*Journal of The Korea Institute of Information Security & Cryptology* ISSN 1598-3986(Print)
VOL.31, NO.4, Aug. 2021 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2021.31.4.587

# 테스트 프레임워크를 활용한 라이브러리 퍼징 환경 구축 자동화

류 민 수,[1†] 김 동 영,[1] 전 상 훈,[1] 김 휘 강[2‡]
[1,2]고려대학교 정보보호대학원 (대학원생, 교수)

# Automated Building Fuzzing Environment Using Test Framework

Minsoo Ryu,[1†] Dong Young Kim,[1] Sanghoon Jeon,[1] Huy Kang Kim[2‡]
[1,2]School of Cybersecurity, Korea University (Graduate student, Professor)

## 요 약

라이브러리는 독립적으로 실행되지 않고 많은 응용 프로그램에서 사용되므로, 라이브러리의 취약점을 사전에 탐지하는 것은 중요하다. 라이브러리 취약점을 탐지하기 위해 동적 분석 방법인 퍼징이 사용되고 있다. 퍼징 기술은 코드 커버리지 및 크래시 발생 횟수 측면에서 개선된 결과를 보여주지만, 그 효과를 라이브러리 퍼징에 적용하기는 쉽지 않다. 특히, 라이브러리의 다양한 상태를 재현하려면 특정 함수 시퀀스를 호출하고 퍼저의 입력을 전달하여 라이브러리 코드를 실행하는 퍼징 대상 파일과 시드 코퍼스가 필요하다. 그러나 퍼징 환경(시드 코퍼스, 퍼징 대상 파일)을 준비하는 것은 라이브러리에 대한 이해와 퍼징에 대한 이해가 동시에 필요한 어려운 일이다. 이에, 본 논문에서는 테스트 프레임워크를 활용하여 라이브러리 퍼징의 용이성을 확보하고, 코드 커버리지와 크래시 탐지 성능을 향상하기 위한 개선 방법을 제안한다. 본 논문에서 제안한 시스템은 9개의 오픈 소스 라이브러리에 적용하여 기존 연구들과 비교를 통한 개선 효과를 검증하였다. 실험 결과 코드 커버리지 31.2%, 크래시 탐지 기준 58.7%의 개선 효과를 확인하였고, 3개의 알려지지 않는 취약점을 탐지하였다.

## ABSTRACT

Because the library cannot be run independently and used by many applications, it is important to detect vulnerabilities in the library. Fuzzing, which is a dynamic analysis, is used to discover vulnerabilities for the library. Although this fuzzing technique shows excellent results in terms of code coverage and unique crash counts, it is difficult to apply its effects to library fuzzing. In particular, a fuzzing executable and a seed corpus are needed that execute the library code by calling a specific function sequence and passing the input of the fuzzer to reproduce the various states of the library. Generating the fuzzing environment such as fuzzing executable and a seed corpus is challenging because it requires both understanding about the library and fuzzing knowledge. We propose a novel method to improve the ease of library fuzzing and enhance code coverage and crash detection performance by using a test framework. The systems's performance in this paper was applied to nine open-source libraries and was verified through comparison with previous studies.

**Keywords:** Library Fuzzing, Automation, Test Framework

Received(04. 06. 2021), Modified(06. 07. 2021), Accepted(06. 14. 2021)

† 주저자, onsoim@korea.ac.kr
‡ 교신저자, cenda@korea.ac.kr(Corresponding author)

## I. 서 론

퍼징은 오늘날 소프트웨어의 보안 취약성 및 안전성 문제를 발견하는 가장 효과적인 테스트 기술 중 하나로 널리 활용되고 있다. 퍼저는 임의로 생성된 값을 입력으로 프로그램을 실행하고, 메모리 손상 문제와 같은 잘못된 작업에 대한 동작을 관찰한다. 퍼저에 의해 뮤테이트(mutate)되는 입력값들은 각각 프로그램의 다른 경로까지 도달하기 때문에 프로그램의 복잡한 상태를 생성할 수 있다. Miller[1] 에 의해 블랙박스 퍼징이 최초로 제안된 이후 최근 커버리지 유도 퍼저와 같은 그레이박스 퍼징 기술의 발전으로 인해 퍼징이 프로그램의 더 깊은 경로에 도달하여 훨씬 더 많은 버그를 발견할 수 있게 되었다[2]. 예를 들어, AFL[3] 을 기반으로 하는 AFLFast[4], AFLGo[5], CollAFL[6] 뿐만 아니라 Driller[7], VUzzer[8], T-Fuzz[9], QSYM[10], Angora[11] 등의 실행 피드백과 코드 커버리지를 기반으로 입력값을 뮤테이트하는 하이브리드 퍼저 등의 방법으로 버그를 찾는 효율성은 지속해서 증가하고 있다.

이러한 그레이박스 퍼징 기술은 코드 커버리지 및 프로그램에서 발견된 크래시 발생 횟수 측면에서 개선된 결과를 보여주지만, 그 효과를 라이브러리 퍼징에 적용하기는 쉽지 않다. 라이브러리는 특성상 개별 함수 간의 종속성 정보 없이 API(Application Programming Interface)만을 노출한다. 호출 간에 공유되는 복잡한 상태를 구축하기 위해서는, 올바른 매개변수를 사용하여 올바른 API 시퀀스로 호출되어야 한다. 이러한 라이브러리 API 호출 간의 종속성은 명시적으로 작성되어 있지 않기 때문에 무작위로 함수를 호출하고 무작위의 매개변수를 사용하면 효율적인 퍼징 캠페인이 발생하지 않을 수 있다. 따라서 라이브러리를 퍼징 하기 위한 기존의 일반적인 접근 방식은 초기 상태를 구축하고 API 함수를 유효한 시퀀스로 호출하는 최소한의 프로그램을 수동으로 작성하는 것이다. 이를 통해 퍼저는 라이브러리에서 제공하는 기능을 테스트하는데 필요한 다양하고 복잡한 상태를 구축할 수 있다. 그러나 퍼징을 위한 최소한의 프로그램을 수동으로 작성하기 위해서는 개발자가 테스트 중인 코드 베이스와 퍼저 엔진의 작동 방식을 이해해야 한다. 즉 API 호출/종속성 및 퍼징 매개변수를 결정하는 것은 전적으로 개발자의 지식에 의존한다. 그뿐만 아니라, 이러한 노동 집약적인 작업은 라이브러리 코드가 변함에 따라 유지보수의 비용까지 발생한다.

이러한 문제를 해결하기 위해 라이브러리 퍼징을 위한 자동화 연구가 수행되고 있다. FUDGE[12], FuzzGen[13] 의 경우 퍼즈 드라이버와 시드 코퍼스를 생성하기 위해 라이브러리를 사용하는 컨슈머(consumer) 프로그램을 분석하는 방법을 제안하였다. 이를 통해 퍼즈 드라이버의 초기 상태를 설정하고 라이브러리의 올바른 API 호출 시퀀스를 충족할 수 있도록 하였다. 이러한 연구는 대상 라이브러리를 실제 사용하고 있는 컨슈머 프로그램의 분석에 기반하여 생성하므로, 컨슈머 프로그램의 특성에 따라 다양한 상태에서의 API 시퀀스가 아닌 일부 기능만을 대상으로 제한적인 퍼징이 수행될 가능성이 크고, 무엇보다 컨슈머 프로그램의 분석을 통해서 라이브러리 퍼즈 드라이버가 생성되므로 독립적인 시스템을 구성하기 어렵다는 단점이 존재한다.

이에 본 연구진은 선행 연구에서 테스트 프레임워크를 기반으로 하여 퍼즈 드라이버를 자동으로 생성하는 시스템인 FuzzBuilder[14] 시스템을 제안하였다. 기존 연구에서 퍼즈 드라이버라 하면 퍼징을 하는 데 필요한 퍼징 대상 파일만을 일컫지만, 본 연구에서는 이를 확장하여 퍼징을 하는 데 필요한 시드 코퍼스와 퍼징 대상 파일을 모두 아우르는 용어로 재정의하여 사용하였다. 이러한 선행 연구를 통해 라이브러리 등의 퍼징 적용과 시드 확보가 어려운 프로젝트들에 적용하여 큰 효과를 거둔 바 있다. 그러나 기존 시스템의 경우 퍼징 적용을 위한 설정 파일을 수동으로 생성해야 하는 문제점이 있고, 자동으로 생성된 퍼즈 드라이버의 낮은 효율성과 코드 커버리지 측면에서의 한계가 있었다. 이에 본 연구에서는 기존 시스템의 한계점을 개선한 *FuzzBuilderEx* 시스템을 제안한다. 기존 연구와의 차이점은 다음과 같다. 첫째, 수동으로 작성해야 했던 설정 파일 생성 등의 초기 과정을 자동화하였다. 둘째, 퍼징을 수행하면서 퍼징 대상 파일에서 불필요한 함수를 제거하고, 퍼징 대상 파일을 세부 기능 단위로 세분화하여 생성함으로써 퍼징의 효율성을 증가하였다. 셋째, 기존 문자열 타입의 매개변수뿐만 아니라 정수형 타입의 매개변수를 대상으로 퍼징 할 수 있도록 확장하여 코드 커버리지가 증가할 수 있도록 개선하였다.

본 연구에서 제안하는 *FuzzBuilderEx* 시스템은 사람의 개입을 최소화하기 위해 라이브러리 프로젝트에 존재하는 테스트 프로그램의 정적/동적 분석을 통해 획득한 정보를 활용하여 퍼즈 드라이버를 자동으

로 생성한다. 또한, 개발자는 *FuzzBuilderEx* 시스템을 통해 퍼징을 위한 코드 작성의 부담에서 벗어날 수 있으며, 검증자는 라이브러리에 대한 적은 이해로 라이브러리에 퍼징을 적용할 수 있다.

## II. 관련 연구

본 연구에는 그레이박스 퍼저를 기반으로 초기 입력 데이터 역할을 하는 시드 코퍼스와 퍼징의 대상이 되는 퍼징 대상 파일을 자동으로 생성하는 연구를 수행한다. 이에 관련 연구로 그레이박스에 관한 관련 연구를 먼저 논의하고, 시드 코퍼스 및 퍼징 대상 파일 생성에 대한 자동화 관련 연구를 소개한다.

### 2.1 그레이박스 퍼징

이 절에서는 관련 연구에 대한 소개에 앞서 그레이박스 퍼징에 대한 배경을 먼저 설명한다. 그레이박스 퍼징에는 다양한 유형이 있지만 많은 그레이박스 퍼저들이 AFL을 기반으로 연구하고 있고, 가장 많이 사용되고 있는 퍼저 중의 하나이므로 AFL의 뮤테이션(mutation) 기반 그레이박스에 중점을 두고 설명한다.

퍼징은 입력값을 가지는 프로그램에서 버그를 찾는 동적 테스트이다. 특히 그레이박스 퍼징은 경량 프로그램 분석을 사용하여 다양한 프로그램 경로를 탐색하기 위한 임의의 입력값을 효율적으로 생성한다. Fig. 1.은 그레이박스 퍼징의 동작을 도식화한 것이다.

그레이박스 퍼저는 초기에 제공된 시드를 기반으로 임의의 입력값을 생성한다. 이 값을 매개변수로 하여 실행한 프로그램이 종료되면, 그레이박스 퍼저는 베이직 블럭 커버리지와 같은 실행 정보를 분석한 다음 생성된 입력값이 새로운 커버리지를 달성하는 경우 시드를 갱신하며, 이 단계의 반복을 통해 시드 코퍼스를 갱신한다. 그러나 초기 시드를 기반으로 임의의 입력값을 생성하는 방식을 사용하기 때문에, 프로그램이 MKV 확장자와 같이 입력값으로 복잡한 구조를 요구하는 경우 그레이박스 퍼저가 높은 코드 커버리지를 달성하는 것은 어렵다. 이러한 문제를 해결하기 위해 다양한 프로그램 경로를 탐색할 수 있는 퍼징 관련 연구가 수행되었다.

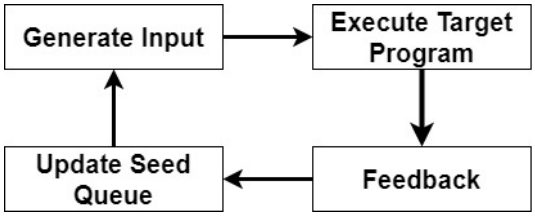

Fig. 1. Process of Grey-box fuzzing

Angora[11], VUzzer[8], Lafintel[15], Steelix[16],[17], TIFF[18], Memfuzz[19] 등의 연구에서는 이러한 문제를 해결하기 위해 입력 데이터 생성 메커니즘으로 퍼징 및 기호 실행을 복합적으로 사용하는 하이브리드 접근 방식을 제안하였다. 이러한 연구를 통해 더욱 효율적인 퍼징을 위한 입력 데이터를 생성할 수 있었지만, 기호 실행의 적용을 위해서는 추가적인 런타임 리소스(runtime resource)가 필요하다는 단점이 존재한다. 그러나 위의 입력 데이터 생성 메커니즘과 달리 효율적인 초기 시드를 제공하는 것은 추가적인 런타임 리소스 없이 충분한 코드 커버리지를 달성하는 데 도움이 되기 때문에 보다 유용한 방법이라 할 수 있다.

그레이박스 퍼저는 퍼징의 효율성을 높이기 위해 최대한 빨리 버그를 찾아야 한다. 이를 달성하는 방법의 하나는 퍼징 중에 실행되는 프로그램의 명령 수를 증가시키는 것이다. 더 많은 명령이 실행되면 코드의 더 많은 부분을 방문하여 버그를 발견할 기회가 증가한다. 따라서 방문한 코드 라인의 수를 측정하여 발견된 크래시 수와 함께 그레이박스 퍼저의 성능을 평가한다.

대부분의 그레이박스 퍼저에는 퍼징을 위한 대상 파일이 필요하며 그 퍼징 대상 파일은 짧은 시간 내에 종료되도록 보장되어야 한다. 또한, 그레이박스 퍼저는 다양한 프로그램 경로를 탐색하기 위해 가능한 다양한 입력값으로 프로그램을 실행해야 한다. 이러한 전제조건으로 인해 요청을 기다리는 라이브러리 또는 서버 프로그램 같은 일부 프로그램에는 그레이박스 퍼징을 적용하기 어렵다. 따라서 퍼징 대상 파일은 그레이박스 퍼징에 적용할 수 있도록 준비되어야 하며, 대상 프로그램에 정의된 호출 기능이 포함되어 짧은 시간 내에 종료되어야 한다.

### 2.2 시드 코퍼스 생성 자동화

시드 코퍼스를 생성하기 위한 관련 연구는 크게 시퀀스 분석을 이용하는 연구와 딥러닝 모델을 사용하는 연구로 구분이 된다. 아래의 절에서 각 항목에

관한 관련 연구를 살펴본다.

#### 2.2.1 시퀀스 분석

시드 코퍼스는 퍼저에서 초기 입력값으로 사용되는 입력 파일들의 집합을 의미한다. 이러한 시드 코퍼스를 생성하기 위해, 퍼징 대상 프로그램의 사용 시나리오를 분석한 이전 연구가 있었다. 먼저, 사용 시나리오를 분석하기 위한 연구에서는 마이닝 또는 코드 스니펫(snippet)을 사용하여 소스 코드 시퀀스를 분석한다. MAPO[20]는 코드 검색 엔진에서 가장 빈번히 사용되는 코드 시퀀스를 분석하기 위해 연구했다. MLUP[21] 는 MAPO를 확장한 연구로, 제어 시퀀스의 멀티 레벨 분석을 사용하여 가장 널리 사용되는 코드 시퀀스를 찾는다. PAM[22] 은 기계 학습을 사용하여 제어 시퀀스를 찾도록 향상되었다. 마지막으로 APIMiner[23], MUSE[24] 및 CLAMS[25] 는 프로그램 슬라이싱 기법을 사용하여 소스 코드 시퀀스를 분석한다.

소프트웨어 엔지니어링 분야에서도 시퀀스를 자동으로 생성하는 연구가 수행되었다. 특히 테스트 시퀀스를 자동으로 생성하는 연구가 수행되었는데, 이는 본 연구에서의 시드 코퍼스 생성과도 밀접한 관련이 있다. Randoop[26] 은 피드백 정보를 지침으로 하여 테스트 시퀀스를 생성한다. RecGen[27] 은 각각 정적 및 동적 접근 방식을 사용하여 함수 시퀀스를 추론하며, 추론된 시퀀스를 바탕으로 하여 무작위로 한 테스트 시퀀스를 생성했다. Palus[28] 는 정적 및 동적 접근 방식을 결합하여 위의 도구를 개선하였다. 또한, Pradel[29] 은 동적 분석을 통해 FSM(Finite State Machine)을 생성하여 FSM을 기반으로 함수 시퀀스를 생성했다.

위의 연구들은 다양한 분석을 통해 일련의 함수 시퀀스를 생성하고, 이에 맞는 입력값 형식을 구체화하여 이를 충족하는 시드 코퍼스를 생성한다. 즉, 위의 연구들은 다양한 분석을 통해 입력값 형식을 구체화하고 이를 충족하는 시드 코퍼스를 생성하는 반면에, 본 연구에서는 개발자가 작성한 테스트 코드의 함수 시퀀스를 사용하여 시드 코퍼스를 생성한다. 특히 개발자가 개발 단계에서 검증을 위해 사용하는 테스트 코드를 그대로 사용하므로 별도의 노력 없이 효율적인 시드 코퍼스를 생성할 수 있으며, 무엇보다 주요 사용 시나리오를 기반으로 작성된 테스트 코드를 컴파일(compile)한 테스트 프로그램을 수정하여 퍼징 대상 파일을 자동으로 생성할 수 있다는 점에서 기존 연구와 차별화된다.

#### 2.2.2 딥러닝 모델

앞선 시퀀스 분석 기반의 접근은 많은 전문지식이 필요하거나 분석에 필요한 리소스가 크므로 확장성이 제한된다는 단점이 있었다. 따라서 이러한 문제를 개선하기 위하여 딥러닝 모델을 사용한 시드 코퍼스 생성을 위한 연구가 진행되었다. Skyfire[30] 에서 제안한 시드 생성 방법은 기존의 시드에 PCSG (Probabilistic Context-Sensitive Grammar)를 사용하여 시맨틱 정보 및 문법 규칙을 자동으로 추출하고, 이를 딥러닝 모델 학습에 사용한다. 이 학습된 모델은 시맨틱 정보와 문법 규칙을 준수하는 새로운 시드를 생성한다. 또한, SmartSeed[31] 는 초기 시드를 AFL을 이용하여 수행한 후에 유니크 크래시 혹은 유니크 경로를 유발하는 시드를 학습 데이터셋으로 확보한 다음 GAN(Generative Adversarial Networks)[32] 모델을 사용하여 이와 유사한 시드를 생성하는 방법을 제안하였다.

이러한 딥러닝 모델을 사용한 연구들은 기존의 시드를 이용해서 또 다른 시드를 생성하는 연구이고, 본 연구는 테스트 코드를 기반으로 시드 코퍼스를 생성하는 것이므로 접근 방식의 차이점이 존재한다. 하지만, 라이브러리 내에 존재하는 테스트 코드가 높은 수준의 테스트를 포함하지 않거나, 테스트 코드가 부족한 경우에는 본 연구를 통해 자동 생성된 시드 코퍼스는 퍼징의 효율성을 높이기는 어렵다는 한계가 있다. 따라서 이러한 문제를 해결하기 위해 향후 연구로 기존 연구에 딥러닝 모델을 적용하는 연구를 진행하고자 한다.

### 2.3 퍼징 대상 파일 생성 자동화

자동화된 퍼징 대상 파일 생성 시스템인 FUDGE[12] 는 오픈 소스 라이브러리에 퍼징을 쉽게 적용할 수 있게 한다. FUDGE는 구글 코드 베이스를 참고하여 라이브러리와 관련된 다양한 코드 스니펫을 추출한다. FuzzGen[13] 은 라이브러리에서 제공하는 API를 인식하여 퍼징 대상 파일을 생성하기 위한 또 다른 최신의 관련 연구이다. FuzzGen은 FUDGE와 유사하게 라이브러리를 사용하고 있는 시스템의 분석을 통해 라이브러리에 대

한 API 호출 시퀀스를 발견하고 이를 LLVM 비트 코드로 컴파일한다. FuzzGen은 라이브러리에 대한 API 의존 관계를 추상화하는 A2DG (Abstraction API Dependence Graph)를 생성한다. FuzzGen은 A2DG를 활용하여 컨슈머 프로그램과 유사한 API 호출 시퀀스를 실행하는 libFuzzer 스텁(stub)을 생성한다.

반면, *FuzzBuilderEx*는 라이브러리를 사용하는 컨슈머 프로그램으로부터 코드 스니펫을 추출하는 대신, 라이브러리 내의 모든 테스트 코드를 계측하고 정의된 FA(Fuzzable API)를 호출하는 특정 명령을 우선순위로 지정하여 더욱 효율적인 퍼징을 위해 최적화된 퍼징 대상 파일을 생성한다. 여기서 FA는 라이브러리 내에 존재하는 API 중에서도 퍼징을 할 수 있는 API를 의미한다. 즉, 퍼징 대상 파일 생성을 위해서 별도의 컨슈머 프로그램의 분석 없이, 개발 과정에서 수행하는 개발자 테스트 코드를 기반으로 퍼징 대상 파일을 생성한다는 점에서 기존 연구와의 차별점을 가지고 있다. 또한, *FuzzBuilderEx*는 컨슈머 프로그램의 분석 없이 라이브러리의 테스트 코드만 필요하므로 FuzzGen이나 FUDGE 등의 기존 관련 시스템보다 가볍고 독립적인 시스템이다.

## III. 제안하는 라이브러리 퍼징 시스템

선행 연구인 FuzzBuilder 시스템은 테스트 프레임워크를 활용하여, 라이브러리 퍼징을 위한 검증자의 개입을 줄여주었다는 기여를 하였다. 하지만 여전히 라이브러리 퍼징을 위해서는 검증자의 개입을 많이 필요로 하며, 제한된 API만을 대상으로 적용할 수 있다는 한계가 있다. 이에 제안하는 *FuzzBuilderEx* 시스템은 선행 연구를 고도화하고 자동화를 더하여 기존보다 더욱 사람의 개입을 최소화하고, 더 많은 FA를 대상으로 퍼징을 할 수 있도록 하였다.

선행 연구에서도 테스트 프레임워크를 이용하여 시드 코퍼스를 생성할 수 있다. 하지만 시드 코퍼스를 생성하기 위해서는 FA를 지정해 주는 역할을 하는 설정 파일이 있어야 하며, 이 FA를 선정하기 위해 목표로 선정한 프로젝트를 분석하여 프로젝트에 존재하는 API들을 파악해야 한다. 또한, API 중에서도 제약사항들을 충족하는 일부 API만이 FA로 사용될 수 있기에 이를 선정하기 위한 사람의 개입이 필요하다.

본 연구에서는 이를 자동화하여 FA의 후보가 될 수 있는 API들을 선정하고, 이를 위한 FA 설정 파일을 생성한다. 이 생성된 FA 후보군을 그대로 또는 이를 참고하여 FA를 지정할 수 있다. 또한, 본 연구에서는 INT와 같은 정수형 타입 등의 추가 매개변수의 확대로 더 다양한 FA를 대상으로 퍼징 할 수 있다. 그 후 *FuzzBuilderEx*는 이전에 이용했던 테스트 프로그램을 이용하여 퍼징을 위한 퍼징 대상 파일을 생성한다. 기존 연구에서는 FA의 매개변수에 들어가는 값을 퍼징에서 사용되는 변수로 대체하는 것만을 수행하였지만, 본 연구에서는 퍼징 수행에 불필요한 함수를 제거하고 기능 단위로 세분화하여 더 향상된 효율성을 기대할 수 있다.

*FuzzBuilderEx*의 동작 개요는 Fig. 2.와 같다. 초록색은 선행 연구 대비 새롭게 추가된 단계를 의미하며, 파란색은 세부 단계가 추가된 것을 의미한다.

1) FA 추론 단계에서는 기존 연구 대비 퍼징 자동화 수준을 향상하기 위하여, FA 선별 과정을 자동화시키는 FA 추론 과정을 추가하였다. FA 추론 단계에서는 라이브러리의 소스 코드, 라이브러리 바이너리, 그리고 테스트 프로그램을 정적 및 동적으로 분석하여 FA를 자동으로 선정하고 선정된 FA를 위한 설정 파일을 생성한다.
2) 시드 코퍼스 생성 단계에서는 수정된 라이브러리와 테스트 프로그램을 사용하여 시드 코퍼스를 생성한다. 특히 커버리지 측면의 퍼징 성능 향상을 위하여 본 연구에서는 문자열을 인자로 사용하는 FA뿐만 아니라 정수를 인자로 사용하는 FA도 대상으로 퍼징 할 수 있도록 확장하였다.
3) 매개변수 치환 단계에서는 퍼저가 시드 코퍼스를 피딩(feeding)할 수 있도록 테스트 프로그램 내에 존재하는 FA의 매개변수를 변수형에 따라 다르게 치환한다.
4) 퍼징 대상 파일 생성 단계에서는 테스트 프로그램을 변모시켜 자동으로 퍼징 대상 파일을 생성한다. 특히 퍼징 속도 측면의 퍼징 성능 향상을

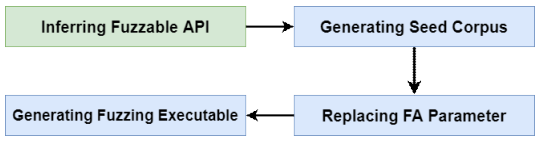


Fig. 2. Process of *FuzzBuilderEx*

위하여 불필요한 함수를 제거하는 최적화 단계의 추가를 통해, 더욱 효율적인 퍼징을 가능하게 한다.

각 단계에 대한 자세한 내용은 아래의 절에서 설명한다.

## 3.1 FA 추론

선행 연구에서는 라이브러리 퍼징을 위한 설정 파일을 생성하기 위하여, 라이브러리를 분석하고 수동으로 FA를 지정해야 한다. 이를 위해서는 사람의 많은 개입이 필요하다. 본 시스템에서는 사람이 수동으로 FA들을 선정하는 방법과 과정을 모방하여, FA를 추론하는 자동화 방법을 제안한다. Fig. 3.은 FA 추론을 하는 과정을 도식화한 것이다.

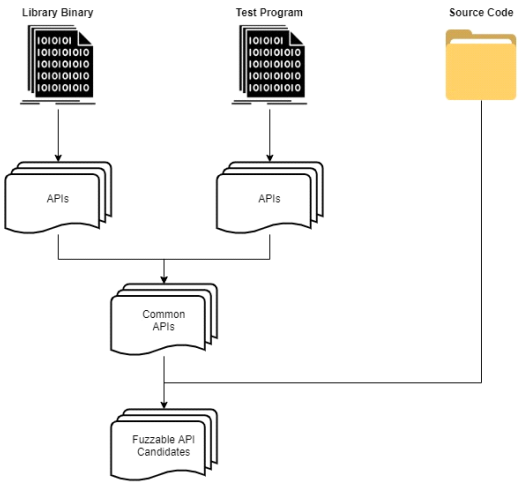


Fig. 3. Process of inferring FA

### 3.1.1 API 목록 생성

$$F_{target} \cap F_{test} \rightarrow F_{common} \quad (1)$$

퍼징을 하고자 하는 라이브러리에서 API들을 추출하여 이를 목록($F_{target}$)으로 만든다. 또한, 라이브러리를 검증하기 위한 테스트 프로그램에서도 API들을 추출하여 이를 목록($F_{test}$)으로 만든다. 이 두 바이너리에서 추출한 각각의 목록들에서 공통된 API들($F_{common}$)을 수집한다. 이렇게 하는 이유는 라이브러리에서 정의가 되어있더라도, 테스트 프로그램에서 사용되지 않는 경우 이를 FA로 사용할 수 없기에 미리 배제하기 위함이다. 그 후 두 프로그램에서 공통으로 존재하는 API들에 대하여 여러 제약사항을 적용한다.

### 3.1.2 API 정보 수집

$$F_{common} \cap S_{spec.} \rightarrow F_{FA} \quad (2)$$

앞서 선정된 API 목록($F_{common}$)에는 함수명만 존재하기에, 소스 코드($S_{spec.}$)에서 함수의 프로토타입 분석을 통한 API들의 정보를 가져와, 이를 바탕으로 FA($F_{FA}$)를 선정한다. 선정 기준으로는 인자의 유형이 지원하는 유형인지를 검사하는 등의 유효성을 확인하는 데 초점을 두고 있다.

### 3.1.3 FA 시드 설정 파일 생성

최종적으로 나온 후보군에 대해서 시드 생성을 위한 설정 파일을 생성 해주며, 이 설정 파일들을 이용하여 퍼징을 위한 시드 코퍼스를 생성할 수 있다.

## 3.2 시드 코퍼스 생성

본 연구에서는 시드 코퍼스를 생성하기 위해서 앞서 선정된 FA의 매개변수 값을 저장하는 방식을 사용한다. 즉, 테스트 데이터 및 테스트 케이스를 통해 입력되는 값이 FA의 매개변수에 전달이 되어, 실행되는 시점에 해당 값을 저장하는 방식을 통해 시드 코퍼스를 자동으로 생성한다.

이를 위해 먼저, 라이브러리의 바이트 코드에서 FA로 선정된 API를 찾은 후, 인터페이스 IR 삽입을 통해 매개변수로 들어온 값을 별도의 파일에 기록한다. 그 후 이를 분배하여 시드 코퍼스를 생성한다. 기존의 시드 생성을 위한 설정 파일은 크게 2가지 형식으로 구분할 수 있다.

Fig. 4.처럼 “char *”형을 가진 매개변수의 인덱스만을 지정하는 형식과 Fig. 5.처럼 “char *”형의 인덱스와 함께 그 길이를 가지는 “int” 형 매개변수의 인덱스를 함께 명시해 주는 형식이 있다.

여기에 더해 본 연구에서는 “int” 형과 같은 정수형 타입을 더 폭넓게 활용할 수 있는, 새로운 2가지 형식을 추가하였다. Fig. 6.처럼 기존의 2번째 형식에서 새로운 “int” 형 매개변수의 인덱스를 추가한

```
# XML_Parse(XML_Parser parser, const char *s, int len, int isFinal)
{
    "targets": [
        [ "XML_Parse", 2 ]
    ],
    "files": [ "xmlparse.bc" ]
}
```

Fig. 4. Seed configuration Type 1.

```
# XML_Parse(XML_Parser parser, const char *s, int len, int isFinal)
{
    "targets": [
        [ "XML_Parse", 2, 3 ]
    ],
    "files": [ "xmlparse.bc" ]
}
```

Fig. 5. Seed configuration Type 2.

형식과 Fig. 7.처럼 기존의 첫 번째 매개변수와 두 번째 매개변수의 인덱스를 "0, 0"으로 설정한 후 세 번째 매개변수의 인덱스를 이어서 위치시키는 형식이 있다. 3번째 형식과 기존의 2번째 형식의 가장 큰 차이점은 기존에는 두 번째 매개변수가 단순히 첫 번째 매개변수의 길이만을 담고 있었지만, 3번째 형식에서는 첫 번째 매개변수와 함께 두 번째 매개변수 역시 퍼징의 대상이 될 수 있다. 특히 4번째 형식의 경우 마지막에 있는 "int" 형 매개변수만을 대상으로 하여 단독으로 퍼징 할 수 있다.

Fig. 8.은 FA에서 테스트 입력값을 수집하기 위한 IR이 삽입된 예제이다. 삽입된 IR은 FA가 구현된 함수 내부의 맨 처음에 한 번 추가되며, 이 FA는 시드 설정 파일의 "files" 키에 지정된 라이브러리 바이트 코드에서 찾는다. 이러한 IR이 삽입된 라이브러리를 사용하는 테스트 프로그램이 실행되면, 테스트 프로그램에서 사용된 입력값들이 로그 파일에 저장되고, 저장된 입력값들을 분배하여 시드 코퍼스를 생성한다. 이렇게 자동으로 생성된 시드 코퍼스는 중복 퍼징을 유도할 수 있으므로, afl-cmin, afl-tmin 등의 시드 최적화를 적용하여 유효한 시드들을 선별한다.

```
# XML_Parse(XML_Parser parser, const char *s, int len, int isFinal)
{
    "targets": [
        [ "XML_Parse", 0, 0, 4 ]
    ],
    "files": [ "xmlparse.bc" ]
}
```

Fig. 6. Seed configuration Type 3.

```
# XML_Parse(XML_Parser parser, const char *s, int len, int isFinal)
{
    "targets": [
        [ "XML_Parse", 2, 3, 3 ]
    ],
    "files": [ "xmlparse.bc" ]
}
```

Fig. 7. Seed configuration Type 4.

```
void FA(const char *str, int len, int flag) {
    // for seed collecting
    ofstream log ("seeds.log");
    log << str;
    log.close();
    ...
}
```

Fig. 8. Inserting IR interface

### 3.3 FA 매개변수 치환

퍼저에서 제공하는 입력값을 퍼징 대상 파일에 전달하여 사용하기 위해서는 퍼저와 퍼징 대상 파일을 연결해주는 인터페이스가 필요하다. 즉, 시드에서 값을 읽어와 FA의 매개변수에 피딩을 해주는 인터페이스가 있어야 한다. 이를 위하여 테스트 프로그램에 전역 변수로 3개의 변수를 추가로 선언하는데, 이는 각각 시드에서 읽어온 값과 이 값의 크기를 가지게 된다. 또한, int를 매개변수로 사용하는 FA의 경우 이 값을 가지게 되는 또 하나의 전역 변수를 추가로 선언한다. 그러면 FA가 호출되기 전에 퍼징 대상 파일은 시드로부터 값을 읽어와서 이를 전역 변수에 저장하고, FA는 이 값을 매개변수로 사용하여 퍼징한다.

Fig. 9. 및 Fig. 10.은 FA 매개변수 값 치환을 위한 예제 코드를 설명하고 있다. Fig. 9.은 FA의 매개변수가 치환되기 전 상태의 코드로, 기존의 테스트 코드의 예제이다. Fig. 10.은 FA의 매개변수가 치환되어 시드를 피딩 하도록 Fig. 9.에 퍼징을 위한 인터페이스를 삽입한 예제 코드이다.

```
void TEST_1() {
    /* init for API_1 */
    API_1(parser, test_input, strlen(test_input), opts);
    /* check result of API_1 */
}

void TEST2() {
    /* init for API_2 */
    API_2(parser, test_input, strlen(test_input), flag));
    /* check result of API_2 */
}

int main(int argc, char **argv) {
    TEST_1();
    TEST_2();

    return 0;
}
```

Fig. 9. Before replacing FA parameter

```c
char *FUZZ_INPUT;
int FUZZ_INPUT_LEN;
int FUZZ_INT;

void TEST_1() {
    /* init for API_1 */
    API_1(parser, FUZZ_INPUT, FUZZ_INPUT_LEN, opts);
    /* check result of API_1 */
}

void TEST2() {
    /* init for API_2 */
    API_2(parser, FUZZ_INPUT, FUZZ_INPUT_LEN, flag));
    /* check result of API_2 */
}

int main(int argc, char **argv) {
    read_from_fuzzer(&FUZZ_INPUT, &FUZZ_INPUT_LEN, &FUZZ_INT);
    TEST_1();
    TEST_2();

    return 0;
}
```

Fig. 10. After replacing FA parameter

### 3.4 퍼징 대상 파일 생성

테스트 프레임워크는 테스트 대상이 되는 라이브러리를 독립적인 실행 파일로 생성하여 테스트를 수행한다. 본 연구에서는 이를 활용하여 여기서 생성된 실행 파일을 퍼징 대상 파일로 변모시킨다. 즉, 앞서 적용하였던 FA 매개변수 값 치환 등의 퍼저와 퍼징 대상과의 연결 부분을 테스트 프로그램에 적용하여 최종적으로 퍼징 대상 파일을 생성한다.

테스트 프로그램을 기반으로 자동 생성된 퍼징 대상 파일은 퍼징과는 상관없는 함수 호출이나 연산이 포함되어 있다. 이를 위해 본 연구에서는 퍼징 대상 파일을 최적화하는 단계를 추가하였다. 기존 연구에서는 단순히 FA에 들어가는 값을 시드의 값으로 치환될 수 있도록 하였기에, 실질적으로 퍼징에서 필요로 하지 않는 FA가 포함되지 않은 함수나 상태를 확인하기 위한 코드들을 포함하고 있다. 이에 함수 간의 호출 관계를 파악하여 불필요한 함수들을 제거함으로써 퍼징 대상 파일을 더 가볍고 효율적으로 만들 수 있다.

Fig. 11.은 퍼징 대상 파일을 생성하면서 최적화가 적용된 예제 코드이다. Fig. 10.은 단순히 매개변수만 피딩 되도록 변경되어 있을 뿐, 퍼징에 불필요한 함수를 포함하고 있다. Fig. 11.은 퍼징에 불필요한 코드들의 제거를 통해 퍼징 대상 파일의 크기를 줄여, 퍼징 속도를 향상할 수 있다. 여기서 삭제되는 불필요한 코드들이란 1) FA를 포함하고 있지 않은 테스트 함수, 2) 퍼징에 사용하기엔 너무 무거운 테스트 함수, 3) 퍼징에 직접 사용되지 않는 코드들을 말한다.

```c
char *FUZZ_INPUT;
int FUZZ_INPUT_LEN;
int FUZZ_INT;

void TEST2() {
    /* init for API_2 */
    API_2(parser, FUZZ_INPUT, FUZZ_INPUT_LEN, flag));
    /* check result of API_2 */
}

int main(int argc, char **argv) {
    read_from_fuzzer(&FUZZ_INPUT, &FUZZ_INPUT_LEN, &FUZZ_INT);
    TEST_2();

    return 0;
}
```

Fig. 11. After optimizing executable

이러한 불필요한 코드들을 제거하기 위한 퍼징 대상 파일의 설정 파일은 Fig. 12.와 같다.

```
{
    "targets"   : [ [ "XML_Parse", 2, 3 ] ],
    "tests"     : [ "test_" ],
    "files"     : [ "runtests.bc" ],
    "skips"     : [ "test_alloc_nested_groups", ... ]
}
```

Fig. 12. Exec configuration

1) FA를 포함하고 있지 않은 테스트 함수
: 시드 설정 파일과 같이 FA를 지정해 주는 "targets" 키가 있으며, 테스트 함수의 매크로를 지정해 주는 "tests" 키가 있다. 테스트 코드에는 여러 개의 테스트 함수들이 포함되어 있으며, 각각의 테스트 함수들은 독립적으로 동작한다. 따라서 테스트 함수 중에서 FA가 포함된 함수는 보존하며, FA를 포함하고 있지 않은 함수는 제거한다.

2) 퍼징에 사용하기엔 너무 무거운 테스트 함수
: "skips" 키를 통해 무거운 작업이 포함된 테스트 함수를 명시적으로 지정하여 이를 제거할 수도 있다. 무거운 작업에는 과도한 중첩 반복문, 메모리 읽기/쓰기 등 퍼징 속도에 큰 영향을 줄 수 있는 작업들을 포함시킬 수 있다.

3) 퍼징에 직접 사용되지 않는 코드
: 테스트 코드의 특성상 테스트 함수의 결과 비교를 위한 테스트 매크로(ASSERT, EXPECT)가 사용된다. 하지만 퍼징에서 이 테스트 매크로는 필요가 없으므로, 퍼징 대상 파일에서 이를 제거해줄 필요가 있다. 이에 본 연구에서는 이를 제거하면서도, 퍼징 과정에는 영향을 주지 않도록 퍼징 대상 파일을 최적화한다.

Fig. 13.은 퍼징에 직접 사용되지 않는 테스트 매크로를 제거하기 위한 설정 파일이다. “skips” 키에 명시된 테스트 매크로를 “files” 키에 명시된 소스 코드에서 이를 제거한다.

```
{
    "skips" : [ "EXPECT_EQ" ],
    "files" : [ "test_utils.cpp", "test_bstr.cpp" ]
}
```

Fig. 13. Optimization configuration for fuzzing executable

## IV. 실험 결과

본 실험에서는 *FuzzBuilderEx*의 효율성과 편리성 검증을 위해 OSSFuzz, 그리고 선행 연구인 FuzzBuilder와의 실험 결과를 비교한다. 실험을 통해 검증하고자 하는 항목은 다음과 같다.

- *FuzzBuilderEx*는 퍼징을 위한 사람의 개입을 얼마나 최소화해주는가? (4.1절에 기술)
- 자동으로 생성된 퍼즈 드라이버는 얼마나 효과적인가? (4.2절 및 4.3절에 기술)
- *FuzzBuilderEx*는 퍼징 수행에 있어 실질적으로 의미 있는가? (4.4절 및 4.5절에 기술)

OSSFuzz와의 비교를 위하여 OSSFuzz가 프로젝트별로 선정한 FA와 같게 퍼징 대상 파일을 생성하였다. 단, 1) 프로젝트 내에 테스트 코드가 없거나, 2) 테스트 코드에 FA와 관련된 테스트 케이스가 없는 경우, 본 연구에서는 퍼징 환경을 자동 생성할 수 없으므로 검증 대상에서 제외하였다. 이러한 기준으로 본 연구에서는 검증을 위한 총 9개의 오픈소스 프로젝트에 *FuzzBuilderEx*를 적용하였다. 이 중에 7개의 프로젝트는 성능 비교를 위해 OSSFuzz와 *FuzzBuilderEx* 모두 적용하였고, 나머지 2개 프로젝트는 취약점 분석 목적으로 *FuzzBuilderEx*만 적용하였다. 실험에 적용한 프로젝트 목록은 Table 1.과 같다.

실험을 위해 사용한 퍼저는 대표적인 그레이박스 퍼징 도구인 AFL 2.52b를 사용하였고 또한 AddressSanitizer도 함께 사용하였다. AFL은 64bit 실행 파일에 대한 AddressSanitizer를 지원하지 않기 때문에, 평가를 위한 검증 대상은 모두 32bit로 빌드하였다.

Table 1. Names of target project with its release version

| Project | Version (commit ID) |
|---|---|
| BoringSSL | v3945 (d2a0ffd) |
| c-ares | v1.15.0 (a9c2068) |
| cJSON | v1.7.8 (08103f0) |
| Expat | v2.2.6 (39e487d) |
| http-parser | v2.9.4 (ec8b5ee) |
| JSON-C | v0.15 (df27756) |
| LibHTP | v0.5.35 (e198e8f) |
| mpc | v1.8.4 (b31e02e) |
| YARA | v3.8.1 (a3784d3) |

### 4.1 퍼징 절차별 자동화 비교

퍼징 준비 과정을 자동화해줌에 따라 얼마나 사람의 개입을 절감시켜주는지 평가하기 위해 퍼징의 절차를 나열하고 각 항목에 대한 사람의 개입이 필요한지 아닌지를 판단하여 각 항목에 대한 자동화 수준을 비교한다. 단 여기서 사람은 검증자뿐만 아니라 개발자도 될 수 있으며, 개발자 또는 검증자에 따른 난이도 차이가 분명하기에 이를 분리하여 기술한다.

Table 2.는 *FuzzBuilderEx*가 OSSFuzz, FuzzBuilder 대비 라이브러리를 퍼징 하기 위해 요구되는 사람의 개입에 대해서 자동화가 되어있는지를 비교한 표이다. 퍼징을 수행하면서 필요한 절차로 대상이 되는 프로젝트를 선정하고, 프로젝트를 분석하여 퍼징의 대상이 되는 FA를 선정해야 한다. 또한, 퍼징의 효율성을 높이기 위해 시드도 직접 작성해야 하고, 이러한 과정이 끝나면 퍼징 대상 파일을 생성하여 퍼징을 수행한다. 절차별 자동화 유무에 따라 O, X로 구분하였으며, 퍼징을 수행하면서 필요한 시간, 전문지식 등의 사람 개입이 많이 요구되는 정도에 따라 난이도를 상, 중, 하로 나누었다. 개발자의 경우 프로젝트에 대한 도메인 지식이 풍부하여 프로젝트 분석 및 FA 선정 단계와 퍼즈 드라이버 생성 단계는 쉬울 수 있으나 퍼징에 대한 도메인 지식이 부족하여 퍼징 단계에서는 어려움이 있다. 반대로 검증자의 경우 퍼징에 대한 도메인 지식이 풍부하기에 퍼징 단계는 쉬울 수 있으나 프로젝트를 분석하여 FA를 선정하고 퍼징 드라이버를 생성하는 데 어려움이 있다. *FuzzBuilderEx*는 검증자 기준뿐만 아니라 개발자 기준으로도 난도가 높은 작업을 모두 자동화해주고 프로젝트 선정 및 생성된 퍼즈 드라이버

Table 2. Process of Fuzzing with its difficulty

| Fuzzing Process | Difficulty | | Automation | | |
|---|---|---|---|---|---|
| | Developer | Tester | OSSFuzz | FuzzBuilder | FuzzBuilderEx |
| Analyze project and Select FA | Low | Mid | X | X | O |
| Generate Seed Corpus | Mid | High | X | O | O |
| Generate Fuzzing Executable | Low | High | X | O | O |
| Fuzzing Test | High | Low | X | X | X |

로 퍼징만 진행하면 되기 때문에 검증자와 개발자는 매우 효율적으로 퍼징을 진행할 수 있음을 보여준다.

### 4.2 자동 생성된 시드 코퍼스의 효율성

*FuzzBuilderEx*를 통해 자동 생성된 시드 코퍼스의 효율성을 검증하기 위하여 프로젝트별 선정된 FA들에 대한 각각의 시드 코퍼스를 생성하여, 커버리지를 비교하였다. 공정한 비교를 위해 퍼징의 대상이 되는 퍼징 대상 파일은 OSSFuzz에서 생성한 퍼징 대상 파일을 같게 사용하고, 시드 코퍼스만 OSSFuzz, FuzzBuilder, *FuzzBuilderEx*에서 각각 생성한 것을 사용하였다. 시드 생성의 기준이 되는 FA의 선정 또한 공정한 비교를 위해 OSSFuzz에서 FA로 사용하고 있는 것으로 제한하였다.

Table 3.은 실험에 사용한 프로젝트별 FA이다.

Table 3. List of FA

| Project | FA |
|---|---|
| BoringSSL | CBS_init<br>BIO_new_mem_buf<br>SSL_SESSION_from_bytes<br>d2i_AutoPrivateKey<br>d2i_X509 |
| c-ares | ares_create_query<br>ares_parse_* |
| Expat | XML_Parse<br>XML_Parse_buffer |
| http-parser | http_parer_execute<br>http_parser_parse_url |
| JSON-C | json_tokenr_parse_ex |
| LibHTP | http_connp_open<br>htp_connp_req_data<br>htp_connp_res_data |
| YARA | yr_rules_scan_mem<br>yr_compiler_add_string |

참고로 c-ares 프로젝트의 "ares_parse_*"는 "ares_parse_"로 시작하는 9개의 FA를 줄여서 표현한 것이다. 프로젝트별로 여러 개의 퍼징 대상 파일이 있으므로, 각 퍼징 대상 파일에 대한 퍼징 수행 후 gcov 도구를 사용해서 결과를 통합하였다. 커버리지는 라인 커버리지를 기준으로 한다.

Fig. 14.는 OSSFuzz가 제공하는 시드 코퍼스와 FuzzBuilder, *FuzzBuilderEx*가 자동으로 생성한 시드 코퍼스를 이용하여 측정한 커버리지의 결과이다. OSSFuzz와의 결과를 비교하면, 총 7개의 프로젝트 중 1개 프로젝트를 제외한 6개의 프로젝트에서 코드 커버리지가 높게 측정이 되었으며, 선행연구와 비교하면 7개의 모든 프로젝트에서 높은 커버리지를 달성함을 확인할 수 있다.

테스트 코드는 개발자가 개발 과정에서 검증을 위해 작성을 하므로 일반적으로 퍼징 테스트보다는 다양하고 보다 복잡한 검증을 수행한다. 따라서 대상으로 선정한 FA 및 이와 관련된 함수 시퀀스가 테스트 코드에 구현된 경우 이를 테스트하기 위한 입력값이 퍼징 테스트보다 상대적으로 다양하며 그 종류가 더 많이 존재한다. 따라서 OSSFuzz와의 결과를 비교하면, 1개 프로젝트를 제외한 6개 프로젝트 모두

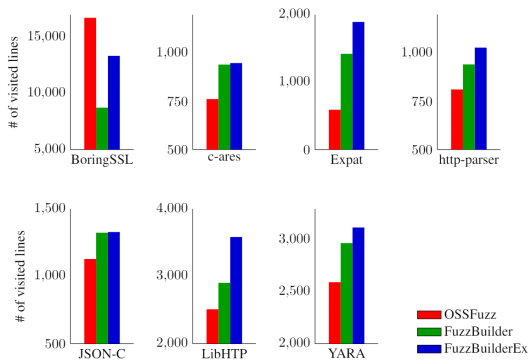


Fig. 14. Comparison line coverage using same seed corpus

에서 높은 커버리지를 달성할 수 있었다. 그러나 BoringSSL 프로젝트의 경우 비교적 커버리지가 낮게 측정되었는데, 이는 해당 퍼징 대상 파일에서의 FA 특성과 FA를 활용하는 테스트 코드와 관련이 있다. "CBS_init"은 라이브러리가 정의한 특정한 메모리 구조에 값을 저장하는 역할을 하며, 이후에 어떤 함수가 불리느냐에 따라서 해당 데이터의 사용법이 결정된다. 그러나 BoringSSL의 테스트 코드에는 이에 대한 함수 시퀀스가 일부만 구현되어 있으므로 낮은 커버리지를 달성하였다. 따라서 프로젝트가 추가로 개발됨에 따라 테스트 코드가 추가되고, 위의 FA에 대한 시퀀스가 보강된다면, 본 연구를 통해 생성된 시드를 사용하여 더 높은 커버리지를 달성할 수 있을 것이라 예상한다.

선행 연구와 결과를 비교하면, 7개 프로젝트 모두에서 높은 커버리지를 달성할 수 있음을 확인하였다. 이는 본 연구에서 커버리지 확장을 위해 FA에 정수형 매개변수를 포함할 수 있도록 추가하였기 때문에 기존 연구보다 더 많은 시드와 함수 시퀀스를 포함하여 검증할 수 있었다. 다만, c-ares와 JSON-C의 경우에는 커버리지 상승 폭이 크지 않았는데, 이는 선정된 FA에 정수형 매개변수를 활용하는 시퀀스가 많이 포함되어 있지 않기 때문이다. 다만, 이러한 결과도 프로젝트가 개발됨에 따라 테스트 코드가 추가되어 정수형 매개변수를 활용하는 시퀀스가 추가된다면, 다른 프로젝트의 결과와 마찬가지로 커버리지의 향상을 기대할 수 있다. 마지막으로 BoringSSL의 경우에는 기존 연구에서는 OSSFuzz와 커버리지 차이가 큰 폭으로 낮은 문제가 있었으나, FA 매개변수 확장으로 인해 OSSFuzz와 커버리지 차이를 상당 수준 감소시킬 수 있었다.

본 실험에서는 생성된 시드 코퍼스의 효율성을 평가하기 위한 목적으로, FA 선정을 OSSFuzz에서 사용하고 있는 FA로 제한하였다. 그러나 *FuzzBuilderEx*의 경우 Fig. 6.에 나열한 FA 이외에, 테스트 코드에 있는 FA를 추가로 선정할 수 있으며, 본 연구에서는 이러한 FA를 앞서 설명한 FA_selector를 통해 자동으로 추출할 수 있도록 하였다. Table 4.는 *FuzzBuilderEx*를 통해 자동으로 추출한 프로젝트별 FA의 개수를 보여준다.

Table 4. The number of FAs in each project

| Project | OSSFuzz | Fuzz Builder | FuzzBuilderEx |
|---|---|---|---|
| BoringSSL | 5 | 3 | 32 |
| c-ares | 10 | 10 | 21 |
| Expat | 2 | 2 | 7 |
| http-parser | 2 | 2 | 6 |
| JSON-C | 1 | 1 | 14 |
| LibHTP | 3 | 3 | 21 |
| YARA | 2 | 2 | 18 |

### 4.3 자동 생성된 퍼징 대상 파일의 효율성

*FuzzBuilderEx*를 통해 자동 생성된 퍼징 대상 파일의 효율성을 검증하기 위하여 각 퍼징 시 발생한 크래시의 개수와 커버리지를 측정하여 비교하였다. 시드 코퍼스의 효율성 평가와 마찬가지로 공정한 평가를 위해 앞서 *FuzzBuilderEx*를 사용하여 생성한 시드 코퍼스를 사용하는 것으로 하였다. 4.2절에서 설명한 바와 같이, 본 실험에서 FA 선정은 공정한 비교를 위해 OSSFuzz에서 FA로 포함하고 있는 것을 대상으로 제한하였기 때문에 *FuzzBuilderEx*가 생성한 시드 코퍼스 또한 OSSFuzz에도 같게 적용할 수 있다.

Table 5.는 퍼징의 대상이 되는 퍼징 대상 파일의 이름을 나타낸다. OSSFuzz는 프로젝트에 포함된 퍼징 대상 파일을 기준으로 생성하였고, *FuzzBuilderEx*는 FA 단위로 퍼징 대상 파일을 생성하였다. FuzzBuilder와 *FuzzBuilderEx*는 모두 같은 FA를 대상으로 퍼징 대상 파일을 생성하였기 때문에 같은 수의 퍼징 대상 파일을 갖는다. c-ares 프로젝트의 "ares_parser_*_fuzzer"의 경우 9개의 "ares_parser_*" 이름으로 시작하는 FA에 대한 퍼징 대상 파일명을 의미한다. OSSFuzz와 퍼징 대상 파일의 개수가 차이가 나므로 공정한 비교를 위해서 AFL의 병렬모드를 이용하였다. 예를 들어, 퍼징 대상 파일이 2개 4개로 차이가 나는 경우, 4개의 인스턴스를 갖는 AFL 병렬모드로 실행하고, 1) 퍼징 대상 파일 4개인 경우, 4개의 인스턴스에 각각 다른 퍼징 대상 파일을 실행하고, 2) 퍼징 대상 파일이 2개인 경우, 4개의 인스턴스에 각각 2번씩 퍼징 대상 파일을 실행하여 같은 수준에서 퍼징이 수행되도록 실험 설계를 하였다.

Table 6.은 OSSFuzz와 FuzzBuilder 그리고 *FuzzBuilderEx*를 통해 각각 생성한 퍼징 대상 파일을 대상으로 크래시 발생 개수와 커버리지를 측정한 결과이다. 앞선 시드 코퍼스 효율성 평가 방법과 같이 7개 프로젝트를 대상으로 측정하였으며,

Table 5. List of fuzzing executable in each project

| Project | OSSFuzz | FuzzBuiler(Ex) |
|---|---|---|
| BoringSSL | vn_div<br>vn_mod_exp<br>client<br>dtls_client<br>dtls_server<br>pkcs8<br>pkcs12<br>read_pem<br>server<br>session<br>spki<br>ssl_ctx_api | CBS_init_fuzzer<br>BIO_new_mem_buf_fuzzer<br>SSL_SESSION_from_bytes_fuzzer<br>d2i_AutoPrivateKey_fuzzer<br>d2i_X509_fuzzer |
| c-ares | ares_create_query_fuzzer<br>ares_parse_reply_fuzzer | ares_create_query_fuzzer<br>ares_parse_buffer_fuzzer |
| Expat | parser_ISO_8859_1_fuzzer<br>parse_US_ASCII_fuzzer<br>parse_UTF_16BE_fuzzer<br>parse_UTF_16_fuzzer<br>parse_UTF_16LE_fuzzer<br>parse_UTF_8_fuzzer | XML_Parse_fuzzer<br>XML_Parse_buffer_fuzzer |
| http-parser | fuzz_parser<br>fuzz_url | http_parer_execute_fuzzer<br>http_parser_parse_url_fuzzer |
| JSON-C | tokenr_parser_ex_fuzzer | json_tokenr_parse_ex_fuzzer |
| LibHTP | fuzz_htp | http_connp_open_fuzzer<br>htp_connp_req_data_fuzzer<br>htp_connp_res_data_fuzzer |
| YARA | dex_fuzzer<br>dotnet_fuzzer<br>elf_fuzzer<br>pe_fuzzer<br>rules_fuzzer | yr_rules_scan_mem_fuzzer<br>yr_compiler_add_string_fuzzer |

OSSFuzz 결과와 비교하면 총 6개 프로젝트에서 크래시와 커버리지 모두 우수한 결과를 도출할 수 있었다. 다만, BoringSSL의 경우 앞선 “CBS_init”과 같은 이유로 다양한 함수 시퀀스에 대한 테스트 코드가 부족하여 OSSFuzz 대비 커버리지와 크래시 모두 낮은 성능을 보였다. 또한, JSON-C도 커버리지는 OSSFuzz 대비 큰 값을 나타내지만, 크래시 발생 횟수는 10개로 같은 성능을 나타내었다. JSON-C의 테스트 코드를 분석해 보면, 테스트 코드 내에 ASSERT 구문을 사용하여 특정 조건이 만족하였는지 여부를 확인하는 코드가 반복적으로 사용됨을 알 수 있다. 해당 부분도 프로젝트를 개발함에 따라 테스트 코드가 추가되거나 개선되면 퍼징의 효율성도 같이 증가할 것을 기대할 수 있다.

선행 연구와 결과를 비교하면, 7개의 모든 프로젝트에서 크래시 발생 개수와 커버리지 모두 성능이 향상됨을 확인할 수 있다. 다만, 시드 코퍼스의 실험 결과와 유사하게 c-ares와 JSON-C 프로젝트의 경우에는 크래시와 커버리지 모두 성능 향상 폭이 크지 않은데, 이는 앞선 원인과 유사하게, FA 매개변수 확장과 퍼징 대상 파일 최적화 관련한 내용이 테스트 코드 내에 많지 않기 때문으로 판단한다. 성능 향상의 원인은 FA의 매개변수 확장과 퍼징 대상 파일의 최적화 적용으로 인해 퍼징의 효율성이 증가하였기

Table 6. Results of crash and coverage about fuzzing executable

| | BoringSSL | | c-ares | | Expat | | http-parser | | JSON-C | | LibHTP | | YARA | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | crash | cover-age | crash | cover age | crash | cover age | crash | cover age | crash | cover age | crash | cover age | crash | cover age |
| OSS Fuzz | **43** | **15,208** | 3 | 729 | 2 | 538 | 4 | 763 | **10** | 1,173 | 11 | 2,351 | 17 | 2,565 |
| Fuzz Builder | 28 | 8,292 | 7 | 912 | 10 | 1,491 | 4 | 928 | 9 | 1,321 | 18 | 2,912 | 16 | 2,814 |
| Fuzz Builder Ex | 39 | 13,162 | **9** | **926** | **21** | **2,125** | **11** | **1,291** | **10** | **1,328** | **25** | **3,274** | **31** | **3,310** |

때문이다. 즉, FA의 매개변수 확장으로 인해 퍼징 가능한 영역과 시나리오가 확장되어 커버리지가 증가하고, 향상된 커버리지로 인해 크래시의 개수도 증가할 수 있다. 또한, 퍼징 대상 파일에서 퍼징에 불필요한 부분을 제거함으로써 퍼징의 효율성이 증가하여, 의미 있는 영역을 보다 집중적으로 퍼징 할 수 있게 되어 그에 따른 크래시 발생 개수가 증가할 수 있다.

### 4.4 취약점 분석 결과

앞선 퍼즈 드라이버의 효율성을 검증하기 위한 실험은 FA를 OSSFuzz 기준으로 제한하여 실험을 진행하였다. 이에 이번 절에서는 OSSFuzz에 포함된 cJSON 프로젝트와 OSSFuzz에 포함되지 않는 mpc 프로젝트 선정하여 실제 취약점을 찾을 수 있는지 확인하기 위한 추가 실험을 진행하였다. 파싱(parsing) 관련 라이브러리 중 가장 빈번하게 사용되는 프로젝트 중 테스트 코드를 프로젝트 내에 포함하고 있는 것을 선정 기준으로 하였고, 이렇게 선정된 2개 프로젝트는 FA의 제약 없이 사용 가능한 모든 FA를 자동 선정하여 실험에 적용하였다.

Table 7.은 프로젝트별 취약점 분석 결과를 보여준다. 2개 프로젝트에서 총 3개의 취약점을 발견하였고, mpc의 2개 취약점은 각각 Stack Buffer Overflow와 Heap Buffer Overflow 취약점으로, 개발자의 검증 이후 다음 버전 배포 시에 패치되었으며, cJSON의 경우 NULL Dereference 취약점을 발견하여 신규 CVE로 등록되었다. 무엇보다도 cJSON에서 신규로 발견한 취약점의 경우 OSSFuzz를 통해 발견하지 못한 취약점을 *FuzzBuilderEx*를 통해 찾았다는 점에서 의미가 크다. 해당 버그는 OSSFuzz가 사용한 퍼징 대상 파일에 정의되어 있지 않은 함수 시퀀스가 우리의 접근법을 바탕으로 생성한 퍼징 대상 파일에 존재하기 때문에 찾은 취약점이다. 또한, mpc 프로젝트는 기존에는 퍼징 적용이 어려웠던 프로젝트에 *FuzzBuilderEx*를 통해 퍼징을 적용하여 찾은 취약점이라는 점에서 의미가 있다.

### 4.5 실험 결과 요약

본 연구에서는 실험을 통해 *FuzzBuilderEx*의 효과에 대해서 검증을 진행하였다. 먼저, 본 연구가 제안하는 퍼징 자동화를 통해 사람의 개입을 얼마나 감소시켰는지를 퍼징의 절차를 구분하여, 절차별로 자동화 가능 여부를 분석하였다. 특히, 본 연구는 사람의 개입을 가장 많이 필요로 하고, 시간이 많이 소요되는 FA 선정과 퍼즈 드라이버 생성 작업을 자동화해서 선행 연구 대비 큰 수준의 개선 효과를 달성할 수 있었다.

본 연구의 실험에서는 총 7개의 오픈 소스 프로젝

Table 7. Results of vulnerability analysis

| Project | Vulnerability Classification | CVE |
|---|---|---|
| mpc | Stack Buffer Overflow<br>Heap Buffer Overflow | Silent Patch<br>Silent Patch |
| cJSON | Null Dereference | CVE-2019-1010239 |

트에 *FuzzBuilderEx*를 적용하여 퍼즈 드라이버의 효율성을 평가하였다. 본 연구를 통해 자동 생성한 시드 코퍼스는 기존 연구 대비 약 31.2% 수준의 커버리지 향상이 있었다. 이는 기존 연구 대비 FA의 매개변수 확장을 통해 퍼징 가능 범위가 확장되었기 때문으로 예측한다. 그뿐만 아니라 자동 생성된 퍼징 대상 파일도 크래시 발생 개수를 기준으로 기존 연구 대비 약 58.7% 향상이 있었다. 이는 FA의 매개변수 확장뿐 아니라, 퍼징 대상 파일의 최적화를 통해 퍼징에 불필요한 부분을 최소화하여 좀 더 의미 있는 영역을 집중적으로 퍼징 할 수 있도록 하였기 때문이다. 마지막으로 위에서 적용했던 프로젝트 이외의 2개 프로젝트를 추가로 선정하여 취약점 분석을 진행하여, 3개의 취약점을 분석할 수 있었다. 이러한 실험 결과를 통해 우리는 *FuzzBuilderEx*가 사람의 개입을 최소화하여 라이브러리의 퍼징을 쉽게 해줄 수 있고, 또한 자동으로 생성한 퍼즈 드라이버를 통해 효율적인 퍼징이 가능함을 확인할 수 있었다. 그뿐만 아니라 실제 취약점 분석에도 효과가 있음을 증명하였다.

## V. 향후 연구

*FuzzBuilderEx*는 라이브러리 퍼징에 광범위하게 적용할 수 있고, 취약성을 감지하는 데 효과적이지만 향후 개선할 여지는 여전히 남아 있다. 이 장에서는 *FuzzBuilderEx*에 대한 제한 사항을 작성하고, 향후 연구를 위한 개선 사항에 관해 설명한다.

### 5.1 복합적인 API 사용 시나리오

*FuzzBuilderEx*는 라이브러리의 테스트 프로그램을 정적/동적 분석하여 퍼즈 드라이버를 자동으로 생성한다. 그러나 퍼즈 드라이버 생성을 위한 분석 시 테스트 프로그램의 기능을 제한적으로 사용한다. 예를 들어, 테스트 프로그램에는 독립적인 테스트 코드를 순차적으로 수행하는 결정론적 수행 기능뿐만 아니라, 무작위로 테스트 코드를 실행시키는 확률론적 수행 기능이나 결함을 주입하여 테스트를 실행시키는 기능 등을 포함하고 있다. 이러한 기능을 이용하여 퍼징 대상 파일을 생성하면 좀 더 다양한 라이브러리의 상태를 재현하고, 올바른 API 호출 시퀀스를 포함하여 전반적인 성능이 향상되고 더 많은 취약점을 효율적으로 찾을 수 있다.

### 5.2 분석 대상 확장

현재 *FuzzBuilderEx*는 C/C++로 개발된 라이브러리만을 대상으로 한다. 본 연구에서는 자동화를 위해서 테스트 프레임워크와 LLVM IR을 이용하였기 때문에 다른 프로그램 언어로의 확장이 비교적 쉬울 수 있으나, 언어적인 특성을 고려하기 위해서 추가 연구를 수행해야 한다. 더욱이 *FuzzBuilderEx*의 현재 설계는 프로그램의 소스 코드를 사용할 수 있다는 가정하에 퍼즈 드라이버를 자동으로 생성하는 것으로 제한된다. 라이브러리 바이너리를 기반으로 퍼즈 드라이버를 자동으로 생성하는 것은 더 어렵고 도전적인 문제이다.

### 5.3 테스트 코드 생성 자동화

*FuzzBuilderEx*는 전적으로 라이브러리의 테스트 프로그램에 의존하여 퍼즈 드라이버를 생성한다. 하지만 라이브러리 내에 존재하는 테스트 코드가 높은 수준의 테스트를 포함하지 않거나, 테스트 코드가 부족한 경우에는 본 연구를 통해 자동 생성된 시드 코퍼스는 퍼징의 효율성을 높이기는 어렵다는 한계가 있다. 따라서 부족한 테스트 코드를 보강하기 위하여 컨슈머 프로그램의 패턴을 적용해 볼 수 있다. 컨슈머 프로그램이란 라이브러리에 의존하고 있는 별도의 프로그램을 말하며, 해당 프로그램에서의 라이브러리 사용법을 기반으로 테스트 코드를 생성할 수 있도록 한다. 이러한 방법 외에도 콘콜릭 테스팅(Concolic Testing)[33] 을 적용하여 자동으로 테스트 코드를 생성할 수 있도록 한다.

## VI. 결 론

많은 보안 커뮤니티에서 소프트웨어 취약성을 발견하기 위한 자동화된 접근 방식으로 퍼징에 관한 연구를 진행했음에도 불구하고, 퍼징에는 실행 환경, 특히 퍼징을 위한 퍼징 대상 파일이 필요하므로 많은 최신의 퍼징 방법이 라이브러리에는 제대로 적용되지 못했다. 이러한 문제를 해결하기 위해 테스트 프레임워크를 기반으로 라이브러리용 퍼즈 드라이버를 자동으로 생성하는 시스템인 *FuzzBuilderEx* 시스템을 제안하였다. 특히, 본 연구진은 테스트 프레임워크를 사용하여 라이브러리용 퍼즈 드라이버를 생성하는 최초의 연구를 제안한 이후, 퍼징 성능과 효율성을 극

적으로 개선한 접근 방식을 제안하였다.

*FuzzBuilderEx*에서 생성된 퍼징 대상 파일은 AFL과 같은 다른 퍼저에서 사용할 수 있어 다양한 베이스 퍼저의 사용을 통한 확장이 가능하고, 퍼징을 위한 소스 코드의 작성 및 유지 관리를 제거하여 개발자에 의한 퍼징을 쉽게 한다. 또한, 이 방법을 사용하면 검증자가 프로젝트에 대한 고수준의 지식 없이 퍼징을 수행할 수 있으며, 개발자 또한 퍼징에 대한 고수준의 퍼징을 수행할 수 있다. 코드 커버리지 비교 결과 *FuzzBuilderEx*는 라이브러리에 대한 최소한의 이해로 라이브러리 퍼징을 위한 다양한 라이브러리 API 함수를 호출할 기회를 제공한다. 또한, 자동으로 생성한 시드 코퍼스를 실험한 결과는 *FuzzBuilderEx*에서 생성된 시드 코퍼스가 실용성을 만족시키면서 효율적인 퍼징에 도움이 될 수 있음을 보여준다. 따라서 *FuzzBuilderEx*는 다양한 유형의 라이브러리에 큰 노력 없이 퍼징을 적용하여 소프트웨어 보안을 강화하는 데 도움이 될 수 있다.

테스트 프레임워크를 사용하는 오픈 소스 프로젝트인 9개의 라이브러리에서 *FuzzBuilderEx*를 평가했다. *FuzzBuilderEx*는 관련 연구와의 비교에서 퍼징 속도와 탐지 능력 측면에서 가장 뛰어난 성능을 나타냄을 확인할 수 있었다. 또한, *FuzzBuilderEx*에서 생성한 퍼즈 드라이버는 3개의 취약점을 발견하고 1개의 CVE를 받았다. *FuzzBuilderEx*의 소스 코드[34]는 GitHub에 제공되므로 다른 연구자들이 우리의 결과를 재현하고 자동화된 퍼즈 드라이버 생성 기술을 확장할 수 있다.

## References


[1] 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.

[2] V.J.M. Manès, H. Han, C. Han, S.K. Cha, M. Egele, E.J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, Oct. 2019.

[3] AFL, "american fuzzy lop," https://lcamtuf.coredump.cx/afl/, Accessed: Feb. 2021.

[4] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, Dec. 2017.

[5] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2329–2344, Oct. 2017.

[6] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 679–696, IEEE, Jul. 2018.

[7] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution.," *NDSS*, vol. 16, pp. 1–16, Jan. 2016.

[8] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing.," *NDSS*, vol. 17, pp. 1–14, Feb. 2017.

[9] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: fuzzing by program transformation," *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 697–710, IEEE, Jul. 2018.

[10] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "{QSYM}: A practical concolic execution engine tailored for hybrid fuzzing," *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pp. 745–761, Aug. 2018.

[11] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," *2018 IEEE Symposium on Security*

*and Privacy (SP)*, pp. 711 - 725, IEEE, Jul. 2018.

[12] D. Babić, S. Bucur, Y. Chen, F. Ivančić, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang, "Fudge: fuzz driver generation at scale," *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 975 - 985, Aug. 2019.

[13] K. Ispoglou, D. Austin, V. Mohan, and M. Payer, "Fuzzgen: Automatic fuzzer generation," *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pp. 2271 - 2287, Aug. 2020.

[14] J. Jang and H.K. Kim, "Fuzzbuilder: automated building greybox fuzzing environment for c/c++ library," *Proceedings of the 35th Annual Computer Security Applications Conference,* pp. 627 - 637, Dec. 2019.

[15] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 475 - 485, Sep. 2018.

[16] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: program-state based binary fuzzing," *Proceedings of the 2017 11th Joint Meeting on Foundationsof Software Engineering*, pp. 627 - 637, Aug. 2017.

[17] J. Choi, J. Jang, C. Han, and S.K. Cha, "Grey-box concolic testing on binary code," *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 736 - 747, IEEE, May 2019.

[18] V. Jain, S. Rawat, C. Giuffrida, and H. Bos, "Tiff: using input type inference to improve fuzzing," *Proceedings of the 34th Annual Computer Security Applications Conference,* pp. 505 - 517, Dec. 2018.

[19] N. Coppik, O. Schwahn, and N. Suri, "Memfuzz: Using memory accesses to guide fuzzing," *2019 12th IEEE Conference on Software Testing, Validation and Verification(ICST)*, pp. 48 - 58, IEEE, Apr. 2019.

[20] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: Mining and Recommending API Usage Patterns," *European Conference on Object-Oriented Programming*, pp. 318 - 343, Jul. 2009.

[21] M.A. Saied, O. Benomar, H. Abdeen, and H. Sahraoui, "Mining multi-level api usage patterns," *2015 IEEE 22nd international conference on software analysis, evolution, and reengineering (SANER)*, pp. 23 - 32, IEEE, Mar. 2015.

[22] J. Fowkes and C. Sutton, "Parameter-free probabilistic api mining across github," *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, pp. 254 - 265, Nov. 2016.

[23] J.E. Montandon, H. Borges, D. Felix, and M.T. Valente, "Documenting apis with examples: Lessons learned with the api miner platform," *2013 20th working conference on reverse engineering (WCRE)*, pp. 401 - 408, IEEE, Oct. 2013.

[24] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, and A. Marcus, "How can i use this method?," *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, pp. 880 - 890, IEEE, May 2015.

[25] N. Katirtzis, T. Diamantopoulos, and C.A. Sutton, "Summarizing software

api usage examples using clustering techniques.," *FASE*, pp. 189 - 206, Apr. 2018.

[26] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," *29th International Conference on Software Engineering (ICSE'07)*, pp. 75 - 84,IEEE, May 2007.

[27] W. Zheng, Q. Zhang, M. Lyu, and T. Xie, "Random unit-test generation with mut-aware sequence recommendation," *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pp. 293 - 296, Sep. 2010.

[28] S. Zhang, D. Saff, Y. Bu, and M.D. Ernst, "Combined static and dynamic automated test generation," Proceedings of the 2011 International Symposium on Software Testing and Analysis, pp. 353 - 363, Jul. 2011.

[29] M. Pradel and T.R. Gross, "Leveraging test generation and specification mining for automated bug detection without false positives," *2012 34th International Conference on Software Engineering (ICSE)*, pp. 288 - 298, IEEE, Jun. 2012.

[30] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," *2017 IEEE Symposium on Security and Privacy (SP)*, pp. 579 - 594, IEEE, May 2017.

[31] C. Lyu, S. Ji, Y. Li, J. Zhou, J. Chen, and J. Chen, "Smartseed:Smart seed generation for efficient fuzzing," *arXiv preprint arXiv:1807.02606*, Jul. 2018.

[32] I.J. Goodfellow, J. PougetAbadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," *arXiv preprint arXiv:1406.2661*, Jun. 2014.

[33] K. Sen, "Concolic testing," in Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering, ASE '07, pp. 571 - 572, Nov. 2007

[34] GitHub, "FuzzBuilderEx", http://github.com/kppw99/FuzzBuilderEx, Accessed: Apr. 2021.

## 〈저 자 소 개〉

류 민 수 (Minsoo Ryu) 학생회원
2020년 8월: 세종대학교 정보보호학과 학사
2020년 9월~현재: 고려대학교 정보보호대학원 정보보호학과 석사과정
〈관심분야〉 Effective Fuzzing, Vehicular Security, Reverse Engineering

김 동 영 (Dong Young Kim) 학생회원
2019년 2월: 학점은행제 정보보호학과 학사
2020년 9월~현재: 고려대학교 정보보호대학원 정보보호학과 석사과정
〈관심분야〉 Offensive Security, Vulnerability Analysis, Fuzzing

전 상 훈 (Sanghoon Jeon) 정회원
2006년 8월: 고려대학교 컴퓨터학과 학사
2011년 8월: 성균관대학교 디지털미디어통신공학 석사
2006년 7월~현재: 삼성전자 종합기술원 전문연구원
2019년 3월~현재: 고려대학교 정보보호대학원 정보보호학과 박사과정
〈관심분야〉 데이터 기반 취약점 및 악성코드 분석, 시스템 보안

김 휘 강 (Huy Kang Kim) 종신회원
1998년 2월: KAIST 산업경학학과 학사
2000년 2월: KAIST 산업공학과 석사
2009년 2월: KAIST 산업및시스템공학과 박사
2004년 5월~2010년 2월: 엔씨소프트 정보보안실장, Technical Director
2010년 3월~2014년 12월: 고려대학교 정보보호대학원 조교수
2015년 1월~2020년 2월: 고려대학교 정보보호대학원 부교수
2020년 3월~현재: 고려대학교 정보보호대학원 교수
〈관심분야〉 온라인게임 보안, 자동차 보안, 침입탐지시스템, 네트워크 보안