ISSN 1342-2812

Research Reports on Mathematical and Computing Sciences

A model and Methods for Moderately-Hard Functions

Takao onodera and Keisuke Tanaka

December 2004, C–202

Department of
Mathematical and
Computing Sciences
Tokyo Institute of Technology

SERIES C: Computer Science

A Model and Methods for Moderately-Hard Functions

Takao Onodera Keisuke Tanaka *

Dept. of Mathematical and Computing Sciences
Tokyo Institute of Technology
W8-55, 2-12-1 Ookayama, Meguro-ku, Tokyo 152-8552, Japan
{onodera0, keisuke}@is.titech.ac.jp

December 22, 2004

Abstract

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 required 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 [7] 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 [12] 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

*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.

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. 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 required 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 construct the computational model and 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. We conclude in Section 5.

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 should model the required properties for moderately-hard functions reflecting the real world. For example, in real contemporary hardware, there are (at least) two kinds of space: ordinary memory (the vast majority) and cache—a small amount of storage on the same integrated circuit chip as the central processing unit. Cache can be accessed roughly 100 times more quickly than ordinary memory, so the computational model needs to differentiate them. In addition, when a desired value is not in cache (cache miss) and an access to memory is made, a small block of adjacent words (a cache line) is brought into cache simultaneously "for free".

2.1 The Model of Computation

Before we specifying the required properties of moderately-hard functions, we provide the the model of computation. We consider the machines with memory. The random-access machine (RAM) models the essential features of traditional computers. The random-access machine has a central processing unit (CPU) and a random access memory unit. The CPU has a small number of storage locations called registers whereas the random-access memory has a large number. All operations performed by the CPU are performed on data stored in its registers.

The CPU implements a fetch and execute cycle in which it alternately reads an instruction from a program stored in the random-access memory and executes it. This memory is called random-access

because the time to access a stored word is the same for all words[1]. The random-access memory has $m = 2^{\mu}$ storage locations each containing a b-bit word, where μ and b are integers. The combination of this memory and the CPU described above is the bounded-memory RAM. When no limit is placed on the number of memory words, this combination defines the unbounded-memory RAM.

A parallel computer can perform more than one operation at a time. Parallelism is common in computer science today.

The parallel random-access machine (PRAM), the canonical structured parallel machine, consists of a bounded set of processors and a common memory containing a potentially unlimited number of words. Each processor is similar to the random-access machine (RAM) except that its CPU can access locations in both its local random-access memory and the common memory. During each PRAM step, the RAMs execute the following steps synchronously: they (a) read from the common memory, (b) perform a local computation, and (c) write to the common memory. Each RAM has its own program and program counter as well as a unique identifying number that it can access to make processor-dependent decisions.

If access by more than one RAM to the same location is disallowed, access is exclusive. If this restriction does not apply, access is concurrent. Four combinations of these classifications apply to reading and writing. The strongest restriction is placed on the Exclusive Read/Exclusive Write (EREW) PRAM, with successively weaker restrictions placed on the Concurrent Read/Exclusive Write (CREW) PRAM, the Exclusive Read/Concurrent Write (ERCW) PRAM, and the Concurrent Read/Concurrent Write (CRCW) PRAM.

In performing a computation on a PRAM, it is typically assumed that the input is written in the lowest numbered locations of the common memory. PRAM computations are characterized by the number of RAMs in use, and that of PRAM steps taken. Both measures are usually stated as a function of the size of a problem instance, i.e. the number of input words and their total length in bits.

In this paper, we consider a restricted version of the random-access machine for our purpose. A restricted PRAM consists of several restricted RAMs and a common memory. Each restricted RAM has a local random-access memory and several counters. We take into consideration plural restricted RAMs for our restricted PRAM setting. We describe the restricted PRAM model in Figure 1. These restricted RAMs run according to each own algorithm. However, these cannot compute even basic calculation, e.g. additions, by themselves. Instead, several oracles are prepared. There exists the addition oracle, the multiplication oracle, the comparison oracle, the exponentiation oracle, and the random oracles (if exist) for computations. The restricted RAMs run and access to these oracles simultaneously according to the descriptions in their algorithms. Let the restricted RAMs be $\mathrm{RAM}_1, \cdots, \mathrm{RAM}_n$. Each restricted RAM has the local counters which count the number of access to the oracles. The counters are prepared to each oracle. For example, if the RAM_1 accesses the addition oracle once, the counter of RAM_1 for the addition oracle is increased by one. Except these counters for the oracles, each restricted RAM has the counter for access to the common memory, and that for the number of steps of the restricted RAM's algorithm. Each restricted RAM computes the intended function by accessing the oracles according to its own description of the algorithm. We estimate the computational complexity as the function of the values of the counters. We call the restricted RAM and PRAM described above RAM and PRAM, respectively, in the rest of this paper.

In our PRAM model, we restrict the access of RAMs to the common memory as done in the standard PRAM model. The strongest restriction is the Exclusive Read/Exclusive Write (EREW) PRAM, with successively weaker restrictions is the Concurrent Read/Exclusive Write (CREW) PRAM, the Exclusive Read/Concurrent Write (ERCW) PRAM, and the Concurrent Read/Concurrent Write (CRCW) PRAM.

For example, in order to estimate the costs in the memory-bound function, we can use the above general model with the following setting. In our PRAM model, the common memory corresponds to the ordinary

[1]The Turing machine has a tape memory in which the time to access a word increases with its distance from the tape head.

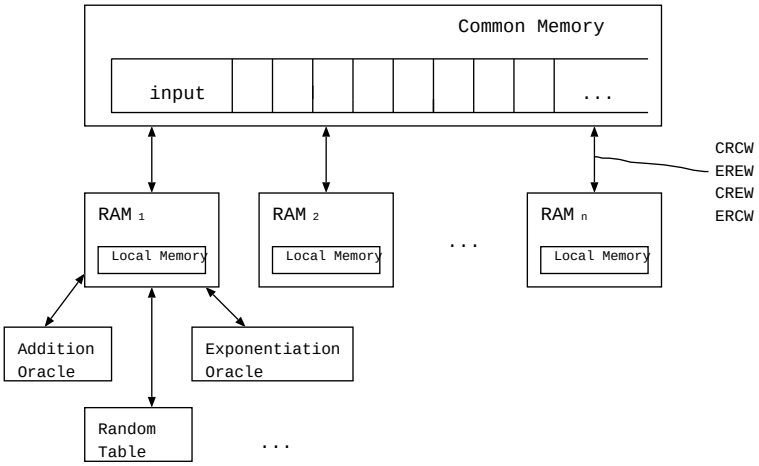

Figure 1: The restricted PRAM.

memory and the local memory of each RAM corresponds to the cache. If there is not desired elements in the local memory, the RAM accesses to the common memory and fetches the elements. In contrast to accessing to the common memory with a cost, we assume each RAM can access to the own local memory as many times as needed for free.

2.2 Properties of Moderately-Hard Functions

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 require 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. For example, we consider the situation that the person who wants to send an electronic mails has to compute a moderately-hard function in order to preventing junk mails. If we use a moderately-hard function which has a shortcut, the shortcut permits the post office to grant bulk mails, such as an announcement of a new product, at a price chosen by the post office, circumventing the cost of directly computing the moderately-hard function for each recipient. The sender pays a fee and prepares a set of letters and one of the trusted agents computes the moderately-hard function as needed for all the letters, using the shortcut. Since the fee is levied to deter junk mail and not to cover the actual costs of the mail, it can simply be turned over to the recipients of the message.

We formally define the moderately hardness, the easy verifiability, and the shortcuts as follows.

The difficulty of functions is decided by the computational complexity in the PRAM model proposed in Section 2. Assume the PRAM has n RAMs, $\mathrm{RAM}_1, \ldots, \mathrm{RAM}_n$. Define a moderately-hard function f as $X \mapsto Y$.

Definition 1 (Moderately Hardness). *A function f is called moderately hard if the following two conditions hold:*

1. *Consider an algorithms A in the PRAM model. For all i, let the numbers of RAM_i access to the addition oracle, the multiplication oracle, the comparison oracle, the exponentiation oracle, the random oracles, and the common memory, and that of steps of A, be a_i, b_i, c_i, d_i, e_i, f_i, and g_i, respectively. For every algorithm A, every positive polynomial $p(\cdot)$, and all sufficiently large s, $\Pr[A(x, 1^s) \in Y] < \frac{1}{p(s)}$, where x is chosen from X at random and s is the size of a element of Y, in time*
$$F(a_1 \ldots, a_n, b_1, \ldots, b_n, \ldots, g_1, \ldots, g_n) < C,$$
where F is a cost function and C is a predetermined number which determines the difficulty of computing f.

2. *Consider an algorithm B in the PRAM model. For all i, let the numbers of RAM_i access to the addition oracle, the multiplication oracle, the comparison oracle, the exponentiation oracle, the random oracles, and the common memory, and that of steps of B, be a_i^*, b_i^*, c_i^*, d_i^*, e_i^*, f_i^*, and g_i^*, respectively. There exists a probabilistic algorithm B, which on input $x \in X$, outputs $y \in Y$, where $f(x) = y$, in time*
$$F(a_1^* \ldots, a_n^*, b_1^*, \ldots, b_n^*, \ldots, g_1^*, \ldots, g_n^*) < C^*,$$
where C^* is a predetermined number.

Definition 2 (Easy Verifiability). *Consider an algorithms A in the PRAM model. For all i, let the numbers of RAM_i access to the addition oracle, the multiplication oracle, the comparison oracle, the exponentiation oracle, the random oracles, and the common memory, and that of steps of A be a_i', b_i', c_i', d_i', e_i', f_i', and g_i', respectively. There exists a probabilistic algorithm A, which on input $x \in X$, $y \in Y$, decides whether $f(x) = y$ or not, in time*
$$F(a_1', \ldots, a_n', b_1', \ldots, b_n', \ldots, g_1', \ldots, g_n') < C',$$
where F is a cost function and C' is a predetermined number.

Let S be the set of shortcuts, which let computing f be easy, of f.

Definition 3 (Shortcuts). *Consider an algorithms A in the PRAM model. For all i, let the numbers of which RAM_i access to the addition oracle, the multiplication oracle, the comparison oracle, the exponentiation oracle, the random oracles, and the common memory, and that of steps of A, be a_i'', b_i'', c_i'', d_i'', e_i'', f_i'', and g_i'', respectively. The function f has a shortcut if there exists a probabilistic algorithm A, which on input $x \in X$ and $s \in S$, computes $y \in Y$, where $f(x) = y$, in time*
$$F(a_1'', \ldots, a_n'', b_1'', \ldots, b_n'', \ldots, g_1'', \ldots, g_n'') < C'',$$
where F is a cost function and C'' is a predetermined number.

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 [8] used a modulus $n = p^2q$ in an electronic cash scheme. Okamoto and Uchiyama [9] used $n = p^2q$ for a public key system. Takagi [13] observed that the RSA decryption can be performed significantly faster by using a modulus of form $n = p^rq$. 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 $n = p^rq$ 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 $n = p^rq$ 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 $n = p^rq$. Set p and q are roughly the same size and $n = p^rq$. 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 $n = p^rq$. The difficulty of computing our functions is based on the size of r. The reason why we employ n which is not an abstract 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 [7] 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 Information: Compute f according to the algorithm **ComputeSquareRoot.**

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

Algorithm ComputeSquareRoot.
Input: a composite number n ($n = p^r q$, where $p \equiv q \equiv 3 \pmod 4$) and an element $x \in Q_n$.
Output: a square root y of $x \bmod n$.

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 $y = r_p tq + r_q sp^r \bmod n$.
6. Return y.

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. In order to see that the output of the algorithm

ComputeSquareRoot is correct, we observe step 3. If p is a prime and $p \equiv 3 \pmod 4$, the square roots of x modulo p are $\pm x^{p+1/4}$.

Theorem 4. *Let s, t be integers where $s \leq t \leq 2s$, p be a prime, and x be an element such that $x = a^2 \bmod p^s$ for an element $a \in Z_n^*$. If we know a, we can compute the square roots c of x modulo p^t as $c = \pm a \pm (x - a^2)/2a \bmod p^t$, which holds $x = c^2 \bmod p^t$*

Proof: We set $c = \pm a + yp^s \bmod p^t$.

We have

$$\begin{aligned} x &= a^2 \pm 2ayp^s + y^2 p^{2s} \bmod p^t \\ x &= a^2 \pm 2ayp^s \bmod p^t. \end{aligned}$$

Therefore,

$$y = \pm \frac{x - a^2}{2ap^s} \bmod p^t.$$

Consequently, $c = \pm a \pm (x - a^2)/2a \bmod p^t$. □

Theorem 4 means if we know the square roots of x modulo p^s, we also know those modulo p^t. Here, t, s are integers, where $s \leq t \leq 2s$. Therefore, if we compute a square root of x modulo p, we can compute a square root of x modulo p^r quickly. More precisely, if we know a square root modulo p then we first compute that modulo p^2, and we next compute that modulo p^4 in a similar way. Finally, we can obtain that modulo p^r.

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.

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 Information: Compute f according to the algorithm **DistinguishQuadraticResidue.**

Computing f with Shortcut Information: 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 Information: Compute f according to the algorithm **ComputingDiscreteLogarithm.**

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

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.

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.

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 Information: Compute f according to the algorithm **Compute n-thRoot**.

Computing f with Shortcut Information: 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 $r = a_0 - lb_0$.
3. While $r > 0$, do the following:
3.1. Compute $s = (t_0 - lt) \bmod p^{2r-1}q(p-1)(q-1)$ 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 $r \leftarrow a_0 - lb_0$.
4. Compute $y = x^t \bmod n^2$.
5. Return y.

3.5 Computing e-th Roots

Rivest, Shamir, and Adleman [11] found the most widely used public-key cryptosystem, which is called RSA cryptosystem. Takagi [13], Okamoto and Uchiyama [9], and Catalano, Gennaro, and Howgrave-Graham [6] also proposed new cryptosystems. 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. We define a function as a moderately-hard one 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. Select a random number e such that $1 < e < p^{r-1}(p-1)(q-1)$ and $\gcd(e, p^{r-1}(p-1)(q-1)) = 1$.

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

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

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

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

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

If we set e small, say $e = 3$, the verification step require an only small operation.

Algorithm Compute e-thRoot.
Input: a composite number n ($n = p^r q$) and an element $x \in Z_n^*$.
Output: an e-th root y of x.
1. Find the prime factors of n.
2. Do the following:
2.1. Compute $a_0 = p^{r-1}(p-1)(q-1)$ and set $b_0 \leftarrow e$, $t_0 \leftarrow 0$, and $t \leftarrow 1$.
2.2. Compute $l = \lfloor \frac{a_0}{b_0} \rfloor$ and $r = a_0 - lb_0$.
3. While $r > 0$, do the following:
3.1. Compute $s = (t_0 - lt) \bmod p^{r-1}(p-1)(q-1)$ 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 $r \leftarrow a_0 - lb_0$.
4. Compute $y = x^t \bmod n$.
5. Return y.

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 [12]. 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 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 Information: Compute $x_i = F(x_{i-1})$ for $i = 2, \ldots, k$ without F's shortcut.

Computing f with Shortcut Information: 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.

5 Concluding Remarks

In this paper, we have proposed the formal model for moderately-hard functions. For this purpose, we have constructed the computational model and investigated the properties required for moderately-hard functions. Then, we have proposed 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 the sequential computation of the primitive functions.

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] Catalano, D., Gennaro, R., Howgrave-Graham, N., and Nguyen, P. Paillier's Cryptosystem Rvisited. In *8th Symposium on Computer and Communications Security* (2001), ACM, pp. 206–214.

[7] 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.

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

[9] 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.

[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 Adleman, L. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM 21*, 2 (1978), 120–126.

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

[13] 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.