17. Twining, D., Williamson, M., Mowbray, M., Rahmouni, M.: Email prioritization: Reducing delays on legitimate mail caused by junk mail. In: Usenix Annual Technology Conference (2004)
18. http://www.soft14.com/Utilities_and_Hardware/Antivirus/ViraLock_5641_Review.html
19. Wong, C., Bielski, S., McCune, J., Wang, C.: A Study of Mass-Mailing Worms. In: Proceedings of ACM Worm (2004)
20. Williamson, M.: Design, Implementation and Test of an Email Virus Throttle. In: Omondi, A.R., Sedukhin, S. (eds.) ACSAC 2003. LNCS, vol. 2823, Springer, Heidelberg (2003)

Suratose Tritilanunt, Colin Boyd,
Ernest Foo, and Juan Manuel González Nieto

Information Security Institute
Queensland University of Technology
GPO Box 2434, Brisbane, QLD 4001, Australia
s.tritilanunt@student.qut.edu.au,
{c.boyd,e.foo,j.gonzaleznieto}@qut.edu.au

Abstract. Client puzzles have been proposed as a useful mechanism for mitigating denial of service attacks on network protocols. Several different puzzles have been proposed in recent years. This paper reviews the desirable properties of client puzzles, pointing out that there is currently no puzzle which satisfies all such properties. We investigate how to provide the property of non-parallelizability in a practical puzzle. After showing that obvious ideas based on hash chains have significant problems, we propose a new puzzle based on the subset sum problem. Despite some practical implementation issues, this is the first example that satisfies all the desirable properties for a client puzzle.

Keywords: Denial of Service Attacks, Client Puzzles, Non-Parallelizable Cryptographic Puzzles.

1 Introduction

Cryptographic puzzles, or *client puzzles*, have been proposed as a mechanism to defeat resource exhaustion denial of service (DoS) attacks in network protocols, particularly in key exchange protocols. Client puzzles counterbalance computational usage between client and server machines. By forcing the client to solve a computational puzzle before attending to a request, the server ensures that the client spends sufficient resources before committing its own. In particular, an adversary who wishes to flood a server with connection requests will have to solve a huge number of puzzles. The idea of using cryptographic puzzles in computer networks was first introduced by Dwork and Naor [5] for combating junk emails. Juels and Brainard [10] extended the concept of puzzles to thwart Denial-of-Service (DoS) attacks in network protocols. Recently, Moskowitz developed the host identity protocol (HIP) [16], which employs a client puzzle mechanism for protecting the server against resource exhaustion attacks.

Although a variety of client puzzles have been proposed to solve DoS attacks, limited analysis of these proposals has appeared in the literature. An exception is the work of Price [17], who introduces a generic attack against hash-based client puzzles. Another investigation of hash-based client puzzles has been carried out by Feng et al. [7]. They examine client puzzles based on six parameters: unit work, range, mean granularity, maximum granularity, exact control and parallel computation.

A client puzzle is *non-parallelizable* if the solution to the puzzle cannot be computed in parallel. Non-parallelizable client puzzles can be used to defend against distributed denial-of-service (DDoS) attacks, where a single adversary can control a large group of compromised machines and launch attacks to the targeted server from those machines. If the client puzzle is parallelizable, such an adversary could distribute puzzles to other compromised machines to obtain puzzle solutions faster than the time expected by the server. A client puzzle is said to provide *fine granularity* if it allows servers to adjust the solution time precisely. Both non-parallelizability and fine granularity are important properties of good puzzles. A survey of existing client puzzles reveals that only time lock puzzles [19] are able to provide both non-parallelizability and fine-grained control. However, these puzzles suffer from being computationally expensive in puzzle construction and verification.

In this paper we propose a new puzzle construction based on the subset sum problem. The primary strengths of this puzzle over others are the simple and cheap construction and verification for the server, as well as non-parallelizability. The main contributions of this work are:

– to provide a summary and analysis of client puzzles for DoS-resistance;
– to compare strengths and weaknesses of existing client puzzles;
– to propose a new construction, called *Subset Sum Client Puzzles*.

In the next section we will summarise existing proposals for client puzzles and review their properties. Section 3 examines possible ways to use hash chains as non-parallelizable puzzles and then Section 4 introduces and analyses our new puzzle.

2 Survey and Analysis of Client Puzzles Approaches

Client puzzles functioning as proofs of work can be constructed from a number of underlying problems. Although many puzzles have been proposed using different techniques, all of them should satisfy seven important properties described by Aura et al. [2]; for instance puzzles should be easy and cheap to construct and verify for the server, but lead to a significant computational effort for adversaries who attempt to flood a large number of bogus requests to the server.

Feng et al. [7] proposed some additional criteria for evaluating *efficiency* and *resolution* of cryptographic puzzles. As defined by Feng et al. [7], the puzzle efficiency represents speed of puzzle generation and verification on the server's machine compared to the puzzle solving on the client's machine. Meanwhile, the resolution or *puzzle granularity* represents the ability of the server to finely control the amount of work done by calibrating the puzzle difficulty to the client. The following list represents the properties that we examine in this paper.

Server's Cost identifies the computational effort on the server's machine. This factor is divided into three subcategories consisting of pre-computation cost, construction cost, and verification cost.

Client's Cost represents the amount of computational effort on the client's machine. We assume that the server and the client have similar resources regarding both CPU and memory units to process these puzzles. We note that this may not be realistic in some applications; for example, some legitimate clients may have restricted resources.

Non-parallelizability describes whether the client puzzles can be distributed and solved in parallel computation. In some circumstances, non-parallelizable puzzles can prevent coordinated adversaries from distributing puzzles to a group of high performance machines to obtain solutions quicker than the specified period assigned by the server. Consequently, the server becomes overloaded by a huge amount of attack traffic and unable to process any upcoming legitimate messages.

Granularity represents the ability of the server to finely adjust puzzle difficulty to different levels. Indirectly, this parameter also affects the traffic flow of arriving packets to the connection queue within a certain time. Three different types of granularity; *linear*, *polynomial*, and *exponential* are compared. Linear granularity is the best that we hope to deal with, while an exponential one is the worst case.

We now conduct a short survey and comparison in term of strengths and weaknesses of existing proposals for client puzzles.

Hash-based Reversal Puzzles: In 1999, Juels and Brainard [10] introduced the construction of client puzzles using a hash function; clients need to calculate a reverse one-way hash value of a puzzle generated by the server. In this technique, the server is able to adjust the difficulty level of the client puzzle by increasing or decreasing the number of hidden bits of the pre-image sent to clients in the puzzle. The client performs a brute-force search to find missing bits of pre-image whose output is given by hashing each pattern until matching the answer. To verify the solution, the server needs to perform only a single hash operation.

An alternative construction was proposed by Aura et al. [2]. Different from Juels and Brainard's construction, the puzzle generation requires only a single hash instead of two hash operations as in Juels and Brainard's scheme. Given part of the pre-image and the length (k) of zero bits at the beginning of the hashed output, clients need to perform a brute-force search to find a matching solution.

In summary, the major strength of these two hash-based reversal schemes is the simple and fast construction and verification. On the other hand, the weaknesses are that they are parallelizable and their granularity is exponential which brings a difficult task to the server to control and adjust the incoming rate of requests.

Hint-Based Hash Reversal Puzzles: As the granularity of hash-based reversal schemes is too coarse, Feng et al. [6] proposed the idea of hint-based hash reversal puzzles to allow the granularity to be linear. The technique of this mechanism is that the server provides extra information called *hints* attached to the puzzle. Instead of checking every possible solution, the client searches for a solution within a range of a given hint. Apart from this action, all remaining processes are similar to the original work from Juels and Brainard [10]. Hence, the simple puzzle generation and verification as well as the linear granularity for fine grained control are the strengths of this construction. However, it is still susceptible to distribution and parallel processing attacks, as is the original hash-based reversal scheme.

Repeated-Squaring or Time-Lock Puzzles: Time-lock puzzles were developed by Rivest et al. [19] in 1996. The major goal of this technique is to defeat the high-end adversaries who attempt to solve puzzles more quickly by using parallel computers. Time-lock puzzles rely on the notion that a client has to spend a pre-determined amount of computation time performing repeated squaring to search for a solution. To achieve this goal, the server estimates the performance of a client by the number of squaring operations a client can perform in a certain period, and determines the amount of time it expects a client to spend solving the puzzle.

To solve the puzzle, the client is required to compute a modular squaring operation repeatedly. This computation must be calculated sequentially so it cannot be distributed and solved in parallel. Since the period of solving the puzzle is easily controlled and determined by the server at puzzle generation time, we can conclude that the time-lock puzzles have a linear granularity. Another strength of this scheme is its non-parallelizable characteristic because it requires an inherently sequential operation to solve a puzzle. In the original paper, the major purpose of this scheme is the long term protection of secret information, for example, in the application of the on-line auction. However, the primary concern of this scheme in DoS mitigation applications is the high-computation in the construction and verification because the underlying technique requires the server to perform a costly modular exponentiation.

DH-based Puzzles: Diffie-Hellman based puzzles were proposed by Waters et al. [25] in 2004. The construction requires an expensive Diffie-Hellman operation, while the verification could be simply done via table lookup, which is considered a cheap operation, because the server has already generated puzzle solutions at the construction and stores them in the memory. Therefore, the expensive construction would be a drawback, while the cheap verification would be the major positive characteristic.

Given the range of a solution as in hint-based schemes, the client searches for a solution by testing each candidate value in the range until it finds a correct solution. Similar to other hint-based puzzles, this scheme then provides a linear-grained control to the server. Considering the non-parallelizability, because clients require a specific range of attempts to find a correct solution, the puzzle can be distributed and computed in parallel to obtain a correct solution. As a result, this scheme does not support non-parallelizability.

Trapdoor RSA-based and DH-based Puzzles: Gao [8] developed two puzzle mechanisms based on trapdoor functions to overcome weaknesses over the hash-based construction. By pre-computing some parameters and expensive operations before starting the protocol, Gao's implementation can reduce the overhead of puzzle construction. However, this pre-computation workload is a disadvantage to these types of puzzles.

On the positive side, the protocol computes and stores the solutions at puzzle generation time to save workload at verification. As a result, the server requires only a single comparison in order to check validity of the solution from the client. In the puzzle solving, the client is given a range of candidates to run a brute-force search for a correct solution. Hence, the granularity of these two constructions is linear-grained.

On the negative side, both trapdoor-function based schemes can be distributed and solved in parallel by a group of adversaries as for other hint-based puzzles. Moreover, these schemes involve modular arithmetical operations which are more expensive than hash functions. Although Gao [8] suggested to perform pre-computations to avoid CPU burden at construction time, puzzle generation still requires a number of modular exponentiations.

Table 1 compares seven cryptographic puzzle constructions based on the analysis criteria previously discussed. For purposes of comparison, we include our new subset sum puzzles in the table. Details will be discussed in Section 4.1. The highlighted field (displayed as the ***bold and italic style***) in individual columns represents the best candidate for each analysis criterion. In the server's and client's cost entry, we use the number of operations as a measurement for comparison. More precisely, the hash-based cryptographic puzzles require a number of hash function computations displayed as *hash* in the table, while the arithmetic-based puzzles require a number of modular exponentiations represented by *mod exp* and modular multiplications represented by *mod mul*. Modular arithmetic consumes much greater resources than hash functions. Hence, the preference for this entry would be the technique which expends a small number of hash operations. We can conclude that the puzzle construction based on hash-based reversal would be the most effective technique.

The non-parallelizability characteristic plays an important role for defending against coordinated adversaries who attempt to distribute puzzles to other users

Table 1. Comparison of existing Client Puzzles for DoS Resistance

Puzzle Type	Server's Cost			Client's Cost	Non Parallel	Granularity
	Pre-Compute	Construction	Verification			
Hash-based Reversal	-	***1 hash***	1 hash	$\mathcal{O}(2^k)$ hash	No	Exponential
Hint-Based Hash Reversal	-	***1 hash***	1 hash	***$\mathcal{O}(k)$ hash***	No	***Linear***
Repeated-Squaring	-	2 mod mul	2 mod mul	$\mathcal{O}(k)$ mod mul	***Yes***	***Linear***
DH-based	-	1 mod exp	***1 comparison***	$\mathcal{O}(k)$ mod exp	No	***Linear***
Trapdoor RSA	1 mod exp 1 mod mul	3 mod mul 2 additions	***1 comparison***	$\mathcal{O}(k)$ mod exp	No	***Linear***
Trapdoor DLP	1 mod exp	2 mod mul 3 additions	***1 comparison***	$\mathcal{O}(k)$ mod exp	No	***Linear***
Subset Sum	n hash	***1 hash***	***1 comparison***	L^3 reduction	***Yes***	Polynomial

or high-performance machines in order to obtain puzzle solutions quicker than the specified time without wasting their own resources. Since non-parallelizability has not been defined as a primary requirement in the original work [10,2], most existing techniques lack this characteristic. From the evaluation shown in Table 1 only repeated-squaring puzzles can thwart this type of attack strategy. Unfortunately, high computation of the puzzle construction causes this technique to be susceptible to flooding attacks. As a result, this gap becomes the most interesting point for our work to develop new schemes which achieve non-parallelizability, while the puzzle construction and verification are also simple and cheap.

3 Hash Chain Puzzles

We have seen in the previous section that currently only time-lock puzzles can provide the characteristic of non-parallelizability but they suffer from an expensive set up operation for the server. One promising method to prevent adversaries from distributing and computing a puzzle in parallel would be a *chaining* technique. Because the characteristic of chaining requires the previous value for constructing the next consecutive items, it will defeat those coordinated adversaries who attempt to solve puzzles by parallel computing. Recently, there are two constructions using the chaining technique based on hash functions proposed by Ma [14] in 2005 and by Groza and Petrica [9] a year later. The aim of these constructions is slightly different from what we have in mind, since they are interested in partial solving of the chained puzzles. Nevertheless it is interesting to examine whether they will be useful as stand-alone puzzles. Following are short descriptions of these two puzzles and an analysis of their suitability.

Ma's Hash Chain Reversal Puzzles: The concept of hash chain puzzles was introduced by Ma [14] in 2005 as password puzzles for use in the IP layer. The construction begins with a random number chosen as an initial value h_0. Then the server applies a one-way function to h_0 repeatedly to generate a hash chain $h_0, h_1, \dots, h_k$ where $h_{i+1} = \text{hash}(h_i)$ and k is the desired length of the chain. According to Ma, this computation would lead to an advantage for the server by storing the entire hash chain for future use. Because the server knows a corresponding solution in advance, the server saves computation and time when verifying the puzzle solution by reducing the cost of verification to a single table lookup.

For puzzle solving, given a puzzle challenge containing the last value of a hash chain (h_k) along with an index value k, a client is required to compute a hash reversal starting from index k back to the beginning point h_0 to obtain the entire hash chain. A characteristic of hash chain operation is that an output from the former state is required to be fed to the next state as an input, similar to a recursion in programming. We conclude that this scheme is a non-parallelizable technique, and the cost of the verification requires k hash operations similar to the construction.

This is a simple and intuitive construction, but there are a number of practical problems. First, it requires the server to store every value of the entire hash chain in order to be able to verify the solution. Although this has an advantage in verification effort, this scheme is susceptible to memory exhaustion attacks. Second, when used with a typical cryptographic hash function the scheme will be too difficult to invert for even one hash value, let alone a chain of many values. Therefore some mechanism must be chosen to make the individual steps in the chain invertible with reasonable effort. Ma [14] suggested that a hash function be used which has 16-bit outputs, but this does not seem to be an acceptable requirement since such a function can be easily stored completely in a look-up table which makes solving the puzzle as easy as constructing it. A more plausible mechanism is used in the next construction that we consider.

Groza and Petrica's Hash Chain Puzzles: This puzzle scheme [9] was constructed from a hash chain of random numbers. Generally, the idea is similar to the puzzle auction proposed by Wang and Reiter [24]; i.e. the more links of the chain computed on a client's machine, the more services from a server a client obtains. At the beginning, the server generates the first element by choosing two state-dependent random numbers, ρ and r, and concatenating them to obtain a value σ. The first output, P_0, is constructed by double hashing σ_0. Hence, the parameter σ_0 serves as an input to the next state of the chain. The rest of the puzzle will be created by XORing two new state-dependent values with hashed output of σ from the previous state. Thus, the puzzle elements challenged to the client would be a series of pairs $[(P_0, r_0), (P_1, r_1), \dots, (P_n, r_n)]$, where $n \geq 1$ is the length of the hash chain. Meanwhile, the client is required to perform a forward process of reconstructing the hash chain by searching for ρ_i values, with $\sigma_i = \rho_i \parallel r_i$.

Unfortunately, this scheme has a major drawback which risks resource exhaustion attacks on the server because it requires three hash operations per state for producing a series of hashes chained either in the construction or verification phase. This action requires a similar amount of computational effort as the solving task on the client's machine. This circumstance violates the fundamental requirement; i.e. client puzzles should be easy to generate and verify by the server but hard to solve by the client. Furthermore, the high-bandwidth consumption required to transmit a puzzle challenge is another drawback of this scheme.

In summary, we have seen that the hash chain puzzle has a major strength in non-parallelizability and linear-grained control because of its structure. Lightweight verification by one comparison is another interesting potential property. However, the proposals so far using this technique require high computation in the construction, high-bandwidth connection for communication, and huge storage to cache an entire chain for avoiding CPU burden at the verification. Therefore, currently it seems impractical to use hash chains as client puzzles and we look for an alternative.

4 Subset Sum Puzzles

Hash based puzzles are the most prevalent due to their simple construction and cheap verification. As shown in Section 2, such puzzles are susceptible to coordinated attacks because they do not provide the non-parallelizability property. In this section, we propose a technique called *subset sum puzzles*. The predominant characteristic of this new approach is not only a simple construction and verification as cheap as hash based puzzles, but also a non-parallelizable characteristic.

A subset sum (or knapsack) system associates a given set of items which have specified weight, with a knapsack which can carry the number of items no more than a certain weight. The solver is required to search for a maximum value by picking as many items as the knapsack can carry in terms of weight. To find whether a solution exists for a specified weight, this becomes a decision problem and the knapsack falls into the NP-completeness category which means no polynomial algorithm can break the knapsack problem within polynomial time as long as P $\neq$ NP. This is why the knapsack problem was long considered a promising underlying technique for constructing a public-key based cryptosystem.

A famous tool used to successfully break subset sum cryptosystems is the *lattice reduction.* There are several lattice reduction algorithms but the best method so far for breaking the subset sum problems is the LLL or L^3 algorithm (details are provided in Appendix A) developed by Lenstra et al. [13] in 1982. The interesting characteristic of the LLL scheme is that it is a polynomial time and non-parallelizable algorithm because it requires highly sequential computation on an iterative function. We remark that practical application of our construction requires clients to implement the LLL algorithm. While this is not a major problem on PC platforms it may be undesirable, particularly on low-powered platforms. Therefore we regard our construction as more a proof-of-concept that non-parallelizable puzzles are feasible, rather than as an ideal solution.

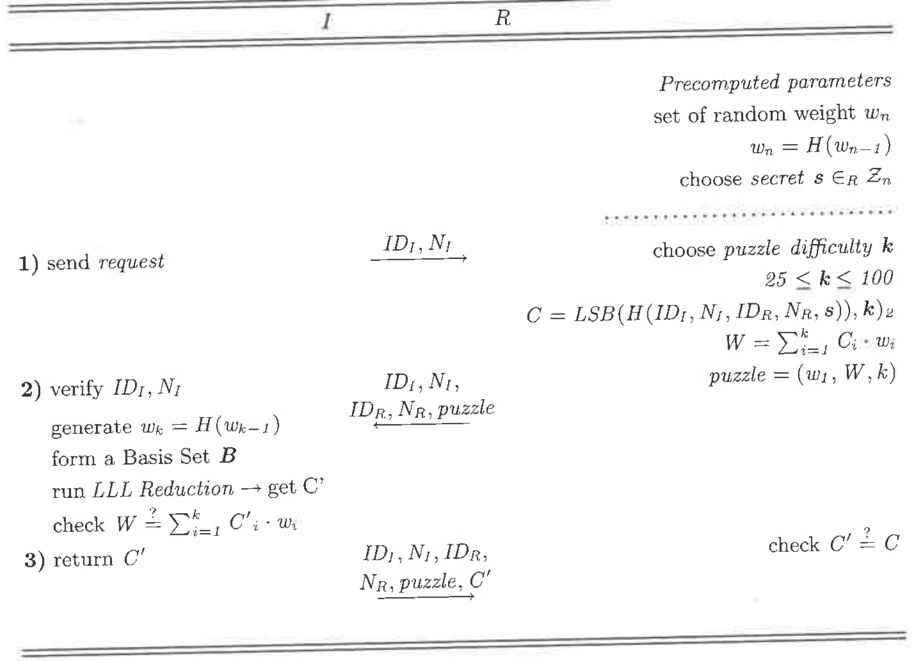
4.1 A New Proposal – Subset Sum Puzzles

We first introduce the notation used in the puzzle challenge-response protocol. I represents a client and R represents a server of the protocol. Communicating messages used in the protocol execution will carry the subscript I or R representing whose these messages are; for instance, ID_I represents the identity of the client and N_R represents a nonce generated by the server. A secret parameter is denoted as s and puzzle difficulty by k. The desired weight of the subset sum problem is W, while the set of candidate weights is $w_1, w_2, \ldots, w_n$. Finally, $H(\cdot)$ represents a hash operation on arbitrary length input messages, and $LSB(\cdot, k)_2$ obtains the k least significant bits from the output of the hash function.

Puzzle Construction

To establish a secure connection to a server, I sends a request containing an identity (ID_I) along with a random nonce (N_I) to R. The server chooses a secret parameter s randomly in order to make the output unique for each communication, and decides a puzzle difficulty k depending on the workload condition. The value of k[1] should be selected to be at least 25 (refer to Table 2 for a

[1] For the definition of subset sum puzzles, the number of items n is used as the puzzle difficulty k.



I
R
Precomputed parameters
set of random weight w_n
$w_n = H(w_{n-1})$
choose *secret* $s \in_R Z_n$
1) send *request*
$\xrightarrow{ID_I, N_I}$
choose *puzzle difficulty* $\boldsymbol{k}$
$25 \leq \boldsymbol{k} \leq 100$
$C = LSB(H(ID_I, N_I, ID_R, N_R, s)), \boldsymbol{k})_2$
$W = \sum_{i=1}^{k} C_i \cdot w_i$
$puzzle = (w_1, W, k)$
2) verify ID_I, N_I
$ID_I, N_I,$
$\xleftarrow{ID_R, N_R, puzzle}$
generate $w_k = H(w_{k-1})$
form a Basis Set $\boldsymbol{B}$
run *LLL Reduction* $\rightarrow$ get C'
check $W \overset{?}{=} \sum_{i=1}^{k} C'_i \cdot w_i$
3) return C'
$ID_I, N_I, ID_R,$
$\xrightarrow{N_R, puzzle, C'}$
check $C' \overset{?}{=} C$

Fig. 1. Subset Sum Puzzles

comparison of the experimental result) to guarantee that the coordinated adversaries approximately requires over a thousand compromised machines to brute-force search or over a hundred compromised machines to run the branch & bound algorithm on the subset sum puzzles at the equivalent proportion to the legitimate user performing LLL lattice reduction. As a practical choice we suggest to take a value of k between 25 and 100 and then if weights are chosen to be of length 200 bits we can ensure that the generated knapsack has density at most 0.5. Practical experimental tests are shown in Section 4.2 which support our proposal.

To construct a puzzle, R computes a hash operation ($H(\cdot)$), and computes ($LSB((\cdot), k)_2$) to obtain k bits from the output of hash function. In practice H could be implemented by truncating the output of SHA-256. Finally, R forms a *puzzle* by computing a desired weight (W) that it wants a client to solve from a pre-computed set of random weight (w_n). To save on protocol bandwidth, the weights can be generated given the initial random weight w_1 by iterative hashing. Hence, a puzzle contains an initial value of weight of the first item (w_1), a desired weight (W), and puzzle difficulty (k). The construction of the subset sum puzzle requires only one hash operation and addition. Figure 1 demonstrates the puzzle challenge-response protocol.

Puzzle Solving

To ensure that the client follows our requirement, we have to configure the puzzle difficulty so that the efficient LLL method of solving is more efficient than brute-force searching, even when the latter is divided amongst many parallel attacking machines. As mentioned above in the description of puzzle construction, when k is in the range between 25 to 100 we can expect that a puzzle would not be solved faster by brute-force technique. Moreover, when k is around 50 or larger the LLL method is more efficient than brute-force search even when the latter is divided amongst 10000 parallel machines in approximation.

By using the LLL algorithm, users can simply treat the subset sum schemes as a lattice problem. In 1985, Lagarias and Odlyzko [11] announced the first successful attack on low density[2] subset sum cryptosystems; i.e. a density below 0.6464 approximately. A few years later, Coster et al. [4] proposed the improved version of the Lagarias and Odlyzko technique. They claimed that their method was able to break *almost all* subset sum problems having density below 0.9408 in polynomial time. This result guarantees that our subset sum puzzle would be solvable in polynomial time by using LLL algorithm.

Consider the client's job when receiving a puzzle challenge from a server. It begins to generate a series of random weights, $(w_1, w_2, \ldots, w_k)$, by computing a hash chain on an initial value w_1. Then, the client constructs a basis reduction set B as follows.

$$b_1 = (1, 0, \ldots, 0, w_1); \quad b_2 = (0, 1, \ldots, 0, w_2)$$
$$\vdots$$
$$b_k = (0, 0, \ldots, 1, w_k); b_{k+1} = (0, 0, \ldots, 0, -W)$$

Finally, the client runs a L^3 lattice reduction [13] which is known from the community to be the most effective method to find moderately short lattice vectors in polynomial time. The algorithm guarantees to return a set of outputs in which one is a solution of the puzzle. To the best of our knowledge, *almost all* subset sum problems having density below 0.9408 can be effectively solved by the improved LLL version of Coster et al. [4]. In addition, this improved version is a highly sequential process because the underlying algorithm requires recursive computation as explained in Appendix A, so the puzzle cannot be distributed for parallel computation.

In terms of the puzzle granularity, there are two possible options for the server to adjust the puzzle difficult; 1) adjusting the item size (n), or 2) adjusting the density (which will cause a change in B because the density relates to the maximum weight of the items). Both modifications affect the running time by a factor $(n^\alpha \cdot \log^\beta B)$, where α and β are real numbers dependent on the version of LLL basis reduction. Since the complexity of LLL basis reduction is a polynomial function, we conclude that our subset sum puzzles provide a polynomial granularity.

[2] The density is defined as $\frac{n}{\log(max\ a_n)}$, where n is a number of items and $max\ a_n$ is the maximum item value.

Puzzle Verification

Puzzle verification is a simple and cheap task for a server which eliminates the risk of puzzle solution flooding attacks. Generally, there are two options for the verification process;

1. *avoiding CPU usage*: this case minimizes CPU usage at verification time. By storing the value of C and W corresponding to the client's identity $(ID_I,\ N_I)$, the verification requires only a table lookup for comparing the claimed solution from a client to the stored solution.
2. *avoiding memory usage*: this option eliminates memory usage prior to verification. The server uses a stateless connection in which no information is stored until the puzzle is solved. Once the server receives a solution, it is required to re-generate C and W from the arriving message. In order to protect against replay attacks, implementation of the timestamp should be used in the computation of the parameter C. The re-constructing process is a very cheap and fast computation that costs little more than a single hash computation, which is the typical cost of verification for hash-reversal puzzles.

We conclude that the upper bound of computational complexity in the former case is $\mathcal{O}(1)$ for the table lookup, whereas the upper bound for computational complexity in the latter case is $\mathcal{O}(k)$ additions which is similar to the construction of the first state. The evaluation and comparison of the subset sum puzzles is previously shown in Table 1.

4.2 Experimental Results

To demonstrate how LLL lattice reduction and the subset sum problems work in practice on client machines, we set up an experiment to create a random set of subset sum problems based on different criteria including density and a number of chosen items. In terms of hardware, we simulated the LLL reduction algorithm using a Sun Enterprise 420R computer operating with four UltraSPARC-II 450 MHz CPUs with 4096 MB of RAM running on Sun Solaris 9 (Sparc). We created MATLAB source code for generating random subset sum problems which have different densities between 0.3 and 0.8 for a range of instance sizes between 20 and 100. To solve these problems we wrote a subset sum solving function for testing the LLL implementation provided in MAGMA. The version of MAGMA installed on our testing machine was a full version patch number V2.13-11 released on April 5, 2007 (details at `http://magma.maths.usyd.edu.au`). The LLL version provided in MAGMA is based on the floating point arithmetic version (*FP-LLL*) proposed by Schnorr and Euchner [21].

The following briefly provides the methods that we used to evaluate our new scheme. Two different searching methods, a backtracking and a branch & bound algorithm [15], are taken into account for comparing with the LLL lattice reduction method.

Backtracking or Brute Force Searching: This is the simplest method which is also known as exhaustive search because it gathers all possible solutions and then checks for one satisfying the solution. This guarantees that it will always return an optimal solution. However, this technique consumes more CPU power as well as running time.

Branch & Bound Technique: To reduce the time of the brute force searching, *pruning techniques* can be used for avoiding some unnecessary nodes during the searching process. By storing and traveling only to states whose total weight does not exceed the limit, it can generate a specified solution faster than brute force. The branch & bound technique is one of those pruning methods. It specifies an upper bound on the output, so any descendant tracks having value above or not below their ascendant node will be eliminated from the possible solution. This can reduce running time and storage space.

LLL Lattice Reduction: This advanced tool, explained in Appendix A, can efficiently solve subset sum problems. This method can solve the subset sum puzzle within polynomial time rather than exponential time as the two previous techniques do. Recently, there have been many implementations for accelerating the running time of LLL reduction. In our experiment, we use two techniques: the first one, Int-LLL, is the original developed in 1982 by Lenstra et al. [13] provided in Mathematica, while the second one, FP-LLL, developed by Schnorr and Euchner [21], is a modified version using floating point arithmetic and provided in MAGMA.

Table 2 shows the experimental result compared among the brute force searching, branch & bound technique, and LLL Lattice Reduction examining puzzles having small size between 5 and 30.

Table 2. Average Running Time of The Subset Sum Puzzle on the specified methods

Number of Items	Average Running Time (seconds)								
	Backtracking			Branch & Bound			LLL		
(n)	Data 1	Data 2	Data 3	Data 1	Data 2	Data 3	Data 1	Data 2	Data 3
5	0.034	0.034	0.025	0.049	0.049	0.053	0	0	0
10	0.086	0.083	0.083	0.06	0.064	0.082	0	0	0
15	1.70	1.69	1.67	0.134	0.40	0.137	0	0	0
20	51.85	52.74	53.74	2.633	3.691	1.43	0	0	0.01
25	2320.70	2262.80	2428.60	315.743	456.97	602.81	0.01	0.01	0.01
30	–	–	–	1437.758	1865.001	1647.246	0.01	0.01	0.01

By evaluating the results from Table 2, we summarize that the reasonable range of puzzle difficulty would be at least 25 for preventing coordinated adversaries who can control a number of compromised machines to obtain puzzle solutions at the same capacity to the legitimate user performing LLL lattice reduction.

Table 3. The Experimental Result of The Subset Sum Puzzle

Number of Items	Average Running Time (seconds)																	
	Random Set 1						Random Set 2						Random Set 3					
	Density						Density						Density					
(n)	0.3	0.4	0.5	0.6	0.7	0.8	0.3	0.4	0.5	0.6	0.7	0.8	0.3	0.4	0.5	0.6	0.7	0.8
60	0.10	0.12	0.23	1.02	2.42	77.11	0.16	0.28	0.19	0.31	3.64	3.70	0.14	0.22	0.21	0.61	0.64	3.21
65	0.14	0.14	0.29	1.59	4.09	190.68	0.18	0.29	0.23	0.57	6.53	6.86	0.17	0.23	0.26	1.70	2.19	18.94
70	0.15	0.15	0.32	2.94	7.33	342.53	0.18	0.29	0.28	1.34	12.97	26.30	0.21	0.25	0.27	2.29	2.29	41.72
75	0.20	0.14	0.78	5.23	13.47	663.24	0.24	0.31	0.38	1.95	27.23	35.65	0.23	0.25	0.34	3.49	4.37	92.37
80	0.27	0.22	0.89	9.63	26.17	1745.97	0.25	0.33	0.52	2.75	58.70	87.12	0.26	0.29	0.45	5.66	8.82	226.76
85	0.37	0.25	1.24	17.38	49.22	4158.73	0.29	0.37	0.72	4.44	120.44	208.86	0.28	0.32	0.62	9.40	18.15	1315.29
90	0.50	0.29	1.63	31.44	96.39	9435.02	0.39	0.40	1.17	7.58	250.52	509.60	0.30	0.37	0.89	16.42	37.75	1344.35
95	0.59	0.34	2.34	55.68	173.30	21351.72	0.43	0.43	1.75	12.78	504.88	1158.45	0.36	0.43	1.28	28.14	79.36	3160.86
100	0.70	0.40	3.43	98.39	317.27	51124.86	0.46	0.47	2.87	21.45	1008.23	2737.79	0.41	0.50	2.03	46.63	168.72	7451.26

Before illustrating the second experimental result, we need to briefly explain the reasoning behind our configuration. By investigating the primary result comparing between FP-LLL and Int-LLL, we have found that Int-LLL works well for low density problems with data size below 100. Once the density grows, the Int-LLL performance drops gradually and becomes ineffective when we run it on high density examples. This behaviour was also observed by LaMacchia [12] as well as by Schnorr and Euchner [21]. Due to this degradation of Int-LLL with large instance and high density problems, we suggest to use FP-LLL in the puzzle solving to avoid the situation that legitimate users are unable to solve their puzzles. The reason is that a floating point arithmetic returns the Gram-Schmidt coefficient in the reduction process more precisely than integer arithmetic. As a result, the FP-LLL reduction provides a more correct output.

Table 3 shows the result of puzzles having size between 60 and 100. We restrict to this range because we are only interested in the values where the LLL performs faster than brute force searching, otherwise the protocol would be vulnerable to parallel attacks if the adversaries are able to run a brute force searching. The table shows that there is a good range of puzzle times suitable for practical use.

5 Discussion and Open Problem

As our main objective has been to design non-parallelizable puzzles, subset sum problems with the LLL lattice reduction bring us this characteristic and fulfill our requirement. However, simplicity and performance of the existing LLL schemes are a concern for deploying them in general applications. As several experiments have shown the failure of original LLL in dealing with large instances and high density problems, recently several attempts have been made to scale down the computation time of the size reduction process as well as increase the accuracy for dealing with the large instances. One example was using dynamic approximation and heuristic technique [3] to speed up the reduction process. To our knowledge, the fastest LLL reduction scheme for solving subset sum problems is the segmentation FP-LLL proposed by Schnorr [20] that minimizes the running time to be $\mathcal{O}(n^3 \log n)$.

Parallelization of the LLL lattice reduction was discussed and proposed by Villard [23]. The idea of that paper is to select non-overlapping parameters and separate them into two independent phases in order to speed up the exchange of parameters during the size reduction of the lattice basis. Thus, these outputs might be able to be computed in parallel by using $n \cdot m$ processors, and dividing them into n columns of m processors. Villard claimed that the running time complexity of this technique may be reduced to $\mathcal{O}(n^5 \log^3 B)$ binary arithmetic steps and $\mathcal{O}(n^4 \log^2 B)$ binary communication steps by using $\mathcal{O}(n)$ processors. This running time complexity could be improved by the factor of n by increasing the number of processors to $\mathcal{O}(n^2)$ units. However, the unclear practical efficiency of the algorithm and the requirement for the larger size of parameters than in the original LLL algorithm [13] mean that future investigation and development are required.

Another disadvantage of the subset sum puzzle is the memory requirement. By investigating instances when the item size n exceeds 100, we found that the memory resource is exhausted in some trials. That is because the LLL reduction constructs a $n \times n$ lattice matrix and allocates it into reserved memory. As a result, the practical range of puzzle difficulty would be up to $n = 100$ for avoiding memory exhaustion. In addition, the running time within this range would be reasonable and acceptable for most users. When we compare this bound with the hash-based reversal puzzles, the reasonable puzzle difficulty for hash-based reversal schemes would have k between 0 and 40 which results in a smaller length puzzle than our construction.

Since we are concerned with the problem of puzzle distribution and parallelizability, we focus on resolving the parallelizable characteristic rather than implementing linear granularity. However, even though our new scheme has coarser granularity than other hint-based schemes, it does offer polynomial granularity which is better than exponential granularity found in hash-based reversal puzzles recently used in some client puzzle protocols. As a result, our new design can be easier to control than many existing ones.

Comparing our construction with repeated squaring (Table 1) we find that, although repeated squaring offers non-parallelism and linear-grained control to the user, it suffers from high computation at construction time which means that a server using these puzzles would be susceptible to flooding attacks. As a result, an interesting open problem for the research community is to explore techniques to find new puzzles providing both non-parallelization and linear granularity.

References

1. Adleman, L M.: On Breaking Generalized Knapsack Public Key Cryptosystems. In: the 15th Annual ACM Symposium on Theory of Computing, pp. 402–412 (1983)
2. Aura, T., Nikander, P., Leiwo, J.: DoS-resistant authentication with client puzzles. In: Security Protocols Workshop 2000, Cambridge, pp. 170–181 (April 2000)
3. Backes, W., Wetzel, S.: Heuristics on Lattice Basis Reduction in Practice. Journal of Experimental Algorithmics (JEA) 7, 1–21 (2002)
4. Coster, M.J., Joux, A., LaMacchia, B.A., Odlyzko, A.M., Schnorr, C., Stern, J.: Improved low-density subset sum algorithms. Computational Complexity 2(2), 111–128 (1992)
5. Dwork, C., Naor, M.: Pricing via Processing or Combatting Junk Mail. In: Brickell, E.F. (ed.) CRYPTO 1992. LNCS, vol. 740, pp. 139–147. Springer, Heidelberg (1992)
6. Feng, W., Kaiser, E., Feng, W., Luu, A.: The Design and Implementation of Network Layer Puzzles. In: Proceedings of IEEE Infocom 2005 (March 13-17, 2005)
7. Feng, W., Luu, A., Feng, W.: Scalable, Fine-grained Control of Network Puzzles. Technical report 03-015, OGI CSE (2003)
8. Gao, Y.: Efficient Trapdoor-Based Client Puzzle System against DoS Attacks. In: Master of Computer Science by Research, School of Information Technology and Computer Science, University of Wollongong, Wollongong, Australia (2005)
9. Groza, B., Petrica, D.: On Chained Cryptographic Puzzles. In: 3rd Romanian-Hungarian Joint Symposium on Applied Computational Intelligence (SACI), Timisoara, Romania, pp. 25–26 (May 2006)
10. Juels, A., Brainard, J.: Client Puzzles: A Cryptographic Defense Against Connection Depletion Attacks. In: NDSS 1999. The 1999 Network and Distributed System Security Symposium, San Diego, California, USA, pp. 151–165. Internet Society Press, Reston (1999)
11. Lagarias, J.C., Odlyzko, A.M.: Solving low-density subset sum problems. Journal of the ACM (JACM) 32(1), 229–246 (1985)
12. LaMacchia, B.A.: Basis Reduction Algorithms and Subset Sum Problems. Master Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology (1991)
13. Lenstra, A.K., Lenstra Jr., H.W., Lovász, L.: Factoring Polynomials with Rational Coefficients. Mathematische Annalen 261(4), 515–534 (1982)
14. Ma, M.: Mitigating denial of service attacks with password puzzles. In: ITCC 2005. International Conference on Information Technology: Coding and Computing, 2nd edn., pp. 621–626 (2005)
15. Martello, S., Toth, P.: Knapsack Problems: Algorithms and Computer Implementations. John Wiley & Sons, Inc., Chichester (1990)
16. Moskowitz, R.: The Host Identity Protocol (HIP). Internet Draft, Internet Engineering Task Force (October 2007), `http://www.ietf.org/internet-drafts/draft-ietf-hip-base-09.txt`
17. Price, G.: A General Attack Model on Hash-Based Client Puzzles. In: 9th IMA International Conference on Cryptography and Coding, Cirencester, UK, pp. 16–18. Springer, Heidelberg (2003)
18. Radziszowski, S., Kreher, D.: Solving subset sum problems with the L^3 algorithm. Journal of Combinatorial Mathematics and Combinatorial Computing 3, 49–63 (1988)
19. Rivest, R.L., Shamir, A., Wagner, D.A.: Time-lock Puzzles and Timed-release Crypto. Technical Report TR-684, Massachusetts Institute of Technology, Cambridge, MA, USA (March 10, 1996)
20. Schnorr, C.P.: Fast LLL-type Lattice Reduction. Information and Computation 204(1), 1–25 (2006)
21. Schnorr, C.P., Euchner, M.: Lattice Basis Reduction: Improved Practical Algorithms and Solving Subset Sum Problems. In: Budach, L. (ed.) FCT 1991. LNCS, vol. 529, pp. 68–85. Springer, Heidelberg (1991)

22. Smart, N.: Cryptography: An Introduction, 2nd edn. McGraw-Hill, New York (2006)
23. Villard, G.: Parallel Lattice Basis Reduction. In: ISSAC 1992. The International Symposium on Symbolic and Algebraic Computation, pp. 269–277. ACM Press, New York (1992)
24. Wang, X., Reiter, M.K.: Defending Against Denial-of-Service Attacks with Puzzle Auctions (Extended Abstract). In: SP 2003. The 2003 IEEE Symposium on Security and Privacy, Berkeley, CA, USA, pp. 78–92 (May 11-13, 2003)
25. Waters, B., Juels, A., Halderman, J.A., Felten, E.W.: New Client Puzzle Outsourcing Techniques for DoS Resistance. In: CCS 2004. The 11th ACM Conference on Computer and Communications Security, pp. 246–256. ACM Press, Washington DC (2004)

Appendix

A A Brief Overview of Lattice Reduction

LLL lattice basis reduction is a polynomial time algorithm developed by Lenstra et al. [13] in 1982. The concept was originally used to solve the shortest vector problem (SVP) and closet vector problem (CVP) of a lattice. Adleman [1] seems to have been the first researcher to apply LLL lattice basis reduction as a cryptanalysis tool to successfully break the subset sum problem. By using the LLL, users simply treat the subset sum schemes as a lattice problem. Since its original use, many researchers have improved not only the performance of the algorithm, but also its accuracy when dealing with large instances of the lattice dimension.

LLL lattice basis reduction algorithm has been widely used in breaking subset sum cryptosystems because the algorithm is able to terminate in polynomial time. Moreover, it is highly sequential because an underlying program requires recursive computation. From this perspective, LLL is a promising technique to fulfill our requirement in terms of non-parallelizability and thwart coordinated adversaries from distributing the client puzzle to calculate the solution in a parallel manner. To explain the LLL lattice basis reduction, we refer to materials provided in Smart's book: *Cryptography: An Introduction (2^{nd} edition)* [22].

Definition 1. *Let $\{b_1, b_2, \dots, b_n\}$ be a set of vectors in $\mathcal{Z}^n$ that are linearly independent over $\mathcal{R}$. Then the set of all integer linear combinations of $\{b_1, b_2, \dots, b_n\}$ is called an integer lattice. In a formula:*

$$B = \left\{ \sum_{i=1}^{n} a_i \cdot b_i \mid a_i \in \mathcal{Z}, 1 \le i \le n \right\} \qquad (1)$$

Definition 2. *The **Gram-Schmidt** algorithm transforms a given basis $\{b_1, b_2, \dots, b_n\}$ into a basis $\{b_1^*, b_2^*, \dots, b_m^*\}$ which is pairwise orthogonal. The algorithm uses equations*

$$\mu_{i,j} = \frac{\langle b_i, b_j^* \rangle}{\langle b_j^*, b_j^* \rangle} \qquad for \quad 1 \le j < i \le n \qquad (2)$$

where $\mu_{i,j}$ is called a Gram-Schmidt coefficient.

$$b_i^* = b_i - \sum_{j=1}^{i-1} \mu_{i,j}\, b_j^* \qquad (3)$$

Definition 3. *A basis $\{b_1, b_2, \dots, b_m\}$ is called LLL reduced if the associated Gram-Schmidt basis $\{b_1^*, b_2^*, \dots, b_m^*\}$ satisfies*

$$|\mu_{i,j}| \le \frac{1}{2} \qquad for \quad 1 \le j < i \le m \qquad (4)$$

$$\|b_i^*\|^2 \ge \left(\frac{3}{4} - \mu_{i,i-1}^2\right) \|b_{i-1}^*\|^2 \qquad for \quad 1 < i \le m \qquad (5)$$

Equation (4), so called *size reduction*, ensures that we obtain a basis in which the vectors are short in length, while equation (5), the so called *Nearly Orthogonal Condition*, guarantees that the obtained vectors are nearly orthogonal. The LLL algorithm works as follows (also in Fig. 2);

1. We examine a fixed column k in which k starts at $k = 2$;
2. If equation (4) does not hold, we need to perform *size reduction* by modifying the basis B;
3. If equation (5) does not hold for column k and $k-1$ (it means the obtained vectors are non-orthogonal), we have to swap those columns and decrease a value of k by one (unless k is already equal to two). Otherwise, we increase k by one;
4. Once k reaches to m, the algorithm stops.

Since attacks on the subset sum problem using LLL reduction were proposed, there have been several experiments set up to compare the practical performance with the theoretical limits. The first such experiment was published by Radziszowski and Kreher [18] in 1988 to run a performance test of LLL on subset sum problems that have an item size (n) between 26 and 98 with different densities. The experimental result showed that when n grows up to 98, their implementation succeeded at density below 0.3 which is lower than the theoretical value proposed by Lagarias and Odlyzko [11]. Later, LaMacchia [12] set up an empirical test on problem sizes between 26 and 106. The result showed that the original LLL worked well for all problems with $n \le 26$ and density ≤ 0.6408, but the accuracy degraded quickly when n grows above 50. By running on the improved version, the performance was improved up to $n = 106$ with density 0.3. In the meantime, Schnorr and Euchner [21] proposed a way to speed up the reduction step by using floating point instead of integer arithmetic as in the original LLL, plus adding the deep insertion technique to their scheme. In comparison with