한국정보전자통신기술학회논문지(jkiiect)'16-02, Vol.9 No.1
논문 15-09-01-015 http://dx.doi.org/10.17661/jkiiect.2016.9.1.015

IoT와 클라우드 서비스를 위한 유연한 암호화 시스템

김석우*

Flexible Crypto System for IoT and Cloud Service

SeokWoo Kim*

요 약 최근 다양한 IoT 기기들의 등장과 데이터 공유를 위하여 DropBox, Amazon S3, Microsoft Azure Storage 등의 클라우드 서비스가 많이 사용되고 있다. 데이터 보안을 위해 AES 같은 암호화 알고리즘이 널리 쓰이고 있지만, 데이터의 보안등급 이나 사용목적에 따라 LEA, SEED, ARIA 같은 여러 가지 다양한 방식의 암호 알고리즘을 선별적으로 유연하게 사용하지 못하고 있다. 이는 적절한 암호화 알고리즘을 사용하지 않음으로 인하여 전력 효율이 중요한 장치들에 더 많은 과부화가 걸리게 할 수 있고 암호화도 필요 이상으로 느려지게 되어 성능이 떨어지는 원인이 된다. 이에 본 논문에서는 IoT 장치 위에서 클라우드 서비스의 클라이언트 프로그램들이 데이터의 보안등급 이나 사용용도에 따라 암호화 알고리즘을 선택 할 수 있는 CloudGate 시스템을 설계 및 구현하였다. 선택적으로 LEA 경량화 알고리즘을 사용함으로써 AES를 사용할 때보다 최대 1.8배 더 빠르고 효율적으로 암호화를 할 수 있음을 확인하였다.

Abstract As various IoT devices appear recently, Cloud Services such as DropBox, Amazon S3, Microsoft Azure Storage, etc are widely use for data sharing across the devices. Although, cryptographic algorithms like AES is prevalently used for data security, there is no mechanisms to allow selectively and flexibly use wider spectrum of lightweight cryptographic algorithms such as LEA, SEED, ARIA. With this, IoT devices with lower computation power and limited battery life will suffer from overly expensive workload and cryptographic operations are slower than what is enough. In this paper, we designed and implemented a CloudGate that allows client programs of those cloud services to flexibly select a cryptographic algorithms depending on the required security level. By selectively using LEA lightweight algorithms, we could achieve the cryptographic operations could be maximum 1.8 faster and more efficient than using AES.

Key Words : IoT, Cloud Service, Crypto System, AES, LEA

1. 서론

최근 IoT 시대로 접어들면서 개개인이 소유하고 사용하는 IT 기기의 수가 증가하고 있는 추세이다. Desktop, Laptop, Tablet, Smart Phone 등을 비롯 각종 스마트 기기들이 쏟아져 나옴으로서 더 많은 데이터가 발생하게 되고 점점 더 많은 사용자들은 자신의 데이터를 유지 관리하기 위하여 다양한 클라우드 서비스를 이용하고 있는 추세이다[1,2].

클라우드 서비스는 기본적으로 클라이언트와 서버의 아키텍처를 지니고 있고, 클라이언트 프로그램을 IoT 장치에 설치하여 사용자 데이터를 프로그램에 로드해주면 원격의 클라우드 서버로 데이터 복사본을 전송. 저장하는 형태를 가지고 있다. 이러한 클라우드 서비스의 장점으로 고가용성과 안정성 및 편리성 등을 들 수 있는데, IoT 장치 사용자는 인터넷만 있다면 언제, 어느 곳에서든 데이터를 저장하고, 업데이트하고, 데이터를 받아 볼 수 있음을 뜻한다. 따라서, 다양한 사용자 장치들이 데이터를 일관성 있게 공유하기 위하여 클라우드 서비스는 필수 불가결한 솔루션으로 떠오르고 있는 상황이다.

* Corresponding Author : Department of IT, Hansei University(swkimone@gmail.com)
Received January 12, 2016 Revised January 20, 2016 Accepted February 01, 2016

그러나, 일반적으로 클라우드 서비스는 제 3자가 운영하고 관리하는 서버들로 구성되며, 데이터의 소유자 입장에서는 데이터의 보안이 많이 우려될 수 있고, 실제로도 클라우드가 해킹을 당함으로서 사용자의 정보가 노출되는 사고도 발생하고 있는 실정이다.

이러한 사용자들의 우려와 문제점을 해결하기 위하여 통상적으로 암호화 알고리즘을 사용하여 클라이언트에서 미리 데이터를 암호화한 후에 클라우드 서버로 보내는 방법이 사용되고 있다. 예를 들어, DropBox 서비스도 데이터를 암호화하여 서버로 보내고, 또한 더 조심스러운 사용자들은 BoxCrypt같은 어플리케이션을 사용하여, 클라이언트 상에서 데이터를 암호화를 한 후 DropBox로 데이터를 전송한다[3, 4, 5, 6].

이처럼 클라우드 서비스들이 암호화를 사용하고 있긴 하지만, 클라이언트 측에서 암호화를 하고, 네트워크를 타고 서버 측으로 데이터가 전송되어, 서버 측에서 데이터가 저장되는 환경을 고려할 때, IoT 기기들의 파워 소모 정도, 네트워크상의 압축 알고리즘 선택, 서버 측에서 데이터 수신확인 방식 선택 등의 여러 가지 선택해야만 하는 상황이 발생한다. 본 논문에서는 CloudGate의 IoT 클라이언트 사용자가 여러 암호 알고리즘 중 LEA를 선택하여 클라우드 서비스를 제공할 때와 AES 사용시의 성능을 비교 분석한다. 전 세계적으로 안전하다고 널리 알려져 있고 또 가장 표준으로 사용되는 AES 암호화 방식 및 국내 표준의 ARIA나 SEED의 경우 안전성 측면에서는 좋지만, 경량화 알고리즘인 LEA 같은 더 가볍고 빠른 알고리즘들보다는 속도가 많이 느리다[7,8,9]. 따라서 일반 서버나 데스크탑과 달리 성능이 떨어지고 배터리 효율이 중요한 IoT 장치들에서는 AES 같이 무거운 암호화 알고리즘만을 사용하는 경우 속도도 느려지고 필요이상으로 전력을 많이 소모할 수 있다. 최근의 발전하는 IoT 및 클라우드 시대를 맞이하면서 우리는 보다 더 다양한 데이터의 비밀성 과 안전성, 특히 파워 소모 및 성능을 고려할 필요가 있고, 따라서 클라우드 서비스들의 클라이언트 프로그램들이 효율적으로 알고리즘을 선택적으로 적절하게 사용할 수 있게 해주는 매커니즘이 필요하다.

이러한 사용자 편리성과 성능향상을 위하여 본 논문에서 클라우드 서비스들이 편리하게 사용할 수 있는 CloudGate 암호화 시스템을 설계 및 구현하였다. CloudGate는 클라우드 서비스의 클라이언트 프로그램이 사용하기 쉬운 Get과 Put 두개의 인터페이스 콜을 제공하고, 라이브러리 형식으로 설계되어 클라이언트 프로그램이 CloudGate 데몬서버와 인터렉션하지 않고도 암호화 서비스를 사용할 수 있게 설계하였다. 또한, 새 암호화 알고리즘을 언제든 쉽게 추가할 수 있도록 확장성을 고려하여 설계하였다. 성능시험 결과에서 LEA를 사용함으로서 AES만 사용하는 경우보다 속도가 1.5 ~ 1.8배 개선된 것을 확인할 수 있었다.

2. 아키텍쳐 개요

CloudGate시스템은 front-end와 back-end컴포넌트와 암복호화 관련 정보 저장소 컴포넌트로 이루어져있다. front-end 컴포넌트는 Client Library로 클라우드 서비스가 링크하여 사용할 수 있는 라이브러리로 설계되었고 RPC(Remote Procedure Call) 을 통해 back-end 컴포넌트인 CloudGate 데몬서버와 통신을 한다. CloudGate 데몬은 로컬 컴퓨터 내에 위치할 수도 있고, 외부 컴퓨터에 설치될 수도 있으며, 직접 클라우드 서버와 인터렉션을 하게 된다. CloudGate Interface 는 Init, Status, Get, Put 의 4가지 Interface 호출을 서비스하는데, CryptoStore 와 CloudeGate 데몬과 데이터 교환기능을 제공한다. CrytoStore 컴포넌트는 암복호화에 쓰이는 각종 키들과 기타 암복호화 관련 정보를 저장하는 저장소로써 CloudGate의 한 구성요소로 동작한다. 따라서, 클라이언트 상의 임의의 앱은 CloudGate Interface API를 통하여 다양한 암호화 알고리즘을 데이터 별 보안등급 및 서비스 형태에 따라 암호화 서비스를 제공받을 수 있으며, 클라우드 서버상에 데이터의 비밀성 과 데이터의 무결성

을 보장받을 수 있다. 기타 확장된 알고리즘 및 보안서비스의 다양한 응용이 가능하지만, 논문에서는 AES와 LEA에 대한 제안된 서비스를 보이는 구현으로만 범위를 제한한다.

[그림 1]에서 처럼 클라우드 서비스의 클라이언트 프로그램(App) 은 Client Library가 제공하는 CloudGate Interface를 통하여 Client Library를 호출하고, Client Library 자체는 넘겨받은 데이터를 선별적으로 암복호화 해주는 역할을 한다. CloudGate Interface를 통해 암호화된 데이터를 CloudGate Daemon에게 넘겨주거나 반대로 서버로 부터 다운되는 암호화된 데이터를 받아서 Crypto Library를 통해 복호화 하는 작업을 하게 된다. CloudGate Daemon은 실제 Cloud Service의 API 함수를 호출하여 클라우드 서비스에 직접 데이터를 읽고 쓰는 작업을 한다. 물론 하나의 클라이언트에 종속된 데몬으로 구성할 수도 있지만, 한 개의 CloudGate Daemon을 여러 개의 클라이언트 프로그램이 물리적인 디바이스 위치에 구애 받지 않고 네트워크를 통해서 동시에 공유하여 사용할 수 있게 해주기 위해서는 클라이언트-서버 아키텍처로 설계하였다. 또, 더 나아가 CloudGate Daemon을 구현하는 방식에 따라 더 확장가능하게 기능을 확장할 수도 있기 때문이다. 예를 들어서, CloudGate Daemon에 부하 분산과 라우팅 기능을 더하면, 하나의 클라우드 서비스만이 아니라 여러 개의클라우드 서비스를 동시에 사용할 수 있게 만들기 용이하다. 게다가, CloudGate Daemon이 좀 더 스마트한 프리페치 기능이나 네트워크 코딩기능을 포함하게 하여 Client Library를 바꾸지 않더라도 성능 향상과 보안성 향상을 한층 더 증대하는 것도 가능하기 때문에 [그림1]과 같은 아키텍처로 설계하여 기본기능을 구현하였다.

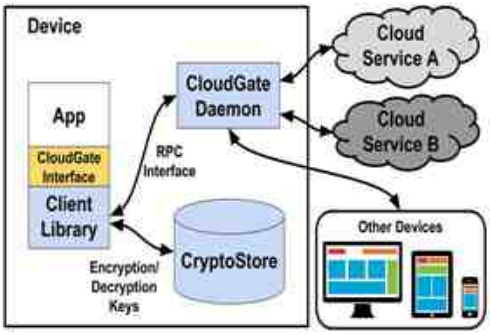

그림 1. CloudGate 아키텍처
Fig. 1. CloudGate Architecture

3. 클라이언트 라이브러리

본장에서는 2장의 아키텍처 컴포넌트 중에서 CloudGate의 Client Library 설계에 대하여 설명한다. [그림 2]에서 보듯 Client Library는 Library Stack과 Extensible Crypto Module 의 두 가지 구성요소로 설계되었다. Library Stack은 4개의 계층(Layer) API Layer, Selector Layer, Crypto Layer, RPC Layer로 구성되어 있고 각각의 계층은 특정한 역할을 수행한다. 한편, Extensible Crypto Module에는 여러 가지 암호 모듈이 장착될 수 있다. 예를 들어, 현재 버전의 CloudGate는 AES, LEA, ARIA, SEED 등의 알고리즘들의 실제 작업이 수행되는 모듈들로 구성되도록 되어 있다.

Library Stack을 좀 더 자세히 들여다보면, 우선 API Layer는 기본적으로 클라이언트 프로그램이 사용할 수 있는 인터페이스를 정의하고 있다. 간단히 Get과 Put 두 가지 기본 인터페이스를 제공한다. 더 자세한 인터페이스 관련 설명은 5장에 기술 되어 있다.

Selector Layer는 Library Stack의 핵심부분으로 클라이언트 프로그램의 요청에 포함된 보안등급 파라미터를 보고 적정한 Crypto Layer 인스턴스의 암호화 함수를 호출 하거나, RPC Layer를 통해 읽어온 데이터를 적정한 Crypto Layer의 인스턴스를 통해 복호화 하는 작업을 수행한다. Selector Layer는 각 데이터를 보안등급과 매핑을 시켜주고 이를 Security Level Table(메모리 해쉬

테이블)에 저장하는 기능을 수행한다. 예를 들어, 클라이언트 프로그램은 원하는 보안등급 및 보안 환경 설정과 대상 데이터를 Client Library에게 요청하게 되면, 관련 매핑(보안과 데이터 ID 연결)이 Security Level Table에 저장하였다가, 추후에 클라이언트 프로그램이 해당 데이터를 읽을 때 제공하는 데이터의 아이디와 RPC Layer로 부터 서버로 받아온 암호화된 데이터를 매핑하여 적절한 보안등급 및 데이터 암호화시 사용한 암호화 모듈을 선택하여 복호화하는 작업을 할 수 있다. Security Level Table에 설정된 최대 용량이 초과될 경우, CryptoStore로 테이블의 매핑 정보가 확장될 수도 있다. Crypto Layer는 Extensible Crypto Module 엔진을 포함하며, Selector Layer에 추상적인 암호화 및 복호화 함수를 제공한다. 따라서 실제로 암호화 및 복호화 되는 알고리즘이 수행되는 모듈을 Selector Layer로 부터 숨길 수 있기 때문에 새로운 암호화 모듈을 Selector Layer에 재구현하지 않고도 붙일 수 있다. 즉 Selector Layer 에서 Crypto Layer가 실제로 어떤 암호화 알고리즘을 구현했건, 한 가지 암복호화 API 함수를 이용해 Extensible Crypto Module내의 다양한 알고리즘으로 처리할 수 있게 해준다.

RPC Layer는 CloudGate Daemon과 인터렉션 하기 위한 RPC을 구현하여 Selector Layer로 부터 하의 계층의 복잡한 네트워킹 작업을 숨겨주는 역할을 한다. RPC Layer는 로컬호스트의 CloudGate Daemon과 통신할 수도 있지만, 원격의 다른 컴퓨터에 존재하는 외부 CloudGate Daemon과도 통신

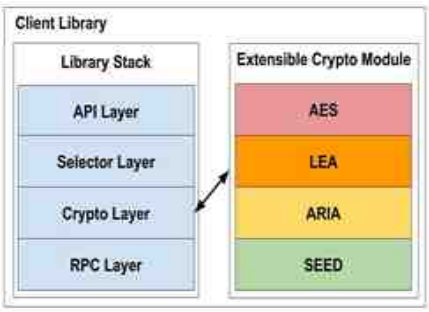

그림 2. Cloud Library 아키텍처
Fig. 2. Cloud Library Architecture

할 수 있다. 심지어, 실제 암호화 작업을 CloudGate Daemon으로 오프로드 시킬 수도 있다. 본 논문에서는 로컬이던 원격이던 모두 하나의 데몬으로 구성된다고 가정한다.

4. CloudGate Daemon

CloudGate Daemon서버는 내부적으로 RPC 서버가 항시 구동하고 있고 Client Library의 RPC Layer로 부터의 연결요청을 기다리고 있다가 연결요청을 처리하여 준다. [그림3]에서 보이는 바와 같이 클라이언트 프로그램이 Client Library를 통해 CloudGate를 사용할 때 다음의 이벤트가 순차적으로 발생한다.

① Client Library의 RPC Layer에서 RPC 서버로 연결 요청을 한다.

② RPC 서버는 여러 개의 Tread을 미리 생성하여 Thread Pool에 저장하고 있다가 Client Library로 부터 요청이 들어오면, 임의의 Tread를 Thread Pool에서 꺼내서 Handler Thread로 할당하여 클라이언트 프로그램(이후App으로 표기)의 요청을 처리하게 해준다.

③ App과 링크된 Client Library에서 Get 또는 Put 호출이 Handler Thread로 넘어 온다.

④ 다시 Handler Thread는 이를 적절한 클라우드 서비스의 API로 변환시켜서 클라우드 서버에 직접 데이터를 읽고 쓰기위한 요청을 한다.

⑤ 그리고 Handler Thread는 클라우드 서버의 응답을 받아

⑥ 다시 App에게 해당 클라우드 서버의 응답을 전달한다. 이제 Handler Thread는 주어진 역할을 다하였으므로, 다시 RPC 서버의 Thread Pool로 귀속되어 다음 요정을 처리할 때까지 대기한다. RPC Layer는 아래의 RPC 서버의 인터페이스를 통하여 CloudGate Daemon과 인터렉션을 한다.

· Get (ObjectID, CloudID)
· Put (ObjectID, EncryptedValue, CloudID)

ObjectID는 읽거나 쓰고자 하는 데이터의 아이디이고, CloudID는 원하는 클라우드 서버의 아이디

이다. 또한 EncryptedValue는 암호화된 데이터를 가리킨다. Get 함수가 Client Library에 의해 호출되면, CloudGate Daemon은 실제로 지정된 클라우드 서버로부터 요청된 데이터를 받아온 뒤에 결과를 Client Library로 돌려준다. Put 함수가 호출된 경우엔 CloudGate Daemon에서 암호화된 데이터를 적절한 클라우드 서버로 전달해주는 역할을 한다.

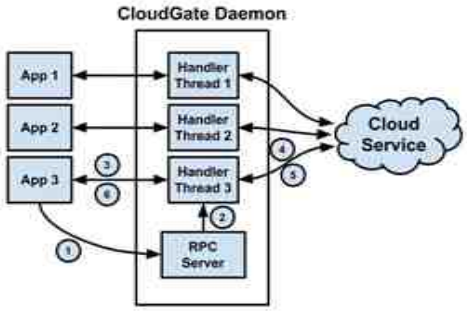

그림 3 CloudGage Daemon 아키텍처
Fig. 3 Cloud Library Daemon Architecture

5. CloudGate 인터페이스

CloudGate의 인터페이스는 다음과 같이 4가지 함수콜로 구성된다.

· Init (SecurityLablel, KeyInfo, EncryptionScheme)
· Status ()
· Get (ObjectID, CloudID)
· Put (ObjectID, Value, SecurityLable, CloudID)

먼저, Init함수는 각 Security level (보안등급) 및 관련된 KeyInfo (키정보)와 EncryptionScheme (암호화 알고리즘)을 지정해준다. 보안등급 과 키 정보 그리고 알고리즘을 매핑 시키고 저장한다. 저장은 메모리 데이터 구조에 동적으로 저장하고, 동시에 디스크 기반 키 값 저장 공간인 CryptoStore에 백업을 저장한다. Status함수는 이 매핑 정보를 표준 값으로 디스플레이 해준다. Get 함수는 CloudGate Daemon을 통해서 데이터를 읽어오게 해준다. 읽어온 데이터는 Selector Layer에 의하여 복호화된다. 이때 필요한 정보는 Security Level Table과 CryptoStore에 저장되어 있고 필요 시 메모리 데이터 구조나 CryptoStore에 저장된 것을 사용한다. Put 함수는 데이터를 클라우드 서버에 암호화 한 뒤에 CloudGate Daemon을 통하여 보내주는 작업을 수행한다. Get과 마찬가지로 필요한 암호화 관련 정보들을 Security Level Table과 CryptoStore에서 읽어와 사용한다.

보안등급은 사용자가 지정할 수 있게 되어있다. 매핑 될 각 보안등급 관련 암호화 알고리즘은 스트링타입으로 역시 사용자가 유일한 이름으로 지정하면 된다. 예를 들면, 보안등급 0과 AES 알고리즘(스트링 "AES" 표현)을 init 함수의 세번째 파라미터로 사용하여 매핑시킬 수 있고, 이와 비슷하게 1과 "LEA"를 매핑시킬 수도 있다. 사용자의 편의에 따라 매핑을 시킨 후에 적절한 알고리즘을 사용하고 싶을시 그 알고리즘과 매핑이 된 보안등급을 사용하도록 한다.

6. 구현

현재, Client Library가 jdk 1.8.0 JRE System Library와 commons-codec-1.10.jar을 참조 라이브러리로 사용하여 Java 코드로 구현하였고, RPC Layer와 데몬 서버는 Java의 XMLRPC 프레임워크를 사용하여 구현하였다. 암호화 알고리즘 AES는 javax.crypto 패키지에 구현되어 있는 것을 이용하였고, LEA는 독립적인 프로젝트로 구현된 것을 사용하였다. 구현된 프로토타입은 CloudGate 인터페이스와 Client Library Stack의 Selector, Crypto Layer들이 구현되어 있으며, 다음 섹션의 성능 실험을 통하여 다양한 알고리즘을 사용함으로서 얻어지는 성능 개선을 측정하기 위한 모듈로 동작한다. 구현된 암호 알고리즘은 AES, LEA, ARIA, SEED 의 4가지이지만, 실제 Counter mode로 구현 가능한 것은 AES 와 LEA로 성능 측정은 상기 두 가지 경우에 대하여만 실험하였다. RPC Layer 와 CloudGate Daemon과의 RPC 통신은 개발 진행 중으로 본 논문에서는 개략적인 API 로만 존재한다.

[표 1]에 프로토타입 CloudGate가 제공하는 실제

Java 인터페이스에 대한 설명이 나와 있다. 이 인터페이스를 활용해 다음의 [표 2]의 클라이언트 프로그램 SimpleExampleClient.java을 만들어 데모를 해본 결과 CloudGate 프로토타입이 성공적으로 작업을 수행하는 것을 확인할 수 있었다. 첫번째 단계는 init함수를 이용하여 보안등급과 해당되는 알고리즘 그리고 키 정보를 매핑해 주고, status 함수를 이용하여 매핑 정보를 질의해 보는 것이다. 그리고 두번째 단계는 실제로 데이터를 put함수로 쓰고 get함수로 읽어 들이는 것이다.

표 1. CloudGate 인터페이스 함수
Table 1. CloudGate Interface functions

함 수	설명
void init(int level, String[] keyInfo, String algo);	사용자가 지정한 보안 등급과 키 관련 정보가 들어 있는 keyinfo 그리고 알고리즘의 이름을 매핑시켜준다. 사용가능한 알고리즘의 이름 "AES", "LEA", "ARIA", "SEED"
void status();	현재, 보안 등급과 관련 알고리즘 이름 그리고 키관련 정보를 표준 출력으로 디스플레이 해준다.
String get(String oid, String cid);	데이터 아이디 oid를 클라우드 서버 cid로부터 읽어온다. Selector Layer에서 RPC Layer를 통해 해당 데이터를 해당 클라우드 서비스로부터 읽어온 후에 데이터가 처음에 클라우드에 보내지기전 사용하였던 암호화 알고리즘을 적용하여 복호화를 한 후에 string타임으로 변경하여 돌려준다.
void put(String oid, String value, int level, String cid);	데이터 아이디 oid에 데이터 값 value를 클라우드 서버 cid에 보안등급 level을 적용하여 암호화하여 저장한다. Selector Layer에서 해당 보안 등급에 적절한 알고리즘을 Security Level Table에서 찾아서 데이터를 암호화 한 후에 클라우드 서비스로 보내준다.

표 2. SimpleExampleClient.java 코드
Table 2. SimpleExampleClient.java code

```
public static void main(String[] args)
{
    CloudGateInterface cgi =
            new CloudGateInterface();
    String[] keyInfo = new String[2];
    keyInfo[0] = "Bar12345Bar12345";
    keyInfo[1] = "RandomInitVector";
    String cid = "testCloud";

    System.out.println("---------------"
            + "Demo 1: init & status");
    cgi.init(0, keyInfo, "AES");
    cgi.init(1, keyInfo, "LEA");
    cgi.init(2, keyInfo, "ARIA");
    cgi.init(3, keyInfo, "SEED");
    cgi.status();

    System.out.println("---------------"
            + " Demo 2: put & get");
    cgi.put("testOid1", "testValue1",
            0, cid);
    cgi.put("testOid2", "testValue2",
            1, cid);
    cgi.put("testOid3", "testValue3",
            0, cid);
    cgi.put("testOid4", "testValue4",
            1, cid);
    System.out.println("testOid1=" +
            cgi.get("testOid1", cid));
}
```

[그림 4]는 [표2]의 코드가 출력한 메세지를 스크린샷을 이용하여 디스플레이 하였다.

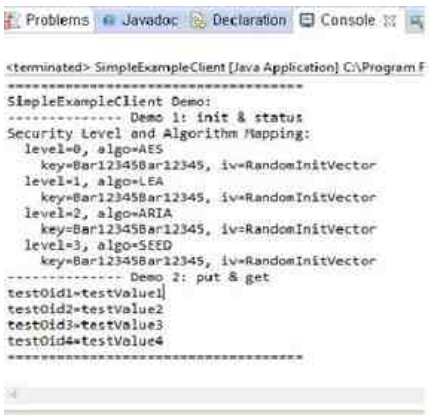

그림 4 코드 출력 메시지
Fig. 4. Code output message

예로, 코드에서 cgi.init() 함수를 이용해서 보안등급 0과 알고리즘 이름 "AES" 을 매핑하였고, status() 함수가 이 매핑을 standard out에 출력하여 보여준다. 또한 코드 1에서 cgi.put으로 "testOid1" 데이터 아이디에 "testVaelu1"을 쓰고 cgi.get으로 "testOid1" 데이터를 읽어서 값을 출력한 것이 "testOid1=testValue1"으로 출력된 것을 확인하였다.

7. 성능 실험

본 논문에서 CloudGate의 Crypto Layer에서 LEA와 같은 경량화 알고리즘을 AES대신 선택하여 사용함으로서 성능이 얼마나 개선될 수 있는지를 알아보는 것을 목적으로 실험을 설계 수행하였다. 데이터의 사이즈는 128바이트이고 키 사이즈는 128비트로 설정하였다. 암호화와 복호화 작업을 각각 10,000번식 수행하였다. 각 암호화 알고리즘은 기본적으로 block cipher로 운영모드는 counter mode를 사용하여 성능 실험을 하였다. 본 논문에서 클라이언트 프로그램의 요청은 Client Library의 API Layer를 통하여 Selector Layer를 거쳐 Crypto Layer에서 암복호화 되어 요청이 처리된다. 이 실험은 먼저 AES를 거쳐 암호화 되고 복호화 되는 시간을 측정하였다. 그리고 LEA를 거쳐 암복호화 되는 시간을 측정하고 두 결과를 비교하였다.

[표 3]에는 위의 작업을 5번씩 반복하여 평균값과 표준 편차를 ms단위로 보여주었다. [표 3]은 Client Library의 API Layer, Selector Layer 그리고 Crypto Layer에서 걸리는 시간을 모두 합한 시간을 의미한다. 결과적으로 LEA를 사용시 AES를 사용할 때 보다 1.53 ~ 1.8배의 성능 개선이 이뤄졌음을 확인하였다. 즉, 데이터를 좀 더 빈번하게 읽고 쓸 경우 CPU 소모량과 전력 소모량을 50 ~ 80% 가량 개선할 수 있음을 보인 것이다.

표 3. Client Library의 AES와 LEA 성능 비교
Table 3. AES and LEA performance comparison of the Client Library

	Get (ms)	Put (ms)
AES	177.8±11.3	213.4±12.4
LEA	116.2±6.2	113.8±10

이어서, Client Library의 각각의 Layer에서 걸리는 시간을 세부적으로 측정하여 [표 4]과 [표 5]를 얻었다. API Layer와 Selector Layer는 Crypto Layer에 비하여 상대적으로 아주 작은 시간이 걸렸다. 반면, Crypto Layer이 총 Client Library에 걸린 시간 의 대부분을 차지하였다. 우리는 Java 표준 라이브러리의 System.currentTimeMillis를 사용하여 시간을 ms 단위로 측정을 하였는데, 더 정확한 결과를 얻기 위해서는 ns로 더 특수한 측정 방법을 사용해야 한다. (따라서, 표 3과 표 4에서는 표준 편차가 별로 의미가 크지 않아 생략했다.) 결과적으로, Crypto Layer만 비교 하였을때 Get의 경우 1.54배 Put의 경우 1.86배 만큼 LEA가 AES에 비해 성능 향상이 되었다.

표 4. Client Library의 각 Layer 세부 측정(AES)
Table 4. Detailed measurements of each Client Library's Layer(AES)

	Get (ms)	Put (ms)
API	0.8	0.8
Selector	4.6	4.2
Crypto	164.4	200

표 5. Client Library의 각 Layer 세부 측정(LEA)
Table 5. Detailed measurements of each Client Library's Layer(LEA)

	Get (ms)	Put (ms)
API	0.8	0.4
Selector	1.8	1.4
Crypto	106.2	107.2

또한, 우리는 Crypto Layer 자체에서 걸리는 시간을 더 세부적으로 측정하여 [표 6]과 [표 7]을 얻었다. 실제 block cipher 내부에서 걸리는 시간을 측정했고 (AES/LEA Fn) cipher를 initialization 하는데 걸리는 시간 (AES/LEA Init)과 데이터 및 키를 String에서 byte[]로 또는 그 반대로 변환하는데 사용한 시간(AES/LEA Conv)을 측정했다.

표 6. AES 성능 세부 측정
Table 6. AES detailed performance measurements

	Encr (ms)	Decr (ms)
AES Fn	56.4±6	122.2±8.3
AES Init	37.4±3.5	15.8±4
AES Conv	95.6±15.1	10.2±2.9

표 7. LEA 성능 세부 측정
Table 7. LEA detailed performance measurements

	Encr (ms)	Decr (ms)
LEA Fn	37.8±5.1	100.6±8.4
LEA Init	2.2±1.3	6.4±0.9
LEA Conv	63.8±3.7	3.2±0.8

[표 6]과 [표 7]에 측정 결과가 같이, 복호화할 때 대부분의 시간은 cipher의 내부 복호화 작업에서 걸렸다. 한편, 암호화 할 때는 데이터의 타입을 변환하는데 더 많은 시간이 걸렸다. (이는 차후에 추가적인 최적화 작업을 통해 개선할 수 있다.) cipher에서만 걸린 시간을 비교해보면, 암호화의 경우 약 49%, 그리고 복호화의 경우 약 21% 가량이 성능 개선이 LEA를 사용함으로서 얻어진 것을 확인할 수 있었다.

8. 결론

본 논문에서는 IoT와 클라우드 서비스의 연동을 좀 더 효율적이고 좀 더 안전하게 사용할 수 있게 도와주는 암호화 시스템의 설계에 대하여 기술하였다. 그리고 CloudGate의 프로토타입 구현과 성능 실험 결과를 통해 유용성을 검증하였고 IoT와 클라우드 서비스 환경에서 효과적이고 유연한 암호화 시스템의 실현 가능성을 보였다. CloudGate는 또한 사용자가 유연하게 보안등급을 설계 지정하여 사용할 수 있고 다양한 암호화 모듈을 투명하게 확장할 수 있게 설계 되었고, 다가오는 클라우드 및 IoT 미래에 필요한 여러 가지 사용자 편리성과 보안서비스 모듈을 유연하게 선택 추가할 수 있는 가능성을 제시하였다. 차후, 여러 가지 암호알고리즘을 모듈화 구현하고, 하드웨어 성능에 기반한 속도향상 , 네트워크 알고리즘의 추가, 프록시 암호알고리즘 등의 보다 구체적인 구현단계 연구를 추가로 진행 할 예정이다.

REFERENCES

[1] J. Cook, “Google Drive Now Has 10 Million Users: Available on iOS and Chrome OS,” http://techcrunch.com/2012/06/28/google-drive-nowhas-10-million-users-available-on-ios-and-chrome-os-offline-editingin-docs/

[2] W. Santos, “76 Storage APIs: Box.net, Amazon S3, Dropbox,” http://www.programmableweb.com/news/76-storage-apis-box.netamazon-s3-dropbox/2012/01/31

[3] C. Soghoian. How Dropbox sacrifices user privacy for cost savings, 2011. http://paranoia.dubfire.net/2011/04/howdropbox-sacrifices-user-privacy-for.html (last accessed: 07/13/2012)

[4] I. Drago, M. Mellia, M. M Munafo, A. Sperotto, R. Sadre, and A. Pras, “Inside Dropbox: Understanding Personal Cloud Storage Services,” in The 2012 ACM conference on Internet measurement conference (IMC), Nov. 2012

[5] B. H. Kim and D. Lie, “Caelus: Verifying the consistency of cloud services with battery-powered devices,” in SP’15. IEEE, 2015.

[6] BoxCryptor, https://www.boxcryptor.com/en

[7] D. Hong, J. Lee, D. Kim, D. Kwon, K. H. Ryu, and D. Lee, “LEA: A 128-Bit Block Cipher for Fast Encryption on Common Processors” in Proc. WISA 2013, pp. 3-27, Jeju Island, Korea, Mar 2014.

[8] J. Park et al., “128-Bit Block Cipher LEA,” TTAK.KO-12.0223, Dec. 2013.

[9] NIST, “Advanced Encryption Standard(AES)“ Federal Information Processing Standard, FIPS PUB 197, Nov. 2001.

[10] J. Park, S. Lee, J. Kim, and J. Lee, The SEED Encryption Algorithm, RFC 4009, Feb. 2005.

[11] Korea Information Security Agency, www.kisa.or.kr , “ARIA , SEED”

[12] Cryptographic Module Validation Authority/Test organization, Cryptographic algorithm validation criteria (LEA, HEIGHT)”,

2015.6

[13] Young-Seok Lee, "Authentication Method for Safe Internet of Things Environments", JKIIECT '15-02. Vol.8 No1, pp. 51-58, Feb. 2015.

[14] Byung-Joon Park, "Study on Face recongnition algorithm using the eye detection", JKIIECT '15-12. Vol.8 No6, pp. 491-496, Dec. 2015.

저자약력

김 석 우(SeokWoo Kim) **[정회원]**

- 1979년 2월 한국항공대학교 통신공학과 공학사
- 1989년 10월 미국 뉴저지공대 전자계산학과 공학석사
- 1995년 2월 아주대학교 컴퓨터공학과 공학박사
- 1980년 8월~1997년 3월 국전자통신연구소 책임연구원 부호5실장
- 997년 3월~현재 한세대학교 정보통신학과 교수

<관심분야> 시스템 보안, 네트워크 보안, 시스템 평가