Practical Client Puzzle from Repeated Squaring

A.J.P. Jeckmans

August 24, 2009

Acknowledgements

My gratitude goes out to dr. Q. Tang, my thesis supervisor, for his support and guidance throughout my master thesis. Thanks also go out to prof. dr. P.H. Hartel for giving his advice on writing this thesis. Special thanks to my family for their continued support. More thanks go out to my friends and flatmates for their support.

Abstract

Cryptographic puzzles have been proposed by Merkle [15] to relay secret information between parties over an insecure channel. Client puzzles, a type of cryptographic puzzle, have been proposed by Juels and Brainard [8] to defend a server against denial of service attacks. However there is no general framework for client puzzle schemes. In this thesis we present a general client puzzle framework.

Since their introduction various types of client puzzles have been developed. One such client puzzle is based on the time-lock secret release scheme created by Rivest et al. [17]. The time-lock secret release scheme uses repeated squaring to hide information. Repeated squaring offers a deterministic puzzle that is not parallelizable by multiple computers. The drawback of the repeated squaring puzzle scheme is that it is computationally expensive and requires the server to keep state information.

In order to develop a generally practical client puzzle scheme the repeated squaring puzzle scheme has been modified. The computational complexity is improved by introducing batch verification. The scheme is also made stateless by altering the way puzzles are created. Analyses show that the result is a client puzzle scheme with more advantages.

Contents

1 Introduction **5**
1.1 Motivation 6
1.2 Organization 6

2 Preliminaries **7**
2.1 Standard Notation 7
2.2 Cryptographic Primitives 7
2.2.1 Hash Functions 7
2.2.2 Encryption 8
2.2.3 Discrete Logarithm 8
2.2.4 RSA 9
2.3 Client Puzzle Scheme Framework 9
2.4 Client Puzzle Evaluation Criteria 10
2.5 Conclusion 12

3 Existing Client Puzzle Schemes **13**
3.1 CPU-bound Puzzles 13
3.1.1 Hash-based Puzzle 14
3.1.2 Discrete Logarithm-based Puzzle 18
3.1.3 Repeated Squaring Puzzle 19
3.1.4 Subset Sum-based Puzzle 20
3.1.5 Conclusion 21
3.2 Memory-bound Puzzles 22
3.2.1 Function Look-up Puzzle 22
3.2.2 Pattern-based Puzzle 24
3.2.3 Conclusion 25
3.3 Comparison of Existing Schemes 26

4 Our Proposals **30**

4.1 Modifying Repeated Squaring 30

4.1.1 Modified Repeated Squaring #1 31

4.1.2 Modified Repeated Squaring #2 38

4.1.3 Combined Modified Repeated Squaring 39

4.2 Conclusion 40

5 Applications of the Proposed Puzzle Scheme **42**

5.1 Web Server Scenario 42

5.1.1 Analysis 44

5.1.2 Comparison 46

5.2 E-mail Server Scenario 49

5.2.1 Analysis 50

5.2.2 Comparison 51

5.3 Conclusion 51

6 Conclusion **52**

A Mathematica Programs **55**

A.1 Triggered Sequential Checking 55

A.2 Divide and Conquer Checking 56

A.3 Random Checking 58

Chapter 1

Introduction

Cryptographic puzzles are puzzles used to hide some secret which can only be uncovered after some computational effort has been made. They are first introduced by Merkle [15]. Cryptographic puzzles form the basis for client puzzles.

Client puzzles defend a server against DoS (Denial of Service) and DDoS (Distributed DoS) attacks. First proposed by Juels and Brainard [8], the client puzzle aimed at removing the limitations left by the solutions against DoS attacks at that time (connection time-out, random connection dropping and syncookies). Since then variations to this client puzzle and new types of client puzzles have been developed. The basic concept remains the same however. By asking the client to perform a simple calculation the server establishes that the client is willing to invest in the connection. This way the client has to use its own resources before the server has to commit its valuable resources to the connection.

There are two main classes of client puzzles. The larger class is that of the CPU-bound puzzle. To solve the puzzle an amount of CPU cycles have to be used by the client. Because of the great disparity in CPU power between various computers, a second class of memory-bound puzzles has been proposed. In this class a number of memory look-ups have to be done in order to solve the puzzle. In various types of client hardware there are smaller differences in memory speed than in processor speed. The solving speed is therefore less dependent on the clients hardware.

Next to these two classes other physical aspects of client hardware could also be used. For example the physical location of a device could serve as input for the puzzle [13].

1.1 Motivation

Most client puzzle schemes tailor to a specific goal, managing well on one requirement, but leaving others lacking. A generic scheme, which scores well on all fronts, could help speed up the widespread adoptation of client puzzles. While a perfect scheme might not present itself in the near future, or even exist, an attempt in this direction can be made. With the goal to meet or come close to as much of the evaluation criteria (Section 2.4), we will present a new client puzzle scheme.

1.2 Organization

The following chapter will present the basic information needed in the rest of the paper. The notations used and the cryptographic primitives are outlined. A general framework for client puzzle schemes is proposed and a standard set of requirements is also given. Chapter 3 will outline existing client puzzle schemes, with an analysis against the set of requirements. The client puzzle schemes will also be compared to each other.

The proposal for an altered client puzzle scheme will be detailed in Chapter 4, which takes the repeated squaring puzzle scheme [17] and improves it in the areas in which it is lacking. Next, Chapter 5 will present two applications in which the client puzzle scheme can solve a problem and outlines this further. The final chapter concludes and provides recommendations for future work.

Chapter 2

Preliminaries

This chapter outlines the basic elements used throughout the rest of this paper. First we introduce some notation, followed by the cryptographic primitives that are used. These aid in understanding the various client puzzle schemes that are presented in this paper. Second we propose a general framework for client puzzle schemes. It is a format to display a client puzzle scheme, which provides a handle to compare different schemes. The final element of this chapter will show the requirements that are imposed on the client puzzle schemes. The requirements are used to test and compare client puzzle schemes.

2.1 Standard Notation

In this thesis the following notations are used, based on the notations of [14]: Exclusive-OR is denoted by $\oplus$, (x, y, z) is the concatenation of the values x, y & z in that order. An integer interval is shown as $[a, b]$. $\stackrel{?}{=}$ represents an equality that has to be fulfilled or, depending on the context, is an equality test generating a 1 if true and a 0 otherwise.

2.2 Cryptographic Primitives

Below are the important cryptographic primitives used in this thesis [14].

2.2.1 Hash Functions

A cryptographic hash function, usually referred to as a hash function, is an efficient one-way operation mapping a variable length bitstring input to a

fixed length bitstring output, called a hash value. A keyed hash function is represented by $h_K(x)$, where K is the specific key used. K can be omitted when the hash function is not keyed. A typical cryptographic hash function has the following properties:

- When the same input is given to the hash function two times, the same output is given in both cases. A cryptographic hash function is a deterministic function.
- Given a hash value z it is computationally infeasible to find an input value x such that $h(x) = z$. This is known as preimage resistance.
- Given a value x it is computationally infeasible to find a value $x' \neq x$ such that $h(x) = h(x')$. This is known as second preimage resistance.
- It is computationally infeasible to find two distinct values x and y such that $h(x) = h(y)$. This is known as collision resistance.

2.2.2 Encryption

Symmetric encryption and decryption are represented by $E_K(x)$ and $D_K(x)$ respectively, where K is the symmetric key. Encryption followed by decryption with the same key yields the initial value, $D_K(E_K(x)) = x$. Asymmetric or public key encryption is represented by $P_{K_{pub}}(y)$ for encryption using the public key K_{pub} of an entity and $S_{K_{priv}}(y)$ for decryption using the private key K_{priv}. The public K_{pub} and private K_{priv} key form a key pair. Consequently public key encryption followed by private key decryption, using the same key pair, yields the initial value: $S_{K_{priv}}(P_{K_{pub}}(y)) = y$. The encryption and decryption functions have the following properties:

- Without knowing the key K it is computationally infeasible to decrypt a message using symmetric encryption.
- Without knowing the key K_{priv} it is computationally infeasible to do private key decryption.

2.2.3 Discrete Logarithm

The discrete logarithm problem is useful in public key cryptographic systems. Given a prime p, $g^x = h \mod p$ has a single solution for h, where $h \neq 0$. Furthermore given g, h, and p it is computationally infeasible to compute x. Another interesting feature is the fact that $(g^x)^y = (g^y)^x \mod p$. The order of the group attained by $g^x \mod p$ is denoted by $\phi(p) = p - 1$.

Any calculations done using the same generator g in mod p can also be rewritten and done without the generator in mod $\phi(p)$. For example, z in $g^{x*y} = g^z \mod p$ can be calculated as $x * y = z \mod \phi(p)$.

2.2.4 RSA

RSA is based on a slightly different property. Instead of a prime, as with discrete logarithm, the group is based on a non prime n that is constructed out of two primes p and q, $n = p * q$. Informally, the security of this system depends on the fact that it is computationally infeasible to calculate the group order $\phi(n) = (p-1)(q-1)$ given only n. Entities that do not know $\phi(n)$ have to do all calculations on this group the normal way. Entities that do know $\phi(n)$, the entity that created the group for example, can do certain calculations more efficiently. By first reducing the exponent formula to a single value, only one exponentiation will be needed. For example, by knowing $\phi(n)$, given x, y, z, the calculation $(g^x * g^y)^z \mod n$ can be simplified by first calculating $w = (x+y)*z \mod \phi(n)$ and then calculating $g^w \mod n$.

2.3 Client Puzzle Scheme Framework

Here we propose a new general framework that can be used to describe any specific client puzzle scheme.

Generally, a client puzzle system involves two entities: a server $\mathcal{S}$ and a client $\mathcal{C}$. The server $\mathcal{S}$ possesses the identifier $id_{\mathcal{S}}$ and the client $\mathcal{C}$ possesses the identifier $id_{\mathcal{C}}$. The identifiers of the server and the client are assumed to be public. This framework implies there is a protocol to pass messages between the server and the client. A client puzzle system consists of four (probabilistic) algorithms ($\mathsf{Setup}, \mathsf{PuzzleGen}, \mathsf{PuzzleSol}, \mathsf{PuzzleVer}$).

- $\mathsf{Setup}(\ell)$: Run by the server $\mathcal{S}$, this algorithm takes a security parameter ℓ as input, and outputs the public system parameter $params$ and a private key mk. The public system parameter $params$ is implicitly part of the input to any of the following algorithms. For reasons of simplicity, this system parameter is omitted in the descriptions.
- $\mathsf{PuzzleGen}(mk, req)$: Run by the server $\mathcal{S}$, this algorithm takes the private key mk and additional request information req received from the client $\mathcal{C}$ as input. It outputs a puzzle puz and some additional information $info$ the server requires for verification. The puzzle puz is sent to the client $\mathcal{C}$.

- PuzzleSol(puz): Run by the client $\mathcal{C}$, this algorithm takes a puzzle puz as input. It outputs a puzzle solution sol, which is consequently sent to the server.

- PuzzleVer($info, mk, sol$): Run by the server $\mathcal{S}$, this algorithm takes the puzzle information $info$, the private key mk and a puzzle solution sol as input. This algorithm outputs 1 if sol is correct or 0 otherwise.

2.4 Client Puzzle Evaluation Criteria

The evaluation criteria covers a set of four aspects, most of which have sub-aspects. Some evaluation criteria are more important in certain settings, depending on the environment. The aspects are:

- Computational complexity: The amount of work that the server has to do to carry out the client puzzle scheme. The server needs to create and verify puzzles. This process is represented by PuzzleGen and PuzzleVer. Computational work done for each puzzle should be as low as possible. With regard to the computational work we consider the following two specific sub-aspects:

  - Initial computation: Initial computation only has to occur at initialization of the client puzzle scheme and possibly upon changing the puzzle environment. Represented by Setup in the framework. This is different from puzzle construction and verification, which has to occur for each single puzzle.
  - Parallel computation resistance: The degree in which the client puzzle scheme provides resistance against parallelization. Parallelization significantly improves the solving speed for multiple clients.

- Hardness granularity: The steps with which the puzzle difficulty can be set. A fine-grained control over the puzzle difficulty is desirable, scaling the difficulty along with the threat level. We also consider the following specific sub-aspect.

  - Deterministic nature: The way a puzzle can be solved is either deterministic or probabilistic. A deterministic scheme will require the same amount of computations from the client each time. A

probabilistic scheme requires an amount of computations on average, but the actual work done by the client is likely to be either more or less.

- Storage space: The amount of storage required by the server to store data that is used in the puzzle scheme. This can be data that is always present when the scheme is active, for example the results of the initial computation, or data that is stored during or after the client solves a puzzle. In the framework this data is represented by $params$, mk and $info$. Memory usage should be as low as possible. With respect to the storage requirement we consider the following specific sub-aspects.

  - Long-term storage: The data that needs to be stored for the client puzzle scheme to create new puzzles. This information is stored in $params$ and mk in the framework.
  - Short-term storage: The additional data that needs to be stored for each instance of the puzzle. A puzzle scheme that requires short-term storage is stateful. This information is stored in $info$ in the framework. Note that multiple instances of $info$ may exist at the same time.

- Communication complexity: The communication complexity is the amount of bandwidth that is needed to transfer the puzzle and the corresponding solution between the client and the server. In the framework req, puz and sol have to be transmitted between the client and the server. A lower communication complexity means a lower overhead for the client puzzle protocol.

For some client puzzle environments certain criteria are more important than others. For example, a server that has little CPU power will require a client puzzle scheme that has a low computational complexity, having low generation and verification costs. Similarly a server that expects a high number of puzzle requests in a short amount of time will also require this. The same can be said for storage space. When a server is likely to be attacked by an adversary with access to multiple computers resistance against parallel computation is very important. Low communication complexity is very important in networks that are lossy and require frequent re-transmissions, or in networks that have low bandwidth.

2.5 Conclusion

In this chapter basic notations and cryptographic primitives have been summarized. Also we have proposed a new framework for client puzzle schemes and outlined a set of evaluation criteria. The client puzzle framework can be used to describe client puzzle schemes in a structured way and provide a handle when comparing schemes. The set of criteria contains four aspects, which also contain four sub-aspects. These aspects are used to rate client puzzle schemes and compare them.

Chapter 3

Existing Client Puzzle Schemes

In this chapter the existing client puzzle schemes are explained, evaluated and compared. Some puzzle schemes are grouped together, because they differ only slightly. A few client puzzle schemes have been omitted on the basis that they are only suited in an restrictive environment or application. For example, Martinovic et al. [13] created a client puzzle using wireless communication and location information, which only works with mobile wireless devices.

In the literature client puzzle schemes are divided into two main classes. CPU-bound puzzles make up the first class. Puzzles of this type require the client to perform a number of computations. The second class consists of puzzles that are bound by memory. In this class instead performing a number of computations, the client is asked to search through information in the memory.

3.1 CPU-bound Puzzles

CPU-bound client puzzles are puzzles based on an amount of computational effort the client has to do in order to solve the puzzle. The more difficult the puzzle, the more computation is required from the client. Described below are the client puzzle schemes that are bound by CPU, they are categorized by the techniques employed. This categorization is partly based on the overview of client puzzle schemes made by Tritilanunt et al. [18] and supplemented with more schemes and additional grouping.

3.1.1 Hash-based Puzzle

Description Several client puzzle schemes have been created in which the client is asked to reverse a cryptographic hash function. Because hash functions are one-way functions, brute forcing the reverse is computationally hard. To counter this Juels and Brainard [8] provide a part of the reverse as the puzzle and let the client look for the remaining ℓ bits.

- $\mathsf{Setup}(\ell)$: mk is a server secret, $h()$ is the chosen hash function and $params = \{\ell, h()\}$.
- $\mathsf{PuzzleGen}(mk, req)$: req contains the request message M. Let t be the current time. This information is hashed together with the server secret to form the basis of the puzzle, $x = h(mk, t, M)$. The first ℓ bits, the prefix, of x are replaced by 0 to form x'. The output is $puz = \{x', h(x), t, M\}$ and $info = \emptyset$.
- $\mathsf{PuzzleSol}(puz)$: The prefix of x' is replaced with another bit string to form a candidate solution z. When ignoring the prefix of z and x, z is equal to x. This candidate solution is checked using $h(z) \stackrel{?}{=} h(x)$. When the right value is found, this algorithm outputs the solution $sol = \{z, t, M\}$.
- $\mathsf{PuzzleVer}(info, mk, sol)$: first check that t is a recent timestamp. If this is not the case, the solution is rejected. Otherwise it is checked with $z \stackrel{?}{=} h(mk, t, M)$.

Aura et al. [2] propose another approach in which nonces and the client identifier as well as a variable of set length act as an input for the hash function. This proposed scheme is outlined below. The goal of the puzzle is to let the result have at least ℓ leading zero bits.

- $\mathsf{Setup}(\ell)$: $h()$ is the chosen hash function and $params = \{\ell, h()\}$ and $mk = \emptyset$.
- $\mathsf{PuzzleGen}(mk, req)$: puz is a server nonce and $info = puz$.
- $\mathsf{PuzzleSol}(puz)$: n is a client nonce and y is a random candidate solution. If the first ℓ bits of $h(id_{\mathcal{C}}, n, puz, y)$ are 0 the candidate solution is accepted. This algorithm then outputs $sol = \{n, y\}$.
- $\mathsf{PuzzleVer}(info, mk, sol)$: the solution is accepted when the first ℓ bits of $h(id_{\mathcal{C}}, puz, n, y)$ are 0.

Analysis The approach by Juels and Brainard requires two hash operations, one at puzzle construction and one at verification. Initial computation is limited to choosing a random bitstring. This random bitstring is the only value that has to be stored by the server. This scheme is stateless, because the server secret is the same for each puzzle. All other relevant information for verification, t and M, is sent to the client. The client then relays this information back to the server when a solution is submitted. The overhead caused by this scheme is size dependent on the chosen hash function. Two hash values, a timestamp and the original request message have to be transmitted to send the puzzle. To send the solution one hash value, a timestamp and the original request message are transmitted. The length of a hash value is, for example, 128 bits for MD5 and 160 bits for SHA-1. A timestamp is 32 bits and the original request message is a few bytes depending on the protocol.

The approach from Aura et al. is computationally faster requiring only one hash operation at puzzle verification. There is no initial computation, though the server nonce created at puzzle generation time can also be calculated at initial computation and can be used for multiple puzzles. Storage requirement is low, as only the identifier of the client, a corresponding nonce and the server nonce have to be stored. This is to prevent replay attacks. The puzzle is stateful, for each puzzle a server nonce has to be stored. It can be made stateless by using a general server nonce. However again two nonces and the client identifier has to be stored to prevent replays. Overhead is minimal: Two nonces and a random bitstring. Aura et al. suggest a minimal 64 bits entropy for the server nonce and a minimal entropy of 24 bits for the client nonce. The nonces could also be used towards achieving other security goals. The size of y is assumed to be around 64 bits.

What both approaches have in common is that both can be computed in parallel by multiple clients working together. Increasing the number of clients working on the solution will decrease, by the same factor, the estimated time in which the solution will be found. Communication overhead between clients and synchronization issues are not taken into account here. Both solutions also lack in hardness granularity. A puzzle with security parameter $\ell + 1$ has twice the difficulty of a puzzle with security parameter ℓ. The resulting hardness granularity is exponential in ℓ. Lei et al. [11] suggest a bit more fine grained solution in which the last four bits of the ℓ leading bits don't have to be zero, but only below a certain threshold. This increases the control over the puzzle difficulty, but is still growing at an exponential rate. These puzzle schemes are probabilistic in nature and a successful solution could be created in one single hash operation.

Parallel Hash Puzzle

Description Along with their original scheme of a single hash puzzle, which is the first scheme in this section, Juels and Brainard [8] already opt to have multiple hash-based puzzles in parallel to get a better puzzle hardness granularity. Because this scheme is identical to the original, except for the fact that several puzzles have to be solved in parallel, specific details have been omitted.

Analysis The number of computations done by the client and server, as well as the overhead generated by the scheme, is multiplied by the number of puzzles that have to be solved in parallel. The advantage of this scheme is that there is a finer-grained control over the hardness granularity. By changing the number of puzzles the client has to solve, the difficulty changes in linear fashion. The resulting difficulty is the number of puzzles times the difficulty of a single puzzle.

Hinted Hash Puzzle

Description Feng et al. [5] propose another alternative solution to improve puzzle granularity. When providing the client with a puzzle, the server should also provide the client with a hint to the range the solution is in. The client then only needs to search within this range for the puzzle solution.

- $\mathsf{Setup}(\ell)$: mk is a server secret, $h()$ is the chosen hash function and $params = \{\ell, h()\}$.
- $\mathsf{PuzzleGen}(mk, req)$: req contains the request message M. Let t be the current time. This information is hashed together with the server secret to form the basis of the puzzle, $x = h(mk, t, M)$. A random number r is chosen in the range $[0, \ell]$, this random number together with x is used to form the range of possible solutions $[x - r, x - r + \ell]$. The output is $puz = \{x - r, h(x), t, M\}$ and $info = \emptyset$.
- $\mathsf{PuzzleSol}(puz)$: A candidate z is chosen from $[x - r, x - r + \ell]$. This candidate solution is checked using $h(z) \stackrel{?}{=} h(x)$. When the right value is found, this algorithm outputs the solution $sol = \{z, t, M\}$.
- $\mathsf{PuzzleVer}(info, mk, sol)$: first check that t is a recent timestamp. If this is not the case, the solution is rejected. Otherwise it is checked with $z \stackrel{?}{=} h(mk, t, M)$.

Analysis When compared to the scheme of Juels and Brainard, this scheme has one main difference. The hardness granularity has become linear over ℓ, due to the range that has to be searched. Instead of having to search the entire possible hash space only a segment of size ℓ has to be searched. Control over ℓ is linear and so the hardness granularity is also linear over ℓ. This grants a fine-grained control over the puzzle difficulty. It is however still probabilistic in nature. When a puzzle is constructed instead of replacing ℓ bits, one random number has to be chosen and one subtraction needs to be done. This change in puzzle construction has little impact on computational complexity. In puzzle transmission the hash value for x' has changed into a range, however since ℓ is known to the client only the first value of the range has to be transmitted. The other aspects remain unchanged.

Chained Hash Puzzle

Description Groza and Petrica [7] present a way to chain cryptographic puzzles, more specifically hash puzzles.

- $\mathsf{Setup}(\ell)$: the number of puzzles in the chain is n, $h()$ is the chosen hash function. $params = \{\ell, n, h()\}$ and $mk = \emptyset$.
- $\mathsf{PuzzleGen}(mk, req)$: For each puzzle in the chain a random starting point is chosen. These starting points are $x_i, 1 \leq i \leq n$. The first puzzle is created by hashing the starting point twice, $y_1 = h(h(x_1))$. Subsequent puzzles are created differently. They are based on the puzzle before it as well as on its own solution. This is done using $y_i = h(h(x_i \oplus h(x_{i-1}))), 2 \leq i \leq n$. The last ℓ bits of all starting points are changed into 0, creating $x'_i, 1 \leq i \leq n$ out of $x_i, 1 \leq i \leq n$. The output is $puz = \{x'_i, y_i\}, 1 \leq i \leq n$ and $info = \{x_i\}, 1 \leq i \leq n$.
- $\mathsf{PuzzleSol}(puz)$: The puzzles have to be solved in order, first a candidate z_1 for x'_1 is created by replacing the last ℓ bits by another bit string. The candidate is checked with $y_1 \stackrel{?}{=} h(h(z_1))$. When the first puzzle is solved, a candidate $z_i, 2 \leq i \leq n$ for the next puzzle is created. The new candidate is checked using $y_i \stackrel{?}{=} h(h(z_i \oplus h(z_{i-1}))), 2 \leq i \leq n$. This process is repeated until all puzzles are solved. The solution will be given by $sol = \{z_i\}, 1 \leq i \leq n$.
- $\mathsf{PuzzleVer}(info, mk, sol)$: For each solution $z_i \stackrel{?}{=} x_i, 1 \leq i \leq n$ is checked.

Analysis In order to create the first puzzle, two hash operations are needed. All other puzzles require three hash operations. Next to that a bit string of length ℓ has to be replaced for each puzzle and $n-1$ exclusive-OR operations are needed, but this is cheap. To verify, a comparison is needed for each puzzle in the chain. There is no initial computation. Because the puzzles are linked in a chain, there is some resistance to parallel processing [18]. Each step in the chain depends on the solution of the previous step, however clients can still work together to speed up each individual step. Hardness granularity for this scheme follows the same line as for parallel hash puzzles. The difficulty per puzzle can be altered in an exponential way. The length of the chain n alters the overall difficulty linearly, with a step size equal to the individual puzzle difficulty which is exponential over ℓ. Again this is a probabilistic scheme. This stateful scheme requires a hash value to be stored for each of the puzzles in the chain. For each client the hash values of one entire chain need to be stored, this takes n hash values worth of storage space. With large chains or many clients the server has a great chance to experience resource exhaustion. The overhead of this scheme is linked to the length n of the chain. Three hash values have to be sent back and forth for each puzzle in the chain, $3n$. When n is large, bandwidth exhaustion can become an issue.

3.1.2 Discrete Logarithm-based Puzzle

Description Another one-way function on which a puzzle can be based is a function based on the discrete logarithm problem. By combining this with a hint based scheme, Waters et al. [19] have created a client puzzle solution. The version described here is a simplified version without a central puzzle distributing authority. The principle however remains the same.

- $\mathsf{Setup}(\ell)$: select a random prime q. It also selects a generator g from the range $[2, q-1]$. $params = \{\ell, g, q\}$ and $mk = \emptyset$.
- $\mathsf{PuzzleGen}(mk, req)$: take a random number r between 0 and $q-1$. x is then randomly chosen over the range $[r, r+\ell \mod q-1]$. $puz = \{r, g^x\}$ and $info = x$.
- $\mathsf{PuzzleSol}(puz)$: take a candidate solution z from $[r, r+\ell \mod q-1]$. Test this candidate with $g^x \stackrel{?}{=} g^z \mod q$. $sol = z$.
- $\mathsf{PuzzleVer}(info, mk, sol)$: $sol \stackrel{?}{=} info$.

Analysis This scheme is computationally expensive due to the modular exponentiation during puzzle creation. All other calculations in the scheme, including the setup, are cheap. There is no defense against parallel computation as clients are given a range in which to search. Due to this search range the hardness granularity of the scheme is linear over ℓ and gives it fine-grained control, however in a probabilistic way. Little storage space is needed, there are a few parameters and for each new puzzle one value has to be stored. This makes the scheme stateful. Communication complexity is minimal as only two values need to be sent to transmit the puzzle and only one to send the solution. One such value is assumed to be around 64 bits, depending on q. Gao et al. [6] suggest an alternative approach in which several puzzles and solutions are pre-computed during the start up phase. The pre-computed puzzles, when a new puzzle is needed, are slightly altered by a a variable linked to the time to keep puzzles fresh. This results in a lower cost for puzzle creation. The drawback is that more memory is needed and more initial calculations have to be done.

3.1.3 Repeated Squaring Puzzle

Description Rivest et al. [17], in their pursuit to send information into the future, created a way to hide information that can only be retrieved after a fixed amount of computations. This secret release scheme, called a time-lock, could be used as a client puzzle. The time-lock puzzle ensures that the client puts the right amount of effort into it. Based on the principles of RSA, the client is asked to do an amount of squarings modulo n.

- $\mathsf{Setup}(\ell)$: $params = \ell$ and $mk = \emptyset$.
- $\mathsf{PuzzleGen}(mk, req)$: this algorithm selects two random large primes p, q and creates $n = p * q$ out of this. Then it chooses a generator g out of the range $[2, n-1]$ and outputs the puzzle $puz = \{n, g\}$ and $info = \{g, p, q\}$.
- $\mathsf{PuzzleSol}(puz)$: $sol = g^{2^\ell} \mod n$.
- $\mathsf{PuzzleVer}(info, mk, sol)$: $sol \stackrel{?}{=} g^{2^\ell \mod \phi(n)} \mod n$.

Repeated squaring is considerably faster than factoring n and guarantees that the client will do the squaring.

Analysis The highest computational cost of this scheme is in the verification where two modular exponentiations are calculated, which is very expensive. However puzzle creation is cheap, two large primes p, q need to be chosen and multiplied and a generator g has to be found. Initial computation is non existent. This scheme also does not allow parallelization by clients as each computational step in the solving process depends on the previous one. Hardness granularity is good; changing the difficulty of the puzzle requires the client to respectively do a greater or a lesser amount of modular squarings. Control is linear and client solve time can be determined beforehand if the modular squaring speed of the client is known. This is due to the fixed number of calculations that have to be done. This fixed amount of calculations also makes this a deterministic client puzzle scheme. Two large primes p, q and a generator g need to be stored for each client puzzle. This scheme is therefore stateful. During the communication between server and client a large number n, corresponding generator g and solution of equal length are passed between them. The exact size of the values p, q, n and g are undefined, but should be large enough to make the factoring of n computationally infeasible, which is at least 1024 bits.

3.1.4 Subset Sum-based Puzzle

Description In the pursuit of creating a non-parallelizable client puzzle Tritilanunt et al. [18] coined the idea of using the subset sum problem as a client puzzle. The subset sum problem, which is detailed in [14], is as follows. Consider a set of positive integers $a_i, 1 \leq i \leq n$ and another positive integer s. Determine if there exists a subset of $a_i, 1 \leq i \leq n$ that sums up to s. The currently fastest algorithm to solve this problem, the LLL algorithm [12], is non-parallelizable. Brute-forcing possible combinations is parallelizable, but far less efficient.

- $\mathsf{Setup}(\ell)$: this algorithm takes $n \geq \ell$ as the number of items in the possible weight set and calculates $w_i, 1 \leq i \leq n$, a set of random weights. This is done by taking the first item w_1 as a random number and each subsequent item as a hash, using hash function $h()$, of the previous, $w_i = h(w_{i-1}), 2 \leq i \leq n$. The server then selects a server secret sk at random from $\mathbb{Z}_n$. The output is $params = \{\ell, w_1, h()\}$ and $mk = \{sk, w_i\}, 1 \leq i \leq n$.

- $\mathsf{PuzzleGen}(mk, req)$: req contains a nonce sent by the client, $N_{\mathcal{C}}$. The server generates a fresh nonce $N_{\mathcal{S}}$. This is used along with the client and servers identity and the server secret to generate a solution to

the problem. The ℓ least significant bits of $h(id_{\mathcal{C}}, N_{\mathcal{C}}, id_{\mathcal{S}}, N_{\mathcal{S}}, sk)$ are stored in $info$. $info_i, 1 \leq i \leq \ell$ represents the i-th bit of $info$. Out of this solution the total weight of the subset is calculated to construct the puzzle, $puz = \sum_{i=1}^{\ell}(info_i * w_i)$.

- PuzzleSol(puz): first the client computes the weight set, $w_i = h(w_{i-1}), 2 \leq i \leq \ell$. Using the LLL algorithm the client solves the subset sum problem received from the server, this generates a solution sol. Before transmitting the solution it is checked against the puzzle, $puz \stackrel{?}{=} \sum_{i=1}^{\ell} sol_i * w_i$.

- PuzzleVer($info, mk, sol$): $sol \stackrel{?}{=} info$.

Analysis Computational complexity of this scheme is low. To create and verify a puzzle only one hash operation and some basic arithmetic is needed. The initial computation requires more computations to be done as $n-1$ hash operations are needed to create a set of weights. The LLL algorithm used by the client is non-parallelizable. Parallelization is possible, but only by using the brute force method which is considerably slower. Because the LLL algorithm has a polynomial complexity over the size ℓ of the weight set and the weight set itself $w_i, 1 \leq i \leq \ell$, the granularity of the puzzle is also polynomial over them [18]. Using the LLL algorithm also makes this a deterministic scheme. The granularity of brute forcing is exponential over ℓ. The storage space needed for this scheme is the space needed by the set of weights, n hash values, and a server secret. The state of each puzzle must also be stored, this takes ℓ bits. Communication overhead is minimal: the server only needs to send the first item in the weights set to transmit the entire set, which is a single hash value. To send the puzzle the server needs a few more bits than a hash value to send the total weight. Sending the solution only requires ℓ bits.

3.1.5 Conclusion

Much research has been done towards hash-based client puzzles [2, 5, 7, 8]. Most drawbacks in the original scheme by Juels and Brainard [8] can be countered. Hardness granularity can be improved by giving the client a range to search in. The scheme can be made stateless by sending the required information to the client along with the puzzle and have the client send this information back with the solution. Protection against parallel computation by clients has only been partly found in the form of a hash chain. There is however no fix yet for the probabilistic nature of hash reversal. The discrete

logarithm puzzle scheme [19] has the same properties with respect to the requirements, but is also having the downside of being more costly for the server than a hash-based scheme.

Repeated squaring [17] shows promise by having an inherent resistance against parallel computation. It is also deterministic in nature, giving another advantage. The drawbacks of this scheme are the high computational cost, requiring modular exponentiations, and high storage requirements, the server needs to store RSA modulus. The subset sum puzzle scheme [18] defends against parallel computation by offering a solving algorithm, the LLL algorithm, that is non-parallelizable and faster than brute-forcing for a solution. The subset sum-based puzzle scheme is computationally cheap. The LLL algorithm used in solving the puzzle is a deterministic algorithm. However the hardness granularity of the subset sum puzzle scheme and the LLL algorithm are both polynomial in nature. Both also share the drawback of being stateful.

3.2 Memory-bound Puzzles

Client puzzles that are based on memory use have been created because there is less disparity in memory speeds than in processing power. These types of puzzles should be better for an environment in which hardware with a large speed range is employed. The client has to pre-compute a database of information, which can later be used to solve the puzzle. The client is asked to perform several searches in this database. Because this database is very large, memory handling has to occur. For all these puzzles it is possible to break them using computations only, but this is considerably slower. Below are the client puzzle schemes bound by memory.

3.2.1 Function Look-up Puzzle

Description Memory can be used to speed up certain actions. For example, a look-up table can be used to speed up reversing a one-way function. Based on this idea Abadi et al. [1] created a puzzle scheme.

- $\mathsf{Setup}(\ell)$: select a one-way many-to-one function $f()$, with input and output having bit length n, and with $f^{-1}()$ representing the inverse. Outputs $params = \{\ell, f()\}$ and $mk = \emptyset$.

- $\mathsf{PuzzleGen}(mk, req)$: the first puzzle element x_0 is a random bitstring with length n. All elements after that are calculated with $x_i =$

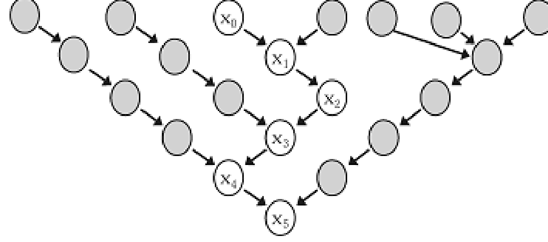

Figure 3.1: Function look-up tree structure

$f(x_{i-1}) \oplus (i-1), 1 \leq i \leq \ell$. A checksum s is computed by concatenating all elements and then applying a checksum function, $s = checksum(x_0, x_1, \ldots, x_{\ell-1}, x_\ell)$. This algorithm outputs $puz = \{x_\ell, s\}$ and $info = x_0$.

- PuzzleSol(puz): it is assumed that the client has pre-computed a look-up table for inverting the function $f()$ and is using this table to solve the puzzle. The client calculates all elements from $x'_\ell = x_\ell$ back to x'_0 using the look-up table and $x'_i = f^{-1}(x'_{i+1} \oplus i), 0 \leq i \leq \ell - 1$. The checksum of the elements in $x'_i, 0 \leq i \leq \ell$ is checked against s and if this matches $sol = x'_0$ is output.
- PuzzleVer($info, mk, sol$): $info \stackrel{?}{=} sol$.

For the function $f()$ Abadi et al. suggest a random function, but this could for instance also be a hash function or a table look-up function. Because the function $f()$ is a many-to-one function, the client will have to backtrack through multiple paths to find the correct solution. Due to this branching the search space will be a tree, rather than a chain. An example for this can be found in Figure 3.1.

Analysis The computational complexity of the server is dependent on the chosen function $f()$. Creating a puzzle requires ℓ applications of the $f()$ and a cheap checksum function. Initial computation consists of selecting a function from a collection of functions. Multiple clients can help solve the puzzle by taking different paths in the tree. Because the tree branches, while being narrow at the base, doubling the number of clients does not automatically halve the solving time. Parallel computation of the solution by the client loses effectiveness as the number of clients increases. The

granularity of this scheme is dependent on the chosen function $f()$ and the depth of the tree ℓ. When there are many collisions for $f^{-1}(x_i), 1 \leq i \leq \ell$, there are many branches that have to be searched. When ℓ is larger more applications of $f^{-1}()$ have to be made to get to the solution. The hardness granularity is therefore polynomial over ℓ, with $f^{-1}()$ determining the order. This is a probabilistic scheme, because there is a tree that has to be searched and wise choices in the branch reduce running time. This stateful scheme requires only one value to be stored per puzzle with size n bits. Besides this it is likely that the server has a collection of possible functions stored. The communication overhead of this puzzle is small as only two values with size n bits and one checksum value need to be transmitted.

3.2.2 Pattern-based Puzzle

Description Doshi et al. [4] suggest a scheme based on solving the sliding tile problem. In this problem there are several tiles on a grid which have to be set to a certain position and one empty space to slide to. The client has to find a specific, possibly not optimal, path from a starting state to a goal state. Because the path is not required to be optimal, existing computational algorithms for finding the shortest path can not always be used to find the solution. Using pattern databases the computation time can be reduced by trading in memory. For this scheme to work, the grid has to be large enough, so that it can not be stored in cache. There is a further need for random access in the memory, this forces the client to swap out memory pages. To achieve this multiple puzzles have to be solved simultaneous. Because the puzzles are in a different state a different part of the memory needs to be addressed.

- $\mathsf{Setup}(\ell)$: this algorithm generates a set G with possible goal states of the puzzle and a secret server key sk used as a key for hashing. The security parameter ℓ is split up in two factors, ℓ_1 the number of puzzles to solve, and ℓ_2 the number of moves per puzzle. Then the algorithm outputs $params = \{\ell_1, \ell_2\}$ and $mk = \{G, sk\}$.
- $\mathsf{PuzzleGen}(mk, req)$: the server chooses ℓ_1 goal states G' from the set G for the client to solve. The server will apply moves to the goal states in the reverse order in which the client will solve them, starting at the goal states and ending at the starting states. Then simultaneous for all goal states G' a random move is applied and a checksum $s_i, \ell_2 \geq i \geq 1$ is computed over the reverse of these moves. Concatenate these reverse moves to the front of the collection of all moves M. This results in a

new set of states on which the next step is applied, this application of moves is then repeated for a total of ℓ_2 times. The result of these steps is the set of starting states S. Finally using the collection of moves M and the current time t a verification value v is computed using $v = h_{sk}(t, M)$. This algorithm then outputs $puz = \{G', S, s_i, v, t\}, 1 \leq i \leq \ell_2$ and $info = \emptyset$.

- PuzzleSol(puz): for all starting states in S apply a move and check the checksum $s_i, 1 \leq i \leq \ell_2$. When the checksum matches concatenate the moves to the collection of all moves M'. This is continued until all goal states G' are reached in paths of the same length with all checksums matching. At this point M' should be identical to M. The solution is then output $sol = \{v, t, M'\}$.
- PuzzleVer($info, mk, sol$): $v \stackrel{?}{=} h_{sk}(t, M')$.

Analysis Creating a puzzle is expensive as $\ell_1 * \ell_2$ moves have to be applied, ℓ_2 checksums have to be calculated and one hash operation is needed. Verifying a puzzle only requires one hash operation. There is not much initial computation needed, the server only has to create a set of goal states. In the puzzle the moves of the solution are based on the previous moves and the client is forced to solve the puzzle one layer at a time. This way the client has to constantly use other parts of the memory database. There is a bit of resistance against parallelization. By working together on the same layer, or by brute forcing the checksum in advance, it is possible for clients to work together. Hardness granularity for this scheme is polynomial over ℓ_1 and ℓ_2. Increasing one will result in an increase with a step size equal to the other. Again this is a scheme that is probabilistic in nature. The only storage needed by the server is a set of goal states along with the server key. There is no additional information stored after a puzzle has been created as a verification ticket is sent to the client along with the puzzle. This makes the scheme stateless. Communication overhead is large: to send the puzzle two sets of states have to be transmitted, along with ℓ_2 checksums, one hash value and one timestamp. To send a solution one hash value, one timestamp and $\ell_1 * \ell_2$ moves have to be transmitted. A move can be represented using 2 bits, but a high number of moves can significantly enlarge the total size.

3.2.3 Conclusion

The memory bound puzzle schemes [1, 4] are probabilistic in nature and either have a high computational complexity or overhead. There is a small

amount of protection against parallel computation.

A common problem shared by memory bound schemes is that the size of the memory that is available to the client alters the speed at which the client can solve the puzzle. If the memory database used by the client is smaller than the available memory the client is not forced to load new pieces of the database into the memory. Loading database pieces into the memory slows down the client in solving the puzzle. Similarly when all items that need to be looked up are clustered together, there is no need to load other pieces of the database into the memory.

Because no valid assumption can be made about the size of the memory that is available to the client, or the way that the client has build the database, the server can not force the client to load new pieces of the database into the memory and slow it down. Some clients might have to load new pieces often and be slowed, creating a difference in speed between these clients and clients that do not need to load new pieces.

3.3 Comparison of Existing Schemes

Table 3.1 shows a performance overview of the existing client puzzle schemes, the notation for this table can be found in Table 3.2. Computational complexity is split into puzzle creation and puzzle verification. Initial computation, puzzle creation and puzzle verification are expressed in computational operations, low cost operations are omitted. Parallel computation resistance represents the opportunity for clients to do parallel computation, this is marked with yes, some or no, with yes being better. Hardness granularity is represented by either linear, polynomial or exponential, with linear being better and exponential being worse. Deterministic is marked with either yes or no, with yes being better. The size needed for storage and transmitting puzzles and solutions is expressed in the size of the security parameter k, the size of a hash values h and the size of an RSA number r. The values ℓ and n represent the puzzle difficulty and number of puzzles respectively. The values s and t represent the upper bound for ℓ and n.

We have the following observations:

- Very few schemes use initial computation. Some schemes use the initial computation to select some values, but only the subset sum-based scheme does some complex calculations in this period. Gao et al. [6] have suggested a method in which puzzles are pre-computed in this phase and later modified to add uniqueness to the puzzles, this results

Client Puzzle Scheme	IC	PC	PV	PR	HG
Hash-based #1	–	$2H$	H	no	exponential
Hash-based #2	–	–	H	no	exponential
Parallel Hash	–	$2n * H$	$n * H$	no	polynomial
Hinted Hash	–	$2H$	H	no	linear
Chained Hash	–	$2n * H$	–	some	polynomial
Discrete Logarithm	–	E	–	no	linear
Repeated Squaring	–	M	$2E$	yes	linear
Subset Sum	$(s-1)H$	$H + \ell * M$	–	yes	polynomial
Function Look-up	–	$(\ell - 1)F + C$	–	some	polynomial
Pattern-based	–	$\ell * C + H$	H	some	polynomial
Client Puzzle Scheme	**DN**	**LS**	**SS**	**CC**	
Hash-based #1	no	k	–	$3h + 2k + 2k^2$	
Hash-based #2	no	–	$3k$	$3k$	
Parallel Hash	no	k	–	$3n * h + 2k + 2k^2$	
Hinted Hash	no	k	–	$3h + 2k + 2k^2$	
Chained Hash	no	k	$n * h$	$3n * h$	
Discrete Logarithm	no	k	k	$3k$	
Repeated Squaring	yes	–	$3r$	$3r$	
Subset Sum	yes	$s * h + k$	h	$3h$	
Function Look-up	no	–	k	$3k$	
Pattern-based	no	$(t+1)k$	–	$h + (2n + \ell + 1)k$	

Table 3.1: Performance of Existing Client Puzzle Schemes

Header		Operation	
IC	Initial Computation	H	Hash Function
PC	Puzzle Creation	M	(Modular) Multiplication
PV	Puzzle Verification	E	Modular Exponentiation
PR	Parallel Computation Resistance	C	Checksum Function
HG	Hardness Granularity	F	One-way Many-to-One Function
DN	Deterministic Nature	**Size**	
LS	Long-term Storage	k	Security Parameter
SS	Short-term Storage	h	Hash Value
CC	Communication Cost	r	Modulus of RSA
Value			
ℓ	Puzzle Difficulty	s	Maximum Puzzle Difficulty
n	Number of Puzzles	t	Maximum Number of Puzles

Table 3.2: Notation for Table 3.1

in an increase of storage space that is needed in order to store these pre-computed puzzles.

- Most schemes fail to provide resistance against parallelization. While the chaining of puzzles offers some resistance against parallelization, each step of the chain is still vulnerable to parallelization if the puzzle on which it is based is vulnerable to parallelization.

- Most schemes are not deterministic in nature. Schemes that are probabilistic in nature have a natural vulnerability to parallelization.

- A scheme can be made stateless by sending the information required to recompute the solution, except for the server secret, to the client. The client can then send this information back to the server along with the solution and the server is able to verify it. This adds to the communication cost. On the other hand storing this information on the server reduces communication cost.

- When the solution is also computed during puzzle creation, storing it instead of having to recompute it reduces the cost for puzzle verification. The drawback is that this information has to be stored for each puzzle resulting in a stateful client puzzle scheme.

- Most client puzzle schemes can be given linear hardness granularity by sending the client a search space in which to find the answer. The hardness then becomes linear over this search space. The drawback of this is that the scheme becomes probabilistic in nature and loses any resistance against parallelization if it had any to begin with. This has been done with the hinted hash puzzle scheme, which is based on hash-based scheme #1. The cost of the puzzle scheme remained the same. Also because hash-based scheme #1 was already probabilistic in nature and had no resistance against parallelization this did not change either. In the evaluation only the hardness granularity changed and did so from exponential to linear.

- By increasing the number of puzzles a client has to solve the hardness granularity of the puzzle scheme is improved. The drawback of this is that more puzzles have to be created and verified. This gives a cost increase in puzzle construction and/or puzzle verification. It could also result in more data that has to be stored for each client. Communication cost will always increase as more puzzles will have to

be send to the client and more solutions will have to be received from the client.

By having a deterministic nature and resistance against parallel computation the repeated squaring and the subset sum-based scheme show promise to have high evaluation marks with regard to the criteria as detailed in Section 2.4. The main differences between the two is that the subset sum puzzle scheme already has a personalized client puzzle, linking the puzzle to the client using the clients identifier. There is also the use of a master key during puzzle creation in the subset sum puzzle scheme, linking the puzzle to the server. The repeated squaring puzzle scheme does not have these linking elements, giving this puzzle scheme more options for alterations.

Chapter 4

Our Proposals

This chapter will outline our proposal for an improved client puzzle scheme. The aim of this new scheme is to be generic, practical and to meet the criteria as detailed in Section 2.4. Repeated squaring [17] was chosen for the basis of this new scheme, because it has inherent resistance against parallel computation and a deterministic nature. Furthermore repeated squaring gives the opportunity to be modified.

The original repeated squaring scheme is repeated below.

- $\mathsf{Setup}(\ell)$: $params = \ell$ and $mk = \emptyset$.
- $\mathsf{PuzzleGen}(mk, req)$: this algorithm selects two random large primes p, q and creates $n = p * q$ out of this. Then it chooses a generator g out of the range $[2, n-1]$ and outputs the puzzle $puz = \{n, g\}$ and $info = \{g, p, q\}$.
- $\mathsf{PuzzleSol}(puz)$: $sol = g^{2^\ell} \mod n$.
- $\mathsf{PuzzleVer}(info, mk, sol)$: $sol \stackrel{?}{=} g^{2^\ell \mod \phi(n)} \mod n$.

4.1 Modifying Repeated Squaring

There are a few drawbacks with the repeated squaring scheme that need to be addressed.

- The cost of verifying a puzzle is high.
- The protocol is stateful.

Here one method is proposed to reduce computational cost and one to remove statefulness. These methods are then combined into one client puzzle scheme.

4.1.1 Modified Repeated Squaring #1

To reduce the computational cost of the repeated squaring puzzle scheme, multiple solutions can be verified in one pass. It is assumed that m clients have submitted solutions and all associated puzzles have the same difficulty k and group size n. Because the difficulty of all puzzles is the same, the exponent $2^k \mod \phi(n)$ only has to be calculated once. The fact that this exponent is the same for all puzzles can be used to verify multiple solutions quicker. Multiple puzzle generators can be multiplied and then exponentiated in one pass, this value can consequently be verified by multiplying the corresponding solutions.

$$(\prod_{i=1}^{m} g_i)^{2^k \mod \phi(n)} \mod n \stackrel{?}{=} \prod_{i=1}^{m} sol_i \mod n \tag{4.1}$$

Instead of m modular exponentiations, $2(m-1)$ modular multiplications and one modular exponentiation are needed.

When calculating the modular exponentiation using the binary exponentiation algorithm it requires a number of modular multiplications, this depends on the exponent. Given the binary representation of an exponent, in our case $2^k \mod \phi(n)$, the number of modular multiplications needed can be calculated by adding the Hamming weight of the exponent h and the bit length of the exponent l and then subtracting two, $h + l - 2$. For example 110100110001 (3,377) will result in 16 modular multiplications, $6 + 12 - 2$. This number of modular multiplications needed for one modular exponentiation will be represented by c.

Sequential verification requires $m * c$ modular multiplications, batch verification requires $2(m-1) + c$ modular multiplications. When comparing these two approaches, $2(m-1) + c < m * c$, the result is that with $m > 1$ batch verification is cheaper as long as $c > 2$. As long as $2^k \mod \phi(n)$ does not equal 0, 1, 2, 3, or 4 batch verification is cheaper than verifying every item sequentially.

This yields the following scheme

- $\mathsf{Setup}(\ell)$: the server selects two random large primes p, q and creates $n = p * q$ out of this. The server also selects a difficulty for the puzzles,

k. The algorithm then outputs $mk = 2^k \mod \phi(n)$ and $params = \{k, n\}$.

- $\mathsf{PuzzleGen}(mk, req)$: the server chooses a generator g out of the range $[2, n-1]$. It then outputs the puzzle and information $puz = info = g$.
- $\mathsf{PuzzleSol}(puz)$: $sol = g^{2^k} \mod n$.
- $\mathsf{PuzzleVer}(info, mk, sol)$: $sol \stackrel{?}{=} g^{2^k \mod \phi(n)} \mod n$.
- $\mathsf{BatchVer}(info_i, mk, sol_i(1 \leq i \leq m))$: assume that the server has stored m solutions to check along with the additional information, $info_i, sol_i, 1 \leq i \leq m$. These solutions are then verified with batch verification using Equation 4.1.

The drawback of batch verification is that when one solution is incorrect the entire batch will fail and additional computations are needed to find the solutions responsible. For our analysis it is assumed that there are w errors at random somewhere in the batch and that w is much smaller than m, $w << m$. We show four approaches to find the false solutions within the batch (sequential checking, triggered sequential checking, divide and conquer checking, random checking), these are detailed below. Because the errors are at random places in the batch triggered sequential checking, divide and conquer checking and random checking can not give an exact complexity value. In these cases the complexity value is given in best, worst and average case scenario.

The average case scenario is under the assumption that the errors are distributed uniformly over the batch. All complexity values are expressed in the number of modular multiplications.

1. **Sequential Checking:** The most straightforward way to check for errors is to check each item individually, using normal verification. There are m solutions in the batch and checking one solution requires c modular multiplications. The complexity will be the same in any scenario and checking for errors requires $m * c$ modular multiplications.
2. **Triggered Sequential Checking:** This approach is a variation to sequential checking. All solutions are checked one by one until an error is found and then the remaining items are re-checked for other errors. The process that this method follows is:

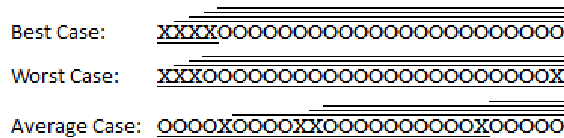

Figure 4.1: Scenarios triggered sequential checking

Step 1: Check the first unchecked solution in the batch. If this solution is incorrect proceed to step 2. If this solution is correct do step 1 again.

Step 2: Check all remaining unchecked solutions using batch verification. If this batch contains errors go to step 1. If this batch does not contain an error, all errors have been found and the check is finished.

Figure 4.1 shows the scenarios for this method. The X represents an incorrect solution and the O represents a correct solution. The lines above the batch represent batch verifications, the line below the batch represents the solutions that are checked sequentially.

- The best case scenario when using this method is all w errors at the beginning of the batch. When all errors are found this gives the biggest batch of correct solutions to check in one pass. All errors are checked sequentially requiring $w * c$ modular multiplications. After each error a batch verification needs to be done, the complexity for this is $2(m - y - 1) + c$, with y ranging from 1 to w. The complexity for the best case scenario is $2w * c + 2w(m - 1) - (w^2 + w)$.
- Worst case scenario is $w - 1$ errors at the beginning of the batch and the last at the end. Because the last solution is incorrect each element of this batch will have to be checked using sequential checking. The other errors at the beginning of the batch create the largest possible batches that have to be checked using batch verification. Checking each element of the batch sequentially requires $m * c$ modular multiplications. The $w - 1$ errors at the start are each followed by a batch verification. The complexity for such a batch verification is $2(m - y - 1) + c$, with y ranging from 1 to $w - 1$. Combining this gives a worst case complexity of

$m * c + (w-1)c + 2(w-1)(m-1) - (w^2 - w)$.

- The average case complexity can be found by averaging all possible complexity values. The set G contains all possible error distributions with batch size m and with w errors. G_i represents the ith element of this set and the number of elements in G is denoted by $n = \binom{m}{w}$. C_i denotes the cost of triggered sequential checking for set element G_i. The average case complexity can then be calculated with: $(\sum_{i=1}^{n} C_i)/n$.

  C_i can be computed by entering G_i into the step process above. Each application of step 1 requires c modular multiplications. Each application of step 2 takes $2(u-1)+c$, with u the number of unchecked solutions remaining at that point.

  Appendix A.1 contains a mathematica program to simulate one instance and calculate the complexity of this instance. The simulation requires the size of the batch m and the number of errors w as input and will be used to determine the average case complexity by averaging 1,000 runs. An overview of the program is also given in text.

3. **Divide and Conquer Checking(x):** This approach is to split the batch into x smaller batches of equal size. The errors will be found using a tree search method. The process is as follows:

   Step 1: If $m = 1$ an error is found. Otherwise split to batch in which errors need to be found into x smaller batches. For each of these x smaller batches do step 2.

   Step 2: Check this batch using batch verification. If this batch contains an error proceed to step 1. If this batch contains no errors, this sub-batch is cleared.

   Figure 4.2 shows the scenarios for divide and conquer checking(3). Again, an X represents an incorrect solution and an O a correct solution. The tree above the batch shows which nodes of that tree the checking method has to verify.

   The complexity of this method is the sum of the complexity of batch verification in all nodes that have to be checked.

   - In the best case the tree is narrow and the number of nodes that have to be checked is small. At each layer in the search tree the number of nodes that have to be checked depends on the number

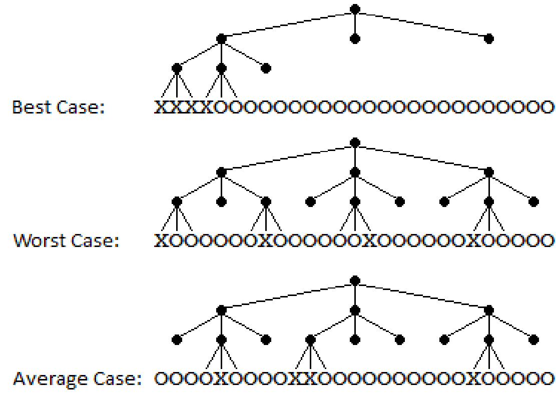

Figure 4.2: Scenarios divide and conquer checking

of errors w and the depth in the tree i. This is represented by $\lceil w * x^{i-y-1} \rceil * x$. In this formula, y is the height of the search tree and $y = \lceil \log_x m \rceil$. The cost of batch verification for a node at depth i is given by $2(\lfloor m * x^{-i} \rceil - 1) + c$. By multiplying the cost of batch verification a node with the number of nodes that have to be checked for each layer in the search tree the total complexity is calculated. The complexity for the best case scenario is $\sum_{i=1}^{y}((2(\lfloor m * x^{-i} \rceil - 1) + c) * (\lceil w * x^{i-y-1} \rceil * x))$.

- In the worst case the errors are distributed and the tree is wide. This gives a different formula for the number of nodes at a given depth i, $\min(w*x, x^i)$. The complexity for the worst case scenario then becomes $\sum_{i=1}^{y}((2(\lfloor m * x^{-i} \rceil - 1) + c) * \min(w * x, x^i))$.
- The average case complexity can be found by averaging all possible complexity values. The set G contains all possible error distributions with batch size m and with w errors. G_i represents the ith element of this set and the number of elements in G is denoted by $n = \binom{m}{w}$. $C(x)_i$ denotes the cost of divide and conquer checking(x) for set element G_i. The average case complexity can then be calculated with: $(\sum_{i=1}^{n} C(x)_i)/n$.
  $C(x)_i$ can be computed by drawing the search tree for G_i, with each node having x branches. At each node that is visited, calculate the cost for batch verifying that node, $2(u-1)+c$, with u the number of solutions at that node. By adding the cost for all nodes, except the root node, the total cost $C(x)_i$ is calculated.

```
Best Case:     OOOOOOOOOOOOOOOOOOOOOOOXXXX
Worst Case:    XXXOOOOOOOOOOOOOOOOOOOOOOOX
Average Case:  OOOOXOOOOXXOOOOOOOOOOXOOOOO
```

Figure 4.3: Scenarios random checking

Appendix A.2 contains a mathematica program to simulate an application of this checking scheme. It calculates a possible complexity for a batch size of m, splitting in x smaller batches and containing w errors. This program will be used to determine the average case complexity by averaging 1,000 runs. This appendix also gives an overview of the program in text.

4. **Random Checking:** This approach is to select sub-batches of random size r and test these sub-batches for errors. The process is as follows:

   Step 1: If $m = 1$ an error is found. Otherwise select a random integer r between 1 and $m-1$. Test the first r elements of the batch using batch verification. If the sub-batch contains an error proceed to step 2. If the sub-batch contains no error proceed to step 3.

   Step 2: Check the remaining items using batch verification. If there is an error in the remaining items do step 1 for both the sub-batch and the remaining items. If there is no error in the remaining items do step 1 only for the sub-batch and clear the remaining items.

   Step 3: Clear the sub-batch and do step 1 for the remaining items.

   Figure 4.3 shows the scenarios for random checking. The line below the batches in the best and worst case scenario shows the assumed initial splitting.

   - In the best case all correct solutions and no erroneous solutions are contained in the first sub-batch, leaving all errors in the remaining items. It takes $2(m-w-1)+c$ modular multiplications to clear the first sub-batch. To check the remaining errors, splitting the batch in half each time results in the fewest computations. Checking only the errors takes $2((\lceil \log_2 w \rceil - 1)w - 2^{\lceil \log_2 w \rceil} + 2) + \max(2(w-1), 1)c$ modular multiplications. The resulting best

case scenario complexity is $2(m + (\lceil \log_2 w \rceil - 2)w - 2^{\lceil \log_2 w \rceil} + 1) + \max(2w - 1, 2)c$.

- In the worst case scenario a sub-batch of size one is selected every time, $r = 1$ in each instance, and this checking method behaves as triggered sequential checking. The worst case scenario is thus the same as the worst case scenario for triggered sequential checking, with $w-1$ errors at the start of the batch and one at the end. The complexity is the same, $m*c+(w-1)c+2(w-1)(m-1)-(w^2-w)$.
- The average case complexity can be found by averaging all possible complexity values. The set G contains all possible error distributions with batch size m and with w errors. G_i represents the ith element of this set and the number of elements in G is denoted by $n = \binom{m}{w}$. Because error checking is probabilistic and not deterministic the cost for checking a set element G_i is not fixed.

  The number of ways the sub-batches can be randomly chosen can be represented by a full binary tree and bound by the number of possible trees with m leaves. The set H contains all possible full binary trees with m leaves. H_j represents the jth tree in this set and the number of trees in this set s is the Catalan number [3] of $m-1$, namely $s = (2m-2)!/(m!*(m-1)!)$. Let $C_{i,j}$ denote the cost of checking set element G_i using the binary tree H_j. The average case complexity can then be calculated with: $(\sum_{i=1}^{n}(\sum_{j=1}^{s} C_{i,j})/s)/n$.
  $C_{i,j}$ is computed by redrawing H_j.

  (a) Each leaf node of H_j has value 1, all other nodes have a value equal to the number of leaf nodes that can be reached from it. One exception to this is the root node which has no value.

  (b) For each node that can only reach leaf nodes with correct solutions, make this node a leaf node, but keep the value of the node the same, and discard the nodes below it.

  (c) In each case that a leaf node with only correct solutions is the left node, remove the value of the right node.

  (d) The cost for a node is $2(u - 1) + c$, with u the value of the node. Nodes with no value do not have a cost. Add all these costs together to get the cost $C_{i,j}$.

  Figure 4.4 gives a small example for the steps to take.

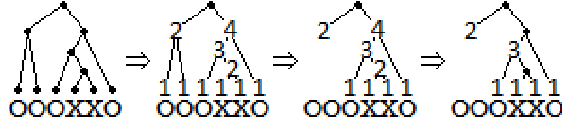

Figure 4.4: Redraw random checking tree

> Appendix A.3 contains the mathematica program that simulates error finding and outputs the corresponding computational complexity. The program is also outlined in text. This simulation requires the size of the batch m and the number of errors w as input and will be used to determine average case complexity by averaging 1,000 runs.

The names for the error finding methods will be shortened in the rest of the paper. Sequential checking will be shortened to Seq, triggered sequential checking to Trig Seq, divide and conquer checking(x) to D&C(x) and random checking to Rand.

When the errors are found the server can take steps to counter a possible attack. For example, ban the client that submitted the false solution.

4.1.2 Modified Repeated Squaring #2

When p and q are fixed, the only variable in the puzzle is the generator g. Instead of having the server generate a g we can have the client generate its own g based on a function and some public parameter controlled by the server. The server should then only check if this generator is freshly created. Furthermore this generator is to be bound to the client. The function for creating a generator becomes $g = h(sp, id_{\mathcal{C}})$, with sp a public parameter chosen by the server with length ℓ bits and $id_{\mathcal{C}}$ the clients identifier.

This results in the following client puzzle scheme.

- $\mathsf{Setup}(\ell)$: the server generates a new public parameter sp with length ℓ bits using a (pseudo)random number generator. It then selects two random large primes p, q and creates $n = p * q$ out of this. The server also selects a difficulty for the puzzles, k. The algorithm then outputs $mk = 2^k \mod \phi(n)$ and $params = \{k, sp, n\}$.
- $\mathsf{PuzzleGen}(mk, req)$: $puz = \emptyset$ and $info = \emptyset$.
- $\mathsf{PuzzleSol}(puz)$: $g = h(sp, id_{\mathcal{C}})$, $sol = g^{2^k} \mod n$.

- $\mathsf{PuzzleVer}(info, mk, sol)$: $g = h(sp, id_{\mathcal{C}})$, $sol \stackrel{?}{=} g^{2^k \mod \phi(n)} \mod n$.

The additional requirements for the server for this modified scheme are the public parameter sp and an additional hash computation. The public parameter has to be stored by the server, adding storage cost, and transmitted to the clients adding communication overhead. It also has to be changed on a regular basis. Because the client computes the generator instead of receiving it from the server, this lowers the communication overhead as g does not have to be transmitted. If the public parameter sp is smaller than g there is less communication overhead compared to the original repeated squaring scheme by Rivest et al. [17]. The hash operation to retrieve the generator is cheap in comparison to the modular arithmetic that is needed to check the solution.

4.1.3 Combined Modified Repeated Squaring

The two schemes can be combined to form a single scheme. This scheme then has the benefits of both. Batch verification provides a lower verification cost and the new puzzle generation mechanism lowers short-term storage requirement. Because in both separate schemes the parameters p, q and n are fixed during the setup phase, $\mathsf{Setup}(\ell)$, there is no clash when combining the two. The client puzzle scheme is then as described below:

- $\mathsf{Setup}(\ell)$: the server generates a new public parameter sp with length ℓ bits using a (pseudo)random number generator. It then selects two random large primes p, q and creates $n = p * q$ out of this. The server also selects a difficulty for the puzzles, k. The algorithm then outputs $mk = 2^k \mod \phi(n)$ and $params = \{k, sp, n\}$.

- $\mathsf{PuzzleGen}(mk, req)$: $puz = \emptyset$ and $info = \emptyset$.

- $\mathsf{PuzzleSol}(puz)$: $g = h(sp, id_{\mathcal{C}})$, $sol = g^{2^k} \mod n$.

- $\mathsf{PuzzleVer}(info, mk, sol)$: this algorithm computes the generator, $g = h(sp, id_{\mathcal{C}})$. The solution is then verified, $sol \stackrel{?}{=} g^{2^k \mod \phi(n)} \mod n$.

- $\mathsf{BatchVer}(info, mk, sol_i(1 \leq i \leq m))$: assume that the server has stored m solutions to check, $sol_i, 1 \leq i \leq m$. Compute the corresponding generators $g_i = h(sp, id_{\mathcal{C}_i}), 1 \leq i \leq m$. These solutions are then verified with batch verification using Equation 4.1.

This combined scheme has high evaluation marks on the criteria as detailed in Section 2.4. In this model, no computations are required for the server to do puzzle generation. Puzzle verification costs one hash operation per solution along with the cost for batch verification, $2(m-1)+c$ if there are no errors. Several values need to be pre-computed during setup, p, q, n, $\phi(n)$ and $2^k \mod \phi(n)$ as well as the public parameter sp. The difficulty of the puzzle can be controlled linearly and is deterministic in nature. There are a few variables that the server needs to store in long-term storage: n, $2^k \mod \phi(n)$, sp and possibly p and q. No additional information has to be stored in short-term storage. The communication complexity consists of the transmission of the parameters n, k, sp and the solution sol.

4.2 Conclusion

In this chapter we have proposed three modified versions of the repeated squaring client puzzle scheme. Two modified versions to combat the drawbacks of the original version of the repeated squaring puzzle scheme and one modified version to combine them both. This combined scheme removes the drawbacks the initial scheme has by adding batch verification and altering the way the puzzles are generated. Computational complexity of repeated squaring has been reduced and the scheme is no longer stateful.

However, there is a potential security concern with our proposals. An adversary can combine generators g_i $(1 \leq i \leq w)$ into $h = \prod_{i=1}^{w} g_i$, and compute a corresponding solution $sol = h^{2^\ell} \mod n$. Then the adversary can randomly split sol into w sub-solutions sol_i $(1 \leq i \leq w)$ and send them to the server. Clearly, as long as sol_i $(1 \leq i \leq w)$ are in the same batch, the verification will pass given that there is no error in other solutions. In reality, this attack could be mitigated by various means. We briefly mention the following two as examples.

- The server can dynamically set the batch size in according to the total number of received puzzle solutions. In addition, the server can randomly mix the solutions before splitting them into batches. By doing this, the probability that all w sub-solutions sol_i $(1 \leq i \leq w)$ fall into the same batch will be kept low.

- Instead of directly multiplying the solutions as in Equation 4.1, the server can assign a random weight w_i (an integer) to each solution sol_i,

then the verification will be as follows.

$$(\prod_{i=1}^{m} g_i^{w_i})^{2^k \mod \phi(n)} \mod n \stackrel{?}{=} \prod_{i=1}^{m} sol_i^{w_i} \mod n \tag{4.2}$$

We will not formally investigate countermeasures against this attack in this thesis, and leave it as an interesting research question in the future.

In the next chapter the combined modified repeated squaring scheme will be applied to two application scenarios.

Chapter 5

Applications of the Proposed Puzzle Scheme

In this chapter the combined modified repeated squaring puzzle scheme will be applied to two example scenarios, a web server scenario and an e-mail server scenario. After the application of the scheme an analysis will be given to the performance. Then this performance will be compared to an application of a subset sum-based puzzle scheme. The subset sum-based puzzle scheme is chosen as a comparison scheme, because it has similar properties with regard to the criteria as described in Section 2.4 and is a potential candidate for an all-round client puzzle scheme.

In the web server scenario the combined modified repeated squaring puzzle scheme will be applied to a web server to protect this web server against DDoS attacks. In the e-mail server scenario the scheme is used to reduce the flow of spam e-mails sent to an e-mail server by spammers. The web server scenario requires the data to be transmitted real-time, while the e-mail server scenario can work with larger delays.

5.1 Web Server Scenario

Consider a company owning a webserver $\mathcal{S}_{web}$ with the only functionality of hosting a website to convey company information to the outside world. Connecting to $\mathcal{S}_{web}$ are a large number of clients $\mathcal{C}_j$. A typical client requests several webpages and disconnects after that. These clients range in hardware from simple PDA's to high end PC's.

However a large number of the clients $\mathcal{C}_j$ are malicious and under control of an adversary. The goal of this adversary is to hurt the business practice

of the company. To this end the adversary launches a DDoS attack on the webserver $\mathcal{S}_{web}$ with the purpose of denying legitimate clients access to the website.

In this scenario the following requirements and assumptions are made.

1. The users wanting to view the website should only perceive minimal delay. An acceptable delay time is about two seconds [16].

2. Delays that are not caused by the client puzzle scheme are not taken into account. These delays are for example transmission times and server webpage fetching. This is beyond the scope of this theoretical framework.

3. Depending on how popular the website is and the time of day, the number of client visits per second differs. The following values will be used.

   - 1,000 connections per second.
   - 10,000 connections per second.
   - 50,000 connections per second.

4. The speed at which a client can perform one squaring, or one modular multiplication, depends on the hardware used. To be able to show some performance statistics the speed values for the clients are assumed values. A high end PC will be able to do 50,000 modular multiplications per second while a PDA can do 5,000 modular multiplications per second. The server $\mathcal{S}_{web}$ will be assumed to be able to do 150,000 modular multiplications per second.

5. The complexity of the modular exponentiation, which is c, is primarily dependent on the value of $2^k \mod \phi(n)$. This is a random value between 1 and $\phi(n)$. The current recommended RSA bitlength for n is 1,024, but this is shifting towards 2,048 [9]. The modular exponentiation complexity c will be between 1 and 2,044 modular multiplications, averaging at 1,022 modular multiplications. This average will be used as a value for c, the number of modular multiplications for one modular exponentiation.

To stop the adversary from reaching his goal the company adopts a client puzzle scheme at $\mathcal{S}_{web}$. The server $\mathcal{S}_{web}$ will use the combined modified repeated squaring puzzle scheme and all clients $\mathcal{C}_j$ are assumed to be able to handle the client side of this scheme. The server $\mathcal{S}_{web}$ is set to verify a

batch size	sequential	batched
1,000	6.81s	0.02s
10,000	68.13s	0.14s
50,000	340.67s	0.67s

Table 5.1: Repeated squaring verification speeds

batch each second, this leaves a maximum of one second for the client to solve the puzzle in order to have an acceptable delay time. When a client $\mathcal{C}_i$ submits a false solution, this client $\mathcal{C}_j$ is banned for a random amount of time.

The variables p and q have been generated by $\mathcal{S}_{web}$ and are assumed static. This also implies a static value for n. The public parameter sp is a short lived value that changes frequently. Based on the modular multiplication speed of the clients, the difficulty parameter k is chosen so that the slowest client can solve the puzzle in one second, this means a value of $k = 5{,}000$. Changing the difficulty parameter k will invalidate all solutions up to that point and re-computation is needed to get the new value for $2^k \mod \phi(n)$. Should the difficulty of the puzzle need to be changed, this can be done together with a change in the public parameter sp which will also invalidate all solutions. Frequently changing k will result in additional computations that could significantly slow the server down. Because the server verifies a batch each second the number of connections per second corresponds to the batch size m.

For the adversary, there is a trade-off between the work he can make the server do and the number of clients that are not blocked by the server and able to continue the attack.

5.1.1 Analysis

Table 5.1 shows the difference in verification speed between sequential verification and batch verification. The difference between sequential verification and batch verification is substantial in each case.

The performance of the combined modified repeated squaring scheme is different when there are incorrect solutions in the batch. Figure 5.1 shows the verification speeds for a batch of size 1,000 with 5 errors distributed at random in the batch. The error finding methods Seq and Trig Seq both need more than a second to find the errors. The methods Rand and D&C(x) keep below a second. In the second graph of Figure 5.1, it is visible that D&C(4)

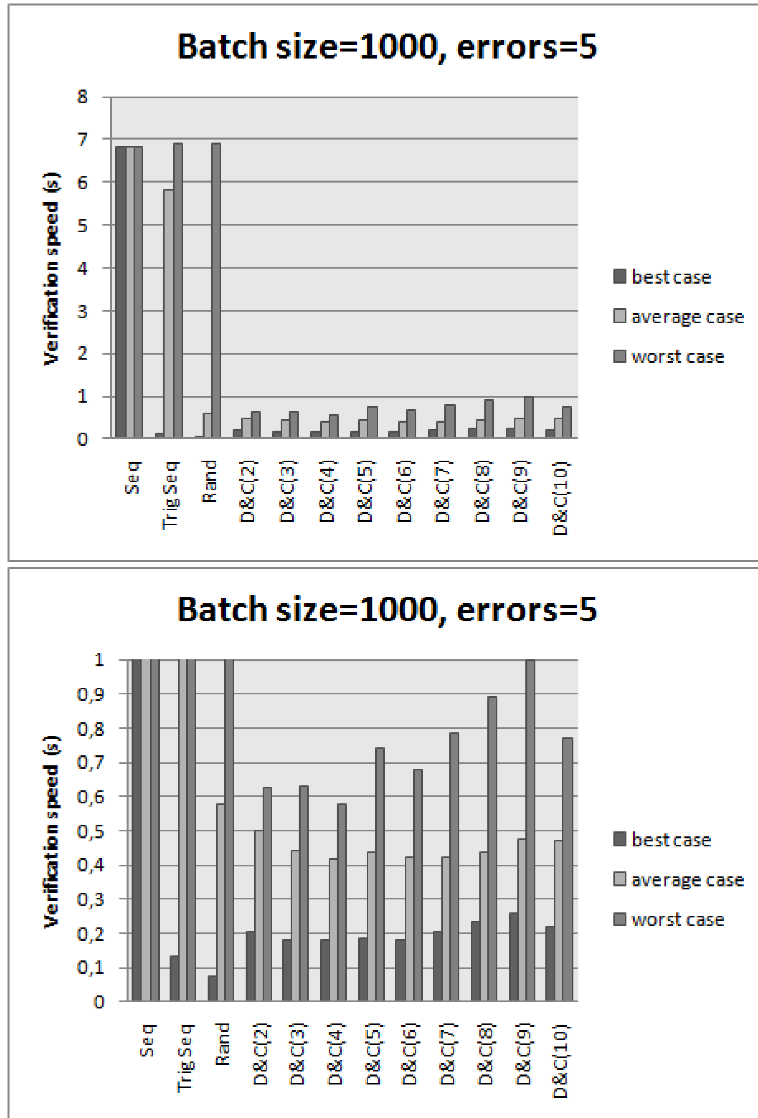

Figure 5.1: Comparison of error checking methods for a batch size of 1,000 and 5 errors

is the fastest in finding the errors, taking 0.42 seconds on average.

Figure 5.2 shows the results when the batch size is increased to 10,000, the number of errors remains the same. In the worst case each method takes more than a second to complete. D&C(5) is the fastest method, taking 0.83 seconds on average.

Figure 5.3 shows the results when the batch size is increased even further to 50,000, the number of errors again remains the same. It can be seen that the difference between Seq and Trig Seq when compared to Rand and D&C(x) is very large. None of the methods are faster than a second in the best case scenario. Of the methods D&C(10) is the fastest, taking 1.73 seconds on average to complete.

In all cases the D&C(x) method is the fastest in finding the errors. However in the best parameter for this method is not always the same. The only element that was varied over the cases was the number of elements in the batch. This number should be the decisive factor when determining the parameter for the D&C(x) method. The number of errors in the batch could be an input as well, but this can not be determined beforehand.

5.1.2 Comparison

We compare these numbers to a web server equipped with the alternative, a subset sum-based puzzle scheme. A subset sum-based puzzle scheme only requires a comparison to verify a solution, however creating a puzzle does require computational effort. To create a puzzle one hash operation, ℓ multiplications and $\ell - 1$ additions are needed. Each verification for combined modified repeated squaring also requires one hash operation, these operations have the same complexity and can be left out of the scope of this comparison. Because addition is a computationally cheap operation, the additions are also left out of the comparison. We assume that the complexity for the modular multiplication used in the combined modified repeated squaring scheme is double the complexity of the multiplications in the subset sum-based scheme.

Tritilanunt et al. [18] suggest a value for ℓ to be between 60 and 100. For this comparison we will take the lower bound of 60. As can be seen in Table 3 of [18] with the right density this gives us the same solving speed for the clients. So creating one puzzle requires 60 multiplications.

To match the verification speed of combined modified repeated squaring, 10,000 verifications in 0.14s, the server needs to be able to do 600,000 multiplications in 0.14s. This translates to a requirement of around 4,280,000 multiplications per second. The server will need to be about 14 times faster

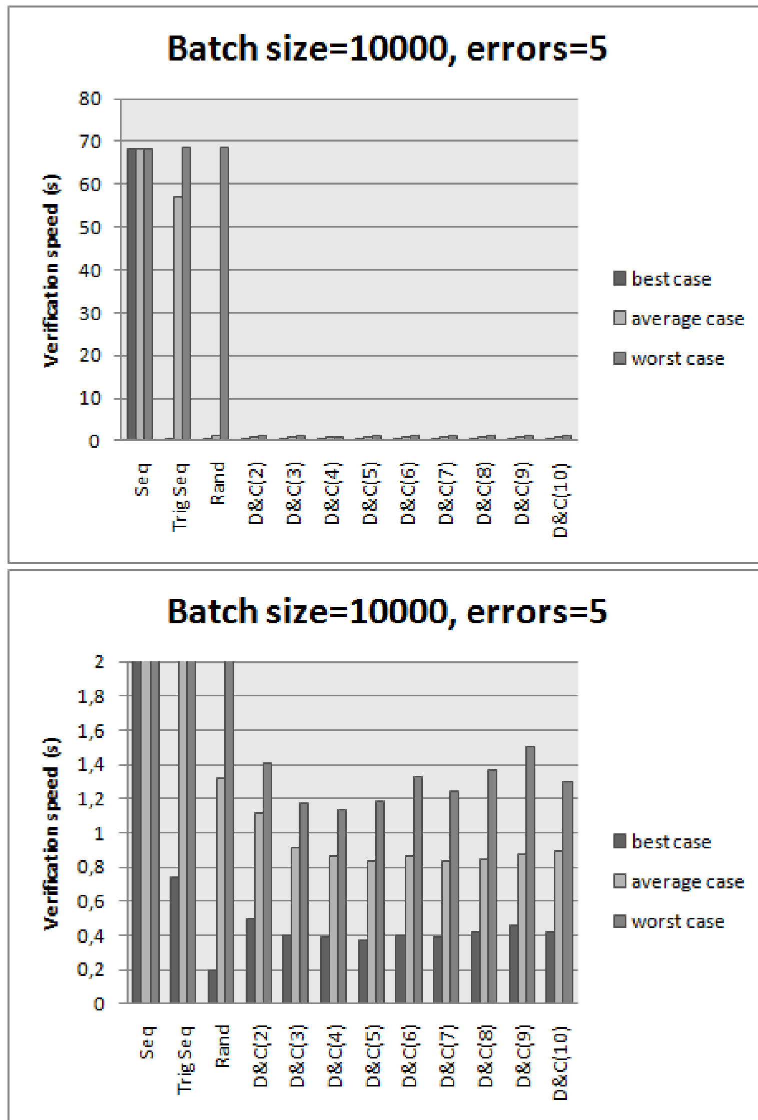

Figure 5.2: Comparison of error checking methods for a batch size of 10,000 and 5 errors

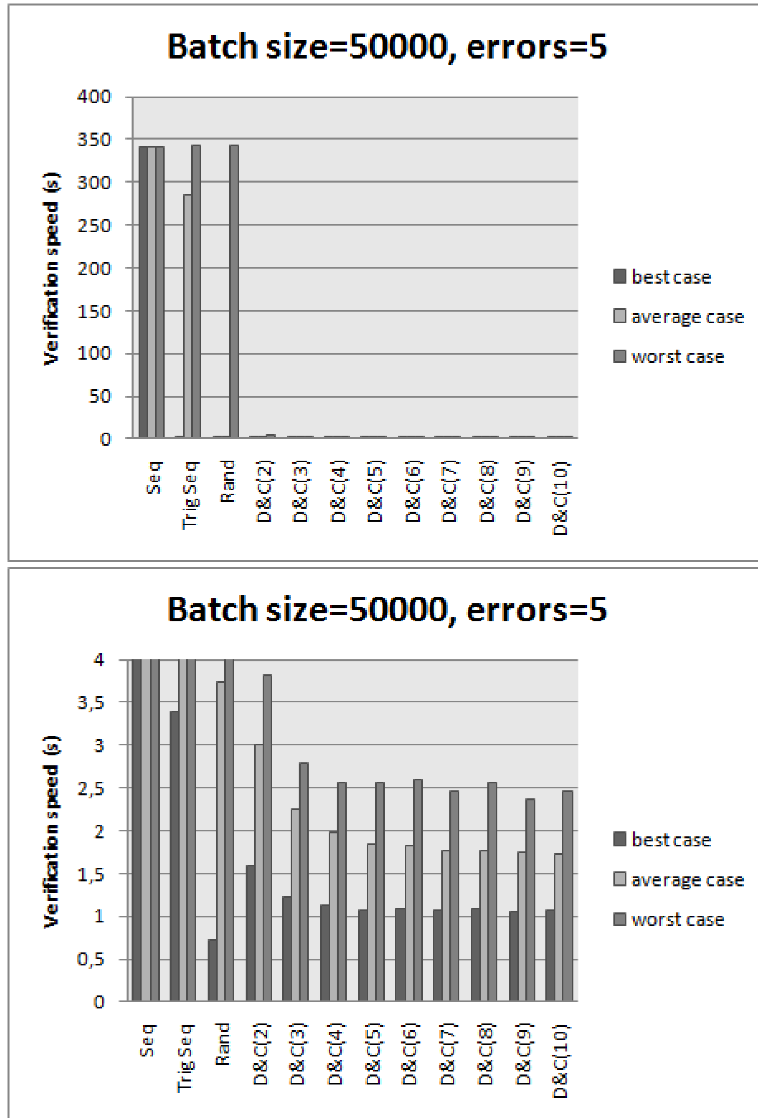

Figure 5.3: Comparison of error checking methods for a batch size of 50,000 and 5 errors

to adopt the subset sum-based scheme instead of the combined modified repeated squaring scheme.

When there are errors in the batch the performance of combined modified repeated squaring drops to about 0.83s. A server would need around 723,000 multiplications per second to match performance. In this case the server will need to be about 2-3 times faster to adopt the subset sum-based puzzle scheme.

In summary the combined modified repeated squaring scheme is significantly faster than the subset sum-based puzzle scheme.

5.2 E-mail Server Scenario

Consider another company owning an e-mail server. Everyone is allowed to send e-mails using this server $\mathcal{S}_{mail}$. There is a large number of clients $\mathcal{C}_j$ connecting to $\mathcal{S}_{mail}$ in order to send e-mails. Again, these clients range in hardware from simple PDA's to high end PC's. $\mathcal{S}_{mail}$ also has a connection to another mail server $\mathcal{M}$ in order to forward the mail received from the clients.

A spammer $\mathcal{A}$ is trying to abuse $\mathcal{S}_{mail}$ by sending a lot of spam e-mails. His goal is to send as many e-mails as possible. These e-mails take up bandwidth and computer resources that the company otherwise could have used for better purposes.

In this scenario the following requirements and assumptions are made.

1. A different protection mechanism exists between $\mathcal{S}_{mail}$ and the other mail server $\mathcal{M}$. $\mathcal{S}_{mail}$ is free to send an unlimited amount of e-mails to $\mathcal{M}$. $\mathcal{M}$ trusts $\mathcal{S}_{mail}$ to only send legitimate e-mails to $\mathcal{M}$.

2. The speed at which a client can perform one squaring, or one modular multiplication, depends on the hardware used. To be able to show some performance statistics the speed values for the clients are assumed values. A high end PC will be able to do 50,000 modular multiplications per second while a PDA can do 5,000 modular multiplications per second. The server $\mathcal{S}_{mail}$ will be assumed to be able to do 150,000 modular multiplications per second.

3. The complexity of the modular exponentiation, which is c, is primarily dependent on the value of $2^k \mod \phi(n)$. This is a random value between 1 and $\phi(n)$. The current recommended RSA bitlength for n is 1,024, but this is shifting towards 2,048 [9]. The modular exponentiation complexity c will be between 1 and 2,044 modular multiplications,

averaging at 1,022 modular multiplications. This average will be used as a value for c, the number of modular multiplications for one modular exponentiation.

To reduce the flow of spam to the mail server $\mathcal{S}_{mail}$ the company wants to adopt a client puzzle scheme as a payment mechanism for sending e-mails. In order for a client $\mathcal{C}_j$ to pay for sending an e-mail some computational work has to be done. This computational work is in the form of a combined modified repeated squaring puzzle that has to be solved and attached to the e-mail. There is a slight difference in that the generator g used in this scheme is now based on the contents of the e-mail $mail_{\mathcal{C}}$ and the recipient of the e-mail $rcpt_{\mathcal{C}}$ instead of the clients identifier $id_{\mathcal{C}}$. This results in the following method for creating a generator: $g = h(sp, mail_{\mathcal{C}}, rcpt_{\mathcal{C}})$. If an incorrect solution is attached to an e-mail, this e-mail is rejected and sent back to the sender.

Because $\mathcal{S}_{mail}$ does not have to handle request in real-time, but can permit a delay, the parameters for this scheme compared to the web server scenario are different. The parameters p, q, n and $\phi(n)$ remain static. But the public parameter sp is changed less often than in the web server scenario. The difficulty parameter k is chosen so that a high end client can solve the puzzle in 2 minutes. 2 minutes equals 120 seconds so the value for k will be 120 * 50,000 = 6,000,000. A PDA will take 6,000,000 / 5,000 = 1200 seconds, which is 20 minutes. The e-mails received by $\mathcal{S}_{mail}$ are stored in a batch and the batch will be verified every 15 minutes.

5.2.1 Analysis

As it takes a high end client 2 minutes to solve a puzzle, the maximum number of e-mails this client can send is 30 per hour and thus 720 per day. In order to be able to send 10 e-mails per hour and 240 per day the client will need to be able to do 16,667 modular multiplications per second. With a PDA it is possible to send 3 e-mails per hour and 72 per day. According to a study by Laurie and Clayton [10] this is enough to accommodate most users. Their figures show that 90% of the legitimate e-mail users send 50 e-mails a day or 12 per active hour. A limit of 240 e-mails a day would inconvenience less than 2%. Laurie and Clayton further state that spammers should be limited below 1,750 e-mails a day to make it uneconomical. The limit for this scheme is 720 a day and is well below that.

When the mail server $\mathcal{S}_{mail}$ verifies a batch and finds that one of the e-mails does not have a valid puzzle solution, it can use the same error finding

method as in the web server scenario. There is however no party in this scenario that would benefit from sending false solutions to the mail server. The goal of the clients $\mathcal{C}_j$ and of the spammer $\mathcal{A}$ is to send e-mails, not to prevent others from sending e-mails.

5.2.2 Comparison

With a combined modified repeated squaring scheme the client can create and solve a puzzle before contacting the e-mail server. In order to send an e-mail using an e-mail server which has a subset sum-based puzzle scheme, the client first has to contact the e-mail server and exchange nonces. The server then generates a puzzle, stores the solution and sends the puzzle to the client. The client can then solve the puzzle.

However, the puzzle difficulty is set so that the estimated time to solve a puzzle is a couple of minutes. During this time the connection to the e-mail server with the subset sum-based puzzle scheme has to be kept open and the server has to store the solution. This requires the server to allocate its resources into a temporarily unused connection.

The combined modified repeated squaring scheme does not suffer from this drawback. The result is that an e-mail server with a subset sum-based puzzle scheme will need to set aside additional resources, next to storing the e-mail, for every puzzle, when compared to the combined modified repeated squaring scheme.

5.3 Conclusion

In both application scenarios, the combined modified repeated squaring puzzle scheme accomplishes the goal set out in that scenario. Under our assumptions the web server is protected against a large number of webpage requests from a client. Also clients are limited in the number of e-mails they can send, but not to an amount which would be impractical.

In both scenarios, the combined modified repeated squaring puzzle scheme requires less resources from the server than the subset sum-based puzzle scheme, while giving the same result.

Chapter 6

Conclusion

The goal of this thesis was to create a new client puzzle scheme that has high scores on the evaluation criteria. These desired criteria scores, described in Section 2.4, are low computational complexity, linear hardness granularity, low storage requirements and low communication complexity. Current client puzzle solutions score high on some criteria, but score low on others.

In the combined modified repeated squaring puzzle scheme the original repeated squaring puzzle scheme has been improved by adding batch verification and a public parameter so clients can create their own puzzles. The result is a client puzzle scheme that has low computational complexity. It is also non-parallelizable, preventing adversaries from working together to rapidly create solutions. The difficulty of the puzzles is scalable in a linear and deterministic way. The storage space required is low and no state information has to be kept.

However combined modified repeated squaring is not perfect, it is not always better than other schemes when only looking at one evaluation criteria. When looking at all evaluation criteria combined modified repeated squaring is the best so far.

The scheme has been applied in two scenarios, the performances were evaluated theoretically. The next step in the process is to create a prototype and do additional testing. This will allow for some fine-tuning of the scheme and determine its practical use.

Bibliography

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

[2] T. Aura, P. Nikander, and J. Leiwo. DoS-resistant authentication with client puzzles. In *Security Protocols, 8th International Workshop*, pages 170–177, 2000.

[3] D. M. Campbell. The computation of catalan numbers. *Mathematics Magazine*, 57(4):195–208, 1984.

[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] W. Feng, E. C. Kaiser, and A. Luu. The design and implementation of network puzzles. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 2372–2382, 2005.

[6] Y. Gao, W. Susilo, Y. Mu, and J. Seberry. Efficient trapdoor based client puzzle against DoS attacks. *Book Chapter in Network Security*, 2006.

[7] B. Groza and D. Petrica. On chained cryptographic puzzles. *3rd Romanian-Hungarian Joint Symposium on Applied Computational Intelligence*, pages 182–191, 2006.

[8] A. Juels and J. G. Brainard. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 1999*, pages 151–165, 1999.

[9] B. Kaliski. TWIRL and RSA key size. Technical report, RSA Laboratories, 2003.

[10] B. Laurie and R. Clayton. Proof-of-work proves not to work. In *WEIS 04*, 2004.

[11] Y. Lei, S. Pierre, and A. Quintero. Client puzzles based on quasi partial collisions against DoS attacks in UMTS. *Vehicular Technology Conference, 2006. VTC-2006 Fall. 2006 IEEE 64th*, pages 1–5, 2006.

[12] A. K. Lenstra, H. W. Lenstra Jr., and L. Lovsz. Factoring polynomials with rational coefficients. *Mathematische Annalen 261*, pages 515–534, 1982.

[13] I. Martinovic, F. A. Zdarsky, M. Wilhelm, C. Wegmann, and J. B. Schmitt. Wireless client puzzles in IEEE 802.11 networks: Security by wireless. In *Proceedings of the First ACM Conference on Wireless Network Security*, pages 36–45, 2008.

[14] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1997.

[15] R. C. Merkle. Secure communications over insecure channels. *Communications of the ACM*, pages 294–299, 1978.

[16] F.H. Nah. A study on tolerable waiting time: How long are web users willing to wait? *Behaviour & IT*, 23(3):153–163, 2004.

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

[18] S. Tritilanunt, C. Boyd, E. Foo, and J. M. González Nieto. Toward non-parallelizable client puzzles. In *Cryptology and Network Security, 6th International Conference, CANS 2007*, pages 247–264, 2007.

[19] B. Waters, A. Juels, J. A. Halderman, and E. W. Felten. New client puzzle outsourcing techniques for DoS resistance. In *Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004*, pages 246–256, 2004.

Appendix A

Mathematica Programs

A.1 Triggered Sequential Checking

```
TrigComplexity[m_, w_] :=
 Module[{e},
  e = Sort[Array[Random[Integer, {1, m}] &, w]];
  (* Determine error positions *)
  (m - First[e] + w + 1)*c + 2 (Total[e] - 2*w)
  (* Compute complexity *)
  ]
```

Explanation of the Program

This program takes the batch size and the number of errors as input.

First the error locations are determined at random and ordered according to their location. The numbers representing the error location show the number of items that still need to be checked, just before the error is sequentially checked.

The complexity is then calculated using the error positions. The error that is last determines the number of elements that need to be checked sequentially. The total of the error locations determines the amount of elements that are checked using batch verification.

The output is the complexity of triggered sequential checking the batch.

A.2 Divide and Conquer Checking

```
SplitComplexity[m_, x_, w_] :=
 Module[{n, y, z, i, b, r},
  n = m; (* Items remaining *)
  y = w; (* Errors remaining *)
  z = 0; (* Complexity variable *)
  If[m < x, (* if less elements than splits just do them *)
   z = m*c,
   For[i = 0, i < x, i++, (* For each split determine the
   number of errors *)
     b = Ceiling[n/(x - i)]; (* Split batch size *)
     n = n - b;
     r = Random[Integer, {Max[0, y - n], Min[y, b]}];
     z = z + If[r == 0, (* If no errors, finish, else split *)
        2 (b - 1) + c,
        y = y - r;
        If[b == 1, (* If batch has size 1, finish *)
         c,
         2 (b - 1) + c + SplitComplexity[b, x, r]
         ]
        ]
     ];
   ];
  z
  ]
```

Explanation of the Program

The input for this program is a batch size, the number of batches to split into and the number of errors that are located in the batch.

The number of items and the number of errors that are in the batch are stored in local variables. These local variables are updated as the batch is split into sub-batches to keep up with the remaining number of items and remaining number of errors. Another variable is created and set to zero to store the checking complexity. If there are less items in the batch than the number of sub-batches to split into, the entire batch is checked using sequential checking. If this is not the case a loop is entered to split the batch.

In this loop the size of a single sub-batch is determined, this is done for each sub-batch as their size may differ. The remaining number of items is divided by the remaining number of sub-batches. The number of items in the sub-batch is subtracted from the total number of items remaining. Then the number of errors in this sub-batch is randomly determined. When there are more errors remaining than there are number of items in the sub-batches that still have to be checked, the difference between these two is the minimum amount of errors. If this is not the case the minimum amount of errors is 0. The maximum amount of errors possible in the sub-batch is the least of either the remaining number of errors or the size of the sub-batch.

If there are no errors in the sub-batch, then the complexity of batch verifying the sub-batch is added to the checking complexity variable. If there are errors in the sub-batch than the number of errors in this sub-batch is subtracted from the remaining number of errors. If there is only one item in the sub-batch the complexity of verifying this item is added to the checking complexity variable. If there are more items in the sub-batch the cost of batch verifying this sub-batch is added to the checking complexity variable as well as the result of recursively invoking this program. The new variables for the program are the size of the sub-batch, the same amount to split into and the number of errors in the sub-batch. After this the loop is continued for a new sub-batch.

The end of the program returns the checking complexity variable of this run.

A.3 Random Checking

```
RandomComplexity[m_, w_] :=
 Module[{r, y, a, b, i},
  If[m == 1, (* If the batch size is 1, we are done,
  else carry on *)
   0,
   r = Random[Integer, {1, m - 1}]; (* Select a batch of
   solutions with random size *)
   y = 0; (* Will become the number of errors in the selected
   batch *)
   a = r;
   b = m;
   For[i = 1, i <= w, i++, (* For each possible error determine
   if it is in the selected batch *)
    If[a >= Random[Integer, {1, b}],
     a = a - 1;
     y = y + 1,
     False
     ];
    b = b - 1
    ];
   2 (r - 1) + c +
    If[y == 0,(* If no error in the selected batch, continue with
    remaining items, else carry on with the selected batch and
    also check all remaining items *)
     If[m - r == 1,(* If only one item remaining, do it and be done *)
      c,
      RandomComplexity[m - r, w]],
     RandomComplexity[r, y] + 2 (m - r - 1) + c +
      If[w - y == 0, (* If no errors in the remaining items,
      no additional calculations needed, else check the remaining
      items as well *)
       0,
       RandomComplexity[m - r, w - y]
       ]
     ]
   ]
  ]
```

Explanation of the Program

This program takes the batch size and the number of errors as input.

When the batch size is 1, then this program assumes the relevant calculations have already been done and terminates. Else it is assumed that there is at least one error in the batch and it takes a proper sub batch of this. A local variable is set to 0, which will represent the number of errors in the selected sub batch. Two local variables are allocated to aid in determining the number of errors in the sub batch. One variable contains the number of items in the sub batch that do not contain an error, the other contains the number of items in total that do not contain an error. It is then checked if an error is in the sub batch or not. If it is in the sub batch, the variable for number of errors in the sub batch is increased and the number of items without errors in the sub batch is decreased. If it is not in the sub batch, both variables remain unchanged. In both cases the number of items without error in total is decreased. This is done for each error and results in a number of errors that is contained in the sub batch.

If there are no errors in the sub batch, then there must be errors in the rest of the batch. The complexity for this is the complexity of checking the sub batch together with a recursive run of the program with input variables the number of items remaining and the same number of errors. One exception to this is when there is only one item remaining in the batch. In that case this item is simply checked without invoking this program again.

If there are errors in the sub batch, there can either be more or no errors remaining in the rest of the batch. If there are no errors in the rest of the batch the complexity of this is batch verifying the sub batch, batch verifying the rest of the batch and invoking this program again with a batch size equal to the sub batch and an amount of errors equal to that in the sub batch. If there are errors in the rest of the batch the complexity is the same with the addition of a run of the program with a number of items equal to the rest of the batch and the number of errors that are in these items.

The output is the complexity of random checking the batch.