情報保護學會論文誌
第18卷 第4號, 2008. 8

서버에서 효율적인 메모리 사용량을 제공하는 공개키 기반 검색 암호 시스템*

권 은 정[1†], 서 재 우[1], 이 필 중[1‡], 박 영 만[2], 이 해 규[2], 김 영 헌[2], 정 학 진[2]
[1]포항공과대학교, [2](주)KT 미래기술연구소

Memory-efficient Public Key Encryption with Keyword Search in Server*

Eun Jeong Kwon[1†], Jae Woo Seo[1], Pil Joong Lee[1‡], Young Man Park[2], Hae Gyu Rhy[2], Yeong Heon Kim[2], Hak Jin Chong[2]
[1]POSTECH, [2]KT Co. Ltd., Future Technology Lab.

요 약

검색 가능 암호 시스템 (Searchable Encryption)은 암호화된 데이터에서 키워드를 검색하는 문제를 해결하기 위하여 2000년에 Song 등에 의해서 처음으로 제안되었다. 지금까지 대칭키 암호화 방식과 공개키 암호화 방식에 기반하는 여러 가지 검색 가능 암호 시스템이 제안되었으나, 공개키 암호화 방식에 기반한 기존의 기법들은 암호화 된 문서의 검색을 위해 서버에서 저장해야 하는 인덱스들의 크기가 키워드 개수에 비례하여 선형적으로 증가하는 단점을 가진다. 본 논문에서는 공개키 기반 검색 가능 암호 시스템에서 서버에서 저장하는 인덱스의 길이를 줄이기 위해 Bloom Filter를 적용하는 방법을 제안하고, Boneh 등이 제안한 PEKS(Public key Encryption with Keyword Search)에 제안한 방법을 적용하여, 메모리 측면에서 그 효율성을 분석한다.

ABSTRACT

In 2000, Song. et. al. firstly proposed the Searchable Keyword Encryption System that treated a problem to search keywords on encrypted data. Since then, various Searchable Keyword Encryption Systems based on symmetric and asymmetric methods have been proposed. However, the Searchable Keyword Encryption Systems based on public key system has a problem that the index size for searching keywords on encrypted data increases linearly according to the number of keyword. In this paper, we propose the method that reduces the index size of Searchable Keyword Encryption based on public key system using Bloom Filter, apply the proposed method to PEKS(Public key Encryption with Keyword Search) that was proposed by Boneh. et. al., and analyze efficiency for the aspect of storage.

Keywords : Searchable Keyword Encryption, Bloom Filter, Public Key Encryption

접수일 : 2007년 11월 26일; 수정일 : 2008년 2월 29일; 채택일 : 2008년 5월 7일
* 본 연구는 KT 미래기술연구소, Brain Korea 21, 지식경제부 및 정보통신연구진흥원의 대학 IT 연구센터 지원사업의 연구 결과로 수행되었음 (IITA-2008-C1090-0801- 0026).
† 주저자, keun0913@postech.ac.kr
‡ 교신저자, pjl@postech.ac.kr

I. 서 론

컴퓨터 기술의 발전으로 인해 많은 정보가 디지털 정보로 저장되거나 처리되고 있으며, 인터넷의 보급은 이러한 디지털 데이터들을 점점 더 외부의 서버를 이용해

서 처리하게 하고 있다. 많은 사람들이 웹을 이용한 e-mail 시스템을 사용하고 있으며, 인터넷 뱅킹은 이미 은행 창구 역할을 상당부분 대신하고 있다. 또한 기업들은 관리를 손쉽게 하기 위해 data warehouse를 이용하고 있다. 우리는 이러한 외부 서버들을 통해 민감한 데이터를 전송하는 경우에 기밀성 유지를 위하여 데이터를 암호화해야 하며, 이는 내부의 서버를 사용하는 경우에도 마찬가지이다. CSI/FBI의 2004년도 자료에 따르면 전체 공격 중 내부자에 의한 공격이 59%를 차지하는 것으로 알려졌다 [13]. 정보를 자신이 직접 관리하는 경우에도 중요한 정보는 도난을 대비해서 암호화를 하게 된다. 즉 데이터의 기밀성을 보장하기 위해서는 우리는 데이터를 암호화해서 저장해야한다. 이렇게 데이터를 암호화해서 저장할 경우, 암호화된 데이터에 대한 검색 문제가 발생한다.

이러한 암호화된 데이터에 대한 검색 문제는 Song 등에 의해 처음 제기되었다[5]. 이 기법은 대칭키 암호화 방식에 기반하는 것으로 암호문의 검색을 위한 인덱스들을 생성할 때와 trapdoor를 생성할 때 동일한 키를 사용한다. 이어서 2004년에 Boneh 등이 공개키 암호화 방식에 기반하는 검색 가능 암호 시스템(Searchable Keyword Encryption)을 처음으로 제안하였으며[1], 이 기법은 인덱스를 생성할 때 수신자의 공개키를 사용하고 trapdoor를 생성할 때는 수신자의 비공개키를 사용한다. 하지만 위의 두 가지 기법들은 모두 단일 키워드(single keyword)에 대한 검색만 가능한 방법들이다. 이 문제를 해결하기 위하여 2004년에 Golle 등이 대칭키 암호화 방식에 기반하면서 다중 키워드 검색(conjunctive keyword)을 지원하는 검색 가능 암호 시스템을 제안하였으며[6], 이어서 Park 등이 다중 키워드 검색을 지원하는 공개키 기반 검색 가능 암호 시스템을 제안하였다[7]. 그 이후로도 암호화된 데이터를 효율적으로 검색하는 방법에 대한 많은 연구가 진행되어 왔는데[8-10, 14], 제안된 검색 가능 암호 시스템들은 데이터의 키워드 개수에 비례하여 암호문의 검색을 위한 인덱스의 길이가 증가한다는 단점이 있었다. 하지만 검색 가능 암호 시스템을 실생활에 적용하고자 할 때 서버에서 저장하는 검색 가능 암호문의 길이는 중요한 요소이므로, 메모리 사용량 측면에서 효율적인 검색 가능 암호 시스템에 대한 연구가 필요하다.

처음으로 이 문제를 다룬 것은 2004년에 Goh가 제안한 Z-IDX에서 였다. Z-IDX는 Bloom Filter를 사용하여 대칭키 암호화 방식에 기반하는 검색 가능 암호 시스템으로[2], 이는 암호화된 키워드를 Bloom Filter로 변환하여 저장함으로써, 서버에서 저장해야 하는 인덱스의 크기를 줄이고, 문서가 가지는 인덱스의 크기를 Bloom Filter의 크기로 고정하였다. 하지만 Goh가 제안한 Z-IDX는 대칭키 암호화 방식에 기반하고 있기 때문에, 이메일 시스템, secure audit log와 같은 공개키 기반 환경에서는 사용이 불가능하며, 사용 범위가 매우 제한적이라는 단점이 있다. 따라서 본 논문에서는 기존의 공개키 기반 검색 암호 시스템에 Bloom Filter를 적용하여 인덱스의 크기를 줄이는 방법을 제안하고, 그 적용과정에서 발생하는 문제점들을 기술한다. 그리고 Boneh 등이 제안한 PEKS(Public key Encryption with Keywrod Search)에 Bloom Filter를 적용하여 메모리 측면에서 그 효율성을 분석한다.

Ⅱ. 배경지식

2.1 Bilinear Maps

G_1과 G_2가 어떤 큰 소수인 p를 위수로 갖는 군이라고 하면, 이 두 군 사이에 성립하는 겹선형 함수(bilinear map) $e: G_1 \times G_1 \rightarrow G_2$는 다음의 세 가지 성질을 만족한다.

- Bilinear : 모든 $P, Q \in G_1$와 $a, b \in Z$에 대해서 $e(aP, bP) = e(P, Q)^{ab}$을 만족하면, bilinear하다고 말한다.
- Non-degenerate : $e(P, Q) \neq I$, I는 군 G_2의 항등원(identity)이다.
- Computable : $e(P, Q)$를 계산할 수 있는 효율적인 알고리즘이 존재한다.

2.2 Bilinear Diffie-Hellman Problem

군 G_1에서 정의되는 BDH 문제는 입력으로 G_1의 생성자 P와 $\alpha P, \beta P, \gamma P$가 주어졌을 때, $e(P, P)^{\alpha\beta\gamma} \in G_2$를 계산하는 문제이다. 다음과 같이 부등식이 만족할 경우 알고리즘 A는 이득 ϵ을 가지고 BDH 문제를 풀 수 있다고 한다.

$$\Pr[A(P, \alpha P, \beta P, \gamma P) = e(P, P)^{\alpha\beta\gamma}] \geq \epsilon$$

2.3 Bloom Filter

Bloom Filter는 특정 집합의 원소들을 고정된 크기의 배열에 표현할 수 있는 자료구조이다[3]. n개의 원소로 이루어진 집합 $S=\{s_1,\cdots,s_n\}$가 존재할 때, S의 원소들은 m-bit 크기의 배열에 나타내어질 수 있다. 이를 위해 Bloom Filter는 ℓ개의 독립적인 해쉬 함수 $h_1,\cdots,h_\ell$을 사용하는데, 여기에서 각 해쉬 함수는 다음과 같이 표현할 수 있다.

$$h_i : \{0,1\}^* \rightarrow [1,m], (1 \le i \le \ell)$$

Bloom Filter의 각 비트들은 초기에 0으로 설정되고, S의 각 원소 $s_i(1 \le i \le n)$에 대해서 $h_1(s_i),\cdots,$ $h_l(s_i)$ 위치의 비트들이 1로 설정된다. 이후에 어떤 데이터 a가 집합 S의 원소인지 판단하려면 위치의 비트들을 확인하고, $h_1(a),\cdots,h_\ell(a)$이 값들이 모두 1이면 a를 S의 원소라고 판단한다. 하지만 Bloom Filter에는 긍정오류율(False Positive Rate)이 존재하는데, 이는 a'이 S의 원소가 아닌데도 불구하고 $h_1(a'),\cdots,h_\ell(a')$ 위치의 비트들이 S의 다른 원소들에 의해서 1로 설정되기 때문이다. 긍정오류율을 f_p라고 할 때, Bloom Filter의 파라미터들과 긍정오류율 사이의 관계는 다음과 같다 [4].

$$f_p = \left(1-(1-1/m)^{\ell n}\right)^{\ell}$$

2.4 의사난수함수 (Pseudo Random Function)

본 논문에서는 Bloom Filter를 생성하기 위해 의사난수함수를 사용하는데, 아래의 2가지 조건을 만족하는 함수 $f:\{0,1\}^n \times \{0,1\}^s \rightarrow \{0,1\}^m$를 (t,ϵ,q)-의사난수함수라고 한다[2].

1. 입력 $x \in \{0,1\}^n$와 키 $k \in \{0,1\}^s$로부터 $f(x,k)=f_k(x)$ 값을 효율적으로 계산할 수 있다.

2. t 시간동안, 최대 q개의 query를 할 수 있는 알고리즘 A에 대해서 아래의 부등식을 만족한 다.

$$|\Pr\left[A^{f(\cdot,k)}=0 \mid k \rightarrow_R \{0,1\}^s\right] - \Pr\left[A^g=0 \mid g \leftarrow_R \{F:\{0,1\}^n \rightarrow \{0,1\}^m\}\right]| < \epsilon$$

Ⅲ. 메모리 측면에서 효율적인 공개키 기반 키워드 검색 암호 기법

3.1 공개키 기반 단일 키워드 검색 암호 기법

본 장에서는 공개키 기반 단일 키워드 검색 암호 기법(Public key Encryption with Single Keyword Search, PESKS)에서 서버의 메모리를 효율적으로 활용하기 위해, 각 문서마다 부가적으로 저장되는 인덱스(Index)의 크기를 줄이는 방법을 제안한다. 제안하는 방법은 사용자에 의해 만들어진 암호화된 인덱스들을 Bloom Filter로 변환하여 서버에 저장함으로써, 서버의 메모리를 효율적으로 활용한다. Bloom Filter는 메모리를 효율적으로 활용하기 위해 고안된 확률적 데이터 구조(probabilistic data structure)로써, 멤버 테스트가 요구되는 분야에서 메모리를 효율적으로 사용하기 위해 적용되었다[4]. 하지만 Bloom Filter를 PESKS에 그대로 적용하는 것은 무리가 있다. 일반적으로 PESKS에서는 각 키워드에 해당하는 인덱스를 생성할 때, 서로 다른 랜덤 값을 사용하기 때문이다. 만약 Bloom Filter를 기존의 PESKS에 그대로 적용할 경우, 서버는 문서에 해당하는 각 키워드들에 대한 인덱스들을 Bloom Filter로 변환한 값을 서버에 저장한다. 이 때, 서버는 각 랜덤 값들이 어떤 인덱스의 생성에 사용되었는지를 알 수 없다. 그리고 서버는 사용자로부터 키워드 검색을 위한 trapdoor가 주어지면, 그 키워드를 포함하는 문서를 검색하기 위해 각 문서에 해당하는 인덱스들의 Bloom Filter와 trapdoor를 비교할 때마다 인덱스 생성에 사용된 모든 랜덤 값들에 대한 Bloom Filter를 구해야 한다. 이와 같은 방법으로는 서버에 저장되는 인덱스의 크기는 줄일 수 있지만 모든 랜덤 값들에 대한 Bloom Filter를 구해야 하기 때문에, 키워드 검색 측면에서 이전 방법들보다 효율성이 떨어진다. 또한 Bloom Filter가 가지는 긍정오류(False Positive) 특성은 요청된 키워드를 포함하지 않은 문서들을 반환하고, 불필요한 통신 대역폭을 차지하여 통신의 효율성을 떨어트린다. 본 절에서 제안하는 방법은 PESKS에 Bloom Filter를 효율적으로 적용하기 위해서,[11,12]에서 제안된 랜덤성 재사용(randomness re-use) 테크닉을 사용한다. 본 논문에서는 키워드를 w, 키워드 집합을 W로 표현한다.

다음은 PESKS의 일반적인 모델을 나타낸다. 일반적

으로 PESKS는 다음의 4가지 알고리즘으로 구성된다.

KeyGen(k) : 안전성 파라미터 k를 입력으로 받아 공개키/비공개키 쌍 A_{pub}, A_{priv}을 생성한다.

Build Index(A_{pub}, w) : 공개키 A_{pub}와 키워드 w가 주어지면, 키워드 w에 대한 인덱스 $S=\{R, I\}$를 생성한다. (여기서 R는 I를 생성 하는 데 사용된 랜덤 값이다.)

Trapdoor(A_{priv}, w) : 사용자의 비공개키 A_{priv} 와 검색하려는 키워드 w가 주어지면, w에 대한 trapdoor T_w를 생성한다.

Test(A_{pub}, S, T_w) : 사용자의 공개키 A_{pub}, 문서의 인덱스 집합 S=Build Index(A_{pub}, w'), trapdoor T_w=Trapdoor(A_{priv}, w)를 이용하여, $w=w'$이면 'Yes'를 출력하고, 그렇지 않으면 'No'를 출력한다.

3.2 Bloom Filter를 적용한 PESKS 모델

기존의 PESKS에서는 각 문서에서 유일 키워드(unique keyword) w_i의 개수 n이 증가할수록 서버에 저장해야 하는 인덱스들의 개수도 비례적으로 증가하였다. 우리는 동일 문서 안에서 Build Index 알고리즘의 출력 값인 인덱스들 중 $I_i(1 \le i \le n)$를 Bloom Filter로 변환해서 저장함으로써, 키워드 검색을 위해 서버에 저장되는 인덱스들의 크기를 줄인다. 다음은 Bloom Filter를 적용한 일반적인 PESKS 모델이다.

KeyGen(k) : 안전성 파라미터 k를 입력으로 받아 공개키/비공개키 쌍 A_{pub}, A_{priv}을 생성한다.

Build Index(A_{pub}, W) : 공개키 A_{pub}와 키워드 들의 집합 $W=\{w_1, w_2, \dots, w_n\}$가 주어지면, 각 키워드 $w_i(1 \le i \le n)$에 대한 인덱스 I_i를 생성한다. (여기에서 각 인덱스 I_i를 생성하기 위해 사용되는 랜덤 값은 R_I로 동일하다.) 그러면 키워드 집합 W에 대한 인덱스 집합 $S=\{R_I, I_1, I_2, \dots, I_n\}$가 된다. 그리고 S에서 모든 I_i를 해당 문서의 Bloom Filter(BF)에 저장 하면 Bloom Filter를 적용한 Build Index 알고리즘의 최종 출력은 $\{R_I, BF\}$이고, 우리는 이를 S_{BF}로 표기한다.

$$S_{BF}=\{R_I, BF\}$$

Trapdoor(A_{priv}, w): 사용자의 비공개키 A_{priv} 와 검색하려는 키워드 w가 주어지면, w에 대한 trapdoor T_w 을 생성한다.

Test(A_{pub}, S_{BF}, T_w): 사용자의 공개키 A_{pub}, 문서의 인덱스 S_{BF}=Build Index(A_{pub}, $W'=\{w_1', w_2', \dots, w_n'\}$)$=\{R_I', BF'\}$, trapdoor T_w = Trapdoor(A_{priv}, w)이 주어지면, T_w 와 R_I' 을 이용하여 키워드 w에 대한 단일 Bloom Filter(BF_w)을 생성하고, BF'에 대한 멤버 테스트를 수행한다. 만약 BF_w가 멤버 테스트를 통과하면 'Yes'를 출력하고, 그렇지 않으면 'No'를 출력한다.

위의 모델에서 해당 문서의 키워드 집합 W에 대한 n개 인덱스 $I_i(1 \le i \le n)$은 모두 Bloom Filter(BF)로 저장되기 때문에, 기존 PESKS에서처럼 유일 키워드 개수에 따라 선형적으로 증가하지 않는다. 또한 각 문서에서 동일한 랜덤 값을 사용하였기 때문에, 기존 PESKS보다 서버에서 저장해야 하는 랜덤 값들이 $1/n$로 줄었으며, 서버는 동일 문서 안에서 여러 번의 Test 알고리즘을 수행할 필요가 없다. 서버는 테스트하고자 하는 문서에 따라 랜덤 값 R_I와 요청된 키워드 w의 Bloom Filter BF_w을 생성하면 된다. 즉 위에서 제안된 모델은 기존 PESKS 모델과 비교하여, 서버에서 저장해야 하는 인덱스들의 크기가 감소하여 서버의 메모리 활용도가 높아졌으며, Bloom Filter의 멤버 테스트 기능을 이용하여 문서 단위로 Test 알고리즘을 수행하기 때문에 요청된 키워드를 포함하는 문서에 대한 검색 시간도 줄어들었다. 하지만 동일한 랜덤 값을 사용했기 때문에 발생할 수 있는 안전성 측면의 문제도 고려해 보아야 한다. 기존 PESKS와 비교하여 제안하는 모델은 안정성이 다소 떨어질 것이라는 것을 예상할 수 있다. 그렇다면 동일한 랜덤 값을 사용함으로써, 안전성이 얼마나 낮아지는지, 그리고 받아들일 수 있는 정도인지를 분석해볼 필요가 있다. 그래서 우리는 Boneh 등이 제안한 키워드 검색이 가능한 공개키 암호화 방식(Public key Encryption with Keyword Search, PEKS)에 위의 모델을 적용하고, 안전성을 증명할 것이다. 이 논문에서 우리는 Bloom Filter를 적용한 PEKS를 BF-PEKS라고 부른다.

3.3 BF-PEKS를 위한 안전성 모델

본 절에서 BF-PEKS를 위한 안전성 모델을 정의한다. 이 안전성 모델은 다음의 게임을 따른다.

1. challenger는 KeyGen() 알고리즘을 수행하여 A_{pub}, A_{priv}을 생성하고, A_{pub}을 공격자에게 준다.

2. 공격자 A는 계속해서 자신이 선택한 임의의 키워드 $w \in \{0,1\}^*$에 대한 trapdoor T_w를 요청할 수 있다.

3. 공격자 A는 challenge하려는 키워드들의 집합 W_0, W_1을 다음과 같이 생성한다.

   $$W_0 = \{w_{0,1}, \cdots, w_{0,n}\},\ \ W_1 = \{w_{1,1}, \cdots, w_{1,n}\}$$

   그리고 W_0, W_1를 challenger에게 보낸다. 이 때, W_0, W_1은 앞에서 공격자가 trapdoor를 요청한 키워드 w를 포함하지 않아야 한다. 그러면 challenger는 랜덤한 값 $b \in \{0,1\}$를 선택하고, 공격자에게 BF-PEKS(A_{pub}, W_b)를 주고, 이를 challenge C라고 한다.

4. 공격자 A는 계속해서 원하는 키워드 w_i에 대한 trapdoor를 요청할 수 있다. 단, w_i는 W_0, W_1의 원소가 아니어야 한다.

5. 최종적으로 공격자 A는 challenger로부터 받은 challenge C가 W_0, W_1 중에서 어떤 키워드 집합의 인덱스인지를 나타내는 $b' \in \{0,1\}$ 을 출력하고, $b' = b$이면 공격자가 게임에서 이기게 된다.

위의 안전성 모델은 Boneh 등이 [1]에서 정의한 안전성 모델과 거의 유사하지만, BF-PEKS는 Boneh 등의 방법(PEKS)과 달리 BF-PEKS 알고리즘의 입력이 단일 키워드가 아닌 키워드들의 집합이라는 점에서 차이가 있다. 또한 PEKS에서는 하나의 문서에 해당하는 인덱스를 생성할 때 각 키워드마다 서로 다른 랜덤한 값을 이용하지만, 제안한 BF-PEKS에서는 하나의 문서에 해당하는 인덱스를 생성하는데 한 개의 랜덤한 값을 사용한다. 하지만 PEKS과 비교하였을 때, BF-PEKS는 안전성 측면에서 여전히 안전하다. 우리는 5절에서 Boneh 등의 스킴과 동일한 가정을 하였을 때, BF-PEKS 또한 선택된 키워드 공격 (adaptive chosen keyword attack)으로부터 semantically secure함을 보일 것이다.

3.4 BF-PEKS

Boneh 등은 2004년에 키워드 검색이 가능한 공개키 암호화 방식(Public key Encryption with Keyword Search, PEKS)을 제안하고, 제안한 기법이 랜덤 오라클 모델에서 선택된 키워드 공격(Chosen Keyword Attack)으로부터 안전하다는 것을 증명하였다 [1]. 하지만 PEKS는 각 문서에 해당하는 유일 키워드의 개수가 많아질수록 인덱스가 길어진다는 단점이 있다. 그래서 우리는 위에서 제시된 모델을 기반으로 BF-PEKS를 설계하고, BF-PEKS가 PEKS보다 메모리 측면과 검색 효율성 측면에서 효율적임을 보이고, BF-PEKS의 안전성을 증명한다. 설명의 편의를 위하여 [1]에서와 마찬가지로 Bob이 Alice에게 암호화된 e-mail을 전송하는 경우를 예로 들어 설명한다. Bob은 전송하려는 e-mail 자체를 암호화한 문서에, 그 문서에 해당하는 키워드들의 암호화된 인덱스들을 덧붙여 Alice에게 보낸다. Bob이 전송하려는 e-mail을 D라 하고, D와 관련된 키워드들의 집합을 $W = \{w_1, \cdots, w_n\}$라고 하면 Bob이 Alice에게 보내는 전체 데이터는 다음과 같다.

$$[E(D), BF-PEKS(A_{pub}, W)]$$

위에서 e-mail D 자체를 암호화 하는 알고리즘 $E(\cdot)$와 D의 키워드들의 집합 $W = \{w_1, \cdots, w_n\}$에 대한 검색 가능 암호문을 생성하는 $BF-PEKS$ 알고리즘은 서로 독립적이다.

BF-PEKS는 Bloom Filter를 이용하기 때문에, 이전 PEKS와 비교하여 서로 다른 ℓ개의 해쉬 함수 $\{h_1, h_2, ..., h_\ell\}$를 더 필요로 한다. 여기에서 사용되는 ℓ개의 해쉬 함수는 Alice의 공개키 A_{pub}를 공개된 의사난수함수(Pseudo Random Function, PSF)에 ℓ번 반복적으로 적용했을 때, 출력되는 ℓ개의 값을 각각의 키로 사용한다.

$$\{h_1 = h_{PSF^1(A_{pub})}, h_2 = h_{PSF^2(A_{pub})}, \dots, h_\ell = h_{PSF^\ell(A_{pub})}\}$$

그리고 BF-PEKS는 [1]에서와 동일하게 2개의 해쉬 함수 $H_1 : \{0,1\}^* \to G_1$과 $H_2 : G_2 \to \{0,1\}^{\log p}$를 사용한다.

BF-PEKS는 다음의 4가지 알고리즘으로 구성된다.

이 중에서 KeyGen과 Trapdoor 알고리즘은 PEKS에서 제안된 방법과 동일하다.

KeyGen(k) : 입력으로 받은 안전성 파라미터 k 를 이용하여, 위수가 p인 군 G_1, G_2을 결정하고, 이를 바탕으로 Z_p^*에서 α를 랜덤하게 선택한다. G_1의 생성자를 g라고 할 때, 이 알고리즘으로부터 생성되는 공개키 A_{pub}와 비공개키 A_{priv}는 다음과 같다.

$$A_{pub} = [g, y = g^{\alpha}], \quad A_{priv} = \alpha$$

BF-PEKS(A_{pub}, W) : Z_p에서 랜덤하게 r을 선택하고, 문서에 해당하는 유일 키워드들의 집합 $W = \{w_1, \cdots, w_n\}$와 공개키 A_{pub}를 이용하여 아래의 값들을 계산한다.

$$s_1 = e(H_1(w_1), y^r), \ldots, s_n = e(H_1(w_n), y^r)$$

그리고 $s_1' = H_2(s_1), \cdots, s_n' = H_2(s_n)$을 계산한 후에, 모든 $s_i'(1 \le i \le n)$을 Bloom Filter (BF)에 넣는다. 즉, 각 s_i'을 ℓ개의 해쉬 함수에 통과시킨 값 $h_1(s_i'), \cdots, h_\ell(s_i')$이 나타내는 위치의 Bloom Filter($BF$) 비트 값들을 1로 설정한다. ($BF$의 초기 값은 모두 0으로 설정된 다.)

$$BF = BF | 2^{(h_1(s_i')-1)} | 2^{(h_2(s_i')-1)} | \cdots | 2^{(h_\ell(s_i')-1)},$$
$$\left(2^{(h_j(s_i')-1)} = 1 \ll (h_j(s_i')-1)\right)$$

여기서 '|'은 bitwise OR 연산을 의미하며, 2의 지수승 연산은 shift 연산과 동일합니다. 그러면 최종적으로 이 T_w 알고리즘으로부터 출력되는 인덱스는 $S_{BF} = \{g^r, BF\}$이다.

Trapdoor(A_{priv}, w) : 검색하려는 키워드 w에 대한 trapdoor 는 $H_1(w)^{\alpha} \in G_1$이다.

Test(A_{pub}, S_{BF}, T_w) : 서버에 저장된 인덱스 $S_{BF} = [g^r, BF]$와 trapdoor T_w를 이용하여 $s' = H_2(e(T_w, g^r))$을 계산하고, 키워드 w에 대한 단일 Bloom Filter BF_w를 생성한다. 즉, BF_w에서 $h_1(s'), \cdots, h_\ell(s')$이 나타내는 위치의 비트 값들을 1로 설정한다. (위에서와 마찬 가지로, BF_w의 초기 값 역시 모두 0으로 설정 된다.)

$$BF_w = 2^{(h_1(s')-1)} | 2^{(h_2(s')-1)} | \cdots | 2^{(h_\ell(s')-1)},$$
$$\left(2^{(h_j(s')-1)} = 1 \ll (h_j(s')-1)\right)$$

그 후에, BF_w의 비트 값이 1인 위치에서 BF 의 비트 값도 1인지 검사한다. 만약 검사한 모든 위치의 비트값이 1이라면, 해당 문서 D가 키워드 w을 포함하고 있는 것으로 판단하여 'Yes'를 출력하고, 그렇지 않으면 'No'를 출력 한다.

BF-PEKS에서 Bob은 전송하고자 하는 문서가 포함하는 모든 유일 키워드들의 $I_i(1 \le i \le n)$을 BF로 변환하여 서버에 전달한다. 그 이후에 서버는 Alice로부터 키워드 검색 요청이 들어오면 그 키워드와 각 문서가 포함하고 있는 랜덤 값 g^r을 이용하여 검색하려는 키워드 w에 대한 단일 Bloom Filter(BF_w)을 생성하고, 서버에서 각 해당 문서마다 저장하고 있는 BF와 멤버 테스트를 수행한다. 따라서 BF-PEKS는 PEKS에 비해 송신자(Bob)가 서버에 전송해야 하는 데이터의 크기를 줄일 수 있으며, 서버에서 저장해야 하는 부가적인 인덱스의 크기도 줄일 수 있다. 그리고 BF-PEKS는 각 문서마다 동일한 랜덤 값을 재사용하기 때문에, 서버는 동일 문서 안에서 여러 번의 Test 알고리즘(최대 n번의 pairing 연산)을 수행할 필요가 없으며, 단 한 번의 Test 알고리즘(1번의 pairing 연산)을 수행해서 키워드 검색이 가능하다. 즉, BF-PEKS는 서버가 trapdoor에서 요청하는 키워드가 포함된 문서를 검색하는데 걸리는 시간을 크게 줄일 수 있다. 만약 특정 문서가 포함하는 유일 키워드의 개수가 n개라고 하면 그 문서가 trapdoor에서 요청한 키워드를 포함하는지 판단할 때, PEKS는 최대 n번의 페어링(pairing) 연산을 요구하는데 비해 BF-PEKS는 단지 1번의 페어링(pairing) 연산만 요구한다. 매 키

〔표 1〕 문서(D)를 기본 단위로 하였을 때, PEKS와 BF-PEKS의 효율성 비교 (P: pairing, H: 해쉬 연산 (H_1, H_2, $h_1, \ldots h_\ell$), E: 모듈러 멱승 연산)

구분	PEKS	BF-PEKS
인덱스 생성	$nP + 2nH$	$nP + (2+\ell)nH$
인덱스 길이	$n\log p - bit$	$(\log p + m) - bit$
Trapdoor 생성	$H + E$	$H + E$
검색(Test)	$nP + nH$	$P + (\ell+1)H$

워드의 인덱스 $I_i(=s_i')$를 Bloom Filter(BF)로 저장하기 위해 ℓ번의 해쉬 연산을 추가적으로 사용해야 하지만 해쉬 연산은 페어링이나 멱승 연산보다 훨씬 가벼운 연산이기 때문에, 서버에서의 검색 측면에서 추가적인 연산량(해쉬 연산)에 비해 줄어든 연산량이 (pairing)이 크다.

3.5 BF-PEKS의 안전성 증명

BF-PEKS의 안전성은 PEKS와 동일하게 BDH 문제가 풀기 어렵다는 가정에 기반한다. 아래의 증명은 [1]에서의 증명 방식과 유사하다. 그리고 본 논문의 증명에서는 설명의 편의와 이해를 돕기 위해 C=BF-PEKS $(A_{pub}, W_b)=\{g^r, BF_b\}$의 출력을 Bloom Filter의 형태가 아닌 $C=\{g^r, s_1', s_2', \ldots, s_n'\}$의 형태로 제시한다. 비록 Bloom Filter가 긍정오류율을 가지지만, Bloom Filter의 긍정오류율은 일반적으로 negligible 하기 때문에 ϵ이 negligible하다는 것을 증명하는 것에 큰 영향을 미치지 않는다.

Theorem 1. BDH 문제가 어렵다고 가정하면 BF-PEKS는 랜덤 오라클 모델에서 선택된 키워드 공격으로부터 안전(semantically secure)하다.

Proof. ϵ보다 높은 확률로 BF-PEKS를 공격하는데 성공하는 알고리즘 A가 존재하고, A가 해쉬 함수 H_2에 대한 요청을 최대 q_{H_2}번, trapdoor 요청을 최대 q_T번 한다고 가정한다. 그러면 우리는 알고리즘 A를 이용하여 적어도 ϵ'의 확률로 BDH 문제를 풀 수 있는 알고리즘 B를 만들 수 있으며, 여기서 ϵ'의 값은 다음과 같다.

$$\epsilon' = \epsilon(1-\delta)^{q_T}\{1-(1-\delta)^{2n}\}/(n q_{H_2})$$

BDH 문제를 풀기 어렵다는 가정이 G_1에서 성립한다면, ϵ'은 negligible function이고 ϵ 또한 negligible function이다. g를 G_1의 생성자라고 하면, B는 $g, u_1 = g^{\alpha}, u_2 = g^{\beta}, u_3 = g^{\gamma} \in G_1$을 입력으로 받아 다음의 값을 출력하고자 한다.

$$v = e(g,g)^{\alpha\beta\gamma} \in G_2$$

우리는 BF-PEKS를 다항시간 안에 높은 확률로 공격에 성공하는 알고리즘 A를 이용하여 다음과 같이 BDH 문제를 푸는 알고리즘 B를 만들 수 있다.

- **KeyGen.** B는 A에게 공개키 $A_{pub}=[g, u_1]$을 준다.

- **H_1, H_2-queries.** A는 언제든지 랜덤 오라클 H_1, H_2에 요청을 할 수 있다. B는 H_1에 대한 A의 요청에 응답하기 위해서 H_1-list를 생성하는데, H_1-list는 $\langle w_j, h_j, a_j, c_j \rangle$와 같이 4개의 원소로 구성된 튜플이며 초기에는 비어 있다. B는 이 list를 저장하고 있다가 A가 $w_i = \{0,1\}^*$에 대한 H_1 요청에 다음과 같이 응답한다.

  1. 요청된 w_i가 H_1-list에 존재하면 B는 $H_1(w_i) = h_i \in G_1$로 응답한다.

  2. 그렇지 않으면 B는 δ의 확률로 c_i값이 0이 되도록 랜덤하게 $c_i \in \{0,1\}$을 선택하고, Z_p에서 랜덤하게 a_i를 선택한다. 여기서 δ의 값이 다음과 같을 때, 가장 최적화된 안전성을 구할 수 있다.

  $$\delta = \sqrt[2n]{q_T/(q_T+2n)}$$

  $c_i = 0$이면 B는 h_i에 $u_2 g^{a_i} \in G_1$을 할당하고, $c_i = 1$이면 $g^{a_i} \in G_1$을 할당한다.

  3. B는 계산한 튜플 $\langle w_i, h_i, a_i, c_i \rangle$을 H_1-list에 추가하고, A의 요청에 대한 응답으로 $H_1(w_i) = h_i$를 A에게 보낸다.

  마찬가지로 A가 새로운 값 s에 대해 H_2 요청을 하면 B는 랜덤하게 $V \in \{0,1\}^{\log p}$를 선택하여 $H_2(s) = V$로 설정하고, (s, V)를 H_2-list에 추가한다. H_2-list 역시 초기에는 비어 있다.

- **Trapdoor queries.** A가 키워드 w_i에 해당하는 trapdoor를 요청하면 B는 다음과 같이 응답한다.

  1. B는 위의 H_1, H_2-queries 단계에서 언급한 알고리즘을 수행하여 $H_1(w_i) = h_i$를 얻는다. 이 값과 관련된 H_1-list의 튜플을 $\langle w_i, h_i, a_i, c_i \rangle$라고 할 때, $c_i = 0$이면 B는 'failure'를 알리고 시뮬레이션을 종료한다.

2. $c_i = 1$이면 $h_i = g^{a_i} \in G_1$이 되고, $u_1^{a_i}$를 T_i라고 정의하면 $T_i = H_1(w_i)^{\alpha}$이다. 따라서 T_i는 키워드 w_i에 대한 정당한 trapdoor가 되고, B는 A에게 T_i를 준다.

- **Challenge.** A는 challenge하려는 키워드들의 집합 $W_0 = \{w_{0,1}, \cdots, w_{0,n}\}$, $W_1 = \{w_{1,1}, \cdots, w_{1,n}\}$을 생성하는데, 여기서 각 키워드들은 서로 다르다고 가정한다. 그러면 B는 다음과 같이 challenge 인덱스들을 생성한다.

1. B는 H_1 요청을 $2n$번 수행하여 $h_{0,1} = H_1(w_{0,1})$, $\cdots, h_{0,n} = H_1(w_{0,n}), h_{1,1} = H_1(w_{1,1}), \cdots, h_{1,n} = H_1(w_{1,n})$ 값들을 얻는다.

2. $i \in [0,1], j \in [1,n]$이고 위에서 얻은 값들과 관련된 H_1-list의 튜플을 $\langle w_{i,j}, h_{i,j}, a_{i,j}, c_{i,j} \rangle$라고 할 때, 모든 $c_{i,j}$ 값이 1이면 B는 'failure'를 알리고 시뮬레이션을 종료한다.

3. 그렇지 않다면 적어도 하나의 $c_{i,j}$는 0이 되고, B는 $b \in \{0,1\}$을 랜덤하게 선택한다. 하지만 $c_{0,j}, c_{1,j} (1 \le j \le n)$ 중에서 모든 $c_{b,j} = 1$이 되는 경우가 있다면, 0이 되는 $c_{b,j}$를 포함하도록 b를 선택한다.

4. B는 $\{0,1\}^{\log p}$로부터 랜덤하게 $J_i (1 \le i \le n)$를 선택하고, 이 값들로부터 선택한 키워드들의 집합 W_b에 대한 인덱스 $C = [u_3, J_1, \cdots, J_n]$을 생성하여 A에게 전달한다. (Bloom Filter가 가지는 긍정 오류율(false positive rate)은 negligible하다는 가정 하에 본 안전성 증명에서는 Bloom Filter의 긍정오류율은 고려하지 않는다.)

- **More trapdoor queries.** A는 계속해서 원하는 키워드 w_i에 대한 trapdoor를 요청할 수 있다. 단, w_i는 W_0, W_1의 원소가 아니어야 한다. B는 A의 요청에 대해 앞에서와 마찬가지로 응답한다.

- **Output.** A는 challenge C가 W_0, W_1 중에서 어떤 키워드 집합의 인덱스인지를 나타내는 $b' \in \{0,1\}$을 출력한다. 그리고 B는 H_2-list에서 랜덤하게 (s, V)를 선택하고 $e(g,g)^{\alpha\beta\gamma}$의 값으로 $s/e(u_1, u_3)^{a_{b,j}}$를 출력한다. 여기서 $a_{b,j}$는 challenge 단계에서 선택된 키워드들의 집합 W_b 중에서, $c_{b,j} = 0$인 키워드 $w_{b,j}$에 해당하는 값이다.

다음으로 B가 정당한 $\hat{e}(g,g)^{\alpha\beta\gamma}$을 출력할 확률을 찾기 위하여, 먼저 B가 시뮬레이션을 하는 동안 'failure'하지 않을 확률을 구한다. 알고리즘 B가 A의 trapdoor 요청 결과로 'failure'하지 않을 확률은 적어도 $(1-\delta)^{q_T}$이고, challenge 단계에서 'failure'하지 않을 확률은 $\{1-(1-\delta)^{2n}\}$이다. A는 challenge하려는 키워드들의 집합 W_0, W_1에 대한 trapdoor를 요청할 수 없기 때문에 결국 시뮬레이션을 하는 동안 'failure'하지 않을 확률은 $(1-\delta)^{q_T}\{1-(1-\delta)^{2n}\}$이다. 이제 B가 주어진 BDH 입력에 대해 정당한 값을 출력할 확률이 적어도 $\epsilon/(nq_{H_2})$ 이상임을 보여야 한다.

Claim. 실제 공격 게임에서 A에게 공개키 $[g, u_1]$가 주어지고, 키워드들의 집합 W_0, W_1 상에서 challenge $C = [g^r, J_1, \cdots, J_n]$가 주어진다고 가정한다. 이 때, A가 W_0, W_1에 대한 H_2 요청을 할 확률은 적어도 2ϵ이다.

Proof. 편의를 위해 실제 공격에서 A가 W_0, W_1에 대한 H_2 요청을 하지 않는 사건을 E라고 하자. E가 발생할 경우, challenge C가 W_0와 W_1 중에서 어떤 키워드 집합의 검색 가능 암호문인지를 나타내는 b와 A가 challenge에 대한 답으로 출력하는 b'이 일치할 확률은 최대 1/2이다. 또한 A의 정의에 따라 다음의 부등식이 성립한다.

$$|\Pr[b=b'] - 1/2| \ge \epsilon$$

다음으로 $\Pr[b=b']$의 상한(upper bound)과 하한(lower bound) 값을 구한다.

$$\begin{aligned}
&\Pr[b=b'] \\
&= \Pr[b=b'|E]\Pr[E] + \Pr[b=b'|\tilde{E}]\Pr[\tilde{E}] \\
&\le \Pr[b=b'|E]\Pr[E] + \Pr[\tilde{E}] \\
&= \frac{1}{2}Pr[E] + \Pr[\tilde{E}] \\
&= \frac{1}{2} + \frac{1}{2}Pr[\tilde{E}],
\end{aligned}$$

$$\Pr[b=b'] \geq \Pr[b=b'|E]\Pr[E] = \frac{1}{2}\Pr[E] = \frac{1}{2} - \frac{1}{2}Pr[\widetilde{E}]$$

따라서 $\epsilon \leq |\Pr[b=b'] - 1/2| \leq \frac{1}{2}Pr[\widetilde{E_1}]$이고, 결국 E가 발생하지 않을 확률은 2ϵ보다 크거나 같다. 이제 B가 시뮬레이션 동안 'failure'하지 않는다고 가정하면, Claim에 의해서 A가 W_0 또는 W_1에 대한 H_2 요청을 할 확률은 적어도 2ϵ 이상이고, W_b에 대한 H_2 요청을 할 확률은 ϵ 이상이다. 또한 B는 H_2 요청을 최대 q_{H_2}번 할 수 있고, challenge 단계에서 W_0와 W_1에 대한 인덱스를 생성하기 위하여 $2n$번의 H_1 요청을 한다. 이러한 경우에 B가 시뮬레이션 동안 'failure'하지 않는다면, $e(H_1(w_{b,j}), u_1^{\gamma}) = e(g^{\beta + a_{b,j}}, g)^{\alpha\gamma}$을 만족하는 H_2-list의 튜플을 B가 선택할 확률은 적어도 $\epsilon/(nq_{H_2})$이다. 또한 B는 적어도 $(1-\delta)^{qT}\{1-(1-\delta)^{2n}\}$의 확률로 시뮬레이션을 하는 동안 'failure'하지 않기 때문에, B가 BDH 문제를 풀 수 있는 확률 ϵ'은 적어도 $\epsilon(1-\delta)^{qT}\{1-(1-\delta)^{2n}\}/(nq_{H_2})$이다.

3.6 공개키 기반 다중 키워드 검색 암호 기법

PESKS에서 다중 키워드 검색을 수행하기 위해서는 각각의 키워드에 대해 개별적인 검색을 수행한 후 검색된 문서 집합들의 교집합을 검색 결과로 반환하는 경우와 가능한 모든 키워드 조합을 인덱스로 저장하는 메타 키워드 검색 방식을 생각해 볼 수 있다[7]. 전자의 경우, 요청된 키워드 개수에 따라 검색 시간이 선형적으로 증가하고, 서버가 요청된 키워드 각각에 대한 검색 결과를 알 수 있기 때문에 필요 이상의 정보가 노출된다. 그리고 후자의 경우, 모든 키워드들의 조합을 인덱스로 저장하기 위해서는 2^n개의 키워드 조합을 고려해야 하기 때문에 메모리 낭비가 심하다. 이 문제를 해결하기 위해 공개키 기반 다중 키워드 검색 암호 기법(Public key Encryption with Conjunctive Keyword Search, PECKS)이 [7]에서 제안되었다. 하지만 PECKS 또한 키워드 필드의 개수가 증가 할수록 서버에 저장해야 하는 암호화된 인덱스의 개수가 선형적으로 증가한다는 단점을 가지고 있다. PECKS에서는 PESKS에서처럼 Bloom Filter를 적용하여 인덱스의 크기를 줄일 수 없다. 일반적으로 PECKS는 다중 키워드 검색을 지원하기 위해서 고유의 키워드 필드를 유지하고, 다중 키워드 검색 요청이 입력으로 들어오면 그 키워드들에 해당하는 키워드 필드의 인덱스 조합들과 Test 알고리즘을 수행한다 [7,9,10]. 만약 서버에 저장되는 인덱스들을 Bloom Filter를 이용하여 압축할 경우, 서버는 요청된 키워드 필드의 인덱스 정보를 변환된 Bloom Filter로부터 추출해 낼 수 없기 때문에, Test 알고리즘을 수행할 수 없게 된다. PECKS에 Bloom Filter를 적용하여 인덱스 크기를 줄이기 위해서는 키워드 필드 없이도 다중 키워드 검색이 가능한 기법이 연구되어야 한다.

Ⅳ. 효율성 분석

4.1 파라미터 설정

PESKS에 Bloom Filter를 적용하는 방법은 키워드 검색 요청이 있을 때, 요청한 키워드가 포함된 문서에 대해서는 정확하게 'Yes'를 출력하지만, 그 키워드가 포함되지 않은 문서에 대해서도 'Yes'를 출력하는 경우가 발생할 수 있다. 이는 Bloom Filter가 가지고 있는 긍정오류 특성(false positive rate)에 기인한다. 이 때, 요청한 키워드를 포함하지 않은 문서가 서버로부터 반환될 확률은 Bloom Filter의 긍정 오류율과 동일하다. 즉, Bloom Filter의 긍정오류율이 f_p라고 하면, 요청한 키워드를 포함하지 않은 문서를 이용하여 Test 알고리즘을 수행하였을 때 'Yes'가 나올 확률도 f_p이다. 이 긍정오류율이 높을수록 요청한 키워드를 포함하지 않은 문서가 많이 반환되며, 서버와 수신자 사이에는 불필요한 데이터 전달량이 발생하게 된다. 이 문제를 고려한다면 PESKS에 적용하는 Bloom Filter의 긍정오류율을 최소화 하는 것이 가장 효율적이지만, 긍정오류율을 최소화 하려면 그 만큼 Bloom Filter의 길이를 증가시켜야 한다. 기존 PESKS에 Bloom Filter를 적용한 방법은 각 문서가 포함하는 유일 키워드들에 대한 인덱스의 크기를 줄이기 위한 목적으로 제안된 것이기 때문에, Bloom Filter의 길이가 늘어난다는 것은 이러한 목적에 어긋나는 일이다. 따라서 Bloom Filter의 긍정오류율을 설정할 때는 주어진 서버 환경에서 긍정오류율을 무시할 수 있는 수준의 값으로 설정해야 하지만, 필요 이상으로 값을 낮출 경우 오히려 효율성이 떨어진다.

[2]에서 Goh는 자신들이 제안한 Z-IDX을 적용할 때, 2654개의 개인 문서가 존재하는 경우와 4개월 동안 보관된 e-mail이 1234개 존재하는 경우에 대해서 Bloom Filter의 적합한 긍정오류율을 $1/2^{10}$로 설정하였다. 이들의 경우, Z-IDX의 검색 결과로 반환된 문서들 중에서 요청된 키워드가 포함되지 않은 문서는 확률적으로 각각 2개, 1개 정도이기 때문에, 이를 무시할 수 있는 수준의 값으로 생각할 수 있다. 하지만 우리가 제안한 방법에서 $1/2^{10}$은 적당한 값이 되지 못한다. Z-IDX에서는 데이터를 업로드하고 다운로드하는 사용자가 온라인 서버를 자신의 개인 공간으로 활용할 수 있는 상황을 고려하였기 때문에, Bloom Filter의 긍정오류율이 적용되는 전체 집합의 크기가 개인의 데이터 개수로 제한된다. 하지만 PESKS에서는 데이터를 제공하는 송신자와 다운로드하여 이용하는 수신자가 서로 다르기 때문에, 서버는 N명의 사용자를 위한 모든 데이터를 저장하고 있으며 수신 받은 데이터가 누구를 위한 데이터인지 알 수 없다. 이러한 환경에서 긍정오류율이 적용되는 전체 집합의 크기는 Z-IDX보다 N배 증가하게 된다. 그렇기에 서버에 등록된 전체 사용자의 수(N)에 따라 긍정오류율을 다르게 설정해야 한다. 전체 사용자가 N이라 하였을 때, Bloom Filter의 긍정오류율을 $1/(N2^{10})$로 설정할 수 있다. 이 경우, Test 과정을 통해 반환되는 문서 중에서 요청된 키워드가 포함되지 않는 문서의 개수는 확률적으로 Z-IDX와 동일하게 된다.

Bloom Filter에서 설정해 주어야 하는 파라미터로는 해쉬 함수의 개수 ℓ, Bloom Filter의 크기 m, Bloom Filter에 저장할 원소의 개수 n, 긍정오류율 f_p이며, 이 파라미터들은 다음의 관계식을 만족한다.

$$f_p = (1/2)^{\ell} \geq (1/2)^{(\ln 2)m/n}$$

이 식은 2장의 관계식을 근사화한 것으로, 긍정오류율이 최소가 되는 파라미터들 사이의 관계를 보여 준다 [4]. 긍정오류율 f_p는 사용자 수 N이 주어지면 $1/(N2^{10})$로 설정되며, Bloom Filter에 저장할 원소의 개수 n은 [2]에서 제시한 e-mail news의 경우를 고려하여, 대략 200개 정도로 설정할 수 있다. 그리고 Bloom Filter의 크기 m과 해쉬 함수의 개수 ℓ은 긍정오류율 f_p와 Bloom Filter에 저장할 원소의 개수 n이 주어지면, 위의 관계식으로부터 구할 수 있다.

$$m = n\log_2(1/f_p)/\ln 2,$$

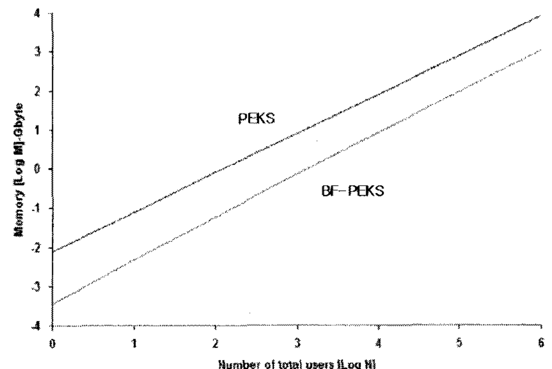
$$\ell = \log_2(1/f_p)$$

4.2 PEKS와의 BF-PEKS의 메모리 효율성 비교

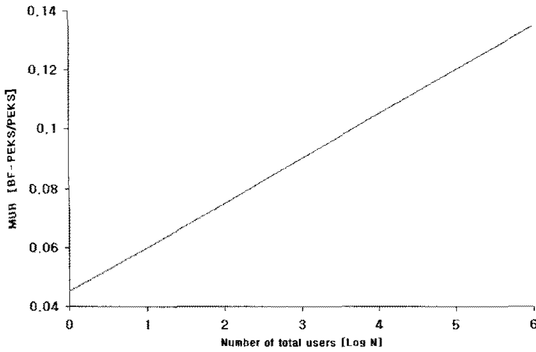
이 장에서는 메모리 측면에서 Boneh 등이 제안한 PEKS와 BF-PEKS의 효율성을 비교하고 분석한다. PEKS는 문서가 포함하고 있는 각 유일 키워드 w마다 $[g^r, H_2(e(H_1(w), y^r))]$ 형태로 키워드 인덱스가 생성된다. 우리는 일정 수준 이상의 안전성을 보장하기 위하여 G_1의 원소가 약 160-bit 정도의 크기를 갖는다고 가정한다. 그러면 g^r은 G_1의 원소이고 H_2 해쉬 함수는 $\log p$-bit 길이의 이진수를 출력하므로, 각 키워드의 인덱스를 저장하는데 부가적으로 320-bit의 메모리가 필요하다. 따라서 각 문서마다 유일 키워드가 n개 있다고 가정하면, 한 개의 문서에 해당하는 키워드 인덱스들을 저장하기 위해서는 $320n$-bit의 메모리가 필요하다. BF-PEKS는 각 문서에 대해서 $[g^r, BF]$의 형태로 인덱스가 저장되기 때문에, 위의 PEKS 경우와 동일한 조건이라면 한 개의 문서를 저장하는데 부가적으로 $(160+m)$-bit의 메모리가 필요하다. 우리는 PEKS와 BF-PEKS의 인덱스 메모리 사용량을 비교하기 위하여 아래와 같이 메모리 사용 비율(Memory Usage Rate)을 정의하고, 수식으로 나타낸다.

$$MUR = \frac{BF-PEKS\text{ 메모리사용량}}{PEKS\text{ 메모리사용량}} = \frac{(160+m)}{320n}$$

$$\approx \frac{n\log_2(1/f_p)}{320n\ln 2} = \frac{\log_2(1/f_p)}{320\ln 2} \approx \frac{10+\log_2 N}{221.8}$$



〔그림 1〕 전체 사용자 수 〔Log N〕에 따른 서버의 인덱스 메모리 사용량 비교



〔그림 2〕 전체 사용자 수 〔Log N〕에 따른 인덱스 메모리 사용 비율

위의 관계식은 앞에서 언급한 파라미터 값들을 적용한 것으로, 한 문서에서 PEKS와 BF-PEKS의 인덱스 메모리 사용 비율이 전체 사용자 수 N에 따라 달라진다는 것을 보여준다.

위의 그림들은 PEKS와 BF-PEKS의 전체 사용자 수 N에 따른 서버에서의 인덱스 메모리 사용량 [그림 1]과 인덱스 메모리 사용 비율(MUR, [그림 2])을 도식적으로 나타낸 것이다. 전체 사용자 수가 증가함에 따라 PEKS와 BF-PEKS의 메모리 사용량의 차이도 점점 커지며, 사용자가 1,000,000의 경우 인덱스에 할당되는 메모리 사용량은 6739 Gigabyte 정도의 차이를 보여준다([그림 1]). 그리고 인덱스 메모리 사용 비율(MUR)이 전체 사용수가 증가함에 따라 커지는 것을 알 수 있는데([그림 2]), 이는 전체 사용자수가 늘어날수록 PEKS의 메모리 사용량 대비 BF-PEKS의 메모리 사용량이 증가한다는 것을 의미한다. 하지만 [그림 2]는 메모리 사용 비율을 log scale로 나타낸 것이며, 실제로는 선형적으로 계속 증가하는 것이 아니라 대략 0.17 정도에서 saturation 된다. 따라서 BF-PEKS를 사용하면 전체 사용자 수에 관계없이 메모리 사용량을 적어도 1/5 이상으로 줄일 수 있으며, 이로부터 BF-PEKS가 메모리 측면 및 검색 효율성 측면에서 PEKS보다 효율적임을 확인할 수 있다.

Goh가 제안한 [2]의 예를 이용하여, 원본 문서를 포함하여 서버에서 저장해야 하는 PEKS와 BF-PEKS의 전체 메모리 사용량을 비교하면 다음과 같다. 서버에 등록된 전체 사용자를 $N=1{,}000$, 개인이 저장하는 문서의 평균적인 개수를 $nD=1{,}000$, 1,000개의 문서를 저장하는 데 필요한 메모리를 4 Megabyte, 문서가 포함하고 있는 유일 키워드의 최대 개수를 $n=200$으로 설정하면, PEKS를 적용하였을 경우, 서버에서 원본 문서를 저장하는데 4,000 Megabyte 정도의 메모리가 필요하며, 전체 인덱스를 저장하는 데 7,800 Megabyte 정도의 메모리가 요구된다. 하지만 BF-PEKS를 동일한 설정에 적용할 경우, 원본 문서를 저장하는 데에는 동일하게 4,000 Megabyte 정도가 요구되며, 전체 인덱스를 저장하는 데에는 720 Megabyte 정도가 요구된다. 위의 예에서 실제 저장하고자 하는 데이터 크기를 고려해보면, 인덱스를 저장하는데 상당히 많은 메모리가 필요하다는 것을 알 수 있다. 이처럼 인덱스의 크기가 커질수록 불필요하게 많은 메모리를 필요로 하게 되고, 이로 인해 실제 데이터를 저장할 수 있는 공간이 줄어들게 되므로 검색 암호 시스템에서 인덱스의 크기를 줄이는 것은 매우 중요한 문제이다. 따라서 기존의 Boneh 등의 방법보다 메모리 사용량을 최소 1/5 이상으로 줄인 본 논문의 BF-PEKS는 상당히 효율적이라는 것을 알 수 있다.

V. 결 론

본 논문에서는 공개키 기반 키워드 검색 암호 시스템에서 서버에 저장되는 인덱스의 크기를 줄이기 위해 기존의 PESKS와 PECKS에 Bloom Filter를 적용하는 방법을 고려하였다. PESKS에 Bloom Filter를 적용하는 방법이 메모리 측면에서 기존의 PESKS보다 효율적이라는 것을 보이기 위해, PEKS로부터 BF-PEKS를 설계하고, 선택적 키워드 공격으로부터 BDH 문제가 어렵다는 가정 하에 안전함을 증명하였다. 그리고 PEKS와 BF-PEKS를 비교하여 BF-PEKS가 송신자의 데이터 전송량, 서버에서 검색 시 사용하는 연산량 및 메모리 저장량 측면에서 효율적임을 보였다. 하지만 기존에 제안되었던 PECKS에 Bloom Filter를 적용하는 방법은 PECKS의 키워드 필드로 인해 적용하기가 어려운 것으로 나타났다. 앞으로의 연구로는 PECKS의 인덱스 사이를 줄이기 위한 새로운 방법의 모색과 키워드 필드 없이도 다중 키워드 검색이 가능한 PECKS 개발이 요구되어진다.

참고문헌

[1] D. Boneh, G. D. Crescenzo, R. Ostrovsky, G. Persiano, "Public Key Encryption with Keyword Search", *Advances in Cryptology - EUROCRYPT*,

LNCS 3027, pp.506-522, 2004.

[2] Eu-Jin Goh, "Secure Indexes", *Cryptology ePrint Archive*, Report 2003/216, 2003.

[3] B. H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors", *communications of the ACM*, vol. 13, No. 7, 1970.

[4] A. Broder, M. Mitzenmacher, "Network Appli cations of Bloom Filters: A Survey", *Internet Mathematics*, vol. 1, No. 4, 2003.

[5] D. Song, D. Wagner and A. Perrig, "Practical Techniques for Searches on Encrypted Data", *IEEE symposium on Security and Privacy*, 2000.

[6] P. Golle, J. Staddon, B. Waters, "Secure conjunctive keyword search over encrypted data", *ACNS 2004*, LNCS 3089, pp. 31-45, 2004.

[7] D. J. Park, K. Kim, P. J. Lee, "Public Key Encryption with Conjunctive Field Keyword Search", *WISA 2004*, LNCS 3325, pp. 73-86, 2004.

[8] M. Abdalla, M. Bellare, D. Catalano, E. Kilz, T. Kohno, T. Lange, J. Malone-Lee, G. Neven, P. Paillier, H. Shi, "Searchable Encryption Revisited: Consistency Properties, Relation to Anonymous IBE, and Extensions", *Advances in Cryptology - CRYPTO 2005*, LNCS 3621, pp. 205-222, 2005.

[9] D. Boneh, B. Waters, "Conjunctive, Subset, and Range Queries on Encrypted Data", *TCC 2007*, LNCS 4392, pp. 535-554, 2007.

[10] Y. H. Hwang, P. J. Lee, "Public Key Encryption with Conjunctive Keyword Search and itx Extension to a Multi-User Syste", *Pairing 2007*, LNCS 4575, pp. 2-22, 2007.

[11] M. Bellare, A. Boldyreva, J. Staddon, "Randomness Re-use in Multi-recepient Encryption Schemes", *PKC 2003*, LNCS 2567, pp. 85-99, 2003.

[12] K. Kurosawa, "Multi-Recepient Public-Key Encryption with Shortend Ciphertext", *PKC 2002*, LNCS 2274, pp. 48-63, 2002.

[13] A. Gordon, M. P. Loeb, W. Lucyshyn, and R. Richardson, "2004 CSI/FBI computer crime and security survey", Ninth annual report of computer security society, CSI, 2004.

[14] J. Katz, A. Sahai, B. Waters, "Predicate Encryption Supporting Disjunctions, Polynomial Equations, and Inner Products", ePrint, refer to "http://eprint.iacr.org/2007/404.pdf"

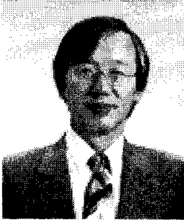
〈著者紹介〉

권 은 정 (Eun Jeong Kwon) 학생회원

2006년 2월 : 이화여자대학교 정보통신학과 졸업
2008년 2월 : 포항공과대학교 전자전기공학과 석사
2008년 3월~현재 : 삼성전자 정보통신총괄
<관심분야> 정보보호, 암호이론

서 재 우 (Jae Woo Seo) 학생회원

2005년 2월 : 경북대학교 전자전기공학부 졸업
2005년 3월~현재 : 포항공과대학교 전자/전기공학과 박사과정
<관심분야> 정보보호, 암호이론, 암호 프로토콜

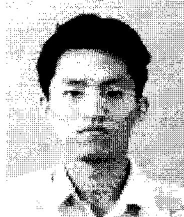


이 필 중 (Pil Joong Lee) 종신회원

1974년 2월 : 서울대학교 전자공학과 학사
1977년 2월 : 서울대학교 전자공학과 석사
1982년 6월 : U.C.L.A System Science. Engineer
1985년 6월 : U.C.L.A Electrical Engineering. Ph.D.
1980년 6월~1985년 8월 : Jet Propulsion Laboratory. Senior Engineer
1985년 8월~1990년 2월 : Bell Communications Research. M.T.S.
1990년 2월~현재 : 포항공과대학교 전자전기공학과 교수
1996년 2월~1997년 2월 : NEC Research Institute 방문 연구원
2000년 9월~2003년 8월 : 포항공과대학교 정보통신 연구소장 (정보통신 대학원장 겸임)
2004년 1월~2004년 12월 : 한국정보보호학회 회장
2004년 1월~2004년 12월 : KT 정보보호 자문위원
2007년 1월~현재 : 한국공학한림원 정회원
<관심분야> 정보보호전반

박 영 만 (Young Man Park) 정회원

1986년 2월 : 한양대학교 전자통신공학 (공학사)
1988년 9월 : 한양대학교 전자통신공학 (공학석사)
2004년 8월 : 한양대학교 전자통신전파공학 (공학박사)
1990년~현재 : KT 미래기술연구소
<관심분야> 정보보호, 무선보안, 암호 프로토콜



이 해 규 (Rhy Hae Gyu) 정회원

1989년 2월 : 서울대학교 컴퓨터공학과 학사
1991년 2월 : 서울대학교 컴퓨터공학과 석사
2001년 6월 : 서울대학교 컴퓨터공학과 박사과정 수료
1992년~현재 : KT 미래기술연구소
<관심분야> 인증, ID관리, 정보보호

김 영 헌 (Yeong Heon Kim)

1981년 2월 : 부산대학교 전기기계공학과 학사
1983년 2월 : KAIST 전기및전자공학과 전문석사
1983년 3월~현재 : KT 미래기술연구소(수석연구원)
<관심분야> 정보보호, Web2.0, IPTV

정 학 진 (Hak Jin Chong) 정회원

1985년~현재 : KT 미래기술연구소(USN연구담당 상무)
<관심분야> Sensor network, 영상처리, USN 서비스