GGM 擬亂函數之效能改進

Performance Improvement for the GGM-Construction of Pseudorandom Functions

陳昱升 洪國寶 劉兆樑
Yu-Sheng Chen Gwoboa Horng Chao-Liang Liu
中興大學資科所 中興大學資科所 中興大學資科所
Institute of Computer Science, National Chung Hsing University
e-mail : { s9356047, gbhorng, s9056001}@cs.nchu.edu.tw

摘要

虛擬隨機函數，即是一個無法以機率式演算法在多項式時間內與真正的隨機函數做出區別的函數，我們亦可稱之為擬亂函數。而擬亂函數由Goldreich、Goldwasser 及 Mical 三位學者首先定義並具體建構之(簡稱 GGM-Construction)。在本文中我們提出一個簡單變形的版本，而此版本可降低原始建構方式的計算量。值得一提的是，若假設產生 ℓ 位元虛擬亂數所需成本之為 $\theta(\ell)$，則可以證明我們變形的方法是所有 GGM-Construction 模式下最佳的。

關鍵詞：密碼學、擬亂函數、虛擬亂數產生器、虛擬亂數。

Abstract

A pseudorandom function is a function that cannot be efficiently distinguished from a truly random function. The first construction of pseudorandom functions was introduced by Goldreich, Goldwasser, and Micali (GGM-construction). In this paper, we propose a simple variant of the GGM-construction that works slightly faster than the original one. Interestingly, we show that our construction is optimal under the assumption that the cost of generating ℓ pseudorandom bits is $\theta(\ell)$.

Keywords: cryptography, pseudorandom function, pseudorandom generator, pseudorandomness.

1 Introduction

Randomness is an important part of cryptographic applications. However, generating true randomness has been one of computer science's most difficult challenges. In practice “pseudorandomness” is often used instead of true randomness.

For cryptographic applications, we require that pseudorandom sequences cannot be efficiently distinguished from truly random sequences. Pseudorandom generators, introduced by Blum and Micali[3] and Yao[14] , are used to produce such pseudorandom sequences. Though they are designed to generate long pseudorandom sequences, yet they do not provide efficiently random access to bits of a huge pseudorandom sequence (even if they do, their security in this “random” usage is not guaranteed).

Pseudorandom functions, introduced by Goldreich et al.[6] , are more powerful than pseudorandom generators. Informally, a pseudorandom function has the property that its input-output behavior is computationally indistinguishable from that of a random function and evaluating such a function can be implemented by an efficient algorithm. Pseudorandom functions have a number of applications[2] [5] [7] [10] . We may first design a scheme allowing the users to have black-box access to a random function and prove its security. Then replace the random function with a pseudorandom function. It can be argued that after this replacement the scheme is still secure.

In addition to introducing pseudorandom functions, Goldreich et al.[6] proposed a construction of length-preserving pseudorandom functions (the GGM-construction) using pseudorandom generators as building blocks. Given a pseudorandom function $f_x : \{0,1\}^k \mapsto \{0,1\}^k$, each evaluation requires k calls to the underlying pseudorandom generator. Though it is not very efficient, yet an advantage of such a *generic* construction is that a pseudorandom function can be built using *any* pseudorandom generators. This work gives a simple variant of the GGM-construction and shows that it can work slightly faster. We also show that our construction is optimal under the assumption that the cost of generating ℓ pseudorandom bits is $\theta(\ell)$.

We remarked that based on specific mathematical hard problems, more efficient pseudorandom functions are possible. Using pseudorandom synthesizers as building blocks, Naor et al.[9] gave a parallel construction of pseudorandom functions which can be evaluated in ***NC*²**.[1] Naor and Reingold[12] showed two constructions of pseudorandom functions: one construction of length-preserving pseudorandom functions based on the Decisional Diffie-Hellman assumption, and the other construction of 1-bit-output pseudorandom functions based on the Factoring assumption. Both constructions use roughly 2*k* modular multiplications per evaluation. In [11] , Naor et al. further improved the later construction so that the functions can be length-preserving and each evaluation uses roughly 3k modular multiplications.

2 Preliminaries

2.1 Definitions

In this section we will briefly review the definitions of Pseudorandom Generators and Pseudorandom Functions. The following notations will be used:

$x \in_R X$: *x* is randomly chosen from the set *X*.

I_k: the set of all k-bit strings.

H_k: be the set of all functions from I_k into I_k.

We say that a set S_k is a *samplable* set if there exists a probabilistic polynomial-time (PPT) algorithm that on input 1^k outputs $r \in_R S_k$.

A pseudorandom generator is a polynomial-time algorithm that can stretch its random input to a polynomially long string. There are several definitions of a cryptographically strong pseudorandom generator[1] [3] [4] [8] . For our purpose, the following definition for pseudorandom generators would be sufficient.[2]

Definition 2.1.1 (Pseudorandom Generators)

Let *P* , P_I be polynomials. Let S_k be a *samplable* set. Let $G_k = \{g_i\}_{i\in S_k}$ be a collection of $I_k \mapsto I_{P(k)}$ functions. If the following conditions hold, then G_k is called a collection of pseudorandom generators:

(1) (*Efficient computation*) There exists a polynomial-time algorithm *A* such that $A(i,x) = g_i(x)$, where $i \in S_k$ (index), and $x \in I_k$ (a seed of the pseudorandom generator g_i).

(2) (*Pseuodrandomness*) Let U_k be a set of $P_I(k)$ strings, each $P(k)$-bit long. For any PPT algorithm A_G, for any polynomials P_1, *Q*, and for all sufficiently large *k*, $| p_k^G - p_k^R | < \frac{1}{Q(k)}$, where p_k^G denotes the probability that $A_G(1^k, U_k) = 1$ if U_k consists of $P(k)$-bit strings output by some pseudorandom generator $g_i \in G_k$ on random *k*-bit seeds, where $i \in_R S_k$, and p_k^R denotes the probability that $A_G(1^k, U_k) = 1$ if U_k consists of random $P(k)$-bit strings.

We now move on to the definition of a pseudorandom function[6] [8] [9] [12] . Loosely speaking, a pseudorandom function is a collection of functions which cannot be efficiently distinguished from a randomly chosen function by any adversary that has access to the functions as a black box. Specifically, we give the following definition:

Definition 2.1.2 (Pseudorandom Functions)

Let S_k be a *samplable* set. Let $F_k = \{f_x\}_{x\in S_k}$ be a collection of $I_k \mapsto I_k$ functions. If the following conditions hold, then F_k is called a collection of pseudorandom functions:

(1) (*Efficient evaluation*) There exists a polynomial-time algorithm *A* such that $A(x,y) = f_x(y)$, where $x \in S_k$ (the key of the pseudorandom function f_x) and $y \in I_k$ (query).

(2) (*Pseudorandomness*) For any PPT algorithm A_F, for all polynomial *Q*, and for all sufficiently large *k*, $| p_k^F - p_k^H | < \frac{1}{Q(k)}$, where p_k^F denotes the probability that A_F outputs 1 on input 1^k and has access to the oracle O_{f_x} for some function $f_x \in F_k$,

[1] ***NC*²** is the class of all problems solvable in O(log²k) paralle time with polynomial amount of total work. See [13] .

[2] Actually, we define a pseudorandom generator as a polynomial-time algorithm whose output sequences on random seeds pass all polynomial-time statistical test. See[6] [14] .

where $x \in_R S_k$, and p_k^H denotes the probability that A_F outputs 1 on input 1^k and has access to the oracle O_h for some function $h \in_R H_k$.

2.2 GGM-Construction

The GGM-construction[6] is a generic method to construct pseudorandom functions from pseudorandom generators. It can use any pseudorandom generator G that stretches a seed $x \in I_k$ into a $2k$-bit-long sequence $G(x) = b_1 b_2 \cdots b_k b_{k+1} \cdots b_{2k}$. By $G_0(x)$ we denote the first k bits in $G(x)$. By $G_1(x)$ we denote the second k bits in $G(x)$. Define the pseudorandom function $GGM_x : I_k \mapsto I_k$ as $GGM_x(\alpha) = G_{\alpha_k}(\cdots G_{\alpha_2}(G_{\alpha_1}(x)))$, where $x \in_R I_k$ is the key of the pseudorandom function, and $\alpha = \alpha_1 \alpha_2 ... \alpha_k \in I_k$ is an input (query) to the pseudorandom function. Its evaluation can be implemented as the following algorithm:

```
$v \leftarrow x$
For $i \leftarrow 1$ to k
    do $v \leftarrow G_{\alpha_i}(v)$
output $v$
```

We may imagine this construction as a full binary tree rooted at the key x. Each left (right) edge of this tree is labeled *0* (1). Each node at the *i*-th level stores $G_{\alpha_i}(v)$ if its parent stores v. To evaluate $GGM_x(\alpha)$, we walk down along one of the all 2^k possible paths, depended on the value of α, and finally reach the leaf, which stores $GGM_x(\alpha)$.

3 The Improvement and Efficiency Analysis

3.1 The Improvement

For the reason of efficiency, we modify the original binary-tree construction to a 4-ary-tree construction. Instead of using a length-doubling pseudorandom generator, we use a pseudorandom generator G that stretches its input seed x to a length-quadrupling sequence $G(x)$. We use $G_{(0,0)}(x)$ to denote the first quarter of $G(x)$. Similarly, $G_{(0,1)}(x)$ denotes the second quarter, $G_{(1,0)}$ denotes the third quarter, and $G_{(1,1)}$ denotes the last quarter. We define the function $GGM_x' : I_k \mapsto I_k$ as $GGM_x'(\alpha) = G_{(\alpha_k, \alpha_{k-1})}(G_{(\alpha_{k-2}, \alpha_{k-3})}(\cdots G_{(\alpha_4, \alpha_3)}(G_{(\alpha_2, \alpha_1)}(x))))$ if k is even and $GGM_x'(\alpha) = G_{(0, \alpha_k)}(G_{(\alpha_{k-1}, \alpha_{k-2})}(\cdots G_{(\alpha_4, \alpha_3)}(G_{(\alpha_2, \alpha_1)}(x))))$ if k is odd, where $x \in_R I_k$ is the key of the pseudorandom function, and $\alpha \in I_k$ is a query to the function. This can be implemented as the following algorithm:

```
$v \leftarrow x$
For $i \leftarrow 1$ to $\lfloor k/2 \rfloor$
    do $v \leftarrow G_{(\alpha_{2i}, \alpha_{2i-1})}(v)$
If k is odd
    then $v \leftarrow G_{(0, \alpha_k)}(v)$
output $v$
```

3.2 **Proof of Correctness**

In this section, we will prove that the functions constructed by the variant, GGM', possess the essential "pseudorandomness" property of pseudorandom functions. The proof is a careful modification of the original proof of the GGM-construction[6] . It uses a proof technique, called the hybrid technique[4] .

Theorem: The functions *constructed by GGM' are pseudorandom functions.*

Proof:

We prove by contradiction. Assume that the functions in F_k constructed by GGM', using pseudorandom generators G_k as build blocks, are not pseudorandom. Then there exists a PPT algorithm A_F on input 1^k and has access to an oracle can distinguish $f \in_R F_k$ from $h \in_R H_k$ with non-negligible advantage, that is, $| p_k^F - p_k^H | \geq \frac{1}{Q(k)}$ for some polynomial Q. We use A_F to construct a PPT algorithm A_G that, on input 1^k and U_k (a polynomially bounded but sufficient large set of $4k$-bit strings), can determine whether the strings in U_k are generated by $G \in G_k$ on random seeds or just randomly chosen $4k$-bit strings, the advantage i.e. $| p_k^G - p_k^R | \geq \frac{1}{Q_1(k)}$ for some polynomial Q_1.

We only prove the case when k is odd since the proof when k is even is a direct modification of the original proof. Consider when A_F*'s* queries are answered by one of the following probabilistic algorithms A_i, $i = 0,1,2,...,\lceil k/2 \rceil$. Let $y = y_1 y_2 \cdots y_k$ be a query of A_F. For $0 \leq i \leq \lfloor k/2 \rfloor$, A_i answers the oracle query as follows:

If y is the first query with prefix $y = y_1 y_2 \ldots y_{2i}$
Then A_i selects a string $r \in_R I_k$,
stores the pair ($y_1 \ldots y_{2i}$, r), and answers $GGM'_{y_{2i+1} \cdots y_k}(r)$.
Else A_i retrieves the pair ($y_1 \ldots y_{2i}$, v) and answers $GGM'_{y_{2i+1} \cdots y_k}(v)$.

For $i = \lceil k/2 \rceil$, A_i answers the oracle query as follows:

If y is the first query with prefix $y = y_1y_2\ldots y_k$

Then A_i selects a string $r \in_R I_k$,

stores the pair ($y_1\ldots y_k$, r), and answers r.

Else A_i retrieves the pair ($y_1\ldots y_k$, v) and answers v.

We may imagine that a hybrid tree associated to A_i is a tree starting at the i-th level. Each node in the i-th level stores a randomly chosen k-bit string. For the nodes of the j-th level, where $j > i$, it stores $G_{(y_{2j+2}, y_{2j+1})}(d)$ if its parent stores d. Define p_k^i to be the probability that A_F outputs 1 when its queries are answered by A_i . Note that $p_k^0 = p_k^F$ and $p_k^{\lceil k/2 \rceil} = p_k^H$. Consequently, $| p_k^0 - p_k^{\lceil k/2 \rceil} | \geq \frac{1}{Q(k)}$.

Let U_k be a set of $P_1(k)$ strings, each $4k$-bit long. We now describe how to construct A_G that, on input k and U_k, can determine whether the strings in U_k are generated by G or just randomly chosen $4k$-bit strings:

Step 1: A_G randomly picks i between 0 and $\lfloor k/2 \rfloor$.

Step 2: A_G gives 1^k as input to A_F and answers A_F's queries using the set U_k properly.

If $0 \leq i < \lfloor k/2 \rfloor$, A$_G$ answers as follows:

If y is the first query with prefix $y = y_1y_2\ldots y_{2i}$

Then A_G picks the next string $u = u_{00}u_{01}u_{10}u_{11}$ in U_k (where $|u_{00}| = |u_{01}| = |u_{10}| = |u_{11}| = k$), stores the pairs ($y_1\ldots y_{2i}$00, u_{00}), ($y_1\ldots y_{2i}$01, u_{01}),and ($y_1\ldots y_{2i}$10, u_{10}), ($y_1\ldots y_{2i}$11, u_{11}), and answers $GGM'_{y_{2i+3}\cdots y_k}(u_{y_{2k+2}y_{2k+1}})$

Else A_i retrieves the pair ($y_1\ldots y_{2i+2}$, v) and answers $GGM'_{y_{2i+3}\cdots y_k}(v)$.

If $i = \lfloor k/2 \rfloor$, A_G answers as follows:

If y is the first query with prefix y=$y_1y_2\ldots y_{k-1}$

Then A_G picks the next string $u = u_{00}u_{01}u_{10}u_{11}$ in U_k (where $|u_{00}| = |u_{01}| = |u_{10}| = |u_{11}| = k$), stores the pairs ($y_1\ldots y_{k-1}$0, u_{00}), ($y_1\ldots y_{k-1}$1, u_{01}), *discards* u_{10} and u_{11}, and answers u_{0y_k} **.**

Else A_i retrieves the pair ($y_1\ldots y_k$, v) and answers v.

Step 3: Finally, A_G outputs A_F's output.

If U_k consists of $4k$-bit strings output by G on randomly selected k-bit input seeds, then A_G acts for oracle A_i. If U_k consists of randomly selected $4k$-bit strings, then A_G acts for oracle A_{i+1}. So we have Pr[$A_G(1^k,U_k)$ =1| U_k consists of $4k$-bit strings output by G] = $\sum_{i=0}^{\lfloor k/2 \rfloor}(\frac{1}{\lceil k/2 \rceil}) \cdot p_k^i$ and

Pr[$A_G(1^k,U_k)$=1| U_k consists of randomly chosen $4k$-bit strings] = $\sum_{i=0}^{\lfloor k/2 \rfloor}(\frac{1}{\lceil k/2 \rceil}) \cdot p_k^{i+1}$. These probabilities differ by at least $\frac{1}{\lceil k/2 \rceil} \cdot | p_k^0 - p_k^{\lceil k/2 \rceil} | \geq \frac{1}{\lceil k/2 \rceil \cdot Q(k)}$ which contradicts to the fact that G is a pseudorandom generator.

3.3 Efficiency Analysis

By processing *2* bits at each level and 1 bit at the last level if necessary, we get a variant of the original binary-tree construction. We claim that this *4*-ary-tree construction can work faster than the binary one under the assumption that the cost of generating ℓ pseudorandom bits is $\theta(\ell)$. And counterintuitively, expanding to 2^c -ary-tree construction, for any c > 2, dose not gain better efficiency. That is, 4-ary-tree construction is optimal among all tree constructions. (Here we do not consider trees other than 2^c-ary-trees, because the input domain and output domain of our PRFs both are binary domains.)

At the (i-1)-th iteration of the evaluating process of the GGM-construction, we compute $G_0(x)$ if $\alpha_i = 0$ and compute $G_1(x)$ if $\alpha_i = 1$. We denote as T_0 and T_1 the cost of time for these two cases, respectively. Since G is a generic generator, we can reasonably assume that in order to generate the second half output $G_1(x)$, we need to first generate the first half output $G_0(x)$. So $T_1 = 2 \cdot T_0$. To estimate the expected cost of time per evaluation of the GGM pseudorandom function, denoted as $E[T_{GGM}]$, we observe that the binary-tree corresponding to GGM-construction has depth k and each call to the generator G takes T_0 or T_1 depending on the i-th bit of input α . So we have

$$\begin{aligned} E[T_{GGM}] &= E[\sum_{i=1}^{k} T_{\alpha_i}] = \sum_{i=1}^{k} E[T_{\alpha_i}] \\ &= \sum_{i=1}^{k}(\frac{1}{2} \cdot T_0 + \frac{1}{2} \cdot T_1) = \sum_{i=1}^{k} \frac{3}{2} \cdot T_0 = \frac{3}{2}k \end{aligned} \quad (1)$$

For our 4-ary-tree construction, the corresponding 4-ary-tree has depth $\lceil k/2 \rceil$ and each call to the generator G will take T_0, $2T_0$, $3T_0$, or $4T_0$ depending on its input accordingly. So its expected cost of time per evaluation, denoted as $E[T_{GGM'}]$, is

$$\begin{aligned}&E[T_{GGM'}] = E[\textstyle\sum_{i=1}^{k/2} T_{\alpha_{2i}\alpha_{2i-1}}]\\&= \textstyle\sum_{i=1}^{k/2}(\frac{1}{4}\cdot T_0 + \frac{1}{4}\cdot 2T_0 + \frac{1}{4}\cdot 3T_0 + \frac{1}{4}\cdot 4T_0)\\&= \frac{5}{4}kT_0 < E[T_{GGM}]\end{aligned} \quad (2)$$

for k even, where $T_{\alpha_{2i}\alpha_{2i-1}}$ denotes the cost of time computing $G_{(\alpha_{2i},\alpha_{2i-1})}(x)$. And if k is odd, we have

$$\begin{aligned}&E[T_{GGM'}] = E[(\textstyle\sum_{i=1}^{\lfloor k/2\rfloor} T_{\alpha_{2i}\alpha_{2i-1}}) + T_{\alpha_k}]\\&= \frac{5}{2}\cdot\left\lfloor\frac{k}{2}\right\rfloor\cdot T_0 + \frac{3}{2}k\\&< \frac{5}{4}kT_0 + \frac{3}{2}k < E[T_{GGM}] \text{ if } T_0 > 6\end{aligned} \quad (3)$$

This shows that GGM$'$-construction performs better than GGM-construction on average.

In general, we can consider the 2^c-ary-tree construction. Let the expected cost of time per evaluation of a pseudorandom function from GGM^c-construction for $c \in N$ be denoted as $E[T_{GGM^c}]$. To simplify the discussion, we assume k is a multiple of c,

$$\begin{aligned}&E[T_{GGM^c}] = E[\textstyle\sum_{i=1}^{k/c} T_{\alpha_{2^c\cdot i}\cdots\alpha_{2^c\cdot(i-1)+1}}]\\&= \textstyle\sum_{i=1}^{k/c}\frac{1}{2^c}(T_0 + 2T_0 + 3T_0 \cdots + 2^c T_0)\\&= \frac{k}{c}\cdot\frac{1}{2^c}\cdot\frac{(1+2^c)\cdot 2^c}{2}T_0 = \frac{1+2^c}{2c}kT_0\end{aligned} \quad (4)$$

It is easy to verify that $E[T_{GGM^c}] > E[T_{GGM'}]$ if $c > 2$. Therefore, the 4-ary-tree construction *GGM'* is optimal.

Acknowledgement

This research was partially supported by the National Science Council, Taiwan, R.O.C., under contract number: NSC 94-2213-E-005-028.

Reference

[1] L. Blum, M. Blum, and M. Shub, "A Simple Unpredictable Pseudo-Random Number Generator", SIAM Journal on Computing, Vol.15, No.2, pp.364-383, 1986.

[2] M. Blum, W. S. Evans, P. Gemmell, S. Kannan, and M. Naor, "Checking the Correctness of Memories", Algorithmica, Vol.12, No.2/3, pp.225-244, 1994.

[3] M. Blum and S. Micali, "How to Generate Cryptographically Strong Sequences of Pseudo-Random Bits", SIAM Journal on Computing, Vol.13, No.4, pp.850-864, 1984.

[4] O. Goldreich, "Foundations of Cryptography (Fragments of a Book)", electronic publication:http://theory.lcs.mit.edu/~oded/frag.html, 1995.

[5] O. Goldreich, S. Goldwasser,and S. Micali, "On the Cryptographic Applications of Random Functions", CRYPTO 1984, pp.276-288, 1984.

[6] O. Goldreich, S. Goldwasser,and S. Micali, "How to construct random functions", Journal of the ACM, Vol.33, No.4, pp.792-807, 1986.

[7] O. Goldreich and R. Ostrovsky, "Software Protection and Simulation on Oblivious RAMs", Journal of the ACM, Vol.43, No.3, pp.431-473, 1996.

[8] S. Goldwasser and M. Bellare, "Lecture Notes on Cryptography", http://www.cs.ucsd.edu/users/mihir/papers/gb.html, 2001.

[9] M. Naor and O. Reingold, "Synthesizers and Their Application to the Parallel Construction of Pseudo-Random Functions", Journal of Computer and System Sciences, Vol.58, No.2, pp.336-375, 1999.

[10] M. Naor and O. Reingold, "On the Construction of Pseudorandom Permutations: Luby-Rackoff Revisited", Journal of Cryptology, Vol.12, No.1, pp.29-66, 1999.

[11] M. Naor, O. Reingold, and A. Rosen, "Pseudorandom Functions and Factoring", SIAM Journal on Computing, Vol.31, No.5, pp.1383-1404, 2002.

[12] M. Naor and O. Reingold, "Number-theoretic constructions of efficient pseudo-random functions", Journal of the ACM, Vol.51, No.2, pp.231-262, 2004.

[13] C. H. Papadimitriou, "Computational Complexity", Addison Wesley, 1995.

[14] A. C. Yao, “Theory and applications of trapdoor functions”, Proceedings of the 23rd IEEE Symposium on Foundations of Computer Science, pp.80-91, 1982.