Cryptol 을 이용한 LSH 해시함수 정형검증

류도현, 최용락, 김태훈, 신영주
광운대학교 컴퓨터정보공학부 정보 및 사이버보안 연구실
Email : ehgus3919@naver.com, cyl941028@naver.com, milk8581@naver.com, yjshin@kw.ac.kr

Formal Verification of LSH Hash Function Using Cryptol

Do-Hyeon Ryu, Tae-Hoon Kim, Yong-Rak Choi, Youngjoo Shin
Information and Cyber Security Lab, School of Computer and Information Engineering
Kwang-Woon University

요 약

암호 알고리즘은 세계적으로 표준화가 진행되고 있으며, 암호 알고리즘의 안전성은 충분히 입증되어 왔다. 하지만, 기존 검증 방법으로 사용되는 테스트벡터 방식으로는 전수조사를 시행할 수 없어 구현상의 취약점을 완벽하게 발견할 수 없기 때문에 심각한 피해를 야기할 수 있다. 그래서 개발한 혹은 현재 존재하고 있는 암호 알고리즘이 표준에 따라 올바르게 구현되었는지에 대한 개선된 검증 방법이 필요하다. 본 논문에서는 KISA 에서 개발한 LSH 해시함수 모듈을 미국의 Galois 사와 NSA 가 함께 공동 개발한 Cryptol 을 이용하여 암호 모듈이 올바르게 구현되었는지 검증하였다.

1. 서론

기존 암호 모듈을 검증하기 위해서는 암호 설계자가 프로그래밍 언어를 직접 학습하여 설계에 맞게 구현되었는지 코드를 검토하고 표준과 함께 제공하는 테스트 벡터를 통해 정상적으로 수행되는지 검사하는 방법을 사용하였다. 기존의 테스트 벡터를 이용한 검증 방법은 입력 값에 대한 정상적인 출력 값을 확인하는 방법이며, 대표적으로 다음 세가지 방법을 사용한다.

- 기지 답안 검사(Known Answer Test)
- 다중 블록 메시지 검사(Multi-block Message Test)
- 몬테 카를로 검사(Monte CarloTest)

기존의 검증방법의 경우 테스트 벡터 내에 있는 입력데이터에 대한 출력데이터만 확인 할 수 있기 때문에 구현상의 취약점을 완벽하게 발견할 수 없었다. 완벽히 구현상의 취약점을 발견하기 위해서는 전수조사가 필요하지만 전수조사는 현실적인 시간 내로 시스템을 검증하기에 불가능하였다. 이러한 테스트 벡터를 이용한 방법의 한계점들을 개선하기 위해 2003 년 SAT/SMT 기반의 기호 실행을 이용하여 기능의 동일성 검사를 수행할 수 있는 Cryptol 이 개발되었다.

Cryptol 을 이용한 검증 방법은 기호 실행(Symbolic Execution)을 활용하여 규격과 동일하게 구현 되었는지 확인할 수 있다. 이를 동치성 검사(Equivalence Check)라 한다. 그리고 검증 기준이 될 수 있도록 참조 구현 모델을 제공하고, 기술된 검증 프로세스를 통해 암호 모듈의 검증을 수행할 수 있도록 제공한다.

암호 모듈의 정형 검증을 수행하는 도구인 Cryptol 특징은 다음과 같다.[1]

- 명세(Specification), 구현(Implementation), 검증(Verification), 인증(Certification) 각 단계에 검증을 위한 도구 제공
- 도메인 지향 언어(Domain-Specific Language) 기반으로 암호 설계자가 높은 수준의 솔루션을설계할 수 있도록 도움
- 플랫폼 독립적 명세로 다양한 환경에 적용 가능
- 명세에 소모되는 비용의 최소
- BSD(Berkeley Software Distribution) 라이선스 3 조항에 따라 오픈 소스로 배포

Cryptol 과 Cryptol 로 검증 시 사용되는 SAW (Software Analysis Workbench)를 개발한 Galois 사는 NSA 와 함께 AES(Advanced Encryption Standard) 블록 암호와 SHA(Secure Hash Algorithm), ECDSA(Elliptic Curve Digital Signature Algorithm) 암호 모듈을 검증하였고 Amazon 에서 웹 서비스로 사용되고 있는 s2n SSL/TLS 라이브러리의 HMAC(Hash-based Message Authentication Code) 모듈을 검증하였다. 그리고 일반 사용자들이 사용하는 라이브러리인 Bouncy Castle 와 libgcrypt 을 검증하였다.

또한 Galois 사는 DARPA(Defense Advanced

Research Projects Agency)의 HACMS(High Assurance Cyber Military Systems) 프로그램을 기반으로 고신뢰(High Assurance) CPS(Cyber-Physical System)를 구축하기 위한 연구를 수행 중이다.[1]

2. 관련 연구

해시함수는 임의 길이의 데이터를 입력 받아 고정된 길이의 데이터를 출력해주는 함수이다. 해시함수가 암호학적으로 사용되기 위해 역상 저항성(Preimage Resistance), 제 2 역상 저항성(2nd Preimage Resistance), 그리도 충돌 저항성(Collision Resistance)을 가지고 있어야 한다. 충돌 저항성과 제 2 역상 저항성은 충돌쌍 찾기 문제에 기반하며. 여기서 충돌 쌍이란 동일한 출력 값을 가지는 서로 다른 두 입력 값을 의미한다. 해시함수는 이러한 성질들을 만족해야 메시지 정보 변조 방지를 위한 인증 기술로서 활용될 수 있다. 해시함수는 파일의 무결성을 검증하는 용도로 사용되며 전자 서명과 의사 난수 생성, 암호 키 유도 등 여러 방면으로도 사용된다.

LSH(Lightweight Secure Hash)는 한국인터넷진흥원(KISA)에서 개발한 암호 알고리즘이며 w 비트 워드 단위로 동작하여 n 비트 출력 값을 가지는 해시함수 LSH-8w-n 으로 구성된 Hash Function Family 이다. 여기에서 w 는 시스템에 사용되는 32bit 또는 64bit 이고, n 은 1 과 8w 사이의 정수이다. 해시함수 LSH-8w-n 은 암호학적 해시함수의 성질을 모두 제공할 수 있도록 설계되었다.

Hash Function Family 에서 주요 해시함수 LSH-224, LSH-256, LSH-512-224, LSH-512-256, LSH-384, LSH-512 이며 LSH-224 와 LSH-256 은 각각 LSH-256-224, LSH-256-256 을 의미하고, LSH-384, LSH-512 는 각각 LSH-512-384, LSH-512-512 를 의미한다.[2]

구분	n	Ns	연쇄 변수 비트 길이	메시지 블록 비트 길이	w
LSH-224	224	26	512	1024	32
LSH-256	256				
LSH-512-224	224				
LSH-512-256	256	28	1024	2048	64
LSH-384	384				
LSH-512	512				

Table 1. 해시함수 LSH 규격

LSH 를 구성하는 각 해시함수는 Fig 1 와 같은 전체 구조를 가지며, 입력 메시지에 대해 다음의 세 가지 단계를 거쳐 해시 값을 출력한다.[2]

- 초기화(Initialization): 입력 메시지를 메시지 블록 비트 길이의 배수가 되도록 패딩을 한 후, 이를 메시지 블록 단위로 분할한다. 그리고 연결 변수를 IV 로 초기화한다.
- 압축(Compression): 32 워드 배열 메시지 블록을 압축 함수의 입력으로 하여 얻은 출력 값으로 연결 변수를 갱신하며, 이를 마지막 메시지 블록을 처리할 때까지 반복하여 메시지를 압축한다.
- 완료(Finalization): 압축 과정을 통해 연결 변수에 최종 저장된 값으로부터 n 비트 길이의 해시함수 출력값을 생성한다.

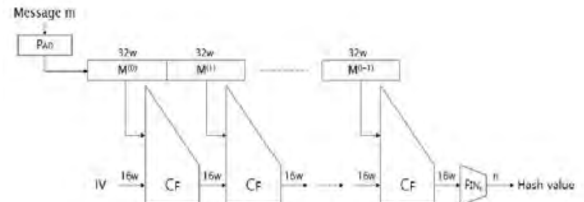

Fig 1. LSH 해시함수의 전체구조[2]

3. 검증환경 구축

3.1 검증절차

Fig 3 은암호 모듈의 검증 절차를 나타낸다.

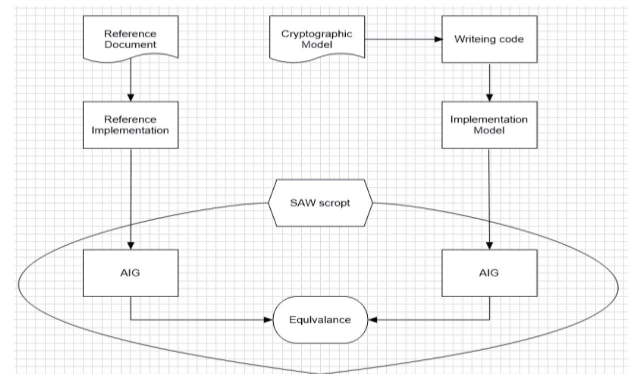

Fig 2. Cryptography Module Verification Process

참조 구현 모델, 래핑 코드(Wrapping Code), SAW 스크립트는 직접 구현해야 하며 표준 문서(Reference Document) 및 암호 모듈은 배포하는 문서나 모듈을 사용하고, 구현 모델 및 AIG(And-Inverter Graph)는 Cryptol과 llvm과 같은 도구를 사용하여 진행한다. AIG파일로 만들기전 clang을 사용하여 비트 코드로 컴파일 해야한다.

참조 구현 모델은 표준 문서를 Cryptol로 구현한 코드로 검증 시 암호 모듈의 비교 기준이 된다. 그렇기 때문에 한 번만 구현해도 되지만, 비교 기준이 되기 때문에 설계와 최대한 유사하게 구현해야 한다.

구현 모델(Implementation Model)은 검증하고자 하는 암호 모듈과 llvm-verifier 라이브러리를 이용하여 래핑 코드를 작성하고, 각 코드를 clang을 사용하여 비트 코드로 컴파일하고 llvm-link를 사용하여 모듈의 비트코드 파일과 래핑코드의 비트 코드를 링킹한다. 이렇게 생성된 구현 모델과 참조 구현 모델이 동일하게 구현되었는지 확인하기 위하여 LSS(LLVM Symbolic Simulator)를 사용하여 AIG파일로 변환시키고 작성된 SAW 스크립트를 실행하여 동치성 검사를

수행한다. SAW 스크립트는 모든 입력에 프로그램이 작동함을 확인하고 코드가 명세와 일치하지 않을 때 Valid를 출력하게끔 작성한다.

3.2 검증도구

암호 모듈의 검증을 위해 본 논문에서는 운영체제는 64bit 환경의 우분투 16.04에서 연구를 진행했으며 도구인 Cryptol은 cryptol-2.5.0 for Ubuntu 14.04-64 버전을 사용하였고, SAW는 saw-0.2-2016-04-12 for Ubuntu 14.04-64 버전을 사용하였다. 표준문서는 KISA에서 제공하는 LSH 해시함수 규격서를 사용하였고, 검증을 수행하기 위한 암호 모듈은 KISA에서 공식적으로 배포하는 LSH 해시함수 모듈 중 no_arch를 대상으로 선정하였다.[3] 또한 비트 코드로 컴파일하기 위한 도구인 Clang 3.6.2 for Ubuntu 14.04가 사용되었다.

3.3 참조 구현 모델 작성

표준 문서로부터 참조 구현 모델을 Cryptol로 구현할 때 표준 문서에 명세 된 내용과 최대한 일치하게끔 구현해야 한다. 참조 구현 모델은 암호 모듈을 검증할 때 기준이 되는 프로그램이기 때문에 최대한 규격에 맞게끔 정확하게 작성해야 한다.

Fig 3는 Cryptol내에서 LSH 해시함수를 구현한 것을 보여준다.

```
LSH : {msgLen} (fin msgLen, msgLen>=0) => [256]->[256]
LSH msg = LSH' pmsg
    where
        pmsg = pad msg

LSH' : {mpLen} (fin mpLen, mpLen>=0) => [mpLen][1024] -> [256]
LSH' mp = h
        where cv = [IV]#[CF(CV,M)|CV <-cv|M<-mp]
              h = FIN(cv!0)

CF : ([16][32], [1024]) -> [16][32]
CF (cv, msgBlock) = msgAdd (t!0, m!0)
        where m = msgExp msgBlock
              t = [cv]#[step(t@j, m@j, j)|j<-[0..25]]

step : ([16][32],[16][32], [8]) -> [16][32]
step (t, mExp, j) = wordPerm (mix (msgAdd (t, mExp), j))

msgAdd : ([16][32], [16][32]) -> [16][32]
msgAdd (X,Y) = [x^y | x<-X | y<-Y]

pad : {msgLen,outLen}
     ( fin msgLen,outLen == (msgLen + 1024)/1024) => [msgLen]-> [outLen][1024]
pad msg = split (msg # [True] # (zero:[padding]))
    where type padding = (1024-((msgLen + 1) % 1024))%1024
```

Fig 3. LSH Function in Cryptol

LSH 함수에서 암호화할 메시지를 전달받아 패딩을 한 다음 압축함수에 전달하고 압축함수에서 스텝을 돌면서 마지막에는 FIN 함수에 값을 전달해 주어 해시 값을 출력해준다. 이 코드는 LSH256-256 기준으로 작성되었으며 표준 문서에서 암호화에 사용되는 모든 기능을 전부 구현하였다.

3.4 래핑 코드 작성

구현 모델을 작성하기 위해서 먼저 암호 모듈을 호출하는 래핑 코드를 작성해야 한다. Fig 5는 KISA에서 제공해준 모듈의 래핑 코드이다.

```c
#include <stdio.h>
#include <string.h>
#include <sym-api.h>
#include "../include/lsh.h"
#include "../include/hmac.h"

int main()
{
    int i;
    lsh_u8* data = lss_fresh_array_uint8(32, 0, NULL);
    size_t databitlen = 256;
    lsh_u8* hashval = malloc(32*sizeof(unsigned char));

    lsh256_digest(LSH_TYPE_256_256, data, databitlen, hashval);
    lss_write_aiger_array_uint8(hashval, 32, "lsh_imp.aig");
}
```

Fig 4. Wrapping code for KISA LSH Module

래핑 코드는 Galois사의 Github 저장소에서 배포하고 있는 llvm-verifier에서 LSS를 수행하는 라이브러리를 사용하여 작성한다. LSS를 수행하는 라이브러리는 기호 실행을 위한 sym-api를 제공하고 입력 변수를 기호화 해주는 lss_fresh_array_uint8() 함수를 제공한다. 래핑 코드는 최대한 간결하게 작성하어야 하며 암호 모듈을 검증할 때 영향을 미치지 않도록 해야 한다. 모듈에서는 평문 부분을 입력 값으로 정의하고 lsh256_digest() 함수를 통해 암호 모듈을 호출하는 방식으로 구현하였다. 래핑 코드의 구현 시 암호 모듈마다 초기화를 먼저 수행해야 하는 경우도 있다. 매개변수 타입이나 반환 타입과 같이 모듈의 사용 방법이 각각 다르기 때문에 래핑 코드는 항상 새로 작성되어야 한다.

3.5 SAW 스크립트 작성

동치성 검사를 수행하기 위해 SAW 스크립트를 먼저 작성해야 한다, Fig 5는 SAW 스크립트의 코드를 나타낸다.

```
import "LSH.cry";
let {{
        lshExtract x = LSH x
}};

print "[+] Loading LSH Implementation Model";
lsh_imp <- time(load_aig "lsh_imp.aig");

print "[+] Loading LSH Reference Implementation Model";
lsh_ref <- time(bitblast {{ lshExtract }});

print "[*] Checking Equivalence (may take about an hour) : ";
res <- time (cec  lsh_ref lsh_imp);
print res;
```

Fig 5. SAW script implementation

SAW 스크립트는 참조 구현 모델과 구현 모델을 비교할 수 있도록 작성한다. 참조 구현 모델의 경우 Cryptol로 작성하였기 때문에 import 키워드를 사용하여 cry 파일을 로드한 다음 let 키워드를 이용하여 참조 구현 모델의 검증할 부분을 정의한다. 구현 모델의 경우 비트 코드로 컴파일하는 작업을 거쳤기 때문에 LSS를 이용하여 AIG 파일을 먼저 만든 다음 load_aig 함수를 호출하여 로드한다. 이후 생성된 두 개의 AIG 파일을 매개변수로 CEC(Combinational Equivalence Checking) 함수를

호출하여 동치성 검사를 수행한다. GNU에서 만든 make 유틸리티를 사용하면 각 파일에 필요한 명령을 정의할 수 있어 일괄 처리를 수행할수 있다.[1]

4. 검증

이전 단계를 직접 수행하였으면, SAW 를 사용하여 Cryptol 을 실행하고 검증을 수행할 수 있다. Fig 6 은 검증한 결과를 나타낸다.

```
[+] Loading LSH Implementation Model
Time: 0.202344s
[+] Loading LSH Reference Implementation Model
Time: 3.508145s
[*] Checking Equivalence (may take about an hour) :
Time: 0.177155s
Valid
```

Fig 6. Equivalence Checking

검증에 소요되는 시간은 참조 구현 모델과 암호 모듈, 혹은 시스템에 따라 다를 수 있으며, 일반적으로 수 시간 이내에 검증이 완료된다. Valid 가 출력되면 구현모델이 참조모델에 맞게끔 구현이 되었다는 것을 뜻한다.

5. 결론

암호 모듈을 검증하는 방법인 테스트 벡터의 문제점들을 해결하기 위해 본 논문에서는 Cryptol 을 사용하여 암호 설계자가 좀 더 쉽게 참조 구현모델을 설계에 맞게 구현할 수 있도록 제시 하였고, 구현 모델을 작성하기 위한 래핑 코드들과 SAW 스크립트를 통해 동치성 검사를 수행할 수 있도록 가이드를 제공하였다. 그리고 이 모델을 사용하여 LSH 해시함수를 검증하여 구현상의 안정성을 검증하였다.

Reference

1. Choi Won Bin, Kim Seung Ju. “Automated Formal Verification of Korean Standard Block Cipher using Cryptol” Journal of The Korea Institute of Information Security & Cryptology VOL.27, NO.6, Dec. 2017
2. KISA. LSH Datasheets-Korean
3. KISA. Hash function _LSH_ source code _ use _ manual (v1.0)