzle supports scalability in an efficient manner, making it more practical in detering attacks like denial-of-service attacks.

References

1. Aura, T., Nikander, P.: Stateless connections. In: Han, Y., Okamoto, T., Qing, S. (eds.) ICICS 1997, vol. 1334 of LNCS, pp. 87–97. Springer (1997)
2. Aura, T., Nikander, P., Leiwo, J.: DoS-resistant authentication with client puzzles. In: Christianson, B., Crispo, B., Malcolm, J.A., Roe, M. (eds.) Security Protocols: 8th International Workshop, vol. 2133 of LNCS, pp. 170–177. Springer (2000)
3. Back, A.: Hashcash: a denial-of-service countermeasure. Available as http://www.hashcash.org/papers/hashcash.pdf (2002)
4. Boyko, V.: A pre-computation scheme for speeding up public-key cryptosystems. Master's thesis, Massachusetts Institute of Technology. Available as http://hdl.handle.net/1721.1/47493 (1998)
5. Boyko, V., Peinado, M., Venkatesan, R.: Speeding up discrete log and factoring based schemes via precomputations. In: Nyberg, K. (ed.) EUROCRYPT '98, vol. 1403 of LNCS, pp. 221–235. Springer (1998)
6. Canetti, R., Halevi, S., Steiner, M.: Hardness amplification of weakly verifiable puzzles. In: Kilian, J. (ed.) Theory of Cryptography Conference (TCC) 2005, vol. 3378 of LNCS, pp. 17–33. Springer (2005)
7. Chen, L., Morrissey, P., Smart, N.P., Warinschi, B.: Security notions and generic constructions for client puzzles. In: Matsui, M. (ed.) ASIACRYPT 2009, vol. 5912 of LNCS, pp. 505–523. Springer (2009)
8. Dwork, C., Naor, M.: Pricing via processing or combatting junk mail. In: Brickell, E.F. (ed.) CRYPTO '92, vol. 740 of LNCS, pp. 139–147. Springer (1992)
9. Feng, W., Kaiser, E., Luu, A.: Design and implementation of network puzzles. In: INFOCOM 2005, vol. 4, pp. 2372–2382. IEEE (2005)
10. Hofheinz, D., Unruh, D.: Comparing two notions of simulatability. In: Kilian, J. (ed.) TCC 2005, vol. 3378 of LNCS, pp. 86–103. Springer (2005)
11. Juels, A., Brainard, J.: Client puzzles: a cryptographic countermeasure against connection depletion attacks. In: NDSS 1999, pp. 151–165. Internet Society (1999)
12. Karame, G., Čapkun, S.: Low-cost client puzzles based on modular exponentiation. In: Gritzalis, D., Preneel, B., Theoharidou, M. (eds.) ESORICS 2010, vol. 6345 of LNCS, pp. 679–697. Springer (2010)
13. Kilian, J. (ed.) TCC 2005, vol. 3378 of LNCS. Springer (2005)
14. Lenstra, A., Verheul, E.: Selecting cryptographic key sizes. J. Cryptology **14**(4), 255–293 (2001)
15. Mao, W.: Timed-release cryptography. In: Vaudenay, S., Youssef, A. M. (eds.) SAC 2001, vol. 2259 of LNCS, pp. 342–358. Springer (2001)
16. Miller, G.L.: Riemann's hypothesis and tests for primality. In: STOC, 1975, pp. 234–239. ACM (1975)
17. Moore, D., Shannon, C., Brown, D.J., Voelker, G.M., Savage, S.: Inferring internet denial-of-service activity. ACM Trans. Comput. Syst. (TOCS) **24**(2), 115–139 (2006)
18. Rangasamy, J., Stebila, D., Kuppusamy, L., Boyd, C., González Nieto, J.M.: Efficient modular exponentiation-based puzzles for denial-of-service protection, The 14th International Conference on Information Security and Cryptology - ICISC 2011, pp. 319–331 (2011)
19. Rivest, R.L., Shamir, A., Wagner, D.A.: Time-lock puzzles and timed-release crypto. Technical Report TR-684, MIT Laboratory for Computer Science, March 1996
20. Shparlinski, I.: On the uniformity of distribution of the RSA pairs. Math. Comput. **70**(234), 801–808 (2001)
21. Stebila, D., Kuppusamy, L., Rangasamy, J., Boyd, C., González Nieto, J.M.: Stronger difficulty notions for client puzzles and Denial-of-Service-Resistant protocols. In: Kiayias, A. (ed.) Topics in Cryptology The Cryptographers' Track at the RSA Conference (CT-RSA) 2011, vol. 6558 of LNCS, pp. 284–301. Springer (2011)

22. Wang, X., Reiter, M.K.: Defending against denial-of-service attacks with puzzle auctions. In: Proceedings 2003 IEEE Symposium on Security and Privacy (SP'03), pp. 78–92. IEEE Press (2003)
23. Wang, Y., Wu, Q., Wong, D.S., Qin, B., Chow, S.M., Liu, Z., Tan, X.: Securely Outsourcing Exponentiations with Single Untrusted Program for Cloud Storage. In: Proceedings of Computer Secuirty - ESORCICS 2014, vol. 8712 of LNCS, pp. 326–343. Springer (2014)