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.2020.30.6.1079

라이브러리 퍼징의 커버리지 향상 방법 연구*

김 서 영,[1†] 조 민 기,[2] 김 종 신,[2] 권 태 경[3‡]
[1]ETRI부설연구소 (연구원), [2,3]연세대학교 정보대학원 정보보호연구실 (대학원생, 교수)

A Study of Coverage Improvement for Library Fuzzing*

Seoyoung Kim,[1†] Mingi Cho,[2] Jongshin Kim,[2] Taekyoung Kwon[3‡]
[1]The Affiliated Institute of ETRI (Researcher),
[2,3]Information Security Lab, GSI, Yonsei University (Graduate student, Professor)

요 약

라이브러리 구현의 결함을 발견하기 위해 동적 분석 방법인 퍼징(fuzzing)이 사용되고 있다. 라이브러리 대상 퍼징은 구현된 함수만 테스트를 할 수 있으므로 더 높은 코드 커버리지를 달성하기 위해서는 구현되지 않은 함수들을 추가로 구현해주어야 한다. 하지만 라이브러리 함수들의 호출 관계를 고려하지 않고 함수를 추가하면 이미 테스트를 수행한 함수가 추가되는 문제가 발생할 수 있다. 본 논문에서는 라이브러리 퍼징의 코드 커버리지 성능을 향상시키기 위한 개선 방법을 제안한다. 먼저, 라이브러리 퍼징의 대상 함수를 효율적으로 추가하기 위해 라이브러리의 함수 호출 그래프 분석하고 구현되지 않은 라이브러리 함수를 추가 구현한다. 그리고 라이브러리의 해결하기 어려운 제약 조건을 가진 분기를 탐색하는 방법으로 하이브리드 퍼징을 적용한다. OpenSSL, mbedTLS, Crypto++을 대상으로 실험한 결과, 제안한 방법이 코드 커버리지를 증가에 효과적인 것을 확인하였다.

ABSTRACT

Fuzzing is used to find vulnerabilities for a library. Because library fuzzing only tests the implemented functions, in order to achieve higher code coverage, additional functions that are not implemented should be implemented. However, if a function is added without regard to the calling relationship of the functions in the library, a problem may arise that the function that has already been tested is added. We propose a novel method to improve the code coverage of library fuzzing. First, we analyze the function call graph of the library to efficiently add the functions for library fuzzing, and additionally implement a library function that has not been implemented. Then, we apply a hybrid fuzzing to explore for branches with complex constraints. As a result of our experiment, we observe that the proposed method is effective in terms of increasing code coverage on OpenSSL, mbedTLS, and Crypto++.

Keywords: Fuzzing, Library Fuzzing, Hybrid Fuzzing, Code Coverage

I. 서 론

라이브러리는 소프트웨어 개발 시 중복된 기능을 효율적으로 재사용하고 기능을 모듈화시키기 위해 널리 사용되고 있다. 라이브러리는 프로그램이 실행될 때 같이 메모리에 로드되어 프로그램이 라이브러리에 구현된 함수를 호출할 수 있다. 라이브러리는 프로그램이 실행될 때 메모리 상에 같이 로드되기 때문에

Received(10. 23. 2020), Modified(11. 23. 2020), Accepted(11. 23. 2020)
* 이 논문은 2020년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구임 (No.2018-0-00513, 기계학습을 활용한 UNIX 기반 커널 취약점 탐지 자동화 연구)
† 주저자, kseoy@nsr.re.kr
‡ 교신저자, taekyoung@yonsei.ac.kr(Corresponding author)

라이브러리에 취약점이 존재하면 해당 프로그램도 라이브러리의 취약점을 공유하게 된다. 라이브러리의 보안도 일반 프로그램만큼 중요하기 때문에 라이브러리를 대상으로 하는 퍼징 기술이 연구되고 있다[1].

라이브러리 대상 퍼징은 라이브러리 함수 일부를 구현한 뒤 랜덤한 입력 값을 사용하여 구현된 함수를 반복적으로 호출한다. 이때 구현된 함수만 테스트 가능하므로 더 높은 코드 커버리지를 달성하기 위해서는 구현되지 않은 함수들을 추가로 구현해주어야 한다. 하지만 라이브러리에 구현된 함수들의 호출 관계를 고려하지 않고 함수를 추가하면 이미 테스트를 수행한 함수가 추가되는 문제가 발생할 수 있다.

본 연구에서는 기존 라이브러리 퍼징의 한계점을 개선하는 라이브러리 퍼징의 코드 커버리지 성능 향상 방법을 제안한다. 먼저, 라이브러리에서 추출된 함수 호출 그래프를 활용하여 라이브러리 함수를 효율적으로 추가 구현한다. 호출 가능한 함수의 개수를 증가시켜 라이브러리의 테스트 범위를 증가시킨다. 그리고 라이브러리 함수에서 복잡한 제약조건 해결이 필요한 경우 새로운 분기문을 탐색하기 위해 하이브리드 퍼징을 적용한다. OpenSSL의 함수 호출 그래프를 활용하여 라이브러리 함수를 추가 구현한 결과 전체 4,553개의 함수 중 653개의 호출 가능한 함수를 708개로 증가시킬 수 있었다. 또 하이브리드 퍼징을 적용한 결과 OpenSSL의 엣지 커버리지를 4,232.5에서 7,898로 약 53.6% 개선하였다.

II. 연구 배경

2.1 라이브러리 취약점

라이브러리는 소프트웨어를 개발할 때 필요한 함수를 제공하는 리소스의 모음이다. 라이브러리를 활용하면 프로그램에 필요한 기능을 모두 구현하지 않고 프로그램을 개발할 수 있어 많은 프로그램은 라이브러리를 활용한다. 하지만 라이브러리에서 취약점이 발생하면 해당 라이브러리를 사용하는 모든 프로그램에 영향을 미친다. 예를 들어, 다양한 암호 기능을 제공하는 대표적인 라이브러리인 OpenSSL에서 2014년도에 HeartBleed라는 취약점이 발견되었으며, 안전 인증을 받은 웹 서버의 17%인 약 50 만대가 영향을 받는 것으로 알려졌다[2]. 또한 안드로이드 미디어 서버에서 사용되는 libstagefright에서 미디어 파일의 파싱 과정에서 취약점이 발견되어 안드로이드 시스템의 권한 상승이 가능하였다[3].

2.2 퍼징

퍼징은 프로그램에 랜덤한 데이터를 무작위로 입력하는 자동화된 소프트웨어 테스팅 기술이다[4]. 퍼징은 주어진 입력 데이터를 변이시킨 뒤 반복적으로 프로그램에 입력하여 프로그램의 비정상 종료를 탐지한다. 가장 단순한 변이 전략을 사용하는 랜덤 기반 퍼징은 유효하지 않은 입력 데이터를 다수 생성하기 때문에 여러 개의 분기문을 동시에 만족하는 입력을 생성하기 어렵다. 이 문제를 해결하기 위해 프로그램의 코드 커버리지 정보를 사용하는 커버리지 기반 퍼징이 사용된다. 커버리지 기반 퍼징은 프로그램 실행 시 코드 커버리지 정보를 기록하여 새로운 코드를 실행한 입력 데이터를 저장한 뒤 이를 변이에 사용한다. 그 결과 커버리지 기반 퍼징은 더 깊은 분기를 효과적으로 탐색한다. 하지만 커버리지 기반 퍼징의 변이 전략은 여전히 랜덤 변이를 사용하므로 복잡한 조건의 분기는 탐색하지 못한다는 한계점이 있다.

이 문제를 해결하기 위해 커버리지 기반 퍼징과 콘콜릭 실행(concolic execution)을 함께 수행하는 하이브리드 퍼징(hybrid fuzzing)이 제안되었다[5]. 콘콜릭 실행은 프로그램 실행 시 제약조건을 생성한 뒤 분기문에 도달하면 제약조건 해결기(SMT solver)를 사용하여 해당 분기문을 탐색할 수 있는 입력값을 구한다. 이를 통해 랜덤 변이로 탐색하지 못하는 복잡한 분기문을 탐색한다[6][7].

III. 퍼징 기법 설계 및 구현

본 장에서는 라이브러리 퍼징의 코드 커버리지를 높이기 위해 함수 커버리지 개선 방법과 하이브리드 퍼징 적용 방법을 설명한다.

3.1 함수 커버리지 개선

Fig. 1.은 함수 커버리지 개선 방법의 전체 구조를 보여준다. 라이브러리 퍼징은 퍼저에 구현된 라이브러리 함수만 테스트하기 때문에 현재 구현된 라이브러리 퍼저가 호출하지 않는 함수 목록을 파악해서 추가해주어야 한다. 이를 위해 먼저 라이브러리에 구현된 전체 함수 목록(all function list)과 라이브러리 퍼저가 호출 가능한 함수 목록(callable

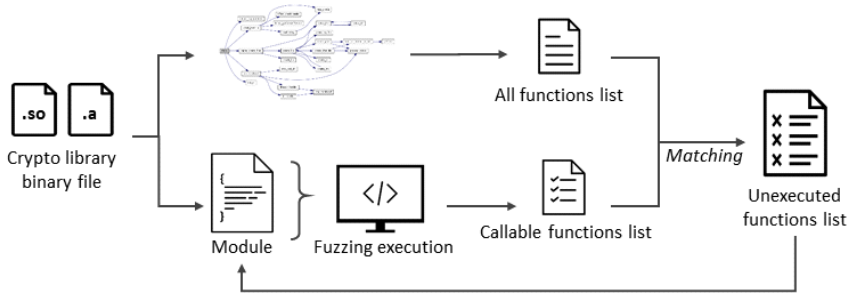

Fig. 1. Coverage improvement flow of library fuzzing

function list)을 알아낸다. 전체 함수 목록을 얻기 위해 컴파일된 라이브러리 바이너리 파일에서 추출된 함수 호출 그래프를 분석하고, 호출 가능한 함수 목록을 얻기 위해 라이브러리 퍼징을 수행한다. 그 결과 생성된 라이브러리의 전체 함수 목록과 호출 가능한 함수 목록을 비교하여 라이브러리 퍼징을 통해 호출 불가능한 함수를 파악하고, 이 함수들을 라이브러리 퍼저에 추가 구현하여 퍼저의 성능을 개선한다.

3.1.1 라이브러리 전체 함수 목록 추출

라이브러리에 구현된 전체 함수 목록을 얻기 위해 먼저 테스트 대상 라이브러리를 컴파일하여 바이너리 파일을 생성한 후, 함수 호출 그래프를 추출한다. 함수 호출 그래프는 함수들 사이의 호출 관계를 표현하는 그래프로 이를 활용하여 라이브러리의 함수 호출 관계를 파악할 수 있다. 함수 호출 그래프에 나타난 전체 함수명에서 중복된 함수명을 제거하여 라이브러리에 구현된 전체 함수 목록을 얻는다.

3.1.2 호출 가능 함수 목록 추출

라이브러리 퍼징은 라이브러리 함수 일부를 사용하는 프로그램을 생성한 뒤 퍼징을 수행한다. 이때 테스트를 위해 구현된 함수에 의해 호출되는 함수들을 정확하게 알기 어렵기 때문에 향후 테스트 프로그램을 확장할 때 테스트가 이미 수행된 함수를 추가하는 문제가 발생할 수 있다. 이 문제를 해결하기 위해 현재 구현된 함수들이 호출할 수 있는 라이브러리 함수의 목록을 구하고, 이를 활용하여 향후 퍼징을 확장할 때 새로운 함수들을 추가할 수 있도록 한다.

호출 가능한 함수 목록 추출은 정적 및 동적 분석 모두 사용할 수 있다. 정적 분석으로 함수 목록을 얻기 위해서는 예제 프로그램에 사용되는 라이브러리 함수 목록과 이 함수들이 호출하는 함수들을 라이브러리의 함수 호출 그래프를 통해 알아낼 수 있다. 하지만 바이너리에서 생성된 함수 호출 그래프는 간접 호출이 발생하는 경우 그래프가 정확하게 생성되지 않는다는 한계점이 존재하므로 호출 가능한 함수 목록을 정확하게 얻을 수 없다. 이에 반해 동적 분석으로 함수 목록을 얻는 방법은 예제 프로그램에 사용된 함수와 이 함수로부터 실제 실행된 함수들을 모두 사용하여 함수 호출 그래프에서 관련 함수들을 얻어오기 때문에 정적 분석과 비교해 더 정확한 결과를 얻을 수 있다. 따라서 본 논문에서는 동적 분석을 수행하여 호출 가능한 함수 목록을 생성한다.

호출 가능한 함수 목록을 얻기 위해 먼저 기존 라이브러리 퍼저가 구현된 모든 라이브러리 함수를 쉽게 호출할 수 있게 수정한다. 라이브러리 퍼저에 구현된 함수 중 특정 조건을 만족해야만 호출되는 함수가 존재하므로 동적 분석을 쉽게 수행하기 위해 위와 같은 과정을 수행한다. 예를 들어, 암호 라이브러리 퍼징에 사용되는 Cryptofuzz[8]는 입력의 특정 오프셋을 암호 알고리즘을 선택하기 위해 사용하고 있으며, 그 결과 많은 수의 분기문이 생성된다. 이때 수행하는 퍼징은 실제 라이브러리를 테스트하기 위함이 아니고 단순히 실행 가능한 함수 목록을 얻기 위함이기 때문에 해당 분기문을 제거하고 모든 라이브러리 함수가 쉽게 호출될 수 있도록 라이브러리 퍼저를 수정한다. 다음으로 수정된 라이브러리 퍼저를 사용하여 퍼징을 수행하고, 실행 결과 호출된 라이브러리 함수들과 해당 함수가 호출할 수 있는 함수 목록을 함수 호출 그래프로부터 추출한다.

3.1.3 라이브러리 퍼징의 함수 호출 개선

퍼징 대상 라이브러리의 전체 함수 목록과 호출 가능한 함수 목록을 비교하여 라이브러리 퍼징으로

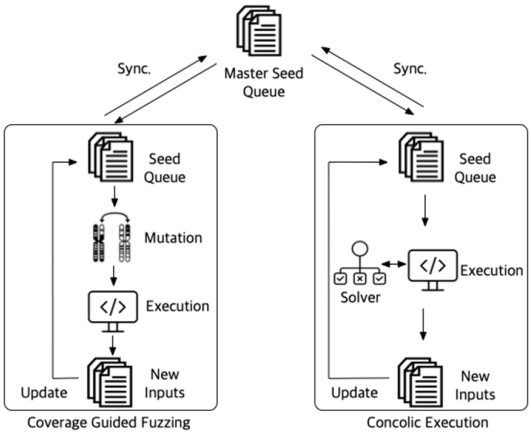

Fig. 2. Hybrid fuzzing structure

호출할 수 없는 함수를 파악할 수 있다. 라이브러리 퍼저에 구현되지 않은 함수는 퍼징을 통해 테스트할 수 없으므로 파악된 함수를 라이브러리 퍼저에 추가 구현한다. 이를 통해 라이브러리 퍼저가 테스트할 수 있는 라이브러리 함수의 수를 늘리고 결과적으로 라이브러리 퍼저의 커버리지 성능을 향상시킬 수 있다.

3.2 라이브러리 대상 하이브리드 퍼징 적용

libFuzzer와 같은 기존의 라이브러리 퍼저는 유전 알고리즘 기반 변이를 통해 새로운 입력 데이터를 생성한다. 랜덤한 변이에 기반하기 때문에 복잡한 제약조건 만족이 필요한 분기를 탐색할 때는 오랜 시간이 소요된다. 이러한 문제를 해결하기 위해 복잡한 제약조건 해결에 효과적인 콘콜릭 실행을 결합하여 복잡한 조건의 분기 탐색을 가능하게 한다.

Fig. 2.는 기존의 라이브러리 퍼징과 콘콜릭 실행을 함께 사용하는 하이브리드 퍼징의 구조를 보여준다. 라이브러리 퍼징과 콘콜릭 실행이 하나의 master seed queue를 공유하고, 기존 라이브러리 퍼징과 콘콜릭 실행은 고유의 seed queue를 가진다. 각각의 seed queue에 새로운 분기를 탐색하는 입력이 생성되면 master seed queue와 연동하여 하이브리드 퍼징을 수행한다.

IV. 실험 결과

본 논문에서 제안한 라이브러리 퍼징의 커버리지 개선 방법을 평가하기 위해 다음과 같은 연구 질문을 설정한다.

Table 1. Number of total function and callable function

Library	Total Function	Callable Function
OpenSSL	4,553	653
mbedTLS	833	104
Crypto++	8,601	965

Table 2. Implemented functions in Cryptofuzz

Library	Implemented	Not implemented
OpenSSL	Digest HMAC CMAC SymmetricEncrypt SymmetricDecrypt KDF_SCRYPT KDF_HKDF KDF_TLS1_PRF KDF_PBKDF1 KDF_PBKDF2 KDF_ARGON2 KDF_SSH	Sign Verify PEM_Read PEM_Write PKCS7_Encrypt D2I

- **연구 질문 1**: 함수 호출 그래프 분석을 통한 함수 커버리지 개선 방법은 라이브러리 퍼징의 커버리지 향상에 유효한 영향이 있는가?
- **연구 질문 2**: 라이브러리 퍼징에 하이브리드 퍼징을 적용하면 적용 전보다 더 높은 커버리지를 달성할 수 있는가?

4.1에서 실험 대상 및 방법에 대해 다루고, 4.2와 4.3에서 각각 연구 질문 1과 연구 질문 2에 대한 실험 결과를 도출한다.

4.1 실험 대상 및 방법

실험 대상으로 암호 라이브러리인 OpenSSL 1.1.0k[10], mbedTLS 2.1.0[11], Crypto++ 8.2[12]를 선정하였다. 암호 라이브러리 퍼저는 가장 최근에 공개된 Cryptofuzz를 사용한다.

퍼징 실험은 12시간씩 20회를 진행하였고, 실험 결과 측정된 엣지 커버리지(edge coverage)와 함수 커버리지(function coverage)를 성능 비교에 사용한다. 실험 결과의 유효성을 검증하기 맨-휘트니 U 검정을 사용해 얻은 p 값(p value)을 사용한다. p 값이 0.05보다 작으면 통계적으로 유의미한 차이가 있다는 것을 의미한다.

Table 3. Result of code coverage by 12h fuzzing of OpenSSL and OpenSSL_NEW

Library	Function coverage	Edge coverage
OpenSSL	233.0	4,232.5
OpenSSL_NEW	145.0	5,156.0

Table 4. Number of functions by 12h fuzzing

Library	Fuzzer	Function coverage	p value
OpenSSL	Cryptofuzz	233.0	-
	QSYM	290.5	$<10^{-4}$
	Intriguer	400.0	$<10^{-11}$
OpenSSL_NEW	Cryptofuzz	145.0	-
	QSYM	289.0	$<10^{-6}$
	Intriguer	338.5	$<10^{-19}$
mbedTLS	Cryptofuzz	33.0	-
	QSYM	51.5	$<10^{-7}$
	Intriguer	51.0	$<10^{-7}$
Crypto++	Cryptofuzz	27.0	-
	QSYM	180.5	$<10^{-10}$
	Intriguer	494.5	$<10^{-18}$

4.2 함수 커버리지 개선 결과

연구 질문 1에 답하기 위해 3.1절에서 제안한 함수 커버리지 개선 방법을 Cryptofuzz에 적용하고 개선된 코드 커버리지 결과를 개선 전과 비교한다. 이를 위해 암호 라이브러리에 구현된 전체 함수 목록과 Cryptofuzz에 구현된 라이브러리 함수 목록을 구하였다. Table 1.은 라이브러리별로 생성된 전체 함수 개수와 호출 가능한 함수 개수를 보여준다. 전체 함수 개수와 비교해 호출 가능한 함수의 수가 적은 것을 알 수 있다.

Table 2.는 Cryptofuzz에 구현된 OpenSSL 라이브러리 기능과 구현되지 않은 기능을 보여준다. 추출된 두 함수 목록을 비교하여 Cryptofuzz에 구현되지 않은 라이브러리의 기능을 쉽게 발견할 수 있었고, 구현되지 않은 기능을 Cryptofuzz에 추가 구현하였다. 그 결과 OpenSSL의 4,553개의 전체 함수 중 호출 가능한 함수 개수는 653개인 14.3%에서 708개인 15.5%로 증가하였다.

Table 3.은 라이브러리의 기능을 추가 구현한 후 퍼징 실험을 수행한 결과 측정된 함수 커버리지와 엣지 커버리지 결과이다. 3.1절에서 제안한 기법을 사용하여 개선된 OpenSSL은 OpenSSL_New라고 표기한다. 실험 결과 프로그램의 엣지 커버리지는 증가하였지만, 함수 커버리지는 감소한 결과가 도출되었다. 이는 추가 구현한 함수가 많은 분기를 생성하였고 그 결과 복잡성이 높아져 많은 수의 함수를 실행하지 못했기 때문이다. 이 문제는 하이브리드 퍼징을 적용하여 해결하였다.

Table 5. Number of edges by 12h fuzzing

Library	Fuzzer	Edge coverage	p value
OpenSSL	Cryptofuzz	4,232.5	-
	QSYM	7,293.0	$<10^{-13}$
	Intriguer	7,898.0	$<10^{-14}$
OpenSSL_NEW	Cryptofuzz	5,156.0	-
	QSYM	10,508.5	$<10^{-10}$
	Intriguer	15,977.5	$<10^{-24}$
mbedTLS	Cryptofuzz	3,602.0	-
	QSYM	5,028.5	$<10^{-6}$
	Intriguer	7,602.5	$<10^{-27}$
Crypto++	Cryptofuzz	4,511.0	-
	QSYM	8,019.0	$<10^{-12}$
	Intriguer	11,157.5	$<10^{-20}$

4.3 하이브리드 퍼징 적용 결과

연구 질문 2에 답하기 위해 복잡한 분기를 해결하는 방법인 하이브리드 퍼징을 라이브러리 퍼징에 적용한 뒤 커버리지 결과를 비교하였다. 이를 위해 최신 콘콜릭 실행 기술인 QSYM과 Intriguer를 Cryptofuzz에 결합하여 하이브리드 퍼징을 수행하였다. 하이브리드 퍼징은 콘콜릭 엔진과 퍼징 엔진에 코어를 하나씩 할당하여 총 두 개의 코어를 사용한다. 하이브리드 퍼징이 아닌 Cryptofuzz의 코어 사용 조건을 동일시하기 위해 Cryptofuzz의 또한 코어를 두 개 사용한다. 4.2절에서 개선된 OpenSSL은 OpenSSL_New라고 표기한다. Table 4.는 하이브리드 퍼징 적용 전과 후의 함수 커버리지 비교 결과이다. 하이브리드 퍼저인 Intriguer를 사용하였을 때 Cryptofuzz를 사용한 것보다 OpenSSL, OpenSSL_NEW, mbedTLS, Crypto++에서 각각 71.7%, 133%, 54.5%, 1,731.4%의 함수 커버리지가 증가하였다.

Table 5.는 하이브리드 퍼징 적용 전후의 엣지 커버리지 측정 결과이다. 하이브리드 퍼저인

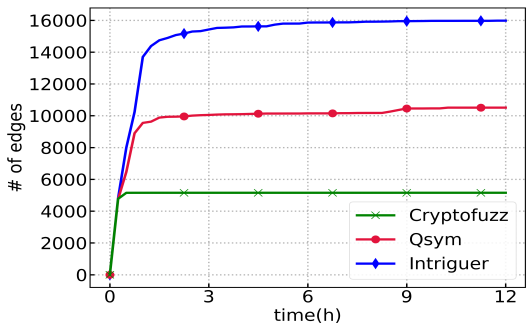

Fig. 3. Number of edges by 12h fuzzing of OpenSSL_New

Intriguer를 사용하였을 때 Cryptofuzz를 사용한 것보다 OpenSSL, OpenSSL_NEW, mbedTLS, Crypto++에서 각각 86.6%, 209.9%, 111.1%, 147.3%의 엣지 커버리지가 증가하였다. Fig. 3.은 OpenSSL_New를 대상으로 퍼징 수행 시 엣지 커버리지 변화를 보여준다. 실험 결과 Intriguer가 가장 높은 엣지 커버리지를 달성하였다.

V. 관련 연구

5.1 라이브러리 퍼징

libFuzzer[9]는 라이브러리 퍼징을 수행하기 위해 테스트 대상 랜덤한 입력을 사용하여 라이브러리 함수를 호출한다. libFuzzer는 랜덤 기반의 변이 전략을 사용하기 때문에 암호 라이브러리와 같은 복잡한 소프트웨어에서 깊은 분기 탐색에 많은 시간이 소요된다. Cryptofuzz[8]는 암호 라이브러리를 대상의 퍼저이다. Cryptofuzz는 일반적인 퍼징으로 탐지할 수 있는 메모리 오류뿐만 아니라 같은 기능을 수행하는 다른 암호 라이브러리의 결과값을 비교하는 차분 테스트를 수행하여 암호 알고리즘의 구현 오류도 발견한다. Cryptofuzz는 랜덤 변이 전략을 사용하는 libFuzzer를 기반으로 구현되었기 때문에 깊은 분기 탐색이 어렵다는 한계점이 존재한다.

5.2 하이브리드 퍼징

QSYM[6]은 기존 콘콜릭 실행의 느린 속도를 개선한 하이브리드 퍼징을 제안한다. QSYM은 명령어 수준의 심볼릭 실행 기술을 적용하여 기존의 베이직 블록 수준의 심볼릭 실행 기술보다 더욱 효율적으로 콘콜릭 실행을 수행한다. Intriguer[7]는 필드 정보를 사용하여 최적화된 콘콜릭 실행을 제안한다. 기존의 하이브리드 퍼징의 콘콜릭 실행 엔진이 지니는 느린 심볼릭 실행 속도, 불필요한 제약조건 해결에 많은 시간 사용, 복잡하지 않은 제약 해결에 많은 자원 할당과 같은 문제를 개선하였다.

VI. 결 론

본 연구에서는 라이브러리 퍼징의 커버리지를 개선하기 위해 함수 호출 그래프 분석 방법와 하이브리드 퍼징 적용 방법을 제시하였다. 함수 호출 그래프 분석을 통해 라이브러리 퍼징 함수를 증가시킨 결과 Cryptofuzz에 구현된 OpenSSL의 함수 653개를 708개로 늘릴 수 있었다. 또한 하이브리드 퍼징 적용 결과 OpenSSL의 엣지 커버리지를 4,232.5에서 7,898로 약 53.6% 개선하였다.

향후 본 연구를 바탕으로 OpenSSL_NEW에 구현되지 않은 OpenSSL 함수를 더 추가하여 더 높은 코드 커버리지를 달성할 수 있을 것이다. 또한 OpenSSL뿐만 아니라 mbedTLS, Crypto++를 포함한 다양한 라이브러리로 대상을 확장하여 본 논문에서 제안하는 함수 커버리지 개선 방법을 적용한다면 더 다양한 결과를 얻을 수 있을 것이다.

References

[1] J. Jang, and H. Kim, "Automated Applying Greybox Fuzzing to C/C++ Library Using Unit Test," Journal of KIISC, 29(4), pp. 807-819, Aug. 2019.

[2] Netcraft, "Half a million widely trusted websites vulnerable to Heartbleed bug," Apr. 2014.

[3] J. Drak, "Stagefright: Scary Code in the Heart of Android," BlackHat USA, Aug. 2015.

[4] B. P. Miller, L. Fredriksen, and B. So, "An Empirical Study of the Reliability of UNIX Utilities," Communications of the ACM, vol. 33, no. 12, pp. 32-44, Dec. 1990.

[5] R. Majumdar and K. Sen, "Hybrid Concolic Testing," Proceedings of the

International Conference on Software Engineering, pp. 416-426, May. 2007.
〔6〕 I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim., "QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing," Proceedings of the USENIX Security, pp. 745-761, Aug. 2018.
〔7〕 M. Cho, S. Kim and T. Kwon, "Intriguer: Field-Level Constraint Solving for Hybrid Fuzzing," Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, pp. 515-530, Nov. 2019.
〔8〕 Guidovranken, "Cryptofuzz," https://github.com/guidovranken/cryptofuzz, 2019.
〔9〕 K. Serebryany, "libFuzzer - a library for coverage-guided fuzz testing." LLVM project, 2015.
〔10〕 OpenSSL, https://www.openssl.org
〔11〕 mbedTLS, https://github.com/ARMmbed/mbedtls
〔12〕 Crypto++, https://www.cryptopp.com

〈저 자 소 개〉

김 서 영 (Seoyoung Kim) 정회원
2018년: 연세대학교 컴퓨터공학과 학사
2020년: 연세대학교 정보대학원 정보보호 석사
2020년~현재: ETRI부설연구소 재직
〈관심분야〉 정보 보안, 소프트웨어 보안, 시스템 보안

조 민 기 (Mingi Cho) 학생회원
2017년: 부산대 정보컴퓨터공학과 학사
2017년~현재: 연세대학교 정보대학원 정보보호 석박통합과정
〈관심분야〉 소프트웨어 보안, 시스템 보안, 정보 보안

김 종 신 (Jongshin Kim) 정회원
2018년: 부산외국어대학교 임베디드소프트웨어학과 학사
2020년: 연세대학교 정보대학원 정보보호 석사
〈관심분야〉 무선 센서 네트워크, 키 교환 프로토콜, 정보 보안

권 태 경 (Taekyoung Kwon) 종신회원
1992년: 연세대학교 컴퓨터과학과 학사
1995년: 연세대학교 컴퓨터과학과 석사
1999년: 연세대학교 컴퓨터과학과 박사
1999년~2000년: U.C. Berkeley EECS Post-Doc.
2001년~2013년: 세종대학교 컴퓨터공학과 교수
2007년~2008년: Univ. of Maryland, College Park 교환교수
2013년~현재: 연세대학교 정보대학원 교수
〈관심분야〉 암호 프로토콜, Usable Security, 소프트웨어/시스템 보안, 기계학습과 보안