Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.28, NO.1, Feb. 2018 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2018.28.1.53

Cryptol을 이용한 국내 표준 블록 암호 모듈의 자동 정형 검증*

최 원 빈,† 김 승 주‡
고려대학교 정보보호대학원

Automated Formal Verification of Korean Standard Block Cipher Using Cryptol*

Won-bin Choi,† Seung-joo Kim‡
Center for Information Security Technologies(CIST), Korea University

요 약

암호 알고리즘은 세계적으로 표준화가 진행되고 있으며, 암호 알고리즘의 안전성은 충분히 입증되어 왔다. 하지만, 기존 검증 방법으로는 구현상의 취약점이 존재하여 심각한 피해를 야기할 수 있기 때문에 표준에 따라 올바르게 구현되었는지에 대한 개선된 검증 방법이 필요하다. 그러므로 본 논문에서는 국가정보원에서 수행하는 128비트 이상 블록 암호 모듈 중에서 검증 대상인 ARIA와 LEA를 선정하였고, 고신뢰 암호 모듈을 위해 Cryptol을 이용하여 올바르게 구현되었는지 검증하는 방법을 제시하고자 한다.

ABSTRACT

Cryptographic algorithms are being standardized globally, and the security of cryptographic algorithms has been well proven. However, there is a need for an improved verification method to verify that the existing verification method is correctly implemented according to the standard, because there is a weakness in implementation and it can cause serious damage. Therefore, in this paper, we selected ARIA and LEA to be verified among 128-bit or more block cipher modules performed by the National Intelligence Service, and propose a method to verify whether it is implemented correctly using Cryptol for high-assurance cryptographic module.

Keywords: formal, verification, cryptography, cryptol, saw

I. 서 론

일반적으로 국내 대칭키 암호 알고리즘을 사용하는 PC환경 시스템에서는 DES(Data Encryption Standard)와 유사한 구조를 가진 알고리즘인 SEED를 주로 사용해왔다. DES는 64비트 길이의 암호블록과 56비트 길이의 비밀키를 사용하는 표준 암호 알고리즘이다. 또한 SEED는 전자 상거래, 금융, 무선 통신 등에서 개인정보를 보호하기 위해 한국인터넷진흥원(KISA, Korea Internet & Security Agency)에서 제작한 블록 암호 알고리즘으로 SEED 128은 1999년에 정보통신단체표준, 2005년에 국제 표준화 기구인 ISO/IEC 국제 블록 암호, IETF 표준으로 제정되었으며 2009년에 암호 알고리즘 활용성 강화를 위해 SEED 256을 개발하

Received(09. 29. 2017), Modified(01. 02. 2018), Accepted(01. 29. 2018)
* "본 연구는 미래창조과학부 및 정보통신기술진흥센터의 대학 ICT연구센터육성 지원사업의 연구결과로 수행되었음" (IITP-2017-2015-0-00403)
† 주저자, bindon@hanmir.com
‡ 교신저자, skim71@korea.ac.kr(Corresponding author)

였다.

최근 IoT(Internet of Things, 사물인터넷)의 발전으로 PC에 비해 제한적인 환경에서 동작하는 임베디드 기기들의 사용이 증가함에 따라 구현 및 운용에 비용이 상대적으로 많이 필요한 SEED보다는 하드웨어 성능이 제한된 경량 환경에서의 구현에 최적화된 암호 모듈이 필요하게 되었다.

국가정보원에서 수행하는 암호 모듈 검증 대상에서 128비트 이상인 블록 암호 알고리즘은 ARIA(Academy Research Institute Agency), LEA(Lightweight Encryption Algorithm)가 있다.

ARIA는 국가기관 및 공공기관에 정보보호 제품을 공급하기 위해서 반드시 구현하고 검증받아야 하는 알고리즘으로서 경량 환경 및 하드웨어 구현을 위해 최적화된, Involutional SPN 구조를 갖는 범용 블록 암호 알고리즘이다. LEA는 빅데이터, 클라우드 등 고속 환경 및 모바일기기와 같은 경량 환경에서 기밀성을 제공하기 위해 개발된 알고리즘이다.

또한 암호 알고리즘을 구현한 암호 모듈을 사용하기 위해서는 반드시 검증이 필요한데, 기존 방법으로 검증된 TLS(Transport Layer Security) 및 SSL(Secure Sockets Layer)을 오픈소스로 구현한 OpenSSL의 경우 2014년 4월에 발견된 구현상의 취약점인 CVE-2014-0160 Heartbleed[1]가 존재하였고, 세계 사이트 조회 통계 사이트인 Alexa의 상위 100개 사이트를 확인한 결과 44개의 사이트가 취약한 OpenSSL의 버전을 사용하고 있는 것으로 나타났다. 이를 해결하기 위해 OpenSSL의 보안 패치를 제공하였으나 현재까지도 패치를 수행하지 않는 사이트가 존재하고 있다. 이와 같이 최초에 잘못 구현된 프로그램이 큰 파장을 일으킬 수 있음을 인지하고, 이를 최초에 확인하기 위해서 프로그램을 명세에 맞도록 안전하게 구현했는지 검증해야 한다.

기존 암호 모듈을 검증하기 위해서는 암호 설계자가 C, 자바, 파이썬과 같이 어려운 프로그래밍 언어를 따로 학습하여 설계와 동일하게 구현되었는지 직접 코드를 검토하고 표준과 함께 제공하는 테스트 벡터를 통해 정상적으로 수행되는지 검사하는 방법을 사용하였다.

기존의 테스트 벡터를 이용한 검증 방법의 경우 입력 값에 대한 정상적인 출력 값을 확인하는 방법을 사용하며, 대표적으로 기지 답안 검사(Known Answer Test), 다중 블록 메시지 검사(Multi-block Message Test), 몬테 카를로 검사(Monte Carlo Test)의 세 가지 시험으로 구성된다.

기지 답안 검사는 ARIA 알고리즘의 세부 구성요소가 정확하게 구현되었는지 확인하는 검사로서 평문을 0으로 고정하고 키를 변경하며 테스트를 수행하는 Variable Key 방식, 키를 0으로 고정하고 평문을 변경하며 테스트를 수행하는 Variable Text 방식, 키를 0으로 고정하고 평문 및 IV값을 임의로 변경하며 S-box가 정상적으로 동작하는지 테스트를 수행하는 S-box 방식이 있다. 기지 답안 검사는 일반적으로 각 운영모드와 키 길이별로 Variable Key 128개, Variable Text 128개, S-box 20개를 합한 총 276개의 테스트 케이스를 통해 검증을 수행한다.

다중 블록 메시지 검사는 긴 메시지의 암호화와 복호화를 올바르게 수행하였는지 구현에 대한 정확성 검사하는 테스트 방법이다. 이 테스트는 한 블록의 정보가 다음 블록에 연속적으로 영향을 미치는지를 확인한다. 한국인터넷진흥원에서는 각 운영모드와 키 길이별로 메시지 블록의 크기가 i×단일블록크기(i는 1부터 10까지)인 총 55블록의 테스트 케이스를 통해 검증을 수행한다.

몬테 카를로 검사는 구현 과정에서의 결함 여부를 확인하기 위해 각 운영모드와 키 길이별로 100개씩 테스트 케이스를 생성하고, 앞서 수행된 테스트의 암호문을 다음 단계의 평문으로 사용하는 방식을 1,000번 반복하여 총 10만회의 암호화를 수행한다.

알고리즘을 완벽하게 검증하기 위해서는 모든 평문, 키, IV의 값을 전부 검사해야 하지만, 테스트 케이스를 이용한 방법은 128비트의 경우 평문, 키, IV의 모든 경우의 수인 2^{384}개의 케이스 중 극히 일부분인 약 10만개의 케이스만 검사한다는 한계가 존재한다. 즉, 테스트 벡터에 해당하지 않는 입력을 공격자가 악의적으로 수정하면 이러한 검증 방법으로는 해결할 수 없다. 하지만 현재 컴퓨터의 연산 능력으로 전수조사를 수행하는 것은 현실적으로 불가능하였고, 국내에서는 이를 더 효율적으로 진행하기 위하여 테스트 벡터와 전수조사를 위한 도구를 개발하여 검증하는 연구[2]가 진행하여 개선되었으나 근본적인 문제점은 해결할 수 없었다.

그러므로 본 논문에서는 국내 대칭키 암호 알고리즘 표준 ARIA 및 LEA를 검증 대상으로, 기존 검증 방법의 문제점들을 개선하기 위해 Cryptol을 이용한 정형 검증 수행 방법과 참조 구현 모델

(Reference Implementation Model)을 제시하고자 한다. 이 검증 방법은 기존의 전수조사가 불가능하다는 문제를 개선하기 위하여, 입력 값과 출력 값의 확인이 아닌, 기호 실행(Symbolic Execution)을 활용하여 설계 내용과 동일하게 잘 구현 되었는지 확인하는 동치성 검사(Equivalence Check)를 수행한다. 또한 검증 시 구현해야 하는 참조 구현 모델을 검증 기준이 될 수 있도록 제공하고, 기술된 검증 프로세스를 통해 암호 모듈의 검증을 수행할 수 있도록 제공하고자 한다.

II. 관련 연구

기존 시스템의 경우 우리가 사용하는 일상 언어(Informal Language)로 명세하였지만 일상 언어로 기술된 시스템은 자동 검증이 불가능하였기 때문에 1987년에 자동제어 및 신호처리가 목적인 동기식 시스템을 정형 언어(Formal Language)로 명세할 수 있도록 LUSTRE[3] 언어를 제시하였다.

하지만 LUSTRE 언어는 동기식 시스템을 명세하기 위한 목적으로 연구되었기 때문에 자동 검증을 수행할 수 없었다. 이후 2003년에 LUSTRE를 기반으로 명세한 시스템을 순차적으로 전수조사를 수행할 수 있는 도구인 SCADE[4]가 개발되었으나 전수조사는 현실적인 시간 내로 시스템을 검증하기 불가능 하였다.

이러한 한계점들을 개선하기 위해 2003년에 SAT/SMT 기반의 기호 실행을 이용하여 기능의 동일함을 증명할 수 있는 동치성 검사를 자동으로 수행하는 Cryptol[5]이 개발되었다.

암호 모듈의 정형 검증을 수행하는 도구인 Cryptol은 NSA(National Security Agency)와 Galois사가 암호 알고리즘의 명세를 위해 공개 표준으로 공동 제작하였고 특징은 다음과 같다.

- 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 솔루션을 설계할 수 있도록 도움
- 플랫폼 독립적 명세로 다양한 환경에 적용 가능
- 명세에 소모되는 비용의 최소화
- BSD(Berkeley Software Distribution) 라이선스 3조항에 따라 오픈 소스로 배포

Galois사는 Cryptol과 SAW(Software Analysis Workbench)를 이용하여 NSA와 함께 AES(Advanced Encryption Standard) 블록 암호, SHA(Secure Hash Algorithm), ECDSA(Elliptic Curve Digital Signature Algorithm) 암호 모듈을 검증하였을 뿐만 아니라 Amazon 웹 서비스에서 실제 사용되고 있는 s2n SSL/TLS 라이브러리의 HMAC(Hash-based Message Authentication Code) 모듈을 검증하였고, 일반 사용자들이 많이 사용하는 라이브러리인 libgcrypt, Bouncy Castle을 검증하였다[6-8].

또한 Galois사는 미국 DARPA(Defense Advanced Research Projects Agency)의 HACMS(High Assurance Cyber Military Systems) 프로그램을 기반으로 고신뢰(High Assurance) CPS(Cyber-Physical System)를 효과적으로 구축하기 위한 연구를 수행하고 있다[9].

III. 검증 환경 구축

3.1 검증 절차

Fig 1은 암호 모듈의 검증 절차를 나타낸다.

표준 문서(Reference Document) 및 암호 모듈은 외부에서 배포하는 문서나 모듈을 수집하고, 참조 구현 모델, 래핑 코드(Wrapping Code), SAW 스크립트는 직접 구현해야 하며 구현 모델 및 AIG(And-Inverter Graph)는 도구를 사용하여 진행한다.

참조 구현 모델은 표준 문서를 Cryptol 도구로 구현한 프로그램으로 검증 시 모든 모듈의 비교 기준이 되기 때문에 한 번만 구현해도 되지만, 가장 중요

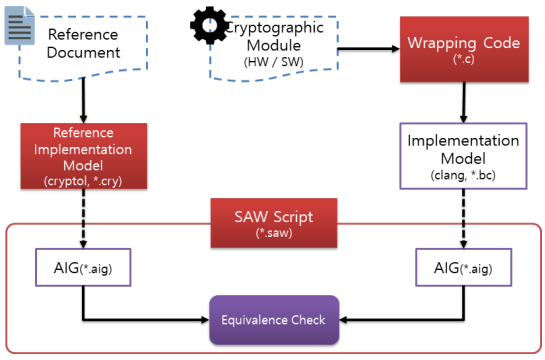

Fig. 1. Cryptography Module Verification Process

하기 때문에 설계와 최대한 유사하게 구현해야 한다.

구현 모델(Implementation Model)은 검증하고자 하는 암호 모듈과 llvm-verifier 라이브러리를 이용하여 래핑 코드를 작성하고, 각 코드를 clang으로 컴파일 하여 비트코드로 변환하고 llvm-link 도구를 사용하여 하나의 비트코드 파일로 변환 한다.

이렇게 생성된 구현 모델과 참조 구현 모델이 동일하게 구현 되었는지 확인하기 위하여 LSS(LLVM Symbolic Simulator)를 사용하여 AIG로 변환한 다음 작성된 SAW 스크립트를 수행하여 동치성 검사를 수행한다.

SAW[10]는 SAT/SMT Solver를 사용하여 프로세스를 최대한 자동화하고 SAW 스크립트를 통해 복잡한 시스템까지 확장하여 검증이 가능하며, 모든 입력에 프로그램이 작동함을 확인하고 코드가 명세와 일치하지 않을 때 반례를 찾는 도구이다.

3.2 검증 도구

암호 모듈의 검증을 위해 본 연구에서는 다음과 같은 환경을 구축하였다.

표준 문서는 한국인터넷진흥원에서 제공하는 ARIA Specification과 블록암호 LEA 규격서를 사용하였고, 검증을 수행하기 위한 암호 모듈은 한국인터넷진흥원에서 공식적으로 배포하는 ARIA 8비트 모듈, ARIA 32비트 모듈, LEA Standalone C 모듈과, Github에서 배포하는 hi-hee의 ARIA 모듈, minjeongJho의 ARIA 모듈, odzhan의 LEA 모듈을 대상으로 선정하였다.

운영체제는 64비트 환경의 칼리 리눅스와 우분투 16.04에서 진행되었으며, 주요 도구인 Cryptol은 cryptol-2.5.0 for Ubuntu14.04-64 버전을 사용하였고, SAW는 saw-0.2-2016-04-12 for Ubuntu14.04-64 버전을 사용하였다.

또한 각 도구의 사용을 돕기 위해 사용된 도구는 비트코드로 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 성능 및 최적화에 초점을 맞추어 구현하기보다는 표준 문서에 명세 된 내용과 최대한 일치하는 방법으로 구현해야 한다. 참조 구현 모델은 모든 암호 모듈을 검증할 때 기준이 되는 프로그램이기 때문에 최대한 정확하게 작성해야 한다.

Fig 2는 표준 문서 5.3 키 확장 알고리즘을 구현한 함수를 나타낸다. 키 확장 알고리즘은 5.3.1 초기화 단계에서 짝수 함수와 홀수 함수를 호출하여 W_0부터 W_3까지의 값을 초기화 하고, 5.3.2 라운드 키 생성 단계에서 순환 이동을 통해 암호화를 위한 13개의 라운드 키를 생성한다.

Fig 3은 표준 문서 5장 알고리즘 구조의 암호화 과정을 구현한 함수를 나타낸다. 암호화 과정은 앞에서 정의된 암호화 키 생성 함수를 호출하여 키를 초기화 한 다음 홀수 라운드에서 홀수 함수를, 짝수 라운드에서는 짝수 함수를 호출하고 마지막 라운드는 최종 함수를 호출하여 암호문을 반환한다.

이와 같은 방법으로 표준 문서에서 암호화 및 복호화에 사용되는 모든 기능을 전부 구현해야 하며, 표준 문서와 최대한 유사하게 작성해야 한다.

```
// 5.3    stretch key
//  5.3.1 initialize
//  5.3.2 generate round key
generateEncKey : [128] -> [13][128]
generateEncKey masterKey = [ek1, ek2, ek3, ek4, ek5, ek6, ek7,
                            ek8, ek9, ek10, ek11, ek12, ek13]
    where
        w0   = masterKey
        w1   = oddFunction(w0, ck @ 0)  ^ zero : [128]
        w2   = evenFunction(w1, ck @ 1) ^ w0
        w3   = oddFunction(w2, ck @ 2)  ^ w1
        ek1  = (w0)          ^ (w1 >>> 19)
        ek2  = (w1)          ^ (w2 >>> 19)
        ek3  = (w2)          ^ (w3 >>> 19)
        ek4  = (w0 >>> 19) ^ (w3)
        ek5  = (w0)          ^ (w1 >>> 31)
        ek6  = (w1)          ^ (w2 >>> 31)
        ek7  = (w2)          ^ (w3 >>> 31)
        ek8  = (w0 >>> 31) ^ (w3)
        ek9  = (w0)          ^ (w1 <<< 61)
        ek10 = (w1)          ^ (w2 <<< 61)
        ek11 = (w2)          ^ (w3 <<< 61)
        ek12 = (w0 <<< 61) ^ (w3)
        ek13 = (w0)          ^ (w1 <<< 31)
```

Fig. 2. Key Generation in Reference Model

```
// encrypt plaintext
// in  : plaintext, key
// out : ciphertext
encrypt : ([128], [128]) -> [128]
encrypt (plainText, key) = cipherText
    where
        ek   = generateEncKey(key)
        r1   = oddFunction (plainText, ek @ 0 )
        r2   = evenFunction(r1,        ek @ 1 )
        r3   = oddFunction (r2,        ek @ 2 )
        r4   = evenFunction(r3,        ek @ 3 )
        r5   = oddFunction (r4,        ek @ 4 )
        r6   = evenFunction(r5,        ek @ 5 )
        r7   = oddFunction (r6,        ek @ 6 )
        r8   = evenFunction(r7,        ek @ 7 )
        r9   = oddFunction (r8,        ek @ 8 )
        r10  = evenFunction(r9,        ek @ 9 )
        r11  = oddFunction (r10,       ek @ 10)
        cipherText = finalFunction (r11, ek @ 11, ek @ 12)
```

Fig. 3. Encryption in Reference Model

3.4 래핑 코드 작성

구현 모델을 작성하기 위해서 먼저 암호 모듈을 호출하는 래핑 코드를 작성해야 한다. Fig 4는 KISA 32bit 모듈의 래핑 코드이다.

래핑 코드를 작성하기 위해서 Galois사의 Github 저장소에서 배포하는 llvm-verifier를 다운로드 하여 LSS 수행을 위한 라이브러리를 사용해야 한다. 이 라이브러리는 기호 실행을 위한 sym-api를 제공하고 입력 변수를 지정할 수 있는 lss_fresh_array_uint8()과 같은 함수를 제공한다.

래핑 코드는 최대한 간결하게 작성하여 암호 모듈을 검증할 때 영향을 미치지 않도록 해야 하며, KISA 32bit 모듈에서는 평문과 키 부분을 입력 값으로 정의하고 encrypt() 함수를 통해 암호 모듈을 호출하는 방식으로 구현하였다.

래핑 코드의 구현 시 암호 모듈마다 초기화를 먼저 수행해야 하는 경우도 있으며, 구조체를 사용하거나 매개변수 타입, 반환 타입과 같이 모듈의 사용 방법이 각자 상이하기 때문에 이를 유의해야 하며 항상 새로 작성되어야 한다.

```c
#include "ARIA.h"
#include <sym-api.h>

// call cryptographic module
void encrypt(
    unsigned char *plainText,
    unsigned char *cipherText,
    unsigned char *key) {
    // initialize parameter
    unsigned char rk[16*17];
    int r = EncKeySetup(key, rk, 128);

    // call encrypt() function
    Crypt(plainText, r, rk, cipherText);
}

int main() {
    // definition symbolic variable
    // plaintext(16 bytes), key(16 bytes)
    unsigned char *plainText  = lss_fresh_array_uint8(16, 0, NULL);
    unsigned char *key        = lss_fresh_array_uint8(16, 0, NULL);
    unsigned char *cipherText = malloc(16 * sizeof(unsigned char));

    // call cryptographic module
    encrypt(plainText, cipherText, key);

    // write result to AIG file
    lss_write_aiger_array_uint8(cipherText, 16, "ARIA_imp.aig");

    return 0;
}
```

Fig. 4. Wrapping code for KISA 32bit Module

3.5 SAW 스크립트 작성

동치성 검사를 수행하기 위해서는 SAW 스크립트를 먼저 작성해야 하며, Fig 5는 SAW 스크립트의 일부를 나타낸다.

```
import "../ARIA.cry";

let {{
  ariaExtract x = encrypt (plainText, key)
    where [plainText, key] = split x
}};

print "[+] Loading ARIA Implementation Model";
aria_imp <- time (load_aig "ARIA_imp.aig");

print "[+] Writing reference AIG";
time (write_aig "../ARIA_ref.aig" {{ ariaExtract }});

print "[+] Loading ARIA Reference Implementation Model";
aria_ref <- time (bitblast {{ ariaExtract }});

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

Fig. 5. SAW script implementation

SAW 스크립트는 참조 구현 모델과 구현 모델을 비교할 수 있도록 다음과 같이 작성한다.

참조 구현 모델의 경우 Cryptol 도구를 사용하였기 때문에 import 키워드를 사용하여 cry 파일을 로드한 다음 let 키워드를 이용하여 참조 구현 모델의 어떤 부분을 검증할지 정의한다. 이후 bitblast 함수를 사용하여 AIG 파일을 생성할 수 있다.

구현 모델의 경우 비트코드로 변환하는 작업을 거쳤기 때문에 LSS를 사용하여 AIG 파일을 먼저 생성한 다음 load_aig 함수를 호출하여 로드한다.

이후 생성된 두 개의 AIG파일을 매개변수로 CEC(Combinational Equivalence Checking) 함수를 호출하여 동치성 검사를 수행한다.

3.6 자동화

암호 모듈을 개발하거나 사용할 때 코드의 변경이

```
CLANG=clang
LLVM_LINK=llvm-link
SAW=saw
LSS=lss
CFLAG=-emit-llvm
LLVM_VERIFIER=/home/bindon/cryptol/llvm-verifier/sym-api

OBJS=ARIA_core.o ARIA_wrap.o

all: verify

ARIA_core.o: ARIA_core.c
    @echo "[+] Compile $<"
    @$(CLANG) $(CFLAG) -o $@ -c $<

ARIA_wrap.o: ARIA_wrap.c
    @echo "[+] Compile $<"
    @$(CLANG) $(CFLAG) -I${LLVM_VERIFIER} -o $@ -c $<

ARIA_imp.bc: $(OBJS)
    @echo "[+] Linking $@"
    @$(LLVM_LINK) -o $@ $^
```

Fig. 6. Makefile for automation of verification

빈번하게 일어나는데 매번 앞에서 실시한 검증 단계를 거치기에는 많은 시간이 소요되므로, Fig 6과 같이 clang을 이용한 컴파일, 링킹, SAW 실행의 과정을 Makefile에 기술하여 자동화 하면 일련의 과정을 make 명령 한 번으로 수행할 수 있다.

GNU에서 제작한 make는 소프트웨어 개발을 위한 빌드 도구로 여러 파일들끼리의 의존성과 각 파일에 필요한 명령을 정의하여 일괄 처리를 수행할 수 있도록 도와주는 도구이다.

IV. 검 증

이전 단계를 직접 수행하였으면, SAW를 사용하여 스크립트를 실행하고, Makefile로 작성하였다면 make를 수행하여 일련의 작업을 한 번에 수행할 수 있다. Fig 7은 make를 수행하여 검증한 결과를 나타낸다.

검증에 소요되는 시간은 참조 구현 모델과 암호 모듈에 따라 상이할 수 있으며, 일반적으로 24시간 이내에 검증이 완료된다.

Table 1은 앞서 선정한 모듈에 현재까지의 모든 단계들을 거쳐 동치성 검사를 수행하여 도출된 결과를 나타낸다.

ARIA 암호 모듈의 경우 KISA 8bit 모듈과 32bit 모듈은 래핑 코드만 작성하여 정상적으로 검증할 수 있었으나, Github에서 제공하는 hi-hee, minjeongJho 사용자의 ARIA 코드는 문제가 발견되어 컴파일이 불가능했기 때문에 일부 소스코드의 수정이 필요하였다.

hi-hee의 ARIA 모듈에서는 2개의 보안 약점(Weakness)이 존재하였는데, 이 부분은 일반적인 C 컴파일러에서는 컴파일이 가능하지만 비트코드로는 오류가 발생하여 컴파일이 불가능하다.

```
bindon@creator:~/cryptol/workspace/ARIA/KISA-32bit$ make
[+] Compile ARIA_core.c
[+] Compile ARIA_wrap.c
[+] Linking ARIA_imp.bc
[+] Generate AIG File: ARIA_imp.aig
Obtained concrete return value from main(): 0
19.71user 0.18system 0:23.38elapsed 85%CPU (0avgtext+0avgdata 91628maxresident)k
44840inputs+1480outputs (190major+18465minor)pagefaults 0swaps

[*] Formal Verification
Loading module Cryptol
Loading file "ARIA.saw"
Loading module ARIA
[+] Loading ARIA Implementation Model
Time: 0.220602s
[+] Writing reference AIG
Time: 2.03372s
[+] Loading ARIA Reference Implementation Model
Time: 1.983277s
[*] Checking Equivalence (may take about an hour):
Time: 2401.322766s
Valid
bindon@creator:~/cryptol/workspace/ARIA/KISA-32bit$
```

Fig. 7. Equivalence Checking using Makefile

Table 1. Result of verification

Module Name	Error	Equivalence
KISA 8bit ARIA	X	Valid
KISA 32bit ARIA	X	Valid
hi-hee ARIA	O	Partial Valid
minjeongJho ARIA	O	Partial Valid
KISA LEA	X	Valid
odzhan LEA	X	Valid

첫 번째 보안 약점은 aria.c 파일에서 정의된 KeySchedule() 함수의 매개변수가 2개인 반면 구현된 같은 이름의 함수는 const char의 매개변수가 추가되어 총 3개의 매개변수를 가져 정의부와 구현부가 상이한 형태를 가지고 있었다.

또한 mode.c 파일에 정의된 MmtWriteFile() 함수의 정의부와 구현부의 배열 크기가 일치하지 않았다. 이 보안 약점의 경우 버퍼 오버플로우 취약점으로 악용될 수 있거나 데이터가 누락될 수도 있기 때문에 반드시 수정이 필요하다.

minjeongJho의 ARIA 모듈에서는 바이트 단위로 저장 및 연산되어야 함에도 불구하고 전부 int형 배열 및 변수로 정의되어 있었기 때문에 저장 공간이 32비트 환경에서 4배만큼 차지하고 각 연산마다 3바이트만큼의 쓸모없는 연산을 추가적으로 수행하기 때문에 공간복잡도와 시간복잡도가 엄청나게 증가한다. 만약 전수조사를 수행할 경우 n비트의 평문과 m비트의 키를 검증할 때 경우의 수는 $2^n \times 2^m = 2^{(n+m)}$ 이지만, 해당 모듈을 사용할 경우 $2^{(n\times4)} \times 2^{(m\times4)} = 2^{4(n+m)}$ 만큼의 수행이 필요하기 때문에 수정이 필요하다.

```
void FE(UNCHAR output[16], const UNC
void FO(UNCHAR output[16], const UNC

void KeySchedule(
    UNCHAR Roundkey[13][16],
    const UNCHAR key[16]);

void ARIAEncryption(UNCHAR C[16], cc
void ARIADecryption(UNCHAR P[16], cc

#endif
void ReadFile(UNCHAR KEY[276][16], U
void MmtReadFile(UNCHAR KEY[10][16],
void WriteFile(const UNCHAR CT[276][
void MmtWriteFile(
    const UNCHAR CT[10][161],
    const UNCHAR TestCT[10][161],
    const char* FileName);
```

```
void FE(UNCHAR output[16], const UNC
void FO(UNCHAR output[16], const UNC

void KeySchedule(
    UNCHAR Roundkey[13][16],
    const UNCHAR key[16],
    const char type);

void ARIAEncryption(UNCHAR C[16], co
void ARIADecryption(UNCHAR P[16], co

#endif
void ReadFile(UNCHAR KEY[276][16], U
void MmtReadFile(UNCHAR KEY[10][16],
void WriteFile(const UNCHAR CT[276][
void MmtWriteFile(
    const UNCHAR CT[10][160],
    const UNCHAR TestCT[10][160],
    const char* FileName);
```

Fig. 8. Weakness in hi-hee crypto module

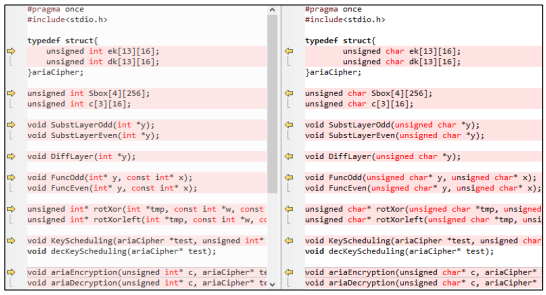

Fig. 9. Weakness in minjeongJho crypto module

LEA 암호 모듈의 경우 KISA LEA모듈에서 for 문에 불필요한 초기화 구문이 존재하여 컴파일 시 경고 메시지가 출력되었지만, 정상적으로 검증되었다. 또한 Github에서 제공하는 odzhan 사용자의 LEA 코드는 레퍼런스 코드와 거의 유사하게 작성되어 정상적으로 검증되었다.

V. 결 론

기존 암호 모듈의 문제점들을 해결하기 위해 본 논문에서는 Cryptol을 사용하여 암호 설계자가 이해할 수 있는 수식과 같은 표현으로 작성된 참조 구현 모델을 동일한 알고리즘에 적용할 수 있도록 제시하였고, 구현 모델을 작성하기 위한 래핑 코드들과 SAW 스크립트를 통해 동치성 검사를 수행할 수 있도록 가이드를 제공하였다. 연구에 대한 모든 자료는 Github의 저장소[11]에서 확인할 수 있다.

향후에는 암호 기능만을 검증하는 동치성 검사뿐만 아니라 버퍼 오버플로우와 같은 보안 취약점을 탐지하는 안전 검사(Safety Checking)에 대한 연구를 진행하고, 64비트 블록 암호 알고리즘인 HIGHT와 메시지 인증, 사용자 인증, 전자서명 등 다양한 분야에 활용 가능한 해시 함수인 LSH와 같은 알고리즘으로 확대하여, 국내 암호 모듈 개발 시 자체적으로 검증할 수 있도록 참조 구현 모델과 가이드를 배포하고 국내 암호 모듈의 구현상 취약점을 최소화 할 수 있는 방안을 제시할 것이다.

References

[1] Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicholas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson, "The Matter of Heartbleed," Proceedings of the 2014 ACM Internet Measurement Conference, pp. 475-488, Nov. 2014.

[2] Kyeoung-Ju Ha, Chang-Ho Seo, and Dae-Youb Kim, "Design of Validation System for a Crypto-Algorithm Implementation," The Journal of Korean Institute of Communications and Information Sciences, 39B(4), pp. 242-250, Apr. 2014.

[3] P. Caspi, D. Pilaud, N. Halbwachs, and J.A. Plaice, "LUSTRE: A declarative language for programming synchronous systems," Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pp. 178-188, Jan. 1987.

[4] Paul Caspi, Adrian Curic, Aude Maignan, Christos Sofronis, Stavros Tripakis, and Peter Niebert, "From Simulink to SCADE/Lustre to TTA: a Layered Approach for Distributed Embedded Applications," Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems, pp. 153-162, June 2003.

[5] Jeffrey R. Lewis and Brad Martin, "Cryptol: high assurance, retargetable crypto development and validation," IEEE Military Communications Conference, pp. 820-825, May 2004.

[6] Levent Erkok and John Matthews, "Pragmatic Equivalence and Safety Checking in Cryptol," Proceedings of the 3rd ACM workshop on Programming languages meets program verification, pp. 73-82, Jan. 2009.

[7] Levent Erkok, Magnus Carlsson, and Adam Wick, "Hardware/Software Co-verification of Cryptographic Algo-

rithms using Cryptol," IEEE Formal Methods in Computer-Aided Design, pp. 188-191, Dec. 2009.
[8] Levent Erkok and John Matthews, "High assurance programming in Cryptol," Proceedings of the 5th ACM Annual Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies, pp. 60, Apr. 2009.
[9] Lee Pike, "Hints for High-Assurance Cyber-Physical System Design," IEEE Cybersecurity Development (SecDev), pp. 25-29, Nov. 2016.
[10] Kyle Carter, Adam Foltzer, Joe Hendrix, Brian Huffman, and Aaron Tomb, "SAW: The Software Analysis Workbench," Proceedings of the 2013 ACM SIGAda annual conference on High integrity language technology, pp. 15-18, Nov. 2013.
[11] Wonbin Choi, "Github Repository," https://github.com/bindon/Cryptol, Dec. 2017.

〈저 자 소 개〉

최 원 빈 (Won-bin Choi) 학생회원
2013년 8월: 한신대학교 컴퓨터공학부 공학사
2013년 7월~2016년 8월: 이니텍(주) 융합보안연구소 대리
2016년 9월~현재: 고려대학교 정보보호대학원 석사과정
〈관심분야〉 취약점 분석, 보안성 분석·평가, 시스템 보안, 소프트웨어 보안

김 승 주 (Seungjoo Kim) 종신회원
1994년~1999년: 성균관대학교 정보공학과(학사, 석사, 박사)
1998년~2004년: 한국인터넷진흥원(KISA) 팀장
2004년~2011년: 성균관대학교 정보통신공학부 부교수
2011년~현재: 고려대학교 사이버국방학과/정보보호대학원 정교수
2017년~현재: 고려대학교 사이버무기시험평가연구센터(CW-TEC) 부센터장
2004년~현재: 한국정보보호학회 이사
2007년: 국가정보원장 국가사이버안전업무 유공자 표창
2010년: 방송통신위원회 정보통신망 침해사고 민관합동조사단 위원
2011년~현재: (사)화이트해커연합 HARU 및 국제해킹대회 SECUINSIDE 설립자 및 이사
2012년: 선관위 디도스 특별검사팀 자문위원
2014년~2015년: 육군사관학교 초빙교수
2014년~2016년: 다음카카오 프라이버시 정책 자문위원회 위원
2015년~현재: 방위사업청 방산기술보호 자문관
2016년~2018년: 개인정보분쟁조정위원회 위원
2016년~현재: 산업통상자원부 전략물자기술 자문위원
2016년~현재: 한국카카오뱅크 정보보호부문 자문교수
2017년~현재: 국방보안연구소 정보보호분야 자문위원
2017년~현재: 여신금융협회 신용카드 단말기 시험 인증위원회 위원
〈관심분야〉 보안공학 및 SDL, 위협 리스크 모델링, 보안성 평가/인증, 암호학, Usable Security