数理解析研究所講究録 1426 巻 2005 年 57-63

中程度の難しさをもつ関数のモデルと方式 (Extended Abstract)

A Model and Methods for Moderately-Hard Functions (Extended Abstract)

小野寺 貴男* 田中 圭介 *
Takao Onodera Keisuke Tanaka

東京工業大学 数理・計算科学専攻
Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology

Summary— Moderately-hard functions are useful for many applications and there are quite many papers concerning on moderately-hard functions. However, the formal model for moderately-hard functions have not been proposed. In this paper, first, we propose the formal model for moderately-hard functions. For this purpose, we construct the computational model and investigate the properties desired for moderately-hard functions. Then, we propose that some particular functions can be used as moderately-hard functions. These functions are based on two ideas: the difficulty of factoring $p^r q$ and sequential computation of primitive functions.

Keywords: Moderately-hard functions, One-wayness, PRAM, Factoring.

1 Introduction

One of key ideas in cryptography is using intractable problems, i.e. problems that cannot be solved efficiently by any feasible machine, in order to construct secure protocols. There are tight connections between complexity theory and cryptography.

However, the concept of hard functions, e.g. the one-way functions, is sometimes considered to be too strong. As we will see later, many tasks, ranging from ones such as combating spam mails to ones such as few-round zero-knowledge, require another notion of intractability called moderately hardness. While there are many applications, where the moderately intractability is needed, the study of moderately hardness has not been much done, compared with the strict intractability.

Dwork and Naor [6] suggested moderately-hard functions for "pricing via processing" in order to deter abuse of resources, such as spam. Bellare and Goldwasser [4, 3] suggested "time capsules" for key escrowing in order to deter widespread wiretapping. A major issue there is to verify at escrow-time that the right key is escrowed. Rivest, Shamir, and Wagner [11] suggested "time-locks" for encrypting data so that it is released only in the future. This is the first scheme that takes into account the parallel power of attackers. They suggested using the "power function", i.e. computing $f(x) = g^{2^{2^x}} \bmod n$, where n is a product of two large primes. Without knowing the factorization of n, the best way that is known is repeated squaring—a very sequential computation in nature. However, in their setting no measures are taken to verify that the puzzle can be unlocked in the desired time. Baneh and Naor [2] introduced and constructed "timed commitment" schemes. Timed commitment is commitment in which there is an optional forced opening phase enabling the receiver to recover with effort the committed value without the help of the committer. They suggested that moderately-hard functions can be used to various applications, e.g. fair contract signing schemes, collective coin flipping schemes, and zero-knowledge protocols.

These proposed functions are all CPU-bound. The CPU-bound approach might suffer from a possible mismatch in processing among different types of machines, e.g. PDAs versus servers. In order to remedy these disparities, Adadi, Burrows, Manasse, and Wobber [1] proposed an alternative computational approach based on memory latency.

* Supported in part by NTT Information Sharing Platform Laboratories and Grant-in-Aid for Scientific Research, Ministry of Education, Culture, Sports, Science, and Technology, 14780190, 16092206.

Their suggestion is to design pricing functions requiring a moderately large number of scattered memory accesses. Since memory latencies vary much less across machines than do clock speeds, memory-bound functions are more equitable than CPU-bound functions.

As we have seen above, moderately-hard functions are useful for many applications and there are quite many papers concerning on moderately-hard functions. However, the formal model for moderately-hard functions have not been proposed. In the first half of this paper, we propose the formal model for moderately-hard functions. For this purpose, we construct the computational model and investigate the properties desired for moderately-hard functions. In the second half of this paper, we propose that some particular functions can be used as moderately-hard functions. These functions are based on two ideas: the difficulty of factoring $p^r q$ and sequential computation of primitive functions.

This paper is organized as follows. In Section 2, we provide the model of moderately-hard functions. We investigate the required properties for moderately-hard functions. In Section 3, we suggest to use some particular functions based on the difficulty of factoring $p^r q$ as moderately-hard functions. In Section 4, we propose moderately-hard functions based on the idea of sequential computing.

Due to lack of space, we omit most of the part of the model for moderately-hard functions. We also omit the details of our schemes. See the full version [9] of our paper.

2 The Model of Moderately-Hard Functions

Moderately-hard functions have many applications, e.g. timed commitment schemes, fair contract signing schemes, collective coin flipping schemes, and zero-knowledge protocols. As above, moderately-hard functions are useful tools. However, there seems no formal model for moderately-hard functions. In this section, we provide this formal model.

We define moderately hardness by dividing the definition into two parts. Let an input of a function f be x and an output be y. We say a function f is moderately hard if,

1. there is no algorithm which can computes y given x in a small amount of time and
2. f can be computed in a certain amount of time.

We also consider the second property, which we name *easy verifiability*. We say a function has the property of easy verifiability if anyone given x and y, can verify that $f(x) = y$ in a small amount of time. The easy verifiability is useful in various applications, e.g. timed commitment schemes.

The third property we desire is that f has a *shortcut*. Shortcuts of moderately-hard functions share a similar idea of trapdoors of one-way functions. A moderately-hard function with a shortcut is easy to compute.

3 Candidates of Moderately-Hard Functions: Idea 1

In this paper, we propose that some particular functions based on two ideas can be used as moderately-hard functions. In this section, we adopt several functions which are based on the difficulty of factoring composite $n = p^r q$, which both p and q are the same size primes.

The security of many cryptographic techniques depends on the intractability of the integer factorization problem. The moduli of the form $n = p^r q$ have found many applications in cryptography. For example, Fujioka, Okamoto, and Miyaguchi [7] used a modulus $n = p^2 q$ in an electronic cash scheme. Okamoto and Uchiyama [8] used $n = p^2 q$ for a public key system. Takagi [12] observed that the RSA decryption can be performed significantly faster by using a modulus of form $n = p^r q$. In all these applications, the factors p and q are approximately the same size. The security of systems relies on the difficulty of factoring n.

Boneh, Durfee, and Howgrave-Graham [5] showed that factoring $n = p^r q$ becomes easier as r gets bigger. For example, when r is on the order of $\log p$, their algorithm factors n in a polynomial time. When $n = p^r q$ with r on the order of $\sqrt{\log p}$, their algorithm is the fastest one for factoring n among the current methods.

We use a non-standard notation and write $\exp(n) = 2^n$. Then, we can recover the factor p from n and r by an algorithm with a running time of:

$$\exp\left(\frac{\log p}{r}\right) \cdot O(r^{12} (\log_2 n)^2).$$

The larger r is, the easier the factoring problem becomes.

The moderately-hard functions that we employ are based on the difficulty of the factorization of composite $n = p^r q$. Set p and q are roughly the same size and $n = p^r q$. As above description, if we deal with a large number as r, we can extract the prime factors p and q of n modestly quickly. We regard the functions in this section as moderately-hard ones by using a composite number n which can be represented as $n = p^r q$. The difficulty of computing our functions is based on the size of r. The reason why we employ n which is not a general composite number represented by $p_1^{e_1} \cdots p_k^{e_k}$ for primes $p_1, \ldots, p_k$, but a product of two primes is to change the difficulty of our functions easily. In our setting, we can change the difficulty of functions by only changing the size of r.

Here, we describe some notations that we use in this section. Let $a \in Z_n^*$. If there exists an $x \in Z_n^*$ such that $x^2 \equiv a \pmod{n}$, a is said to be a *quadratic residue* modulo n. If no such x exists, then a is called a *quadratic non-residue* modulo n. The set of all quadratic residues modulo n is denoted by Q_n and the set of all quadratic non-residues is denoted by $\tilde{Q_n}$. Let J_n be set of the elements $a \in Z_n^*$ with Jacobi symbol $\left(\frac{a}{n}\right) = 1$, where $n \geq 3$ is an odd integer.

We consider the following five functions are moderately hard.

$$f(x) = \sqrt{x} \bmod n \tag{1}$$

$$f(x) = \begin{cases} 1 & x \in Q_n \\ 0 & x \in \tilde{Q_n} \end{cases} \tag{2}$$

$$f(x) = \log_g x \bmod n \tag{3}$$

$$f(x) = \sqrt[n]{x} \bmod n^2 \tag{4}$$

$$f(x) = \sqrt[e]{x} \bmod n. \tag{5}$$

In the rest of this section, we observe these functions.

3.1 Computing Square Roots

Dwork and Naor [6] suggested a moderately-hard function based on the difficulty of computing square roots modulo p. The checking step for verification of computing requires only one multiplication. However, there is no shortcut for their function, i.e. no one can compute the function easily.

We describe their function as follows.

Preparation: A prime p of length depending on difference parameter.

Definition of f: The domain of f is Z_p. $f(x) = \sqrt{x} \bmod p$.

Verification: Given x and $y = f(x)$, check that $y^2 = x \bmod p$.

The checking step for verification of computing requires only one multiplication. In contrast, no method of computing square roots mod p is known that requires fewer than about $\log p$ multiplications. Thus, the larger we take the length of p, the larger the difference between the time needed to evaluate f and the time needed for verification.

Let p and q both primes of the same size, and r a positive integer. The first implementation of our idea is based on the difficulty of computing square roots modulo a composite number n, where $n = p^r q$. We describe the moderately-hard function as follows.

Preparation: Let $n = p^r q$, where p and q are both primes of the same size where $p \equiv q \equiv 3 \pmod{4}$, and r is a positive integer.

Definition of f: The domain of f is Q_n. $f(x) = \sqrt{x} \bmod n$.

Verification: Given x and $y = f(x)$, check that $y^2 = x \bmod n$.

Shortcut: The factors p and q of n and the integer r.

Computing f without Shortcut: Compute f according to the algorithm **ComputeSquareRoot.**

Computing f with Shortcut: Compute f according to the algorithm **ComputeSquareRoot** except for step 1.

The square roots of an element x modulo n can be extracted by the Chinese Remainder theorem quickly, if we know the prime factors p and q of n.

The function computing a square root modulo n has shortcuts. When we want to change the difficulty of the function, the only thing we have to do is changing the size of r. We do not need to change the size of outputs of the function. Therefore, even if we want to change the difficulty of the functions, we can use the same size moduli. On the other hand, the function computing a square root modulo p seems not to have shortcuts, and in order to change the difficulty of the function, we have to change the size of modulo.

Algorithm ComputeSquareRoot.

Input: a composite number n and an element $x \in Q_n$. ($n = p^r q$, where $p \equiv q \equiv 3 \pmod 4$)

Output: a square root y of x mod n.

1. Find the prime factors of n.
2. Do the following:
   2.1. Compute $r_q = x^{(p+1)/4} \bmod q$ by using the Square-and-Multiply algorithm.
3. Compute r_p such that $x \equiv r_p^2 \pmod{p^r}$ as follows where p^r is represented by $\sum_{i=0}^{l} k_i 2^i$:
   3.1. Compute $r_p = x^{(p+1)/4} \bmod p$.
   3.2. For i from 1 to l do the following:
   3.2.1. Compute $r_p = r_p + \frac{x - r_p^2}{2r_p} \bmod p^{2^i}$.
   3.2.2. Compute $r_p = r_p + \frac{x - r_p^2}{2r_p} \bmod p^r$.
4. Do the following:
   4.1. Set $a_0 \leftarrow p^r$, $b_0 \leftarrow q$, $t_0 \leftarrow 0$, $t \leftarrow 1$, $s_0 \leftarrow 1$, and $s \leftarrow 0$.
   4.2. Compute $u = \lfloor \frac{a_0}{b_0} \rfloor$ and $b = a_0 - ub_0$.
   4.3. While $b > 0$, do the following:
   4.3.1. Compute $v = t_0 - ut$ and set $t_0 \leftarrow t$ and $t \leftarrow v$.
   4.3.2. Compute $v = s_0 - us$ and set $s_0 \leftarrow s$, $s \leftarrow v$, $a_0 \leftarrow b_0$ and $b_0 \leftarrow b$.
   4.3.3. Compute $u = \lfloor \frac{a_0}{b_0} \rfloor$ and $b = a_0 - ub_0$.
5. Compute $y = r_p t q + r_q s p^r \bmod n$.
6. Return y.

3.2 Deciding Quadratic Residuosity

We can also use the decisional version of square roots problem for moderately-hard functions. In this section, we describe the function which is based on the difficulty of distinguishing $x \in J_n$ is a quadratic residue or not. Unfortunately, this function seems not to be able to check the validity of the solution directly. To solve this problem, we use prime factors of a composite number n as a shortcut and give it to the verifier. Then, the verifier can compute f quickly and verify the value by using the shortcut.

Preparation: Let $n = p^r q$, where p and q are both primes of the same size and r is a positive integer.

Definition of f: The domain of f is J_n.

$$f(x) = \begin{cases} 1 & x \in Q_n \\ 0 & x \in \tilde{Q}_n \end{cases} \tag{6}$$

Verification: Given x, $y = f(x)$, p, and q, check by using the algorithm **DistinguishQuadraticResidue** except for step 1.

Shortcut: The factors p and q of n and the integer r.

Computing f without Shortcut: Compute f according to the algorithm **DistinguishQuadraticResidue.**

Computing f with Shortcut: Compute f according to the algorithm **DistinguishQuadraticResidue** except for step 1.

Algorithm DistinguishQuadraticResidue.

Input: a composite number n ($n = p^r q$) and an element $x \in J_n$.

Output: $y = 1$ if $x \in Q_n$ and $y = 0$ if $x \in \tilde{Q}_n$

1. Find the prime factors of n.
2. Compute the Legendre symbol $\left(\frac{x}{p}\right)$ of x mod p by the equation $\left(\frac{x}{p}\right) = x^{(p-1)/2} \bmod p$.
3. In a similar way, compute the Legendre symbol $\left(\frac{x}{q}\right)$ of x mod q.
4. Return $y = 1$ if $\left(\frac{x}{p}\right) = \left(\frac{x}{q}\right) = 1$ and $y = 0$ otherwise.

3.3 Computing Discrete Logarithms

The security of many cryptographic techniques depends on the intractability of the discrete logarithm problem. Let P be a prime. We consider a group Z_P^* of order $P - 1$ with generator α. Let $P - 1 = 2p^r q$, where p and q are both primes of the same size and r is a positive integer. In cryptographic settings, we assume that there is no algorithm for solving the discrete logarithm problem in practical time. However, if we take a large number for r to factorize $P - 1$, the discrete logarithm problem modulo a large prime is reduced to that modulo a small prime. This means that the discrete logarithm problem of this type is a modestly difficult (not infeasible) one. In this way, a function for computing discrete logarithms can be used for a moderately-hard function.

We describe the moderately-hard function based on the difficulty of the discrete logarithm problem.

Preparation: Let $n = 2p^r q$, where p and q are both primes of the same size, $n + 1$ is also a prime, and r is a positive integer. Let α be a generator of Z_{n+1}^*.

Definition of f: The domain of f is Z_{n+1}^*. $f(x) = \log_\alpha x$.

Verification: Given α, x, and $y = f(x)$, check that $x = \alpha^y \bmod n + 1$.

Shortcut: The factors p and q of n and the integer r.

Computing f without Shortcut: Compute f according to the algorithm **ComputingDiscreteLogarithm.**

Computing f with Shortcut: Compute f according to the algorithm **ComputingDiscreteLogarithm** except for step 1.

The domain Z_P^* of this function is dense, and this property is useful for many applications. For example, if we use this function for combatting spam mail, we make the sender compute $f(m)$, where m is a message, to charge some computational cost to the sender. Here, the message space is restricted to the domain of the functions. Therefore, this dense property is useful. Note that the other functions we employ in this section do not have this property.

Algorithm ComputingDiscreteLogarithm.
Input: a composite number n ($n = 2p^r q$), where $n+1$ is a prime, a generator α of Z_{n+1}^*, and an element $x \in Z_{n+1}^*$.
Output: the discrete logarithm $y = \log_\alpha x$.

1. Find the prime factors of n.
2. Do the following:
   (Compute $y_2 = y \bmod 2$.)
   2.1. Compute $\bar{\alpha} = \alpha^{p^r q} \bmod n+1$ and $\bar{x} = x^{p^r q} \bmod n+1$.
   2.2. Compute $y_2 = \log_{\bar{\alpha}} \bar{x}$.
3. Do the following:
   (Compute $y_p = l_0 + l_1 p + \cdots + l_{r-1} p^{r-1}$,where $y_p = y \bmod p^r$.)
   3.1. Set $\gamma \leftarrow 1$ and $l_{-1} \leftarrow 0$.
   3.2. Compute $\bar{\alpha} = \alpha^{n/p}$.
   3.3. For i from 0 to $r-1$ do the following:
      3.3.1. Compute $\gamma = \gamma \alpha^{l_{i-1} p^{i-1}}$ and $\bar{x} = (x\gamma^{-1})^{n/p^{i+1}}$.
      3.3.2. Compute $l_i = \log_{\bar{\alpha}} \bar{x}$.
   3.4. Compute $y_p = l_0 + l_1 p + \cdots + l_{r-1} p^{r-1}$.
4. Do the following:
   (Compute $y_q = y \bmod q$.)
   4.1. Compute $\bar{\alpha} = \alpha^{2p^r} \bmod n+1$ and $\bar{x} = x^{2p^r} \bmod n+1$.
   4.2. Compute $y_q = \log_{\bar{\alpha}} \bar{x}$.
5. Compute the integer y which satisfies, $y = y_2 \bmod 2$,$y = y_p \bmod p^r$, and $y = y_q \bmod q$ by using the Chinese Remainder theorem.
6. Return y.

3.4 Computing n-th Roots

Paillier [10] proposed an encryption scheme whose one-wayness is based on the problem of finding an integer $x \in Z_{n^2}^*$, where n is a product of two primes, which is represented as $x = a^n \bmod n^2$ for an integer $a \in Z_{n^2}^*$. Paillier assumed that there is no efficient algorithm for this problem. However, this problem can be solved in a small amount of time if we know the prime factors of n. This means if we take a large number for r to factorize n, a function for computing an n-th residue can be used for a moderately-hard function.

We describe the moderately-hard function as follows.

Preparation: Let $n = p^r q$, where p and q are both primes of the same size and r is a positive integer.

Definition of f: The domain of f is Z_n^*. $f(x) = \sqrt[n]{x} \bmod n^2$.

Verification: Given x and $y = f(x)$, check that $y^n = x \bmod n^2$.

Shortcut: The factors p and q of n and the integer r.

Computing f without Shortcut: Compute f according to the algorithm **Compute n-thRoot**.

Computing f with Shortcut: Compute f according to the algorithm **Compute n-thRoot** except for step 1.

Algorithm Compute n-thRoot.
Input: a composite number n ($n = p^r q$) and an element $x \in Z_n^*$.
Output: an n-th root y of x.

1. Find the prime factors of n.
2. Do the following:
   2.1. Compute $a_0 = p^{2r-1} q(p-1)(q-1)$ and set $b_0 \leftarrow n$, $t_0 \leftarrow 0$, and $t \leftarrow 1$.
   2.2. Compute $l = \lfloor \frac{a_0}{b_0} \rfloor$ and $r = a_0 - l b_0$.
3. While $r > 0$, do the following:
   3.1. Compute $s = (t_0 - lt) \bmod a_0$ and set $t_0 \leftarrow t$ and $t \leftarrow s$.
   3.2. Set $a_0 \leftarrow b_0$ and $b_0 \leftarrow r$ and compute $l = \lfloor \frac{a_0}{b_0} \rfloor$, and $r \leftarrow a_0 - l b_0$.
4. Compute $y = x^t \bmod n^2$.
5. Return y.

3.5 Computing e-th Roots

The RSA scheme and these variants employ different composite moduli. As a part of the public key, each scheme employs e relatively prime to the modulus used in the scheme. These cryptosystems are based on the difficulty of the factorization of a composite number. We employ a modulus $n = p^r q$, where p and q are both primes of the same size and r is a positive integer.

See [9] for details.

3.6 Observation

We briefly mention the distinguished properties of the functions mentioned above. The function 2, which distinguishes an element is a quadratic residue or not, is only a function whose way of computation for the verification step and the computing step with shortcut information are the same. For the other functions, the verification steps require only one exponentiation. In particular, the function 1 computing a square root requires one multiplication and the function 5 computing an e-th root requires two multiplication when $e = 3$.

The domains of the functions 1-5 are Q_n, Q_n, Z_p^*, Z_n^*, and Z_n^*, respectively. The function 3 is especially useful when we choose input elements randomly from the domain.

4 New Moderately-Hard Functions: Idea 2

The functions in the Section 3 are not immune to the parallel power of the attacker. The first scheme taking into account such attackers is one by Rivest, Shamir, and Wagner [11]. They suggested using the "power function," i.e. computing $f(x) = g^{2^{2^x}} \bmod n$, where n is a product of two large primes. Without knowing the factorization of n, the best way that is known is repeated squaring—a sequential computation in nature. The power function was also employed by Boneh and Naor [2]. In their paper, several applications which are immune to parallel exhaustive search attack were proposed by using the power function. As far as we know, the proposed functions which are immune to the parallel attackers seem to be only the power function.

In this section, we propose another functions which are immune to the parallel attackers. First, we define moderately-hard functions by using an abstract function. Then, we apply the random oracles to this construction.

4.1 The Moderately-Hard Functions with Abstract Functions

Our suggested functions are based on the idea of the sequential computation. We use the cut and choose technique for verification, which reduces the communication cost to verify.

We define the functions by using an abstract function F whose domain and range are exactly the same. We let the domain and range of our function be the same as those for F. Since our proposed function is defined in a general way, we can construct multiple functions depending on the choice of F. For example, if we set a function F as $F(x) = x^2$, the resulting function is the same as the power function.

We describe our proposed function f as follows.

Preparation: Let the domain and range of F be G.

Definition of f: The domain of f is G. $f(x_1) = \{x_1, \ldots, x_k\}$, where $x_i = F(x_{i-1})$ for $i = 2, \ldots, k$. We can change the difficulty by choice of k.

Verification: Choose s numbers $a_1, \ldots, a_s$ at random, where $s \leq k$, and check whether $x_{a_i} = F(x_{a_i - 1})$ or not for $a_1, \ldots, a_s$. Here, s is a number which is large enough not to deceive the verifier.

Shortcut: F's shortcut.

Computing f without Shortcut: Compute $x_i = F(x_{i-1})$ for $i = 2, \ldots, k$ without F's shortcut.

Computing f with Shortcut: Compute $x_i = F(x_{i-1})$ for $i = 2, \ldots, k$ by using F's shortcut at computation of F for each i ($i = 1, \ldots, k$).

The existence of the shortcut for F implies that for f.

4.2 The Moderately-Hard Functions with the Random Oracles

We observe an example of F. As far as we know, there exists no memory-bound function which is immune to the parallel power of the attacker. In this section, by using the random oracles, we can construct a function which has both properties.

We employ the random oracles for constructing a function. We describe the computing algorithm for the function. Assume $\mathcal{A}$ is the person who wants to compute the function and $\mathcal{B}$ is the verifier of the validity of the function. Our function involves a large fixed forever table T of truly random integers. Both $\mathcal{A}$ and $\mathcal{B}$ have the table T. We consider $\mathcal{A}$ and $\mathcal{B}$ as PRAM algorithms. Before we present the algorithm, we introduce hash functions H_0, H_1, H_2, H_3, and H_4 for changing the sizes of inputs. We model them as idealized random functions, which we call the random oracle. The function H_0 is used during initialization. It takes as input an element x and a trial number k and returns an array A. The function H_1 takes an array A as input and returns an index c into the table T. The function H_2 takes as input an array A and an element of T and returns a new array, which obtains assigned to A. The function

H_3 takes as input an array A and returns a string of s bits. The function H_4 takes as input a string of e bits and returns a new trial number k. We described our function as the algorithm **Computing** f.

Algorithm Computing f.
Input: a table T, an element x, and a trial number k
Output: $\{(c_i, T[c_i])\}$ for $i = 1, \ldots, t$ if $\mathcal{A}$ computes H_1 t times.

1. Set $j \leftarrow 0$
2. Compute $A = H_0(x, k)$.
3. For i from 0 to l do the following:
   3.1. Compute $c_j = H_1(A)$ and $A = H_2(A, T[c_j])$.
   3.2. Compute $j = j + 1$.
4. If all bits of $H_3(A)$ are zero, do the following:
   4.1. Set $k = H_4(H_3(A))$.
   4.2. Go to Step 1.
5. Return $((c_1, T_1), \ldots, (c_j, T[c_j]))$.

The part except step 4 in this algorithm is F represented in Section 4.1. We now estimate the expected running time of $\mathcal{A}$ and $\mathcal{B}$. Since $\mathcal{A}$ obtains an array such that $H_3(A) = 0^s$ with probability $1/2^s$ in step 3, the expected number of evaluating H_3 is 2^s. Furthermore, we require and l times computation of H_1 and H_2 before evaluating $H_3(A)$. Therefore, the expected number of evaluating H_1 for $\mathcal{A}$ is $l2^e$.

On the other hand, $\mathcal{B}$ can verify the value in parallel. If the size of T is twice as the number of $\mathcal{A}$'s local memory, $\mathcal{A}$ has to access the table T $(l2^e)/2$ times on average. If $\mathcal{B}$ stores $\mathcal{A}$'s outputs $\{(c_i, T[c_i])\}$ for $i = 1, \ldots, l$ and each RAM of s RAMs in $\mathcal{B}$ accesses to T and checks the validity of l/s $(c_i, T[c_i])$ separately, $\mathcal{B}$ can verify efficiently with respect to the number of table access.

References

[1] Adadi, M., Burrows, M., Manasse, M., and Wobber, T. Moderately Hard, Memory-Bound Functions. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium* (February 2003).

[2] Baneh, D., and Naor, M. Timed Commitments. In *Advances in Cryptology – CRYPTO 2000* (Santa Barbara, California, USA, August 2000), M. Bellare, Ed., vol. 1880 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 236–254.

[3] Bellare, M., and Goldwasser, S. Encapsulated Key Escrow. MIT laboratory for Momputer Science Technical Report 688, April 1996.

[4] Bellare, M., and Goldwasser, S. Verifiable Partial Key Escrow, 1997.

[5] Boneh, D., Durfee, G., and Howgrave-Graham, N. Factoring $n = p^r q$ for Large r. *Lecture Notes in Computer Science 1666* (1999), 326–337.

[6] Dwork, C., and Naor, M. Pricing via Processing -or- Combatting Junk Mail. In *Advances in Cryptology – CRYPTO '92* (Santa Barbara, California, USA, August 1992), E. F. Brickell, Ed., vol. 740 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 139–147.

[7] Fujioka, A., Okamoto, T., and Miyaguchi, S. ESIGN: an Efficient Digital Signature Implementation for Smartcards.

[8] Okamoto, T., and Uchiyama, S. A New Public Key Cryptosystem as Secure as Factoring. In *Advances in Cryptology – EUROCRYPT '98* (Espoo, Finland, May 1998), K. Nyberg, Ed., vol. 1403 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 308–318.

[9] Onodera, T., and Tanaka, T. A Model and Methods for Moderately-Hard Functions. Research Report C-202, Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology, 2004.

[10] Paillier, P. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *Advances in Cryptology – EUROCRYPT '99* (Prague, Czech Republic, May 1999), J. Stern, Ed., vol. 1592 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 223–238.

[11] Rivest, R., Shamir, A., and Wagner, D. Time Lock Puzzles and Timed Release Cryptography. Technical report, MIT/LCS/TR-684.

[12] Takagi, T. Fast RSA-Type Cryptosystem modulo $p^k q$. In *Advances in Cryptology – CRYPTO '98* (Santa Barbara, California, USA, August 1998), H. Krawczyk, Ed., vol. 1462 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 318–326.