Efficient Modular Exponentiation-based Puzzles for Denial-of-Service Protection

Jothi Rangasamy, Douglas Stebila, Lakshmi Kuppusamy, Colin Boyd, and Juan Gonzalez Nieto

Information Security Institute, Queensland University of Technology,
GPO Box 2434, Brisbane, Queensland 4001, Australia
{j.rangasamy,stebila,l.kuppusamy,c.boyd,j.gonzaleznieto}@qut.edu.au

Abstract. Client puzzles are moderately-hard cryptographic problems — neither easy nor impossible to solve — that can be used as a countermeasure against denial of service attacks on network protocols. Puzzles based on modular exponentiation are attractive as they provide important properties such as non-parallelisability, deterministic solving time, and linear granularity. We propose an efficient client puzzle based on modular exponentiation. Our puzzle requires only a few modular multiplications for puzzle generation and verification. For a server under denial of service attack, this is a significant improvement as the best known non-parallelisable puzzle proposed by Karame and Čapkun (ESORICS 2010) requires at least $2k$-bit modular exponentiation, where k is a security parameter. We show that our puzzle satisfies the unforgeability and difficulty properties defined by Chen *et al.* (Asiacrypt 2009). We present experimental results which show that, for 1024-bit moduli, our proposed puzzle can be up to 30× faster to verify than the Karame-Čapkun puzzle and 99× faster than the Rivest *et al.*'s time-lock puzzle.

Keywords: client puzzles, time-lock puzzles, denial of service resistance, RSA, puzzle difficulty

1 Introduction

Denial-of-Service (DoS) attacks are a growing concern due to the advancement in information technology and its application to electronic commerce. The main goal of DoS attacks is to make a service offered by a service provider unavailable by exhausting the service provider resources. In recent years, DoS attacks disabled several Internet e-commerce sites including eBay, Yahoo!, Amazon and Microsoft's name server [17].

Since millions of computers are connected through the Internet, DoS attacks on any of these systems would lead to a large scale impact on the whole network. Many essential services such as communications, defense, health systems, banking and financial systems have become Internet-based applications. There is an immense need for keeping these services alive and available on request. However mounting a DoS attack is very easy for the sophisticated attackers while defending them is very hard for the victim servers. A promising way to deal with this

problem is for a defending server to identify and segregate the malicious requests as early as possible.

CLIENT PUZZLES, also known as *proofs of work*, can guard against resource exhaustion attacks such as DoS attacks and spam [2,8,11]. When client puzzles are employed, a defending server will process a client's request only after the client has provided the correct solution to its puzzle challenge. In this way, a client can prove to the server its legitimate intentions in getting a connection. Although employing client puzzles adds an additional cost for legitimate clients, a big cost will be imposed on an attacker who is trying to make multiple connections. In this case, the attacker would need to invest its own resources in solving a large number of puzzles before exhausting the server resources.

1.1 Puzzle Properties

The essential property of a client puzzle is that it be *difficult* to solve: not impossible, but not too easy, either. Many cryptographic puzzles are based on inverting a hash function [3,9,11].

Puzzles based on modular exponentiation have the potential to provide additional properties:

- NON-PARALLELISABILITY. A client puzzle is called non-parellelisable if the time to solve the puzzle cannot be reduced by using many computers in parallel. This property ensures that a DoS attacker cannot divide a puzzle into multiple small tasks and therefore gains no advantage in puzzle solving with the large number of machines it may have in its control.
- DETERMINISTIC SOLVING-TIME. If the puzzle issuing server has specified a value as a difficulty parameter, then a client needs to do at least the specified number of operations to solve a puzzle. This will help the server decide the minimum work each client must do before getting a connection. Many puzzles in contrast, only determine the average work required to obtain a solution.
- FINE GRANULARITY. A puzzle construction achieves finer granularity if the server is able to set the difficulty level accurately. That is, the gap between two adjacent difficulty levels should be small. This property helps servers switch between different difficulty levels easily. If there is a large gap between two difficulty levels, then increasing the difficulty to the next level might have an impact on computationally-poor legitimate clients.

It is imperative that both the puzzle generation and solution verification algorithms add only minimal computation and memory overhead to the server. Otherwise, this puzzle mechanism itself may become a target for resource exhaustion DoS attacks when a malicious client sends a large number of fake requests for puzzle generation or a large number of puzzle solutions for verification.

Rivest *et al.* [18] described a concrete modular exponentiation puzzle based on the RSA modulus factorisation problem. This was the first puzzle to provide the three properties listed above: non-parallelisability, deterministic solving time, and finer granularity. However, the main disadvantage of these puzzles is that

Puzzle	Verification Cost	Verification Time (μs) 512-bit n	1024-bit n
Rivest *et al.* [18]	$\lvert n\rvert$-bit mod. exp.	474.68	2903.99
Karame-Čapkun [12]	$2k$-bit mod. exp.	263.35	895.17
This paper	**3 mod. mul.**	**14.75**	**29.24**

Table 1. Verification costs and timings (in microseconds) for modular exponentiation-based puzzles; n is an RSA modulus, k is a security parameter. Timings are for 1024-bit modulus n with $k = 80$ and for 512-bit modulus n with $k = 56$, both with puzzle difficulty 1 million.

they require a busy server to perform computationally intensive exponentiation to verify solutions. Recently, Karame and Čapkun [12] improved the verification efficiency of Rivest *et al.*'s puzzle by a factor of $\frac{|n|}{2k}$ for a given RSA modulus n, where k is the security parameter. More details on these two puzzles will be provided in Section 2.

Although Karame and Čapkun's performance gain in verification cost is impressive, it is still sufficiently expensive that it could be burdensome on a defending server, as verification still requires modular exponentiations. This is the main reason preventing modular exponentiation-based puzzles being deployed widely, despite having some attractive characteristics. Construction of modular exponentiation-based puzzles which avoid a big modular exponentiation for puzzle generation and solution verification has not been attained until now.

1.2 Contributions

1. We propose an efficient modular exponentiation-based puzzle which achieves non-parallelisability, deterministic solving time, and finer granularity. Our puzzle can be seen as an efficient alternative to Rivest *et al.*'s time-lock puzzle [18]. Our puzzle requires only a few modular multiplications to generate and verify puzzle solutions. The verification costs and timings for our puzzle and other puzzles of the same type are presented in Table 1.
2. We analyse the security properties of our puzzle in the puzzle security model of Chen *et al.* [7] and show that our puzzle is unforgeable and difficult.
3. In order to validate the performance of our puzzle, we give experimental results and compare them with the performances of Rivest *et al.*'s time-lock puzzle [18] and Karame and Čapkun's puzzle [12], which is the most efficient non-parallelisable puzzle in the literature. Our results suggest that our puzzle reduces the solution verification time by approximately 99 times when compared to Rivest *et al.*'s time-lock puzzle and 30 times when compared to Karame and Čapkun, for 1024-bit moduli.

Organization of paper: The rest of the paper is organized as follows. Section 2 presents the background and motivation for our work. Section 3 describes our proposed puzzle and Section 4 analyses the security properties of the proposed puzzle in the Chen *et al.* model. Section 5 presents our experimental results

validating the efficiency of the proposed puzzle scheme and we conclude the paper in Section 6.

2 Background: Modular Exponentiation-Based Puzzles

In this section, we review known modular exponentiation-based puzzles and follow the definition of a client puzzle proposed by Chen *et al.* [7].

Notation. If n is an integer, then we use $|n|$ to denote the length in bits of n, and $\phi(n)$ is the Euler phi function of n, which is equivalent to the size of the multiplicative group $\mathbb{Z}_n^*$. We denote the set $\{a, \ldots, b\}$ of integers by $[a, b]$. We use $x \leftarrow_r S$ to denote choosing x uniformly at random from S. If A is an algorithm, then $x \leftarrow A(y)$ denotes assigning to x the output of A when run with the input y. If k is a security parameter, then $\mathrm{negl}(k)$ denotes a function that is negligible in k (asymptotically smaller than the inverse of any polynomial in k). By p.p.t. algorithm, we mean probabilistic polynomial time algorithm.

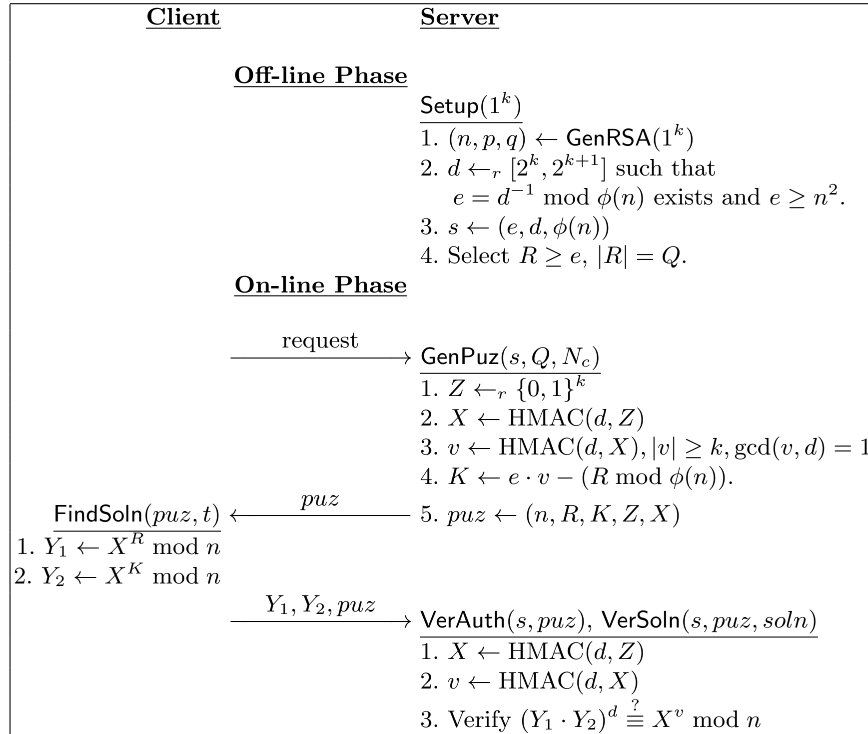

Fig. 1. KCPuz: Karame-Čapkun's Variable-Exponent Client Puzzle [12]

2.1 Rivest *et al.*'s puzzle

Given a RSA modulus n, Rivest *et al.*'s puzzle [18] requires $|n|$-bit exponentiation to verify puzzle solutions. In detail, the server generates two RSA primes p and q, and computes the associated RSA modulus $n = pq$ and the Euler totient function $\phi(n) = (p-1) \cdot (q-1)$. Then sets the difficulty level Q or the amount of work a client needs to do. Now, the server picks an integer $a \leftarrow_r \mathbb{Z}_n^*$ and sends the client the tuple (a, Q, n). The client's task is to compute and return $b \leftarrow a^{2^Q} \bmod n$. The server first computes $c = 2^Q \bmod \phi(n)$ and then checks if $a^c \stackrel{?}{\equiv} b \bmod n$. Here the server can compute c once and use it for all the solution verifications unless it changes the difficulty level Q. With the trapdoor information $\phi(n)$, the server is able to verify the solution in one $|n|$-bit exponentiation whereas the client should perform Q repeated squarings and typically $Q \gg |n|$.

2.2 Karame-Čapkun puzzle

Recently, Karame and Čapkun [12] reduced $|n|$-bit exponentiation in time-lock puzzle verification to $2k$-bit exponentiation modulo n, thereby significantly reducing the computational burden of the server by a factor of $\frac{|n|}{2k}$, where k is a security parameter. The Karame and Čapkun puzzle KCPuz is depicted in Figure 1. Karame and Čapkun showed that their puzzle is unforgeable and difficult in the puzzle security model of Chen *et al.* [7]. In this paper, we are considering the variable-exponent version of Karame and Čapkun's puzzle.

Although the verification cost is improved significantly in KCPuz, the server still needs to engage in at least $2k$-bit exponentiation for each puzzle solution it receives. Since it is expected that the defending server may receive a large number of fake requests/solutions, puzzle generation and solution verification should be as efficient as possible. Otherwise this mitigation mechanism itself opens door for resource exhaustion DoS attacks when a malicious client sends a number of fake requests/solutions for puzzles triggering the server engage in those expensive operations.

Parallelisability. Puzzle solving in KCPuz can be partially parallelised by decomposing the exponent R into multiple parts. For example, consider a malicious client C with two compromised machines, namely M_1 and M_2, under its control. In order to parallelise the computation of $x^R \bmod n$, C first decomposes R into two parts R_1 and R_2 such that $R = R_1 || R_2$, where $||$ denotes the concatenation. Then C gives $R_1 || 0^{\frac{\ell}{2}}$ to M_1 and R_2 to M_2, along with the public values (X, n). Now, using the square and multiply algorithm, M_1 computes $X^{R_1 || 0^{\frac{\ell}{2}}} \bmod n$ in $\frac{5\ell}{4}$ modular multiplications and M_2 computes $X^{R_2} \bmod n$ in $\frac{3\ell}{4}$ modular multiplications. Note that, without decomposition, $\frac{3\ell}{2}$ modular multiplications would have been required if the malicious client chose to compute $X^R \bmod n$ itself. Since M_1 and M_2 could work in parallel, the time taken by C to compute $X^R \bmod n$ is the time taken by M_1 to compute $X^{R_1 || 0^{\frac{\ell}{2}}} \bmod n$,

which requires to do more operations than M_2. Therefore, this decomposition saves the malicious client $\frac{1}{6}$ of the total time needed to solve the puzzle. This parallelisation via exponent decomposition is gainful only if R is not a power of 2. Rivest *et al.* set R to be power of 2 to achieve non-parallelisability. Moreover when $|R| \gg 2^{20}$ bits and $R = 2^Q$ for some $Q \in \mathbb{N}^+$, sending Q for each puzzle instead of R will save communication cost as well.

Granularity. Unlike Rivest *et al.*'s modular exponentiation-based puzzle, the Karame-Čapkun puzzle does not provide fine control over granularity of difficulty levels. In KCPuz, a client is given the pair (K, R) where $K \leftarrow e \cdot v - (R \bmod \phi(n))$ and therefore for security reasons, R must be large enough so that $R > n$. This condition rules out difficulty levels between 0 and n. Also, if R is the current difficulty level, then the next difficulty level R' must satisfy the following: $\frac{R'}{R} \geq n^2$. This implies that there will be a large gap between the two successive difficulty levels. Hence, KCPuz does not support fine granularity.

Example parameter sizes. In a DoS scenario, a client is given a puzzle whose hardness is typically set between 0 to 2^{25} operations. Since a client needs to perform at most 2^{25} for each puzzle, the 40-bit security level is enough for the puzzle scheme and is higher than the work needed to solve a puzzle. Lenstra and Verheul [14] suggest using a 512-bit RSA modulus n which is widely believed to match the 56-bit security of Data Encryption Standard (DES). Since $|n| = 512$, $|R| \geq 512$. Suppose $R = 2^{512}$. From $\frac{R'}{R} \geq n^2$, the possible values for the next two difficulty levels are $R' = 2^{1536}$ and $R'' = 2^{2560}$.

In this work, we give an efficient modular exponentiation-based puzzle which achieves both non-parallelism and finer granularity.

3 Our Client Puzzle Protocol

In this section, we present a non-parallelisable client puzzle scheme that requires *only* a few modular multiplications for puzzle generation and solution verification. First we review the cryptographic ingredients required and then present our puzzle construction.

3.1 Tools

Our puzzle construction makes use of algorithm GenRSA that generates an RSA-style modulus $n = pq$ as follows:

Definition 1 (Modulus Generation Algorithm). *Let k be a security parameter. A* modulus generation algorithm *is a probabilistic polynomial time algorithm* GenRSA *that, on input 1^k, outputs (n, p, q) such that $n = pq$ and p and q are k-bit primes.*

In our puzzle generation algorithm, the server needs to produce a pair (x, x^u) for each puzzle. Since the generation of these pairs are expensive, we utilise a technique due to Boyko *et al.* [5] for efficient generation of many pairs $(x_i, x_i^u \bmod n)$ for a fixed u using a relatively small amount of pre-computation.

Definition 2 (BPV Generator). *Let k, ℓ, and N, with $N \geq \ell \geq 1$, be parameters. Let $n \leftarrow \mathsf{GenRSA}(1^k)$ be an RSA modulus. Let u be an element in $\mathbb{Z}_{\phi(n)}$ of length m. A* BPV generator *consists of the following two algorithms:*

- $\mathsf{BPVPre}(u, n, N)$*: This is a pre-processing algorithm that is run once. The algorithm 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. Finally, it returns a table $\tau \leftarrow ((\alpha_i, \beta_i))_{i=1}^N$.*
- $\mathsf{BPVGen}(n, \ell, \tau)$*: This is run whenever a pair $(x, x^u \bmod n)$ is needed. Choose a random set $S \subseteq_r \{1, \ldots, N\}$ of size ℓ. Compute $x \leftarrow \prod_{j \in S} \alpha_j \bmod n$. If $x = 0$, then stop and generate S again. Otherwise, compute $X \leftarrow \prod_{j \in S} \beta_j \bmod n$ and return (x, X). In particular, the indices S and the corresponding pairs $((\alpha_j, \beta_j))_{j \in S}$ are not revealed.*

Indistinguishability of the BPV Generator. Boyko and Goldwasser [4] and Shparlinski [19] showed that the values x_i generated by the BPV generator are statistically close to the uniform distribution. To analyse the security properties of the proposed puzzle, we use the following results by Boyko and Goldwasser [4, Chapter 2]. Let N be the number of pre-computed pairs (α_i, β_i) such that α_i's are chosen independently and uniformly from $[1, n]$ and $\beta_i = \alpha_i^u \bmod n$. Each time a random set $S \subseteq \{1, \ldots N\}$ of ℓ elements is chosen and a new pair (x, X) is computed such that $x = \prod_{j \in S} \alpha_j \bmod n$ and $X = \prod_{j \in S} \beta_j \bmod n$. Then, 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 $x' \in \mathbb{Z}_n^*$. Here we also note that although BPV outputs a pair (x, x^u), only x is made available to clients and x^u is kept secret by the server. That is, each time clients are given the pair $(x, 1)$, not (x, x^u).

Theorem 1. *[4, Chapter 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,*

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

Parameters for BPV. As discussed in Section 2, in a DoS scenario, the difficulty level Q for a puzzle is typically set between 0 and 2^{25} operations. Therefore, it can be anticipated that factoring of n and hence computing $\phi(n)$ for solving puzzles easily, will be much more difficult than performing Q squarings

as $Q \ll n$ when $Q \leq 2^{25}$ and $|n| \geq 512$. Lenstra and Verheul [14] suggest using a 512-bit RSA modulus n to match the 56-bit security of Data Encryption Standard (DES). Since a client needs to perform at most 2^{25} for each puzzle, the 40-bit security level could be enough for the puzzle scheme and hence breaking the scheme is much harder than solving a puzzle.

Boyko *et al.*[4,5] suggest to set N and ℓ so that subset product problem is intractable and birthday attacks becomes infeasible. To achieve the above security level, we can select N and ℓ such that $\binom{N}{\ell} > 2^{40}$. Boyko *et al.* [4,5] suggest setting $N = 512$ and $\ell = 6$ for the BPV generator. Alternatively, we could achieve this with $N = 2500$ and $\ell = 4$; this increases the amount of precomputation required in BPVPre but reduces the number of modular multiplications performed online in BPVGen from 12 to 8.

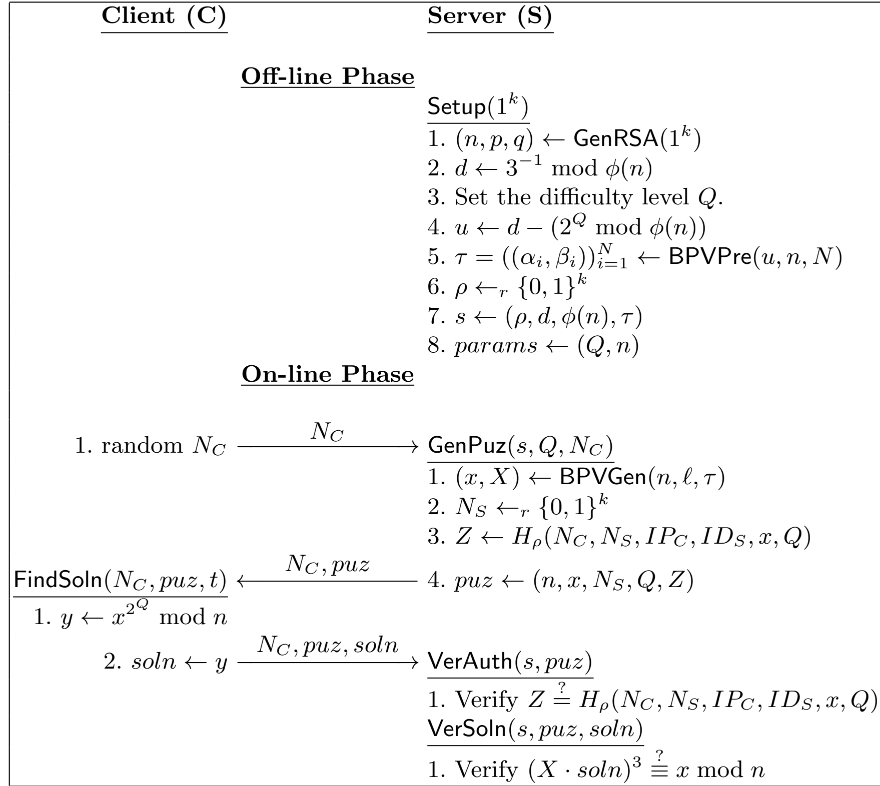

Fig. 2. RSAPuz: A new client puzzle based on modular exponentiation

3.2 The Proposed Puzzle: RSAPuz

The main idea behind our puzzle construction is: given a RSA modulus n, an integer Q and $X \in \mathbb{Z}_n^*$, the task of a client is to compute $X^{2^Q} \bmod n$.

Our client puzzle RSAPuz is presented in Fig 2 as an interaction between a server issuing puzzles and a client solving them. RSAPuz is parameterized by a security parameter k and a difficulty parameter Q. In practice, a server using puzzles as a DoS countermeasure can vary Q based on the severity of the attack it is experiencing. However once a difficulty level is set, it is increased only if the server still receives a large number of requests with correct puzzle solutions.

In RSAPuz, the server does the following:

- Puzzle pre-computation. Generating (n, p, q) and computing d is a one time process. Whenever a server is required to change the difficulty parameter Q, it selects an integer R such that $|R| = Q$ and computes u. Then it runs the BPV pre-processing step with inputs (u, n, N) and obtains N pairs of (α_i, β_i). Since all the required pre-computations are done off-line, the defending server can be more effective on-line against DoS attacks.

In a DoS setting, an attacker could mount a resource depletion attack by asking the server to generate many puzzles and to verify many fake puzzle solutions. Hence the following algorithms run online by the server many times should be very efficient to resist such flooding attacks.

- Puzzle generation. The dominant cost in puzzle generation is the BPV pair generation BPVGen, which requires $2(\ell - 1)$ modular multiplications: $\ell - 1$ to compute x and $\ell - 1$ to compute X. There is also a single call to the pseudo-random function H_ρ to compute the authentication tag Z. As suggested by Boyko *et al.*, ℓ could be set between 4 and 16 so that our puzzle requires only 8 modular multiplications in the best case.
- Puzzle authenticity verification. Puzzle authenticity verification is quite cheap, requiring just a single call to the pseudo-random function H to verify the authentication tag Z.
- Puzzle verification. To verify correctness of a solution, the server has to perform only 3 modular multiplications.

Our puzzle construction dramatically reduces the puzzle verification cost incurred by the server and is the only modular exponentiation-based puzzle that does not require a big exponentiation to be performed by the server on-line. The efficiency of our puzzle is compared with the efficiency of Karame-Čapkun and Rivest *et al.*'s puzzles in Section 5.

After receiving the puzzle, the client finds the solution to the puzzle as follows:

- Puzzle solving. One typical method for a legitimate client to implement the FindSoln algorithm is to use square-and-multiply algorithm, which is the most commonly used algorithm for computing modular exponentiations. Upon receiving a puzzle *puz* from the server with an integer Q, the client computes y as $x^{2^Q} \bmod n$. We note however that a client could also choose to factor n first and then can solve the puzzle efficiently.

Puzzle	Non-parallelisable & Deterministic Solving Time	Finer Granularity	Puzzle Solving Cost	Generation Cost	Verification Cost
TLPuz [18]	Yes	Yes	Q mod. mul.	1 hash	1 hash $\lvert n\rvert$-bit mod. exp.
KCPuz [12]	Yes	No	$O(Q)$ mod. mul. $O(\lvert n^2\rvert)$ mod. mul.	2 HMAC (4 hash) 1 gcd	2 HMAC (4 hash) $2k$-bit mod. exp.
RSAPuz	Yes	Yes	Q mod. mul.	1 hash $2(\ell - 1)$ mod. mul.	1 hash 3 mod. mul.

Table 2. Puzzle properties and operation counts for puzzle solving, generation and verification. Q is the difficulty level, n is an RSA modulus, k is a security parameter, and ℓ is a small integer.

The best known method to solve our puzzle is to perform Q repeated squarings and this is an inherently sequential process [10,12,18]. Therefore a client needs to do *exactly* Q sequential modular multiplications to correctly solve the given puzzle, thereby achieving deterministic solving time property and non-parallelisability. We also get finer granularity as Q can be set to any positive integer regardless of the previously used difficulty values.

In Table 2, we compare the puzzle properties and asymptotic costs for FindSoln, GenPuz and VerSoln algorithms for the non-parallelisable puzzles examined in this paper. In particular, we compare the performance of the proposed puzzle RSAPuz with that of Rivest *et al.*'s time-lock puzzle (TLPuz) and Karame-Čapkun's variable exponent puzzle (KCPuz).

Remark 1. In RSAPuz as illustrated in Figure 2, the server requires a short-term secret X for verifying the puzzle solution. Storing X for each puzzle may introduce a memory-based DoS attack on the server. Fortunately, the server may avoid this type of attack by employing *stateless connections* [1] to offload storage of X to the client. That is, the server can use a long-term symmetric key s_k to encrypt X and send it along with each puzzle. Then the client has to send back this encrypted value while returning the solution to the puzzle. In this way, the server remains stateless and obtains X by decrypting the encrypted value using the same key s_k. With an efficient symmetric encryption algorithm, the server will not experience any significant computational burden.

4 Security Analysis of RSAPuz

We analyse the security properties of RSAPuz using the security model of Chen *et al.* [7] which appears in Appendix A. In particular, we show that RSAPuz satisfies the unforgeability and difficulty properties introduced by Chen *et al.* Since we use a secure pseudo-random function H in puzzle generation, proof of unforgeability for RSAPuz is quite straightforward and is given in Appendix B.1.

Puzzle	512-bit modulus, $k = 56$ Setup (ms)	GenPuz (μs)	FindSoln (s)	VerSoln (μs)	1024-bit modulus, $k = 80$ Setup (ms)	GenPuz (μs)	FindSoln (s)	VerSoln (μs)
Difficulty: 1 million								
TLPuz [18]	13.92	4.80	1.54	474.68	56.10	4.86	4.13	2903.99
KCPuz [12]	11.52	8.37	1.59	263.35	42.30	8.66	4.27	895.17
RSAPuz	140.11	16.66	1.54	14.75	8510.92	35.15	4.29	29.24
Difficulty: 10 million								
TLPuz [18]	49.99	4.80	15.17	474.83	103.95	4.87	42.62	2917.25
KCPuz [12]	28.95	8.37	15.18	265.28	85.09	8.62	43.31	907.03
RSAPuz	1419.78	16.66	15.34	14.53	8669.75	34.72	43.08	28.97
Difficulty: 100 million								
TLPuz [18]	416.29	4.81	157.10	470.61	607.87	4.84	429.31	2924.01
KCPuz [12]	218.76	8.35	160.97	259.39	327.46	8.70	426.04	899.00
RSAPuz	1609.83	16.76	158.22	14.88	8966.74	34.76	422.58	29.18

Table 3. Timings for modular exponentiation-based puzzles. For RSAPuz, $N = 2500$ and $\ell = 4$.

4.1 Difficulty of RSAPuz

The time-lock puzzle was first proposed in 1996 and to date the best known method of solving the puzzle is sequential modular squaring, provided that factoring the modulus is more expensive. Indeed, it has been widely accepted that given a large RSA modulus n, the computation of $a^{2^Q} \bmod n$ can be obtained by Q repeated squarings and no algorithm with better complexity than Q squarings is known [10,12,15,18].

Karame and Čapkun proved that their puzzle KCPuz is $\epsilon_{k,R}(t)$- difficult in the Chen *et al.* model, where

$$\epsilon_{k,R}(t) = \min\left\{ \left\lfloor \frac{t}{\log R} \right\rfloor + O\left(\frac{1}{2^k}\right), 1 \right\}$$

for all probabilistic algorithms $\mathcal{A}$ running in time at most t.

If a solver knows a multiple of $\phi(n)$, then it can compute $\phi(n)$ and the factors of n very efficiently [16]. Then the solver can efficiently compute $x^R \bmod n$ by computing $c \leftarrow R \bmod \phi(n)$ first and then computing $x^c \bmod n$. However, it is computationally infeasible for a client to compute a multiple of $\phi(n)$ from the transcripts of the puzzle scheme, so computing $x^R \bmod n$ requires at least $O(\log R)$ modular multiplications. Hence, the success probability of solving the puzzle is bounded by $\epsilon_{k,R}(t)$ for any algorithm running in time at most t.

Detailed examination of Karame and Čapkun's proof reveals that they are essential making the assumption that the best approach for solving the time-lock puzzle is sequential modular squaring and multiplication. Moreover, their proof further makes the assumption that the time-lock puzzle is difficult in the Chen *et al.* model, in other words, when the adversary is allowed to see valid puzzle-solution pairs returned from the CreatePuzSoln query.

We show in the following theorem that our puzzle RSAPuz is difficult in the Chen *et al.* model [7] as long as Rivest *et al.*'s time-lock puzzle is difficult. Due to lack of space the proof of the theorem appears in Appendix B.2.

Theorem 2 (Difficulty of RSAPuz). *Let k be a security parameter and let Q be a difficulty parameter. Let* GenRSA *be a modulus generation algorithm. If* TLPuz *with* GenRSA *is $\epsilon_{k,Q}(t)$-difficult, then* RSAPuz *is $\epsilon'_{k,Q}(t)$-difficult for all probabilistic algorithms $\mathcal{A}$ running in time at most t, where*

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

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

5 Performance Comparison

Table 3 presents timings from an implementation of these puzzle variants for both 512-bit and 1024-bit RSA moduli with $k = 56$ and $k = 80$, respectively, for puzzle difficulty levels 1 million, 10 million, and 100 million. The software was implemented using big integer arithmetic from OpenSSL 0.9.8ℓ and run on a single core of a 3.06 GHz Intel Core i3 with 4GB RAM, compiled using `gcc -O2` with architecture x86_64.

In the 512-bit case, our puzzle reduces the solution verification time by approximately 32 times when compared to TLPuz and 17 times when compared to KCPuz. For the 1024-bit case, the gain in the verification time is approximately 99 times when compared to TLPuz and 30 times when compared to KCPuz.

Since VerSoln cost in RSAPuz is independent of k the security parameter, the verification gain increases as the size of RSA moduli increases. Note that, for both the moduli, the puzzle generation algorithm GenPuz is 2 to 7 times slower than the GenPuz in TLPuz and KCPuz. However, the cumulative puzzle generation and puzzle verification time in RSAPuz is still substantially less than in TLPuz or KCPuz. Furthermore, GenPuz cost in RSAPuz can still be improved by reducing ℓ from 4 to 2 and increasing the number N of pairs precomputed by BPVPre in the puzzle setup algorithm.

6 Conclusion

In this paper, we presented the most efficient non-parallelisable puzzle based on RSA. A DoS defending server needs to perform only $2(\ell - 1)$ modular multiplications online, where ℓ could be as low as 2, for a given RSA modulus. For the comparable difficulty level, the best known non-parallelisable puzzle requires a busy server perform online at least $2k$-bit modular exponentiation, where k is a security parameter.

We have also proved that the proposed puzzle satisfies the two security notions proposed by Chen *et al.* In particular, we have reduced the difficulty of solving our puzzle to the difficulty of solving Rivest *et al.*'s time-lock puzzle.

Experimental results show that our puzzle reduces the solution verification time by a factor of 99 when compared to Rivest *et al.*'s time-lock puzzle and a factor of 30 when compared to Karame and Čapkun puzzle, for 1024-bit moduli.

Acknowledgements. The authors are grateful to anonymous referees for their comments. This work is supported by Australia-India Strategic Research Fund project TA020002.

References

1. T. Aura and P. Nikander. Stateless connections. In Y. Han, T. Okamoto, and S. Qing, editors, *Proc. 1st International Conference on Information and Communications Security (ICICS) 1997*, volume 1334 of *LNCS*, pages 87–97. Springer, 1997.
2. T. Aura, P. Nikander, and J. Leiwo. DoS-resistant authentication with client puzzles. In B. Christianson, B. Crispo, J. A. Malcolm, and M. Roe, editors, *Security Protocols: 8th International Workshop*, volume 2133 of *LNCS*, pages 170–177. Springer, 2000.
3. A. Back. Hashcash: A denial-of-service countermeasure. 2002. Available as `http://www.hashcash.org/papers/hashcash.pdf`.
4. V. Boyko. A pre-computation scheme for speeding up public-key cryptosystems. Master's thesis, Massachusetts Institute of Technology, 1998. Available as `http://hdl.handle.net/1721.1/47493`.
5. V. Boyko, M. Peinado, and R. Venkatesan. Speeding up discrete log and factoring based schemes via precomputations. In K. Nyberg, editor, *Advances in Cryptology – Proc. EUROCRYPT '98*, volume 1403 of *LNCS*, pages 221–235. Springer, 1998.
6. R. Canetti, S. Halevi, and M. Steiner. Hardness amplification of weakly verifiable puzzles. In Kilian [13], pages 17–33.
7. L. Chen, P. Morrissey, N. P. Smart, and B. Warinschi. Security notions and generic constructions for client puzzles. In M. Matsui, editor, *Advances in Cryptology – Proc. ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 505–523. Springer, 2009.
8. C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In E. F. Brickell, editor, *Advances in Cryptology – Proc. CRYPTO '92*, volume 740 of *LNCS*, pages 139–147. Springer, 1992.
9. W. Feng, E. Kaiser, and A. Luu. Design and implementation of network puzzles. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 4, pages 2372–2382. IEEE, 2005.
10. D. Hofheinz and D. Unruh. Comparing two notions of simulatability. In Kilian [13], pages 86–103.
11. A. Juels and J. Brainard. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In *Proc. Network and Distributed System Security Symposium (NDSS) 1999*, pages 151–165. Internet Society, 1999.
12. G. Karame and S. Čapkun. Low-cost client puzzles based on modular exponentiation. In D. Gritzalis, B. Preneel, and M. Theoharidou, editors, *Proc. ESORICS 2010*, volume 6345 of *LNCS*, pages 679–697. Springer, 2010.
13. J. Kilian, editor. *Theory of Cryptography Conference (TCC) 2005*, volume 3378 of *LNCS*. Springer, 2005.
14. A. Lenstra and E. Verheul. Selecting cryptographic key sizes. *J. Cryptology*, 14(4):255–293, 2001.
15. W. Mao. Timed-release cryptography. In S. Vaudenay and A. M. Youssef, editors, *Proc. Selected Areas in Cryptography (SAC) 2001*, volume 2259 of *LNCS*, pages 342–358. Springer, 2001.

16. G. L. Miller. Riemann's hypothesis and tests for primality. In *Conference Record of Seventh Annual ACM Symposium on Theory of Computation (STOC), 5-7 May 1975, Albuquerque, New Mexico, USA*, pages 234–239. ACM, 1975.
17. D. Moore, C. Shannon, D. J. Brown, G. M. Voelker, and S. Savage. Inferring internet denial-of-service activity. *ACM Transactions on Computer Systems (TOCS)*, 24(2):115–139, 2006.
18. R. L. Rivest, A. Shamir, and D. A. Wagner. Time-lock puzzles and timed-release crypto. Technical Report TR-684, MIT Laboratory for Computer Science, March 1996.
19. I. Shparlinski. On the uniformity of distribution of the RSA pairs. *Mathematics of Computation*, 70(234):801–808, 2001.
20. D. Stebila, L. Kuppusamy, J. Rangasamy, C. Boyd, and J. M. González Nieto. Stronger difficulty notions for client puzzles and Denial-of-Service-Resistant protocols. In A. Kiayias, editor, *Topics in Cryptology – The Cryptographers' Track at the RSA Conference (CT-RSA) 2011*, volume 6558 of *LNCS*, pages 284–301. Springer, 2011.
21. D. Stebila and B. Ustaoglu. Towards denial of service resilient key agreement protocols. In C. Boyd and J. M. González Nieto, editors, *Proc. 14th Australasian Conference on Information Security and Privacy (ACISP) 2009*, volume 5594 of *LNCS*, pages 389–406. Springer, 2009.

A Security Notions for Client Puzzles

Several computational security models for the difficulty of client puzzles have been proposed [6,7,20,21]. In this section, we briefly describe the definition and security notions for a client puzzle proposed by Chen *et al.* [7].

Definition 3. *A client puzzle Puz is a tuple consisting of the following algorithms:*

- Setup *is a p.p.t. setup algorithm that accepts the security parameter* k *as input and returns output as follows:*
  - *Selects the key space* S*, the hardness space* T *, the string space* X*, the puzzle instance space* I *and puzzle solution space* P*.*
  - *Selects a secret* $s \leftarrow_r S$*.*
  - *Selects the puzzle parameters* $\mathsf{params} \leftarrow (S, T, X, I, P)$ *required for the client puzzle.*
  - *Returns* (params, s)
- GenPuz *is a p.p.t. puzzle generation algorithm that returns a puzzle instance* $\mathsf{puz} \in I$ *on input* $s \in S, Q \in T$ *and* $a \in X$*.*
- $\mathsf{FindSoln}$ *is a probabilistic solution finding algorithm that on input* $\mathsf{puz} \in I$ *returns a potential solution* $\mathsf{soln} \in P$ *after at most* t *clock cycles of execution.*
- $\mathsf{VerAuth}$ *is a puzzle authenticity verification algorithm that accept inputs* $s \in S$ *and* $\mathsf{puz} \in I$ *and returns* `true` *or* `false`*.*
- $\mathsf{VerSoln}$ *is a deterministic solution verification algorithm that accept inputs* $s \in S$*,* $\mathsf{puz} \in I$ *and a solution* $\mathsf{soln} \in P$ *and returns* `true` *or* `false`*.*

In the following unforgeability experiment, the adversary is allowed to query the $\mathsf{CreatePuz}$ oracle by choosing a str and a puzzle difficulty level Q on its own. This is to ensure that the adversary cannot create a valid looking puzzle even after seeing puzzles with different difficulty levels, .

Definition 4 (Puzzle Unforgeability [7]). *Let* Puz *be a client puzzle,* k *be a security parameter and* $\mathcal{A}$ *be a probabilistic algorithm. The unforgeability experiment* $\mathsf{Exp}^{\mathsf{UF}}_{\mathcal{A},\mathsf{Puz}}(k)$ *is defined as follows:*

1. *Run the set up algorithm* $\mathsf{Setup}(1^k)$ *to obtain* s *and* $params$*.*
2. *Run the adversary* $\mathcal{A}$*with input* $params$*. The adversary has oracle access to* $\mathsf{CreatePuz}(\cdot)$ *and* $\mathsf{CheckPuz}(\cdot)$*, which are answered as follows:*
   - $\mathsf{CreatePuz}(str, Q)$*: Generate a puzzle* puz *by running the* $\mathsf{GenPuz}(s, Q, str)$ *algorithm. Return* puz *to* $\mathcal{A}$*.*
   - $\mathsf{CheckPuz}(puz)$*: If any of the previously made* $\mathsf{CreatePuz}(str)$ *query did not output* puz *and* $\mathsf{VerAuth}(s, puz) = \mathsf{true}$ *then stop the experiment and output* 1*. Otherwise, return* 0 *to* $\mathcal{A}$*.*

We say that $\mathcal{A}$ *wins the game if* $\mathsf{Exp}^{\mathsf{UF}}_{\mathcal{A},\mathsf{Puz}}(k) = 1$ *and loses otherwise. The advantage of* $\mathcal{A}$ *is defined as:*

$$\mathsf{Adv}^{\mathsf{UF}}_{\mathcal{A},\mathsf{Puz}}(k) = \Pr\left(\mathsf{Exp}^{\mathsf{UF}}_{\mathcal{A},\mathsf{Puz}}(k) = 1\right) \ .$$

If this advantage is negligible in k *for all probabilistic algorithms* $\mathcal{A}$ *running in time polynomial in* k*, then a puzzle* Puz *is said to be* unforgeable.

In the following difficulty experiment, the adversary is allowed to query the $\mathsf{CreatePuzSoln}$ oracle by choosing a str at will. This is to ensure that the adversary does not gain any advantage in solving a new puzzle even after seeing many puzzles and their corresponding solutions.

Definition 5 (Puzzle Difficulty [7]). *Let* k *be a security parameter. Fix a difficulty parameter* Q *throughout the experiment. Let* Puz *be a client puzzle and* $\mathcal{A}$ *be a probabilistic algorithm. The puzzle difficulty experiment* $\mathsf{Exp}^{\mathsf{Diff}}_{\mathcal{A},\mathsf{Puz},Q}(k)$ *is defined as follows:*

1. *Run the set up algorithm* $\mathsf{Setup}(1^k)$ *to obtain* s *and* $params$*.*
2. *Run the adversary* $\mathcal{A}$*with input* $params$*. The adversary has oracle access to* $\mathsf{CreatePuz}(\cdot)$ *and* $\mathsf{CheckPuz}(\cdot)$*, which are answered as follows:*
   - $\mathsf{CreatePuzSoln}(str)$*: Generate a puzzle* puz *by running the* $\mathsf{GenPuz}(s, Q, str)$ *algorithm. Return* puz *to* $\mathcal{A}$*. Find a solution* $soln$ *using* $\mathsf{FindSoln}$ *algorithm such that* $\mathsf{VerSoln}(puz, soln) = \mathsf{true}$*. Return* $(puz, soln)$ *to* $\mathcal{A}$*.*
   - $\mathsf{Test}(str^*)$*: At any point during the game, the adversary* $\mathcal{A}$*may ask this query once. For this query, the challenger generates a puzzle* puz^* *by running the* $\mathsf{GenPuz}(s, Q, str)$ *algorithm and returns* puz^* *to* $\mathcal{A}$*. Then* $\mathcal{A}$ *may continue to ask* $\mathsf{CreatePuzSoln}$ *queries.*
3. $\mathcal{A}$ *outputs a potential solution* $soln^*$*.*
4. *Output* 1 *if* $\mathsf{VerSoln}(puz^*, soln^*) = \mathsf{true}$ *and* 0 *otherwise.*

We say that $\mathcal{A}$ wins the game if $\mathsf{Exp}^{\mathsf{Diff}}_{\mathcal{A},\mathsf{Puz},Q}(k) = 1$ and loses otherwise. The advantage of $\mathcal{A}$ is defined as:

$$\mathsf{Adv}^{\mathsf{Diff}}_{\mathcal{A},\mathsf{Puz},Q}(k) = \Pr\left(\mathsf{Exp}^{\mathsf{Diff}}_{\mathcal{A},\mathsf{Puz},Q}(k) = 1\right) \ .$$

Let $\epsilon_{k,Q}(t)$ be a family of functions monotonically increasing in t. A puzzle Puz *is $\epsilon_{k,Q}(t)-$*difficult *if*

$$\mathsf{Adv}^{\mathsf{Diff}}_{\mathcal{A},\mathsf{Puz},Q}(k) \leq \epsilon_{k,Q}(t) \ .$$

for all probabilistic algorithms $\mathcal{A}$ running in time at most t,

Remark 2. The bound $\epsilon_{k,Q}(t)$ in the difficulty definition is quite abstract. Let Puz be a puzzle scheme such that each instance requires approximately Q steps to solve. If Puz is $\epsilon_{k,Q}(t)$-difficult, then a concrete function for $\epsilon_{k,Q}(t)$ might be of the form $\epsilon_{k,Q}(t) \approx t/Q + \mathrm{negl}(k)$.

B Security Analysis of RSAPuz

In this section, we show that RSAPuz satisfies the unforgeability and difficulty properties introduced by Chen *et al.* [7].

B.1 Unforgeability of RSAPuz

Puzzle unforgeability follows in a straightforward manner from the pseudorandomness of the function H used to compute the authentication tags.

Theorem 3 (Unforgeability of RSAPuz). *Let H be a pseudo-random function. Then* RSAPuz *is unforgeable.*

Proof. We use a sequence of games to prove this theorem. Let $\mathcal{A}$ be a probabilistic algorithm that runs in time t. Let E_i be the event that $\mathcal{A}$ wins in game G_i.

Game G_0 Let G_0 be the original unforgeability game. Then

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

Game G_1 We replace the pseudo-random function H_ρ which is used to compute Z by a truly random function F in game G_0 to obtain game G_1. Since H_ρ is a pseudo-random function, this modification has a negligible effect on adversary $\mathcal{A}$. Hence,

$$|\Pr(E_0) - \Pr(E_1)| \leq \mathsf{Adv}^{\mathsf{prf}}_{\mathcal{B}}(k) \leq \mathrm{negl}(k) \tag{2}$$

where $\mathcal{B}$ runs in time $O(t)$. The second inequality is due to the pseudo-randomness of H_ρ.

Since F is a truly random function, the probability that an adversary can guess an output without having access to F is negligible:

$$\Pr(E_1) \leq \frac{1}{2^k} \ . \tag{3}$$

Thus, the adversary's success in forging a puzzle is negligible and is obtained by combining equations (1)–(3). □

B.2 Difficulty of RSAPuz

In this section, we show that our puzzle RSAPuz is difficult in the Chen *et al.* model [7] as long as Rivest *et al.*'s time-lock puzzle TLPuz [18] is difficult. That is, we show in the following theorem that the probability of solving RSAPuz is bounded for all probabilistic algorithms $\mathcal{A}$ running in time at most t.

Theorem 4 (Difficulty of RSAPuz**).** *Let k be a security parameter and let Q be a difficulty parameter. Let* GenRSA *be a modulus generation algorithm. If* TLPuz *with* GenRSA *is $\epsilon_{k,Q}(t)$-difficult, then* RSAPuz *is $\epsilon'_{k,Q}(t)$-difficult for all probabilistic algorithms $\mathcal{A}$ running in time at most t, where*

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

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

Proof. We prove the theorem using a sequence of games. Let $\mathcal{A}$ be a probabilistic algorithm with running time t. Let E_i be the event that $\mathcal{A}$ wins in game G_i. We will use an adversary $\mathcal{A}$ that wins the puzzle difficulty experiment of RSAPuz to construct an algorithm $\mathcal{B}$ that solves the TLPuz easily.

Game G_0 Let G_0 be the original difficulty game $\mathsf{Exp}^{\mathsf{Diff}}_{\mathcal{A},\mathsf{RSAPuz}}(k)$.
For clarity, we write the full definition of this game:

1. The challenger obtains $s \leftarrow (\rho, d, \phi(n), (\alpha_i, \beta_i)_{i=1}^{N})$ and $params \leftarrow (Q, n)$ by running the Setup algorithm. The challenger keeps the secret s and gives the parameters $params$ to $\mathcal{A}$.
2. For the $\mathsf{CreatePuzSoln}(N_C)$ query issued by $\mathcal{A}$, the challenger responds as follows:
   - the challenger first obtain a pair (x, X) by running the BPV pair generator BPVGen and then it computes Z and y as per the puzzle specification (Figure 2).
   - The challenger responds to $\mathcal{A}$ with $(puz, soln) \leftarrow ((N_S, Z, x), y)$.
3. At some time during the game, $\mathcal{A}$ may issue the $\mathsf{Test}(N_C^*)$ query to the challenger. To respond to this, the challenger generates a puzzle $puz^* = (N_S^*, Z^*, x^*)$ using $\mathsf{GenPuz}(s, Q, N_{C^*})$ and returns puz^* to $\mathcal{A}$. Then $\mathcal{A}$ may continue to ask $\mathsf{CreatePuzSoln}(N_C)$ queries.
4. $\mathcal{A}$ outputs a potential solution $soln^* = y^*$.
5. The challenger outputs 1 if $\mathsf{VerSoln}(puz^*, soln^*) = \mathsf{true}$, otherwise outputs 0.

Then

$$\Pr\left(\mathsf{Exp}^{\mathsf{Diff}}_{\mathcal{A},\mathsf{RSAPuz}}(k) = 1\right) = \Pr(\mathsf{E}_0) \ . \tag{4}$$

Game G_1 Game G_1 is very similar to the Game G_0 except that we use the TLPuz challenger to answer the $\mathsf{CreatePuzSoln}$ queries from $\mathcal{A}$ and we insert a TLPuz challenge into the response to the Test query. In particular, the experiment proceeds as follows:

1. Obtain $params \leftarrow (Q, n)$ from the TLPuz challenger.
2. Run $\mathcal{A}(params)$ with oracle access to $\mathsf{CreatePuzSoln}(\cdot)$ and $\mathsf{Test}(\cdot)$, which are answered as follows:
   - $\mathsf{CreatePuzSoln}(str)$: When $\mathcal{A}$ makes a $\mathsf{CreatePuzSoln}(\mathtt{str})$ query, our challenger asks $\mathsf{CreatePuzSoln}$ query to the TLPuz challenger. Upon receiving a pair of the form $(\mathsf{puz}, \mathsf{soln}) = (a, b)$ where $b = a^{2^Q} \bmod n$, our challenger does the following:
     - Sets $x \leftarrow a$ and $y \leftarrow b$. Note that the value a received each time from TLPuz challenger is a random value in $[1, n]$, where as in RSAPuz, x is an output of the BPV generator and hence it is not chosen at random from $[1, n]$.
     - Return $(\mathsf{puz}, \mathsf{soln}) = (x, y)$ to $\mathcal{A}$.
   - $\mathsf{Test}(str^*)$: When $\mathcal{A}$ makes a $\mathsf{Test}(\mathtt{str}^*)$ query. Then our challenger simply passes the same query as its Test query to the TLPuz challenger. In return, our challenger receives its challenge puzzle $\mathsf{puz}^* = a^*$, where a^* is a random integer in $[1, n]$. Then our challenger simply passes it to $\mathcal{A}$. That is, the target puzzle for $\mathcal{A}$ is $\mathsf{puz}^* = a^*$.
3. $\mathcal{A}$ may continue its $\mathsf{CreatePuzSoln}$ queries and the challenger answers them as above.
4. When $\mathcal{A}$ outputs a potential solution soln^*, the challenger outputs the same soln^*.

If $\mathcal{A}$ wins game G_1, then the challenger wins the puzzle difficulty experiment of TLPuz. Hence,

$$\Pr(\mathsf{E}_2) \leq \mathsf{Adv}^{\mathsf{Diff}}_{\mathcal{B},\mathsf{TLPuz},Q}(k) \leq \epsilon_{k,Q}(t) \ . \tag{5}$$

where $\mathcal{B}$ is our challenger which runs in time $t(B) = t(A)+(q_{\mathsf{C}}+1)\,(2(\ell-1)T_{\mathrm{Mul}})+N \cdot T_{\mathrm{Exp}}$ where q_{C} is the number of $\mathsf{CreatePuzSoln}$ queries asked by $\mathcal{A}$ in G_0, T_{Mul} is the time complexity for computing a multiplication modulo n, and T_{Exp} is the time complexity for computing an exponentiation modulo n.

In G_0, a puzzle is of the form (N_S, Z, x) where x is an output from the BPV generator BPVGen whereas in G_1 it is uniformly random and is output by the TLPuz challenger.

Hence by Theorem 1, we get

$$|\Pr(\mathsf{E}_0) - \Pr(\mathsf{E}_1)| \leq 2^{-\frac{1}{2}\left(\log \binom{N}{\ell}+1\right)} \leq \epsilon_{k,Q}(t) \ , \tag{6}$$

where the second inequality follows from appropriate choices of N and ℓ.

Final result Combining equations (4) through (6) yields the desired result. □