Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.30, NO.6, Dec. 2020 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2020.30.6.1225

차분 퍼징을 이용한 국내 공개 암호소스코드 안전성 검증*

윤 형 준,[1†] 서 석 충[2‡]
[1,2]국민대학교 (학생, 교수)

Security Verification of Korean Open Crypto Source Codes with Differential Fuzzing Analysis Method*

Hyung Joon Yoon,[1†] Seog Chung Seo[2‡]
[1,2]Kookmin University (Undergraduate student, Professor)

요 약

퍼징은 예상되는 범위를 벗어난 입력값을 무작위로 생성해 소프트웨어를 동적으로 테스팅 하는 방법으로, 취약점 분석을 자동화하기 위해 주로 쓰인다. 현재 한국인터넷진흥원에서는 국내 표준 암호 알고리즘들에 대한 소스코드를 공개하고 있으며, 많은 암호모듈 개발업체들이 이 소스코드를 이용하여 암호모듈을 개발하고 있다. 만약 이러한 공개 소스코드에 취약점이 존재할 경우, 이를 참고한 암호 라이브러리는 잠재적 취약점을 가지게 되어 향후 막대한 손실을 초래하는 보안 사고로 이어질 수 있다. 이에 본 연구에서는 SEED, HIGHT, ARIA와 같은 블록암호 소스코드의 안전성을 검증하기 위한 적절한 보안 정책을 수립하였고, 차분 퍼징을 이용해 안전성을 검증하였다. 최종적으로 스택 버퍼 오버플로우와 널 포인터 역참조를 포함하는 메모리 버그 항목과 오류 처리 항목에서 총 45개의 취약점을 발견하였으며, 이를 해결할 수 있는 취약점 개선 방안을 제시한다.

ABSTRACT

Fuzzing is an automated software testing methodology that dynamically tests the security of software by inputting randomly generated input values outside of the expected range. KISA is releasing open source for standard cryptographic algorithms, and many crypto module developers are developing crypto modules using this source code. If there is a vulnerability in the open source code, the cryptographic library referring to it has a potential vulnerability, which may lead to a security accident that causes enormous losses in the future. Therefore, in this study, an appropriate security policy was established to verify the safety of block cipher source codes such as SEED, HIGHT, and ARIA, and the safety was verified using differential fuzzing. Finally, a total of 45 vulnerabilities were found in the memory bug items and error handling items, and a vulnerability improvement plan to solve them is proposed.

Keywords: Software Testing, Differential Fuzzing, Cryptographic Library, Vulnerability

I. 서 론

퍼징(fuzzing)은 무작위 입력값을 생성해 프로그램에 반복적으로 대입하는 동적 테스팅 방법이다. 퍼징이 처음 고안된 1980년대 후반에는 랜덤 문자열을 테스트 대상 프로그램의 입력으로 사용해 버그를

Received(10. 12. 2020), Modified(11. 05. 2020), Accepted(11. 05. 2020)
* 이 논문은 2019년도 정부의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(No.2019R1F1A1058494)
† 주저자, topcue1019@kookmin.ac.kr
‡ 교신저자, scseo@kookmin.ac.kr(Corresponding author)

탐지했다. 처음에는 무작위 입력에 의해 버그가 우연히 발견되길 기대하는 형태의 분석 방식이었다. 이후 2010년대에 퍼징을 수행하면서 획득하는 정보를 이용해 더 효율적인 입력값을 생성하는 기법이 제안되었다[1]. 덕분에 효율적으로 버그를 찾는 방법론에 관한 연구들과 함께 퍼징 기법에 대한 연구가 폭발적으로 증가했다.

하지만 퍼징의 개념과 용어들이 정립되지 않은 채로 퍼징 연구가 급격히 증가했기 때문에 부작용이 나타나기도 했다. 퍼저마다 사용하는 용어가 다르고 적용하는 개념이나 매뉴얼이 모호한 경우도 있었다. 그래서 퍼징 분야에서 사용하는 용어들을 정의하고 퍼저의 체계적인 모델을 제시하는 연구[2]가 있었다.

현재의 퍼저들은 다양한 기준으로 분류할 수 있는데, 크게는 테스트 대상 프로그램의 내부구조를 아는지에 따라 분류한다. 대상 프로그램의 입력과 출력만 알 수 있는 경우 블랙박스 퍼저, 프로그램의 내부구조와 프로그램 실행 중 발생하는 정보를 알 수 있는 경우 화이트박스 퍼저, 제한적인 정보만을 알 수 있는 경우 그레이박스 퍼저로 분류한다. 퍼저의 입력을 생성하는 전략에 따라 변이 기반 퍼징과 생성 기반 퍼징으로 구분할 수 있다. 변이 기반 퍼징은 주어진 시드를 기반으로 변이를 주면서 새로운 입력값을 생성하는 퍼징 방법이다. 생성 기반 퍼징은 대상 프로그램이 처리할 수 있는 입력의 모델을 정의하고, 이 모델을 기반으로 입력값을 생성하는 퍼징 방법이다.

퍼저의 테스트 대상은 다양하며, 암호 라이브러리가 그 대상이 되기도 한다. 암호 라이브러리에 취약점이 존재하는 경우 잠재적으로 큰 보안 위협이 된다. 2014년에 OpenSSL[3]의 취약점인 하트 블리드[4]가 발견되면서 일반 IT업계뿐만 아니라 국내외 금융업계에도 큰 위협이 되었다. 그 이후에도 CVE-2017-3735, CVE-2017-3738, CVE-2018-0739와 같은 OpenSSL 취약점이 계속 발견되고, 그중 일부는 퍼저를 통해 발견되었다[5].

한국인터넷진흥원(KISA)에서 제공하는 공개 암호 알고리즘 소스코드를 활용하여 국가 및 공공기관에서 사용되는 국가용 암호모듈이 개발되고 있는 실정이다. 따라서, 공개된 소스코드에 취약점이 존재하는 경우 이를 기반으로 개발된 암호 모듈을 통해 보안 사고로 이어질 가능성이 존재한다. 특히 KCMVP (Korea Cryptographic Module Validation Program) 인증을 받은 암호 모듈이 취약한 공개 암호 알고리즘 소스코드를 기반으로 개발되었다면, 이는 잠재적으로 심각한 보안 사고를 야기할 수 있다.

본 연구의 대상은 상용 암호 라이브러리가 아닌, 암호 알고리즘을 사용하거나 암호 라이브러리를 개발할 때 참고하도록 공개한 소스코드다. 소스코드에 존재하는 취약점과 그로 인해 발생 가능한 많은 문제들이 사용자 혹은 개발자에게 전가되고 있다.

안전성 검증 대상이 암호 알고리즘 소스코드임을 고려하여 적절한 보안 정책을 수립하였고, 차분 퍼징 방법을 이용해 보안 정책을 위반하는 취약점이 존재하는지 분석했다. 그 결과 보안 정책을 위반하는 취약점 총 45개를 발견하였으며, 발견한 취약점에 대하여 취약점 개선 방안을 제시한다. 본 논문 기여할 수 있는 바는 다음과 같다.

- 바이너리 동적 분석 방법 중 하나인 퍼징이 암호학 분야에서도 활용될 수 있도록 퍼징의 대상을 확장하였다.
- 암호 알고리즘 소스코드나 암호 라이브러리에서 발생 가능한 취약점을 식별하여 적절한 보안 정책을 제시하였다.
- 차분 퍼징을 적용하여 한국인터넷진흥원에서 배포하는 국내 표준 블록암호 알고리즘 소스코드에서 다수의 취약점을 발견하였다.

본 논문의 구성은 다음과 같다. 2장에서는 관련 연구를 소개한다. 3장에서는 암호 알고리즘 소스코드 퍼징을 위한 보안 정책을 제시하고, 본 연구에서 적용한 차분 퍼징 방법과 발견한 취약점을 설명한다. 4장에서는 발견한 취약점에 대한 개성 방안을 제시하며, 5장에서는 본 논문의 결론을 짓는다.

II. 관련연구

2.1 퍼징 테스트

퍼즈 테스팅은 전처리(preprocess), 스케줄(schedule), 입력 생성(input generation), 입력 평가(input evaluation), 설정 업데이트(configuration update) 다섯 단계로 구분할 수 있다. 이때 전처리를 제외한 나머지 네 단계를 반복하는데, 이 반복을 퍼즈 반복(fuzz iteration)이라고 한다. 먼저 전처리 단계에서 사용자가 설정한 환경설정값에 따라 퍼즈 설정 정보를 수정한다. 스케줄

단계에서는 퍼즈 반복을 위한 적절한 퍼즈 설정 정보를 지정한다. 입력 생성 단계에서는 테스트 케이스(test case)라고 불리는 대상 프로그램의 입력을 생성한다. 입력 평가 단계에서는 테스트 케이스를 이용해 대상 프로그램을 실행하고 보안 정책을 위반하는지 버그 오라클을 이용해 평가한다. 설정 업데이트 단계에서는 퍼즈 설정 정보와 퍼즈 반복에서 획득한 정보를 이용해 퍼즈 설정 정보를 업데이트한다.

AFL[6]은 대표적인 그레이박스 퍼저로서, 커버리지 기반 퍼징 도구다. AFL은 컴파일 시 특정한 코드를 삽입해 코드 커버리지를 측정하면서 효율적으로 크래시(crash)를 찾는다. 특히 포크 서버(fork server)를 이용한 인메모리(in-memory) 퍼징 기법으로 성능을 극대화했는데, 프로그램이 시작될 때 수행하는 링커 및 라이브러리 초기화와 같은 루틴을 한 번만 수행하고 자식 프로세스를 포크 한다. 이후 포크 서버는 목표 기능에 대해서만 퍼징을 수행하는 방법을 이용해 퍼저를 최적화했다.

LibFuzzer[7]는 그레이박스 퍼저로, 시드를 기반으로 변이한 입력을 테스트 케이스로 사용하는 변이 기반 퍼징 도구다. LibFuzzer는 라이브러리를 퍼징하기 위해 만들어졌으며, 특정한 인터페이스를 통해 퍼징코드를 작성할 수 있다. 또한 AFL처럼 인메모리 퍼징 기법을 적용하는데, 대상 프로그램의 메모리를 스냅샷(snapshot) 해두고 퍼즈 반복을 수행한 뒤, 다시 스냅샷을 복원하는 방식으로 구현되어 있다.

OSS-Fuzz[8]는 구글이 제안한 퍼징 프레임워크로서 크롬의 퍼징 테스트 환경을 다른 오픈소스 프로젝트들에 공유하기 위해 공개되었다. 오픈소스를 OSS-FUZZ에 추가하면 구글의 퍼징을 위한 분산 환경인 clusterfuzz 형태로 배포된다. 현재 OSS-FUZZ는 300개 이상의 오픈소스를 테스트하고 있으며, OpenSSL과 같은 암호 모듈과 pycryptodome과 같은 파이썬 저수준 암호 프리미티브 패키지도 포함되어있다.

그 외에도 지금까지 VUzzer[9], Angora[10], Honggfuzz[11]와 같은 퍼저들이 연구되었고, 다수의 취약점을 발견하면서 그 성능을 입증해오고 있다.

2.2 암호 라이브러리 취약점 분석

암호 라이브러리의 취약점을 분석하기 위한 여러 연구가 진행되어왔다. Wycheproof[12]는 제3기관의 소프트웨어 암호 라이브러리들을 대상으로 안전성을 검증하기 위해 구글 보안 팀에서 개발한 자동화 도구다. 알려진 공격을 탐지하기 위해 암호학적 이론이나 역사적 보안 약점을 기반으로 유닛 테스트를 수행한다. RSA, 타원곡선 암호, 인증기술 등을 포함한 여러 암호 기술을 지원한다.

차분 퍼징을 적용해 암호 라이브러리의 취약점을 분석한 연구도 있다. 차분 퍼징이란 같은 기능을 제공하지만 다른 방식으로 구현된 두 프로그램을 동일한 입력으로 실행한 뒤 출력을 비교하는 소프트웨어 테스트 방식인 차분 테스팅[13]에서 기인한 방식이다. 차분 퍼징은 같은 암호 기술을 지원하는 서로 다른 암호 라이브러리가 같은 기능을 수행하는지, 혹은 특정 평문을 암호화하고 이어서 복호화한 뒤 복호화한 데이터가 처음의 평문과 일치하는지를 비교하는 등의 전략을 이용한다.

DIFFUZZ[14]는 부채널 취약점을 찾기 위해 차분 퍼징을 적용했다. 리소스 기반 휴리스틱으로 리소스 소비 차이를 최대화하는 테스트 케이스를 생성하여 취약점을 찾는다. CDF(Crypto Differential Fuzzing)[15]는 BlackHat에서 발표된 블랙박스 퍼저다. 동일한 입력에 대해 출력의 차분을 비교하는 차분 퍼징을 전략을 이용한다. CryptoFuzz[16]는 암호 라이브러리를 퍼징하기 위해 개발된 도구로, 차분 퍼징을 적용해 메모리 오염 취약점뿐만 아니라 내부 일관성(internal consistency)이나 다중 라이브러리 차분(multi-library differential) 또한 탐지할 수 있다.

2.3 KISA 공개 암호 알고리즘 소스코드

한국인터넷진흥원은 정보보호의 기반 암호기술 및 정책을 연구, 개발하고 다양한 IT 서비스에 적용하기 위한 기술적, 제도적 방안을 마련함으로써 안전한 IT 환경 조성에 기여하기 위해 암호이용활성화 페이지를 운영하고 있다[17]. 암호 이용 활성화를 위해 암호 알고리즘 소스코드를 배포하고 있는데, 국내 표준 블록암호, 해시함수, 전자서명뿐만 아니라 패스워드 안전성 검증 라이브러리나 모바일 환경에 적합한 암호 알고리즘 소스코드도 제공한다.

Table 1. 은 KISA가 배포하는 블록암호의 종류와 운영모드, 지원하는 프로그래밍 언어를 표로 작성한 것이고, Table 2.는 해시함수, Table 3.은 전자서명의 종류와 지원 프로그래밍 언어를 표로 작성한

Table 1. Block cipher algorithm source code provided by KISA

	Mode of operation	Language
SEED	ECB, CBC, CTR, CCM, GCM, CMAC	C/C++, Java, ASP, JSP, PHP
HIGHT	ECB, CBC, CTR, CMAC	C/C++, Java
ARIA	ECB	C/C++, Java
LEA	ECB, CBC, CTR, CFB, OFB, CCM, GCM, CMAC	C, Java, Python

Table 2. Hash functions algorithm source code provided by KISA

	Detail	Language
LSH	LSH-224, LSH-256, LSH-512-224, LSH-512-256, LSH-384, LSH-512, HMAC(LSH)	C, Java, Python
SHA-256	SHA-256	C/C++, Java, ASP, JSP, PHP
SHA-3	SHA3-224, SHA3-256, SHA3-384, SHA3-512, SHAKE128, SHAKE256	C/C++, Java, ASP, JSP, PHP

Table 3. Digital signatures algorithm source code provided by KISA

	Language
KCDSA	C/C++, Java
EC-KCDSA	C/C++, Java

것이다. 이와 같은 다양한 암호알고리즘이 KISA의 관리하에 오픈소스로 제공되고 있다. 예를 들어, SEED 블록암호는 2009년을 기준으로 nCipher, RSA Security 등의 정보보호업체를 포함해 약 3000개 이상의 국내외 기업 및 학교에 배포되었다. 이렇게 다양한 곳에 배포되고, 일반 사용자들도 쉽게 접근해 제공받을 수 있는 오픈소스 암호 알고리즘 코드의 안전성 검증은 필수적이다.

금융보안원은 하트블리드 같은 해킹 사고가 결국 오픈소스 소프트웨어 관리의 소홀함 때문이라고 말하면서, 이를 해결하기 위해 오픈소스 소프트웨어 보안 관리 방안 관련 규격을 제공하기도 했다[18]. 하지만 KISA의 암호 알고리즘 소스코드는 지속적인 관리가 이루어지고 있다고 판단하기 힘들다. 예를 들어 SEED 암호는 2018년 이후에 CCM, GCM, CMAC 운영모드가 추가되었지만, 다른 운영모드는 2013년 이후로 소스코드가 갱신되고 있지 않다.

III. 차분 퍼징을 이용한 취약점 검증

3.1 타겟 알고리즘

KISA에서 국내 표준 블록암호인 SEED, HIGHT, ARIA의 소스코드를 제공하고 있다. SEED는 ECB, CBC, CTR, CCM, GCM, CMAC 운영모드를 구현해 소스코드로 제공한다. HIGHT는 ECB, CBC, CTR, CMAC 운영모드를 구현해 소스코드로 제공한다. ARIA는 ECB를 제외한 다른 운영모드를 제공하지 않는 대신, ARIA ECB 모드의 32비트 최적화 구현 소스코드를 추가적으로 제공한다.

SEED는 1999년 한국인터넷진흥원과 국내 암호 전문가들이 개발한 Feistel 구조의 블록암호 알고리즘이다. 1999년에 정보통신단체표준(TTA)으로 제정되었으며, 2005년에 ISO/IEC 국제 블록암호 알고리즘 IETF 표준으로 제정되었다.

HIGHT는 저전력·경량 환경에서 용이하게 사용하기 위해 2005년에 KISA, ETRI 부설연구소, 고려대학교가 공동으로 개발한 블록암호 알고리즘이다. 2006년에 정보통신단체표준으로 제정되었으며, 2010년에 ISO/IEC 국제 블록암호 IETF 표준으로 제정되었다.

ARIA는 경량 환경 및 하드웨어 구현에 최적화하기 위해 개발된 Involutional SPN 구조의 블록암호 알고리즘이다. ARIA는 개발에 참여한 학계(Academy), 연구소(Research Institute), 정부기관(Agency)의 첫 글자를 따서 만들어졌다. 2004년에 지식경제부에 의해 국가 표준(KS)으로 제정되었으며, 2010년에 국제 표준으로 제정되었다.

각 블록암호 소스코드마다 제공되는 운영모드도 다르고, 내부 함수 구현 방식에 통일성이 없어서 공통된 퍼징 방법론을 적용하는데 어려움이 있었다. 따라서 각 블록암호 소스코드의 특징에 맞게 퍼저를 구

현하고 한 인터페이스로 통합하는 과정이 필요하다.

본 연구에서 SEED의 CBC, CTR, CCM 모드와 HIGHT의 CBC, CTR 모드, ARIA ECB 모드와 ARIA ECB 32-bit 최적화 구현 소스코드를 퍼징의 대상으로 하였으며, C/C++로 구현된 코드를 퍼징 대상으로 한다. KISA에서 대상 암호 알고리즘과 함께 빌드 방법을 제시하는 경우 해당 방법을 따랐으며, 빌드 방법을 제시하지 않는 경우 다른 블록암호와 유사하게 빌드하였다.

3.2 보안 정책

퍼즈 테스팅은 퍼징 대상 프로그램이 특정 보안 정책을 위반하는지 점검하는 것이다. 예를 들어 초기 퍼저의 보안 정책은 테스트 케이스가 대상 프로그램에 크래시를 발생시키는지였다. 하지만 최근 연구되는 여러 퍼저는 대상이나 사용자에 따라 보안 정책이 다양하다. 본 연구 역시 대상 프로그램이 암호 알고리즘 소스코드이므로 그에 맞는 적절한 보안 정책이 필요하다. Table 4. 는 KISA의 암호 알고리즘 소스코드 퍼징을 위해 수립한 보안 정책이다. 대부분의 항목은 한국인터넷진흥원에서 제공하는 “전자정부 SW 개발·운영자를 위한 소프트웨어 개발보안 가이드[19]”와 “공개용 소스코드 보안약점 분석도구 개발 동향[20]”을 기반으로 한다. 해당 자료에서는 구현단계 소프트웨어 보안 약점을 입력데이터 검증 및 표현, 보안기능, 시간 및 상태, 에러처리, 코드오류 총 다섯 가지 항목으로 분류했으며, 총 30가지의 보안 약점을 식별했다.

Table 4. Security policy for KISA cryptographic algorithm source code

Security Policy		
Classification	No	Vulnerability
Memory Bugs Testing	1.1	Stack Buffer Overflow
	1.2	Heap Buffer Overflow
	1.3	Use-After-Free
	1.4	Double-Free
	1.5	Memory Leak
	1.6	Null Pointer Dereference
Error Handling Testing	2.1	Generation of Error Message Containing Sensitive Information
	2.2	Lack of Handling to Error Conditions
	2.3	Improper Exception Handling
Differential Testing	3.1	Implementation Discrepancies

하지만 이 보안 정책은 본 연구의 퍼징 대상인 암호 소스코드에 그대로 적용하기에는 부적절하다. 예를 들어 입력 데이터 검증 및 표현 항목 중 하나인 SQL 삽입 취약점은 암호 소스코드에 존재할 수 없는 취약점이다. 따라서 한국인터넷진흥원에서 식별한 30개의 항목 중 암호 소스코드에서 발생 가능한 취약점들을 분류해 본 연구의 보안 정책에 포함했다.

또한 OpenSSL이나 mbedTLS[21]와 같은 해외 암호 모듈을 퍼징하는 CryptoFuzz가 발견한 취약점을 토대로, 암호 소스코드에서 발생 가능한 취약점을 추가로 식별했다.

CryptoFuzz는 사용자가 부적절한 입력값을 함수에 전달하는지도 검증한다. 그에 비해 KISA의 암호알고리즘 소스코드는 부적절한 입력값에 대한 검증을 거의 하지 않는다. 물론 암호기술 명세를 따르지 않는 부적절한 입력값을 사용하는 것은 사용자의 잘못으로 치부할 수도 있다. 하지만 국내외 상용 암호 라이브러리는 부적절한 입력값을 라이브러리 내부에서 검증하고 있으며, 만약 이러한 검증이 생략된 경우 취약점으로 판단해 적절한 패치를 적용하고 있다. 따라서 입력 데이터의 적절한 검증 여부 또한 보안 정책에 포함했다.

3.3 차분 퍼징 방법론

Fig. 1. 은 퍼저에 적용한 차분 퍼징 기법을 나타낸 그림이다. 평문과 암호키는 퍼저가 생성한 입력 데이터이며, 이 테스트 케이스를 이용해 암호화와 복호화를 연달아 진행한다. 복호화한 데이터가 처음 암호화 전 평문과 일치하는지 비교하는 방식의 차분 퍼징 전략이다. 이 차분 퍼징 방식은 CDF의 전략과 유사한 방식으로, CVE-2019-1543[22]을 발견하기도 했다.

Fig. 2. 는 구현한 차분 퍼저를 시각화하여 그림으로 나타낸 것이다. 오른쪽 Entry에서 시작하며

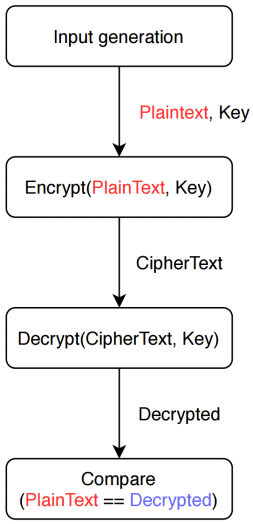

Fig. 1. Differential fuzzing method

전처리, 스케줄, 입력 생성, 설정 업데이트는 기반 퍼징 엔진인 LLVM의 LibFuzzer의 동작 과정을 따른다. 퍼징의 대상인 암호 알고리즘 소스코드는 함수 형태로 제공되므로, LibFuzzer를 기반 퍼징 엔진으로 사용하는 것이 적합하다고 판단했다.

퍼즈 타겟에서 퍼징하려는 함수를 호출해 랜덤한 데이터를 전달할 수 있으며, Executor가 특정 암호 알고리즘을 선택해 퍼징한다. Executor는 퍼징의 대상인 블록암호에 따라 암호화할 평문과 암호키를 전달하며, 필요한 경우 nonce, 초기화 벡터 등을 추가로 전달한다.

메모리 버그 테스팅과 오류 처리 테스팅은 LibFuzzer를 기반으로 검증할 수 있으며, 특히 메모리 관련 버그는 Address Sanitizer[23]를 이용하여 효율적으로 감지할 수 있다. Address Sanitizer는 C/C++로 작성된 프로그램의 메모리 오류를 감지하는 도구로서, 스택 버퍼 오버플로우, 힙 버퍼 오버플로우, 자원 해제 후 재사용, 메모리 누수 등의 버그를 감지한다. 이러한 차분 퍼징 방법을 이용해 3.2장의 보안 정책을 준수하는지를 평가할 수 있다.

3.4 취약점 분석 결과

Fig. 3.은 시간에 따른 코드 커버리지 그래프를 나타낸 그림이다. 대상 프로그램은 암호 알고리즘 소스코드로, 일반적인 응용 프로그램과 비교했을 때 전체 코드 크기가 매우 작다. 덕분에 6시간 만에 퍼저가 코드 전체를 커버할 수 있었다. Address Sanitizer가 프로그램을 중단하거나, 구현 불일치(implementation discrepancies)로 인하여 abort() 함수가 호출되었을 때마다 차후 정확한 취약점 분석을 위해 테스트 케이스를 저장했다.

Table 5. 는 3.2에서 수립한 보안 정책을 기준으로 암호알고리즘 소스코드들에서 발견한 취약점을 표로 나타낸 것이다. 총 45개의 취약점을 발견하였으며, SEED에서 25개, HIGHT에서 14개, ARIA에

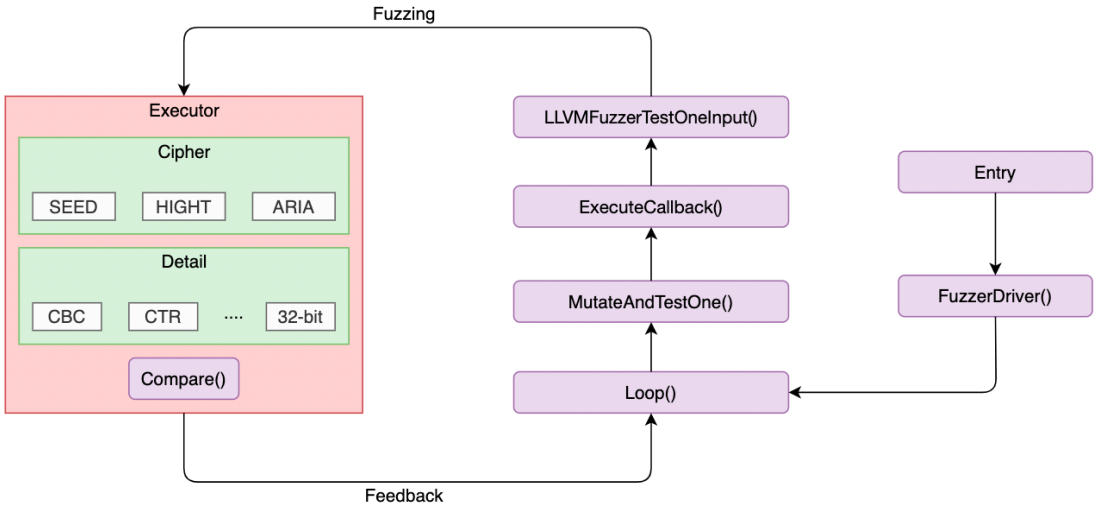

Fig. 2. Structure of fuzzer with differential fuzzing method

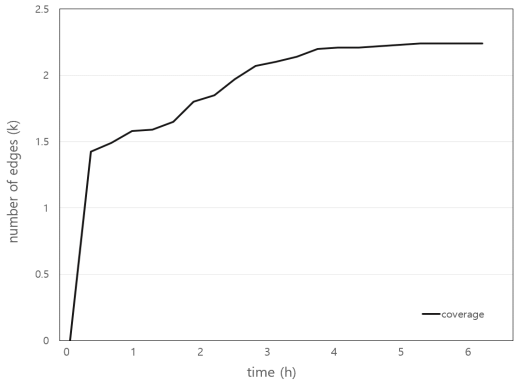

Fig. 3. Code coverage graph

서 6개를 발견했다. 발견한 취약점 중 버퍼 오버플로우나 메모리 누수, 널 포인터 역참조와 같은 메모리 관련 버그가 약 84%로 가장 많았다. 오류 처리 관련 버그는 7개가 발견되었고, 구현 불일치는 발견되지 않았다.

취약점 분석 중 근본적인 원인 A로 인해 취약점 B가 발생하는 경우, A만을 취약점으로 분류했다. 예를 들어 SEED 블록암호 CBC 운영모드의 SEED_CBC_init() 함수에서 pbszUserKey 변수에 대한 널 포인터 역참조를 탐지한 경우, 0을 반환

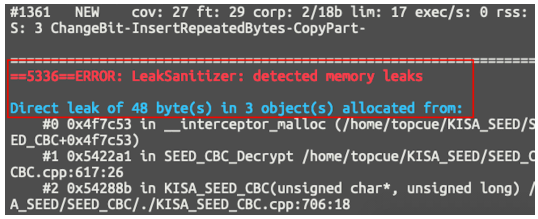

Fig. 4. Memory leak detected by fuzzer

하면서 함수를 중단하는 오류 처리 코드가 있다. 하지만 이후 오류 상황 대응 부재로 인해 코드가 계속 진행되고, 결과적으로 SEED_CBC_Decrypt() 함수에서 메모리 누수 취약점이 발생했다. 이 경우 메모리 누수의 근본적인 원인인 오류 상황 대응 부재를 취약점으로 분류했다.

Fig. 4. 는 해당 취약점을 발생시키는 테스트 케이스를 이용해 크래시를 재현한 사진이다. 근본적인 원인은 오류 상황 대응 부재이지만, 퍼저의 Address Sanitizer는 메모리 누수를 감지했다고 화면에 출력한다. 이렇게 퍼저가 발견한 취약점과 근본적인 취약점 원인이 다른 경우가 있었다. 따라서 정확한 취약점 원인을 분류하기 위해 퍼저가 크래시를 발생시킬 때마다 저장한 모든 테스트 케이스를 분류하고 분석했다.

Table 5. Vulnerabilities that violate security policy found by fuzzing

Security Policy			Target Algorithm						
Classification	No	Vulnerability	SEED			HIGHT		ARIA	
			CBC	CTR	CCM	CBC	CTR	8-bit	32-bit
Memory Bugs Testing	1.1	Stack Buffer Overflow	-	1	-	-	-	1	1
	1.2	Heap Buffer Overflow	3	2	1	-	3	-	-
	1.3	Use-After-Free	-	-	-	-	-	-	-
	1.4	Double-Free	-	-	-	-	-	-	-
	1.5	Memory Leak	3	2	-	2	2	-	-
	1.6	Null Pointer Dereference	3	3	4	3	-	2	2
Error Handling Testing	2.1	Generation of Error Message Containing Sensitive Information	-	-	-	-	-	-	-
	2.2	Lack of Handling to Error Conditions	1	2	-	3	1	-	-
	2.3	Improper Exception Handling	-	-	-	-	-	-	-
Differential Testing	3.1	Implementation Discrepancies	-	-	-	-	-	-	-

IV. 취약점 개선 방안

발견된 취약점은 스택 버퍼 오버플로우, 힙 버퍼 오버플로우, 메모리 누수, 널 포인터 역참조 그리고 오류 상황 대응 부재 취약점이다.

스택 버퍼 오버플로우 취약점은 RTL[24]이나 ROP[25]와 같은 공격에 악용될 수 있다. 예를 들어 CVE-2015-7547은 glibc에 존재하는 스택 버퍼 오버플로우를 이용해 원격 코드 실행 공격이 가능한 취약점이다[26]. 본 연구에서 발견한 스택 버퍼 오버플로우 취약점은 입력값에 대한 검증 없이 함수를 호출하거나 매크로를 사용했기 때문에 발생하였다. ARIA 소스코드의 EncKeySetup() 함수에서 사용자로부터 입력받는 KeyBits 변수의 크기를 검증하지 않은 채로 (KeyBits - 128)을 수행하여, 이후 버퍼 오버플로우 취약점이 발생한다. 스택 버퍼 오버플로우 취약점을 이용한 공격은 Canary나 DEP, ASLR과 같은 메모리 보호 기법을 이용해 완화하고 있지만, 근본적인 문제를 해결하기 위해선 사용자의 입력값 검증을 통해 스택 오버플로우를 방지해야 한다[27][28].

힙 버퍼 오버플로우 취약점은 힙풍수(heap feng shui[29])와 같은 공격에 함께 사용되거나 함수 포인터 변조 공격에 악용될 수 있다. 예를 들어 CVE-2016-0778은 OpenSSL에서 발생한 힙 버퍼 오버플로우 취약점이다[30]. HIGHT 블록암호 CTR 운영모드의 HIGHT_CTR_Process() 함수에서 BLOCK_XOR_HIGHT() 매크로를 사용하는데, 인자의 크기를 고려하지 않아 힙 버퍼 오버플로우 취약점이 발생한다. 본 연구에서 발견한 힙 버퍼 오버플로우는 대부분 객체가 동적으로 할당받은 메모리의 크기를 고려하지 않고 메모리에 접근하거나 memcpy() 함수의 인자로 전달했기 때문에 발생했다. 따라서 객체를 이용하기 전에 할당받은 메모리의 크기를 초과하는지를 검사해야 한다.

메모리 누수 취약점은 ASLR과 같은 메모리 보호 기법에 대한 공격의 근간이 될 수 있는 취약점이다. 예를 들어 CVE-2015-3079나 CVE-2015-3044는 어도비 플래시 플레이어(adobe flash player)에서 발견된 취약점으로, 메모리 누수 취약점을 이용해 ASLR을 우회할 수 있는 취약점이다[31]. 본 연구에서 발견한 취약점들은 Valgrind[32]와 같은 도구로도 충분히 발견할 수 있다. Fig. 5. 는 Valgrind를 이용해 HIGHT 블록암호 CBC 운영

```
==6178== LEAK SUMMARY:
==6178==    definitely lost: 128 bytes in 1 blocks
==6178==    indirectly lost: 0 bytes in 0 blocks
==6178==      possibly lost: 0 bytes in 0 blocks
==6178==    still reachable: 0 bytes in 0 blocks
==6178==         suppressed: 0 bytes in 0 blocks
==6178==
==6178== For counts of detected and suppressed errors,
==6178== ERROR SUMMARY: 1 errors from 1 contexts (supp
```

Fig. 5. Memory leak detected by valgrind

모드 소스코드의 메모리 누수를 검사한 결과로, 메모리 누수가 감지되었음을 알 수 있다. 따라서 Valgrind와 같은 메모리 누수 탐지 전용 도구를 이용해 검사하고, 사용한 메모리를 잘 해제하여 메모리 누수 취약점을 방지해야 한다.

널 포인터 역참조는 널 포인터에 값을 대입할 때 발생하는 취약점으로, 공격자가 의도적으로 널 포인터 역참조를 발생시키는 경우 추후 공격에 악용될 수 있다. 예를 들어 CVE-2020-5183은 FTPGetter 프로그램에서 발생한 취약점으로, 널 포인터 역참조를 이용해 메모리 오염 공격에 악용할 수 있는 취약점이다[33]. 널 포인터 역참조는 객체를 사용하기 전, 객체가 Null 인지 검증함으로써 방지할 수 있다.

오류 상황 대응 부재 취약점은 공격자가 오류 상황을 악용하여 개발자가 의도하지 않은 방향으로 프로그램을 동작시킬 수 있는 취약점이다. 본 연구의 대상 프로그램에서는 널 포인터 역참조나 비정상적인 입력값에 대한 오류 상황을 인지하고 있으나, 후속 조치가 이루어지지 않아 다른 취약점으로 이어지는 경우가 있었다. 따라서 오류가 발생할 수 있는 부분에 대해 제어문을 사용하여 적절한 예외 처리가 필요하다.

V. 결 론

본 논문에서 KISA가 제공하는 공개 암호알고리즘 소스코드 중 블록암호에 대한 안전성을 검증했다. 바이너리 동적 분석 방법 중 하나인 퍼징을 이용했으며, 특히 대상 프로그램이 암호 알고리즘 소스코드임으로 고려하여 차분 퍼징 방법을 적용했다. 또한 퍼징 대상에 맞는 적절한 보안 정책을 수립하고, 이 보안 정책을 기준으로 취약점 여부를 판단하였다. 퍼징을 통해 다수의 취약점을 발견하였으며, 이를 해결하기 위한 취약점 개선 방안을 제시했다.

4장에서 전술하였듯이 발견한 취약점 대부분은 입력값 검증을 통해 해결할 수 있다. 대상 프로그램이

상용 암호 라이브러리가 아니라는 점을 감안하면, 암호 기술 명세를 따르지 않는 입력값을 검증할 필요가 없다고 생각할 수도 있다. 하지만 해킹은 이렇게 사소한 취약점을 악용해 이루어지며, 향후 막대한 피해를 수반하는 보안 사고로 이어진다. 특히 시스템의 보안 강도를 결정하는 중추 중 하나인 암호 기술의 안전성을 도리어 간과하는 것은 어불성설이다. 본 연구가 차분 퍼징 방법론에 대한 소개와 암호 소스코드 취약점 분석의 좋은 선행 연구가 되길 바란다.

본 연구에서는 새로운 차분 퍼징 기법을 제안하고 있지는 않다. 따라서 향후 연구에서는 새로운 차분 퍼징 기법을 적용하겠다. 또한 대상 프로그램에서 제외되었던 여러 운영모드들과 LEA 블록암호, 그리고 해시함수나 전사서명에 대한 안전성 검증을 수행하겠다.

References

[1] S. Embleton, S. Sparks, and R. Cunningham, "sidewinder: An evolutionary guidance system for malicious input crafting," in Proceedings of the Black Hat USA, 2006.

[2] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz and Maverick Woo, "The Art, Science, and Engineering of Fuzzing:A Survey," IEEE Transactions on Software Engineering, pp. 1-21, Oct. 2019.

[3] OpenSSL, "OpenSSL Cryptography and SSL/TLS Toolkit", https://www.openssl.org, Oct. 2020.

[4] HeartBleed, "HeartBleed CVE-2014-0160", https://heartbleed.com, Oct. 2020.

[5] OpenSSL, "OpenSSL Vulnerabilities", https://www.openssl.org/news/vulnerabilities.html, Oct. 2020.

[6] Github, "American Fuzzy Lop", https://github.com/google/afl, Oct. 2020.

[7] Github, "LibFuzzer", http://llvm.org/docs/LibFuzzer.html, Oct. 2020

[8] M. Aizatsky, K. Serebryany, O. Chang, A. Arya and M. Whittaker, "Announcing OSS-Fuzz: Continuous fuzzing for open source software," Google Testing Blog, 2016.

[9] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida and Herbert Bos, "VUzzer: Application-aware Evolutionary Fuzzing," NDSS, Vol. 17, pp. 1-14, Feb. 2017.

[10] Peng Chen and Hao Chen, "Angora: Efficient fuzzing by principled search," 2018 IEEE Symposium on Security and Privacy (SP), pp. 711-725, May. 2018

[11] Github, "Honggfuzz", https://github.com/google/honggfuzz, Oct. 2020.

[12] Github, "Wycheproof", https://github.com/google/wycheproof, Oct. 2020.

[13] Muhammad Ali Gulzar, Yongkang Zhu and Xiaofeng Han, "Perception and Practices of Differential Testing," 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), pp. 71-80, May. 2019.

[14] Shirin Nilizadeh, Yannic Noller and Corina S. Pasareanu, "DifFuzz: Differential Fuzzing for Side-Channel Analysis," ICSE '19: Proceedings of the 41st International Conference on Software Engineering, pp. 176-187, May. 2019.

[15] Jean-Philippe Aumasson and Yolan Romailler, "Automated Testing of Crypto Software Using Differential Fuzzing," Blackhat USA 2017, Jul. 2017.

[16] CryptoFuzz, "Differential fuzzing of cryptographic libraries", https://guidovranken.com/2019/05/14/differential-fuzzing-of-cryptographic-libraries/, Oct. 2020.

[17] KISA, "KISA Seed Algorithm", https://seed.kisa.or.kr/kisa/index.do, Oct. 2020

[18] FSEC, "Threats and countermeasures for using open source SW", https://www.fsec.or.kr/user/bbs/fsec/42/312/bbsDataView/539.do, Oct. 2020.

[19] KISA, "Software development security

guide for e-government software developers and operators", https://www.kisa.or.kr/public/laws/laws3_View.jsp?mode=view&p_No=259&b_No=259&d_No=50, Oct. 2020.

[20] KISA, "Development trend of security weakness analysis tool for public source code", https://www.kisa.or.kr/public/library/IS_View.jsp?mode=view&p_No=158&b_No=158&d_No=161, Oct. 2020.

[21] Github, "MbedTLS", https://github.com/ARMmbed/mbedtls, Oct. 2020.

[22] MITRE, "CVE-2019-1543", https://www.openssl.org/news/secadv/20190306.txt, Oct. 2020.

[23] Github, "Address Sanitzier", https://github.com/google/sanitizers/wiki/AddressSanitizer, Oct. 2020.

[24] David J Day, Zhengxu Zhao and Minhua Ma, "Detecting Return-to-libc Buffer Overflow Attacks Using Network Intrusion Detection Systems," 2010 Fourth International Conference on Digital Society (IEEE), pp. 127-177, Mar. 2010.

[25] Erik Buchanan, Ryan Roemer, Stefan Savage and Hovav Shacham, "Return-oriented Programming: Exploitation without Code Injection," in Proceedings of Black Hat USA 2008, Aug. 2008.

[26] MITRE, "CVE-2015-7547", https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-7547, Oct. 2020.

[27] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," SSYM'98: Proceedings of the 7th conference on USENIX Security Symposium vol.7, pp. 1-16, Jan. 1998.

[28] Jonathan Ganz and Sean Peisert, "ASLR: How Robust Is the Randomness?," 2017 IEEE Cybersecurity Development (SecDev), pp.34-41, Sep. 2017.

[29] Alexande Sotirov, "Heap Feng Shui in JavaScript," in Proceedings of Black Hat Europe 2007, Jul. 2007.

[30] MITRE, "CVE-2016-0778", https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-0778, Oct. 2020.

[31] MITRE, "CVE-2015-3079", https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3079, Oct. 2020.

[32] Valgrind, "Valgrind", https://valgrind.org, Oct. 2020.

[33] MITRE, "CVE-2020-5183", https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-5183, Oct. 2020.

Appendix A. Details of security policy

Security Policy			
Classification	No	Vulnerability	Detail
Memory Bugs Testing	1.1	Stack Buffer Overflow	Occurs when a program uses contiguous memory space tries to read or write a value to the allocated stack or to an address outside the usable range of a specific variable. It can cause a malfunction or be exploited in an attack such as RTL or ROP to execute malicious code. e.g. CVE-2015-7547
	1.2	Heap Buffer Overflow	Occurs when an object accesses outside the allocated memory without considering the size of the dynamically allocated memory. It can be used with attacks such as Heap Feng Shui, or it can be exploited in attacks that manipulate function pointers. e.g. CVE-2016-0778
	1.3	Use-After-Free	Occurs when an object that has been dynamically allocated memory is freed and then the object is accessed and used again. If an attacker can access a desired location on the heap and read or write memory, it can be exploited for arbitrary code execution attacks. e.g. CVE-2015-0311
	1.4	Double-Free	Occurs when free is performed twice, it is caused by the merge operation of unlink, an internal function of free function. It causes a heap buffer overflow and can be exploited for attacks such as function pointer manipulation. e.g. CVE-2020-25559
	1.5	Memory Leak	Occurs when the allocated memory is not freed, and a computer program continues to occupy unneeded memory. It can be the basis of attacks against memory protection techniques such as ASLR. e.g. CVE-2015-3079
	1.6	Null Pointer Dereference	Occurs when assigning a value to a null pointer. If an attacker intentionally generates a null pointer dereference, it can be exploited for attacks such as memory corruption. e.g. CVE-2020-5183
Error Handling Testing	2.1	Generation of Error Message Containing Sensitive Information	Occurs when important information system internal information is included in the error information due to insufficient or insufficient handling of errors. Important information may be exposed as it is or may help an attacker's malicious behavior.
	2.2	Lack of Handling to Error Conditions	Occurs when the part where an error may occur is checked, but exceptions are not handled for such error. An attacker can exploit an error condition to run a program in a direction not intended by the developer.
	2.3	Improper Exception Handling	Occurs when the function result is not properly processed or the condition for an exception condition is not properly checked during program execution. An attacker can exploit an error condition to run a program in a direction not intended by the developer.
Differential Testing	3.1	Implementation Discrepancies	Occurs when the implemented cipher algorithm differs from the actual specification. It can be exploited for backdoors or other cryptographic attacks. e.g. bugzilla 1575923

〈저 자 소 개〉

윤 형 준 (Hyung Joon Yoon) 학생회원
2018년 3월~현재: 국민대학교 정보보안암호수학과 학사과정
〈관심분야〉 소프트웨어 보안, 시스템 보안, 취약점 분석

서 석 충 (Seog Chung Seo) 정회원
2011년 8월: 고려대학교 정보보호학과 박사
2013년 11월: 삼성전자 종합기술원 전문연구원
2014년 4월: 삼성전자 DMC 연구소 책임연구원
2019년 2월: 국가보안기술연구소 선임연구원
2019년 3월~현재: 국민대학교 정보보안암호수학과 조교수
〈관심분야〉 암호최적화, 공개키 암호, 암호모듈검증, 네트워크보안