Efficient Client Puzzle Schemes to Mitigate DoS Attacks

Qiang Tang and Arjan Jeckmans
DIES, Faculty of EEMCS
University of Twente
Enschede, the Netherlands
{*q.tang, a.j.p.jeckmans*}*@utwente.nl*

***Abstract*—A (computational) client puzzle scheme enables a client to prove to a server that a certain amount of computing resources (CPU cycles and/or Memory look-ups) has been dedicated to solve a puzzle. In a number of different scenarios, researchers have applied client puzzle schemes to mitigate DoS attacks. In this paper, we introduce two batch verification modes for the RSW client puzzle scheme in order to improve the verification efficiency for the server, and investigate three methods for handling incorrect solutions in batch verifications.**

I. Introduction

A (computational) client puzzle scheme enables a client to prove to a server that a certain amount of computing resources (CPU cycles and/or Memory look-ups) has been dedicated to solve a puzzle. It has been applied to mitigate denial-of-service (DoS) attacks in a number of scenarios such as email systems, web servers, and critical communication infrastructures. In a DoS attack, an attacker attempts to prevent legitimate users from accessing information or services by sending a large number of fake requests; furthermore, in its distributed form (referred to as a DDoS attack [9]), an attacker may use the controlled Zombie computers to simultaneously launch the attack. In more details, there are two categories of DoS attacks.

- One is the exhaustion of *specific* types of very limited computer resources, such as TCP connections. For example, the SYN flood attack falls into this category [6]. With a client puzzle scheme implemented, the server can mitigate an attack by asking every client to solve a puzzle before allocating any resource. The rationale is that, the number of "valid" requests from a malicious client will drop to some extent because the client has only limited resources to find puzzle solutions.
- The other is the exhaustion of bandwidth or *general* CPU cycles or memory usages, for this purpose, the adversary just congests the communication links or sends nonsense messages to the victim. For example, the jamming attack in wireless sensor networks falls into this category [7]. With a client puzzle scheme implemented, if malicious clients send non-sense data as their puzzle solutions, the attack will become even worse because the server has to spend resources in verifying the fake puzzle solutions. Therefore, client puzzle schemes will not help here.

In this paper, we focus on using client puzzle schemes to mitigate the first category of Dos attacks. How to mitigate the second type of DoS attacks is beyond the scope of this paper. For the related work in using client puzzles to combat DoS attacks, refer to [10] or the full version of this paper.

To effectively mitigate DoS attacks, the deterministic computation and parallel computation resistance properties, formally defined in [10], are desirable for a client puzzle scheme. The deterministic computation property implies that the server can precisely determine the required resource required from the client in solving a puzzle. Without this property, the server never knows what is the exact amount of computation required from a client to solve a puzzle, and therefore is unable to set an appropriate hardness for the puzzle. The parallel computation resistance property implies that the client cannot accelerate the puzzle solving process by letting more than one computer work in parallel. In practice, it is very difficult for a server to determine the amount of computing resources a client can access, especially in the presence of malicious clients which control a large number of Zombie computers. To some extent, this property will eliminate the computation disparity between clients and help create a fair situation for them. It is worth noting the memory-bound client puzzles [1], [4], [5] also aim at eliminating such disparity, however, they have not been proven with the parallel computation resistance property. Most existing client puzzle schemes, such those based on hash functions [3], [6], do not achieve these properties. Interestingly, the RSW scheme, which was originally proposed by Rivest, Shamir, and Wagner to realize timed-encryption, achieves both properties [10]. To our knowledge, this is the only scheme that has been rigorously proven achieving both properties.

The downside with the RSW scheme is that it incurs heavy overhead for the server, which needs to perform one exponentiation to verify a puzzle solution. In this paper, we first investigate how to improve the efficiency of the server by verifying multiple puzzle solutions. We apply the batch verification techniques, which were originally introduced for signature schemes [2], to the RSW scheme, and introduce two batch verification modes. The application is rather straightforward and our contribution lies in the handling of incorrect solutions in the batch verification process. To this end, we propose three methods for handling incorrect solutions in batch verifications, and provide comparison results based on our simulations.

The rest of the paper is organized as follows. In Section II, we introduce two batch verification modes for the RSW scheme. In Section III, we introduce three methods to handle incorrect solutions in batch verifications. In Section IV, we conclude the paper.

II. Batch Verification of the RSW Scheme

The scheme, described below, is a slightly modified version of the original RSW client puzzle scheme proposed by Rivest, Shamir, and Wagner [8]. For simplicity, we still call it the RSW scheme.

- $\mathsf{Setup}(\ell)$: Run by the server, this algorithm takes a security parameter ℓ as input. It selects two random large primes p, q and a hash function $\mathsf{H} : \{0,1\}^* \rightarrow \mathbb{Z}^*_{pq}$, and outputs the public parameter pq and the master key $mk = (p, q)$.
- $\mathsf{PuzzleGen}(mk, d, req)$: Run by the server, this algorithm takes the server's master key mk, a puzzle hardness d, and some additional information req as input, and computes $g = \mathsf{H}(r||req)$ where $r \in_R \mathbb{Z}^*_{pq}$. The server sends $puz = (g, d)$ to the client as the puzzle, while keeps the related puzzle information $info = (r, d, req)$ by itself.
- $\mathsf{PuzzleSol}(puz)$: Run by a client, this algorithm takes a puzzle puz as input and outputs $sol = g^{2^d} \mod pq$.
- $\mathsf{PuzzleVer}(mk, info, sol)$: Run by the server, this algorithm takes the master key mk, the related puzzle information $info$, and the puzzle solution sol as input. It returns 1 if $sol \equiv g^{2^{d \bmod \phi(pq)}} \pmod{pq}$, where $g = \mathsf{H}(r||req)$, and returns 0 otherwise.

Note that the puzzle hardness parameter d is an integer, denoting the number of multiplications in $\mathbb{Z}^*_{pq}$.

In the above scheme, g is computed as $g = \mathsf{H}(r||req)$, while g is randomly chosen from $\mathbb{Z}^*_{pq}$ in [8]. In our case, if needed, the g can be bound to situational information (such as the identity information of the client) contained in req. With respect to computing the verification complexity of the server, we omit that of computing $2^d \mod \phi(pq)$ for two reasons. One is that it could be pre-computed and stored by the server. The other is that, in many cases, multiple puzzles might share the same hardness so that the computation only needs to be done once. As a consequence, it is straightforward to calculate that the average verification complexity for the server is $\frac{3L}{2} - 2$ multiplications in $\mathbb{Z}^*_{pq}$, where L is the bit-length of $\phi(pq)$.

This scheme has been proven with the deterministic computation and parallel computation resistance properties [10], and we skip the details here. In the rest of this section, we introduce two batch verification modes for the RSW scheme. Due to the lack of space, the proofs for all lemmas will appear in a full version of this paper.

A. A Batch Verification Mode - Attempt

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

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

Based on this observation, suppose that there are n puzzles $puz_i = (g_i, d)$ $(1 \leq i \leq n)$ and solutions h_i $(1 \leq i \leq n)$, we can verify the solutions using a batch verification mode, by checking the following equality.

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

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

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

$$C_n = 1732 << (\frac{3L}{2} - 2) \cdot n = 153400.$$

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

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

replaced with any h'_i $(1 \le i \le n)$, where

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

the equality still holds.

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

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

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

B. A Batch Verification Mode - Improvement

Suppose that there are n puzzles $puz_i = (g_i, d)$ $(1 \le i \le n)$ and solutions h_i $(1 \le i \le n)$, the improved batch verification mode is as follows. Select $x_i \in \mathbb{Z}^*_N$, where N is an integer and smaller than pq, and check the following equality.

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

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

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

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

C. Further Improvement

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

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

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

Lemma 2. *Suppose that the server divides the received solutions into Y subgroups. The probability that a* false accept *occurs is $(\frac{1}{Y})^{n-1}$.*

Clearly, when Y becomes larger (or, the size of subgroup become smaller), the *false accept* rate will drop much faster. In practice, the *divide-and-verify* strategy and the improved batch verification mode (described in Section II-B) can be integrated, namely the server first divides the received puzzle solutions into several subgroups, and then performs batch verification for each subgroup. The *false accept* rate is described by the following lemma.

Lemma 3. *Suppose that the server divides the received solutions into Y subgroups. With the improved batch verification mode, the probability that a* false accept *occur is $(\frac{1}{Y})^{n-1} \cdot \frac{1}{N}$.*

III. Handling Incorrect Solutions in Batch Verification

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

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

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

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

Next, we take the basic verification mode as example and consider three different methods to figure out the incorrect solutions, namely *sequential searching*, *sequential searching with batch verification*, and *dividing-and-conquering*. Our analysis will focus on the average complexity for the server.

A. The Case of sequential searching

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

B. The Case of sequential searching with batch verification

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

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

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

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

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

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

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

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

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

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

C. The Case of dividing-and-conquering

Generate a puzzle set list $\mathcal{L}$ and initialize it to be $\{\mathcal{B}\}$. The algorithm of *dividing-and-conquering* is as follows.

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

D. A Comparison of Different Methods

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

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

From Table I, we can roughly draw the following conclusions. When the rate of incorrect solutions (namely $\frac{t}{n}$) is small, the method *dividing-and-conquering* is more efficient than the other two, and the method *sequential searching with batch verification* is also more efficient than the method *sequential searching*. When the rate increases, the advantage of the method *dividing-and-conquering* becomes less obvious, while *sequential searching with batch verification* may become less efficient than the method *sequential searching*. Intuitively, let the bit-length of $\phi(pq)$ be 1024 (i.e. $L = 1024$), a

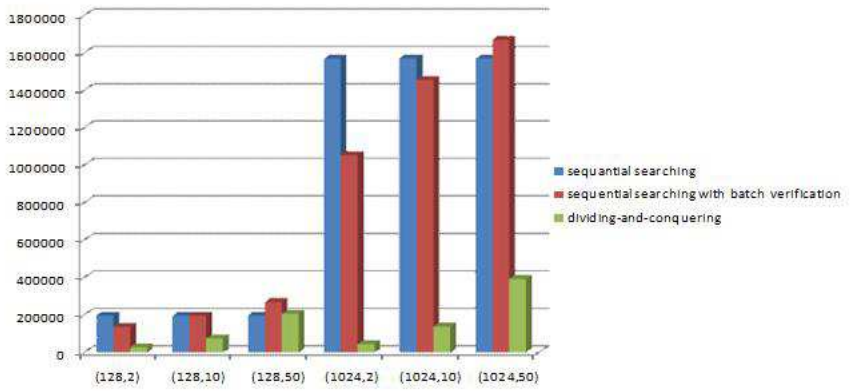

Figure 1. Comparison Results

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

Table I
Complexity Comparison

visual comparison is shown in Figure 1. Overall, the method *dividing-and-conquering* is a preferred one.

IV. Conclusion

In this paper, we have shown that the RSW scheme supports batch verification modes, which greatly improve the efficiency for the server. While our proposal is theoretical and abstract at the moment, an interesting future work is to instantiate the proposal in a real-world application, such as defeating junk emails, and to further investigate the effectiveness.

References

[1] M. Abadi, M. Burrows, M. Manasse, and T. Wobber. Moderately hard, memory-bound functions. *ACM Transactions on Internet Technology*, 5(2):299–327, 2005.

[2] M. Bellare, J. Garay, and T. Rabin. Fast batch verification for modular exponentiation and digital signatures. In *Eurocrypt '98*, pages 236–250, 1998.

[3] L. Chen, P. Morrissey, N. Smart, and B. Warinschi. Security notions and generic constructions for client puzzles. In *Advances in Cryptology — Asiacrypt 2009*, volume 5912 of *LNCS*, pages 505–523. Springer, 2009.

[4] S. Doshi, F. Monrose, and A. D. Rubin. Efficient memory bound puzzles using pattern databases. In *Applied Cryptography and Network Security, 4th International Conference, ACNS 2006*, pages 98–113, 2006.

[5] C. Dwork, A. Goldberg, and M. Naor. On memory-bound functions for fighting spam. In Dan Boneh, editor, *Advances in Cryptology — CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 426–444. Springer, 2003.

[6] A. Juels and J. G. Brainard. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In *Proceedings of NDSS'99*, pages 151–165, 1999.

[7] D. R. Raymond and S. F. Midkiff. Denial-of-service in wireless sensor networks: Attacks and defenses. *IEEE Pervasive Computing*, 7(1):74–81, 2008.

[8] R. L. Rivest, A. Shamir, and D. A. Wagner. Time-lock puzzles and timed-release crypto. Technical Report MIT/LCS/TR-684, MIT, 1996.

[9] S. M. Specht and R. B. Lee. Distributed denial of service: Taxonomies of attacks, tools, and countermeasures. In D. A. Bader and A. A. Khokhar, editors, *Proceedings of the ISCA 17th International Conference on Parallel and Distributed Computing Systems*, pages 543–550, 2004.

[10] Q. Tang and A. Jeckmans. On non-parallelizable deterministic client puzzle scheme with batch verification modes. Technical Report TR-CTIT-10-02, CTIT, University of Twente, 2010. http://eprints.eemcs.utwente.nl/17107/.