Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.29, NO.4, Aug. 2019 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2019.29.4.807

유닛테스트를 활용한 c/c++ 라이브러리 그레이박스 퍼징 적용 자동화

장 준 언,† 김 휘 강‡
고려대학교 정보보호대학원

Automated Applying Greybox Fuzzing to C/C++ Library Using Unit Test

Joon Un Jang,† Huy Kang Kim‡
Graduate School of Information Security, Korea University

요 약

그레이박스 퍼징은 소프트웨어에 존재하는 알려지지 않은 보안 취약점을 찾는 효과적인 방법으로 최근까지 활발하게 연구되고 있다. 단, 대부분의 그레이박스 퍼징 도구들은 실행파일을 필요로 하기 때문에 직접 실행할 수 없는 라이브러리는 별도의 실행파일을 준비해야 한다. 이러한 실행파일을 만드는 것은 라이브러리에 대한 이해 및 퍼징에 대한 이해가 동시에 필요한 어려운 일이다.

본 연구에서는 라이브러리를 위한 실행파일을 자동으로 생성하는 방법을 제안하고 이를 LLVM 기반의 도구로 구현한다. 제안하는 방법은 대상 라이브러리 프로젝트에 존재하는 유닛테스트에 대한 정적/동적 분석을 통해 라이브러리를 테스트할 수 있는 실행파일 및 시드파일을 자동으로 생성한다. 생성한 실행파일은 기존 그레이박스 퍼징 도구들이 주로 사용하는 인터페이스를 보유하여 AFL과 같은 다양한 그레이박스 퍼징 도구와 호환된다. 우리는 이 도구를 사용해 오픈소스 프로젝트로부터 생성한 실행파일과 시드파일을 바탕으로 코드 커버리지 및 알려지지 않은 취약점을 찾음으로써 제안하는 방법의 성능을 보인다.

ABSTRACT

Greybox fuzzing is known as an effective method to discover unknown security flaws reside in software and has been actively researched today. However, most of greybox fuzzing tools require an executable file. Because of this, a library, which cannot be executed by itself requires an additional executable file for greybox fuzzing. Generating such an executable file is challengeable because it requires both understanding of the library and fuzzing.

In this research, we suggest the approach to generate an executable file automatically for a library and implement this approach as a tool based on the LLVM framework. This tool shows that executable files and seed files can be generated automatically by static/dynamic analysis of a unit test in the target project. A generated executable file is compatible with various greybox fuzzers like AFL because it has a common interface for greybox fuzzers. We show the performance of this tool as code coverage and discovered unknown security bugs using generated executable files and seed files from open source projects through this tool.

Keywords: security testing, greybox fuzzing, library fuzzing

Received(04. 15. 2019), Modified(05. 29. 2019), Accepted(05. 29. 2019)

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

I. 서 론

퍼징은 임의의 입력값을 통해 프로그램의 버그를 찾는 테스트이다. 주로 보안 취약점으로 연결될 수 있는 버그를 찾기 때문에 보안 분야에서 취약점을 찾기 위한 자동화 도구로 널리 활용하고 있다.

그레이박스 퍼징은 기호 실행과 같은 복잡한 분석 대신 커버리지 분석과 같은 간단한 분석을 통해 프로그램의 다양한 영역을 탐색할 수 있는 입력값을 생성하여 버그를 찾는다.

최근까지 AFL[1], AFLFast[2], VUzzer[3], T-Fuzz[4], Angora[5], CollAFL[6]과 같은 도구들이 연구, 개발되었고 Table 1.과 같이 다수의 취약점을 찾음으로써 성능을 보였다.

그레이박스 퍼징은 바이너리만 가지고 테스트할 수 있다는 장점으로 많이 알려져 있다. 하지만 Table 1.에서 집계된 취약점 대부분은 오픈소스 프로그램에서 발견된 것으로 코드가 공개된 상태에서도 그레이박스 퍼징을 통해 테스트를 수행하는 것이 의미가 있음을 알 수 있다. 따라서, 코드가 공개된 개발단계에서 그레이박스 퍼징을 적용하여 보안 취약점에 대비하는 개발 프로세스 역시 의미가 있다. 이미 일부 큰 IT 기업에서는 퍼징을 개발 프로세스에 포함했다. 특히, 구글에서는 그레이박스 퍼징 서비스 프레임워크인 ossfuzz[7]를 통해 오픈소스 프로젝트들을 지원하고 있다.

그레이박스 퍼징은 실행을 필요로 하는 동적 테스트이다. 따라서 직접 실행이 불가능한 경우 적용이 어렵다. 적용을 위해서는 해당 프로그램을 퍼징할 수 있도록 별도의 실행파일이 만들어져야 하며, 이를 위해서는 해당 프로그램과 퍼징을 모두 이해하고 있는 사람에 의한 코드 작성이 필요하다.

라이브러리는 위에서 언급한 적용의 어려움이 따르는 대상 중 하나이다. 하지만 라이브러리에 존재하는 보안 취약점은 잠재적으로 이를 활용할 많은 프로그램에 영향을 줄 수 있어 치명적이다. 따라서, 적용의 어려움에도 불구하고 그레이박스 퍼징을 통해 미리 취약점을 찾아내려는 시도는 필요하다.

Table 1. The number of vulnerabilities discovered by greybox fuzzing

fuzzer	# of vulnerabilities
AFL	more than 350
AFLFast	12
VUzzer	8
T-Fuzz	3
Angora	175
CollAFL	157

라이브러리에 그레이박스 퍼징을 적용하기 위해서는 해당 라이브러리가 제공하는 API(Application Programming Interface)들을 사용하는 실행파일이 필요하며, 이를 위한 코드 작성이 요구된다. 이 코드는 크게 개발자 혹은 테스터에 의해 작성될 수 있다.

개발자의 경우, 프로그램을 가장 잘 이해한 상태로 확실하고 다양한 코드를 작성할 수 있다. 하지만 개발 일정뿐 아니라, 이미 존재하는 많은 테스트 관련 프로세스에 별도의 부담스러운 업무가 추가되는 점, 해당 코드는 단순히 대상 라이브러리의 API를 활용하는 것이 아닌 그레이박스 퍼징을 위해 작성하는 것이므로 퍼징에 대한 교육이 필요한 점 등의 현실적인 어려움이 존재한다.

Table 2.는 깃허브에서 조사한 주요 파싱관련 라이브러리 프로젝트들의 유닛테스트와 퍼징테스트 현황을 조사한 것이다. 잘 정립된 유닛테스트와 비교했을 때, 퍼징테스트는 거의 구현되어 있지 않은 것을 볼 수 있다. 이를 30개의 프로젝트로 확장하여도 유닛테스트는 26개의 프로젝트에 구현된 반면, 퍼징테스트는 단 4개의 프로젝트에만 구현되었다. 즉, 개발자에 의한 퍼징 지원은 그레이박스 퍼징의 효과에 비해 거의 이루어지고 있지 않다.

반면, 테스터의 경우 퍼징에 효율적인 코드를 작

Table 2. The test status of top 10 open source projects from GitHub which are searched based on keywords: c/c++ language, library, parse, most stars

project	unittest	fuzztest
libphonenumber	o	x
rapidjson	o	x
simdjson	o	x
http-parser	o	x
redcarpet	o	x
grbl	o	x
cJSON	o	o
tinyxml	o	x
libpostal	o	x
yajl	o	x

성할 수 있다. 하지만 라이브러리 API를 어떻게 활용해야 하는지에 대한 이해, 코드 작성, 실행파일로 빌드하는 것까지 시간이 많이 소요될 수 있다. 이는 테스터가 동시에 여러 개의 프로그램에 대해 테스트를 해야 하는 상황을 고려하면 마찬가지로 현실적인 어려움이 있다고 볼 수 있다.

무엇보다도, 이러한 작업은 노동 집약적 작업으로, 라이브러리 코드가 변함에 따라 유지보수의 비용까지 발생하는 부담이 큰 작업이다. 따라서 본 연구에서는 이러한 어려움을 해결하기 위하여 라이브러리에 그레이박스 퍼징을 수행할 수 있도록 자동으로 실행파일을 생성하는 방법을 제시한다. 이 방법은 개발 단계에서의 적용을 목표로 하므로 소스 코드에 대한 접근이 가능한 상황임을 전제한다.

본 연구에서 제안하는 방법의 핵심은 프로젝트에 존재하는 유닛테스트의 정적/동적 분석을 통해 획득한 정보를 활용하여 자동으로 실행파일을 생성하는 것이다. 이 방법을 통해 개발자는 퍼징을 위한 코드 작성의 부담에서 벗어날 수 있으며, 테스터는 라이브러리에 대한 이해 없이도 쉽게 그레이박스 퍼징을 라이브러리에 적용할 수 있다.

우리는 이 방법을 LLVM[8]기반의 도구로 개발하였고 6개의 오픈소스 라이브러리 프로젝트에 적용하여 라인 커버리지 및 찾은 버그를 공개함으로써 성능을 입증하였다. 또한, 실용성을 위하여 기존의 잘 알려진 그레이박스 퍼징 도구로 바로 테스트할 수 있도록 실행파일을 생성한다.

II. 관련 연구

2.1 그레이박스 퍼징

퍼징은 임의의 입력값을 통해 프로그램의 버그를 찾는 동적 테스트이다. 그레이박스 퍼징은 간단한 분석 기술을 활용하여 임의의 입력값이 좀 더 효율적으로 생성될 수 있도록 하는 방법이다. 그레이박스 퍼징은 일반적으로 Fig.1.과 같이 동작한다.

그레이박스 퍼징은 시드로부터 임의의 입력값을 생성하고 프로그램을 실행한 뒤, 생성한 입력값을 프로그램에 전달한다. 프로그램이 종료되면, 프로그램 실행정보에 대한 분석과정을 거친 뒤 보유한 시드를 업데이트한다. 그리고 이 업데이트한 시드로부터 다시 입력값을 생성하는 과정을 반복한다.

시드는 입력값을 생성하기 위해 사용하는 기준값

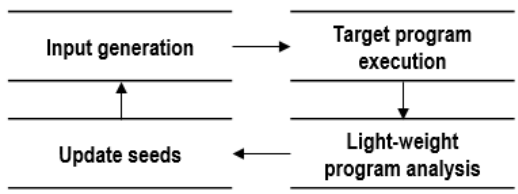

Fig. 1. The general process of greybox fuzzing

이다. 그레이박스 퍼징은 특성상 프로그램이 요구하는 입력값의 형태가 복잡하면 복잡할수록 높은 커버리지를 달성하기가 어렵기 때문에 프로그램이 요구하는 입력값의 형태와 유사한 값을 시드로 제공하여 문제를 해소하는 접근법을 취한다. 물론, lafintel[12]과 같이 시드 없이도 효율적으로 내부를 탐색할 수 있도록 하는 그레이박스 퍼징 방법이 존재하지만, 일반적으로는 제공되는 것이 유리하다.

그레이박스 퍼징의 목적은 버그를 많이 찾는 것이고 이 문제는 얼마나 많은 코드 영역을 탐색할 수 있는 입력값을 생성할 수 있는지의 문제로 바꾸어 생각해 볼 수 있다. 그 이유는 탐색하는 코드 영역이 많으면 많을수록 더 많은 버그를 찾을 수 있는 것이 자연스럽기 때문이다. 따라서 찾아낸 버그의 개수 뿐 아니라, 얼마나 높은 코드 커버리지를 달성했는지 역시 그레이박스 퍼징의 성능을 측정하는 데 주로 사용된다.

현재까지 개발된 대부분의 그레이박스 퍼징 도구들은 퍼징 입력으로 특정 시간 내에 종료가 보장되는 실행파일을 전달받는다. 따라서, 그레이박스 퍼징을 통해 보안 취약점을 탐색하기 위해서는 해당 조건을 만족하는 실행파일이 제공되어야 하며 라이브러리와 같은 직접 실행이 불가능한 프로그램의 경우에는 이를 위한 코드를 직접 작성해야 한다.

2.2 ossfuzz

ossfuzz는 구글에서 제안한 퍼징 서비스 프레임워크로 구글의 크롬 컴포넌트에 대한 성공적인 퍼징 테스트 환경을 오픈소스 프로젝트들에 공유할 목적으로 공개되었다. 현재까지 100개 이상의 오픈소스 프로젝트가 테스트 되고 있으며, 10000개 이상의 버그가 발견되었다.

ossfuzz에 오픈소스 프로젝트를 적용하기 위해 먼저 담당자는 퍼징을 위한 코드를 작성해야 한다. 작성된 코드는 ossfuzz에 저장된 뒤, clusterfuzz

환경으로 배포된다. clusterfuzz는 구글의 퍼징을 위한 분산 환경으로 실제 퍼징이 수행되고 발견된 버그를 해당 오픈소스 프로젝트에 보고하는 것까지 담당하고 있다.

현재 공식적으로 c/c++언어로 작성된 오픈소스 프로젝트에 대해 지원하고 있고 퍼징도구로는 잘 알려진 그레이박스 퍼징 도구인 AFL과 libfuzzer[9]를 지원한다.

libfuzzer는 라이브러리 퍼징을 목적으로 만들어진 그레이박스 퍼징 도구로, 간단한 인터페이스를 통해 쉽게 퍼징코드를 작성할 수 있도록 한다. 작성된 퍼징코드는 퍼저와 함께 실행파일로 만들어지고 이렇게 생성된 파일을 실행하는 것으로 퍼징이 수행된다. libfuzzer는 최대한 쉽게 퍼징코드를 작성할 수 있도록 배려함으로써 기존의 그레이박스 퍼징과 차별점을 갖는다. 하지만, 여전히 퍼징코드는 직접 작성되어야 하고, 시드파일도 별도로 준비되어야 하는 등 본 논문에서 제시한 문제를 해결하지는 못한다.

III. 제안하는 방법론

3.1 문제 정의

라이브러리 퍼징을 위한 코드를 작성하는 것은 특정 라이브러리 함수를 호출하는데 사용할 매개변수에 무작위 입력값이 전달될 수 있도록 코드를 작성하는 것을 시작으로 한다. 하지만, 단순히 하나의 함수는 코드 커버리지를 보장할 수 없으므로 해당 함수의 호출과 관련된 다른 함수의 호출도 고려해야 한다.

본 논문에서는 호출되는 함수의 집합을 함수 시퀀스로 정의한다. Fig.2.는 함수 시퀀스가 필요한 이유를 보여준다. 이 코드에서 기준이 되는 API는 입력값을 전달받는 insert 함수이다. 하지만, insert 함수는 initflag 가 true인 경우에만 전체 코드가 동작한다. 따라서, 먼저 init 함수를 호출해야 해당 함수의 커버리지를 보장할 수 있다. 또한, insert 함수는 단순히 메모리 영역에 전달받은 입력값을 저장하는 역할을 할 뿐이고 실제 이 값을 활용하는 함수는 각각 parse_A, parse_B이다. 그러므로 이 코드 내에서 가장 높은 코드 커버리지를 달성하기 위해서는 {init - insert - parse_A}, {init - insert - parse_B}, 두 개의 함수 시퀀스가 호출되도록 코드를 작성해야 한다.

결과적으로 그레이박스 퍼징을 위한 실행파일을

```
1 static bool initflag = false;
2
3 static char* buffer = nullptr;
4 static size_t buffer_len = 0;
5
6 API init() {
7   ... // initialize code
8   initflag = true;
9 }
10
11 API insert(char* input, size_t input_len) {
12   if(initflag == fasle) return;
13
14   buffer = calloc(input_len + 1, 1);
15   memcpy(buffer, input, input_len);
16   bufffer_len = input_len;
17 }
18
19 API parse_A() {
20   ... // complex logic for parsing buffer
21 }
22
23 API parse_B() {
24   ... // complex logic for parsing buffer
25 }
```

Fig. 2. The example for understanding why a function sequence is required to achieve higher code coverage

자동으로 생성하기 위해서는 외부로부터 입력을 전달받고, 해당 입력값을 특정 함수의 매개변수로 전달하며, 해당 특정 함수와 관련된 가능한 다양한 함수들이 호출되어야 하는 세 가지의 요구사항이 만족 되어야 함을 알 수 있다.

또한, 퍼징을 통해 다양한 프로그램 영역을 탐색하기 위해서는 시드파일이 제공되어야 한다. Fig.3.은 본 논문의 실험에서 사용한 boringssl 프로젝트

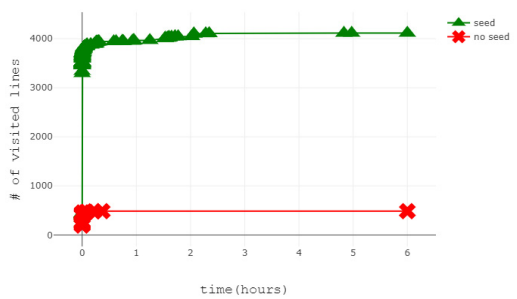

Fig. 3. Line coverage comparison between when seeds are given and when not given

의 pkcs12 실행파일을 6시간 동안 AFL을 통해 퍼징한 결과이다. 시드파일이 제공된 경우 시작부터 높은 라인 커버리지를 달성하며, 최종적으로 10배 이상의 라인 커버리지의 차이를 보인다.

3.2 유닛테스트

유닛테스트는 프로그램이 정상적으로 동작하는지 확인하기 위해 유닛 단위의 테스트를 수행하는 방법으로 일반적으로 함수 단위로 수행된다. 효과적인 유닛테스트를 위해서는 프로그램의 최대한 많은 기능을 테스트할 수 있어야 한다. 여기서 최대한 많은 기능을 테스트한다는 것은 유닛테스트의 라인 커버리지가 높아야 한다고 볼 수 있다.

유닛테스트는 이를 위해 다양한 함수 시퀀스를 테스트하며, 특정 함수를 호출할 때 사용하는 매개변수 값도 다양하게 준비하여 테스트한다. 즉, 유닛테스트는 실행파일 자동생성을 위한 함수 시퀀스와 특정 함수 테스트를 위해 매개변수로 사용되는 입력값이 충분히 준비되어 있다.

따라서 본 연구는 유닛테스트에 존재하는 함수 시퀀스와 입력값을 바탕으로 실행파일과 시드파일을 생성하는 방법을 제안한다. 단, 이 방법은 해당 프로젝트에 유닛테스트가 있어야 하는데, Table 1.에서 본 것과 같이 대부분의 프로젝트가 유닛테스트를 구현하고 있으므로 충분히 실용적인 접근법이라 할 수 있다.

3.3 Fuzzable API

본 논문에서는 라이브러리 퍼징의 기준이 되는 API를 명시하기 위해 FA(Fuzzable API)라는 명칭을 사용하도록 한다. FA란, 외부 데이터를 전달받는 라이브러리 함수를 말하며, 이 함수의 매개변수를 통하여 퍼징 도구가 생성한 입력값이 라이브러리가 사용할 메모리에 적재된다. 3.1에서 소개한 insert 함수가 대표적인 예이다.

Table 3.은 실험에서 사용한 FA 중 일부의 정보이다. XML_Parse, CBS_init 모두 외부 입력을 매개변수로 전달받아 라이브러리의 메모리에 적재하는 함수이며 두 함수 모두 2번째 3번째 매개변수를 통해 입력값을 전달받는다. 이 FA를 기준으로 FA가 포함된 함수 시퀀스 및 FA를 호출하는데 사용되는 입력값 정보가 유닛테스트로부터 분석된다.

Table 3. Examples of fuzzable API

library	function	parameters
expat	XML_Parse	[1] XML Parser* p [2] const char* s [3] int len [4] int isFinal
boringssl	CBS_init	[1] CBS* cbs [2] const uint8* data [3] size_t len

입력값을 위한 매개변수의 데이터 타입으로는 사용자 정의 타입을 포함하여 다양한 타입이 있을 수 있지만, 본 연구의 목적은 다양한 타입의 지원이 아닌 자동생성 방법을 제시하는 것이므로 매개변수의 타입을 바이트 버퍼의 주소를 가리킬 때 주로 사용되는 char 포인터로 한정한다.

이 데이터 타입은 퍼징 테스트에 효과가 좋은 파싱 관련 함수들의 대부분이 매개변수로 이용하고 있어 구현의 범위 대비 효과적일 뿐 아니라, 평가를 위해 비교할 대상인 ossfuzz에서 주로 선정한 매개변수 데이터 타입이다.

3.4 사용자 설정

사용자는 분석에 사용할 LLVM 비트코드 파일의 이름과 FA를 명시해야 한다. 이 과정은 비록 사용자의 개입이 있어야 하지만, 이것은 해당 프로그램에 대한 높은 수준의 지식을 요구하지 않는 단순한 작업으로 전체적인 자동화 프로세스에 큰 영향을 미치지 않는다.

FA를 명시하기 위해서 사용자는 함수의 이름과 몇 번째 매개변수가 입력값으로 사용될지를 명시해야 한다. 예를 들어, Table 3.의 FA들을 설정하기 위하여 사용자는 아래와 같이 함수의 이름, 몇 번째 매개변수가 입력값을 의미하는지, 만약 매개변수로 길이 정보가 필요하다면 몇 번째가 길이 정보로 사용하는지를 아래와 같이 설정한다.

```
XML_Parse 2 3
CBS_init 2 3
```

유닛테스트 프레임워크를 사용하는 경우 테스트에 활용되는 함수의 이름은 특정한 패턴을 가지고 생성되므로 관련 함수를 자동으로 찾을 수 있다. 하지만

그렇지 않은 경우는 테스트 함수의 이름 역시 사용자가 명시해야 한다. 다행히 대부분의 프로젝트는 테스트 함수의 이름을 특정한 prefix 혹은 postfix를 갖도록 구현하기 때문에 astreisk(*)를 활용하면 쉽게 테스트 함수를 명시할 수 있다.

3.5 실행파일 자동 생성

본 연구에서는 유닛테스트를 통해 실행파일을 자동으로 생성하기 위해서 다음의 두 조건이 만족되었음을 가정한다.

· 하나의 테스트는 하나의 함수로 구성된다.
· 각 테스트는 서로 독립적이다.

Google Test[10]와 같이 잘 알려진 유닛테스트 프레임워크는 위의 두 조건을 따르며 많은 프로젝트가 위와 같은 구조를 갖도록 유닛테스트를 구성하고 있다. 따라서, 위의 두 조건으로 인한 본 접근법의 실용적인 측면에는 큰 영향을 미치지 않는다.

Fig.4.는 전체적인 동작 과정을 보여주는 알고리즘이다. 이 알고리즘은 전달받은 LLVM 비트코드를 분석하여 함수들이 모두 추출된 상태에서 시작한다. 각 과정에 대해서는 아래에서 자세히 소개한다.

```
1: procedure ALGORITHM(functions)
2:     tests, entry ← preprocessing(functions)
3:     entry ← insert_interface(entry)
4:     for all test ∈tests do
5:         if is_FA_exist(test) then
6:             test ← insert_operands(test)
7:         else
8:             test ← remove_test(test)
9:         end if
10:    end for
11:    modify(entry, tests)
```

Fig. 4. Algorithm for generating an executable file

3.5.1 preprocessing

각 함수를 순회하면서, 테스트케이스로 활용되는 함수들과 가장 먼저 실행되는 함수를 추출한다. 테스트케이스는 이미 소개한 대로, 잘 알려진 테스트 프레임워크 사용 시에는 특정 키워드를 바탕으로 추출할 수 있으며, 그렇지 않으면 사용자 설정을 이용할 수 있다. 가장 먼저 실행되는 함수는 일반적으로는 main 함수이며 존재하지 않을 때는 생성자 함수를 만들어 이를 반환하는 것으로 구성된다.

3.5.2 insert_interface

유닛테스트의 시작점에 외부 입력을 전달받을 수 있는 인터페이스와 전역 변수를 생성한다.

생성되는 인터페이스에는 외부로부터 입력을 받고, 이를 생성한 전역 변수에 저장하는 코드가 포함된다. 외부로부터 입력을 받는 부분은 그레이박스 퍼징 도구들이 주로 사용하는 인터페이스인 특정 파일이나 stdin을 통해 전달받을 수 있도록 구현된다. 그 외에도 다른 방법을 이용하여 다양한 그레이박스 퍼징 도구와 호환될 수 있다.

이 과정은 생성된 실행파일을 퍼징 시, 퍼징 도구가 생성한 입력값이 삽입된 인터페이스를 통해 해당 실행파일의 메모리에 적재되도록 한다.

3.5.3 is_FA_exist

테스트 함수의 명령어를 순회하면서 명시한 FA를 호출하는 명령어가 있는지 확인한다. 존재하는 경우 해당 테스트 함수에 대하여 *insert_operands*를 수행하며, 존재하지 않는 경우에 *remove_test*를 수행한다.

3.5.4 insert_operands

테스트 함수에 FA를 호출하는 명령어가 있는 경우에 수행된다. FA호출 시 3.4를 통해 명시한 매개변수에 3.5.2에서 생성한 전역 변수를 사용하도록 변경한다. 이 과정을 통해 퍼징도구가 입력한 입력값이 실제 라이브러리 코드로 전달된다.

3.5.5 remove_test

테스트 함수에 FA를 호출하는 명령어가 없는 경우에 수행된다. 해당 함수는 테스트할 필요가 없는 함수이므로 삭제하여 생성되는 실행파일이 조금 더 빠른 실행속도를 갖도록 한다. 퍼징은 최대한 많은 입력값을 테스트 하는 것이 유리하므로 실행속도도 성능에 중요한 영향을 미친다. 이 함수를 삭제하는

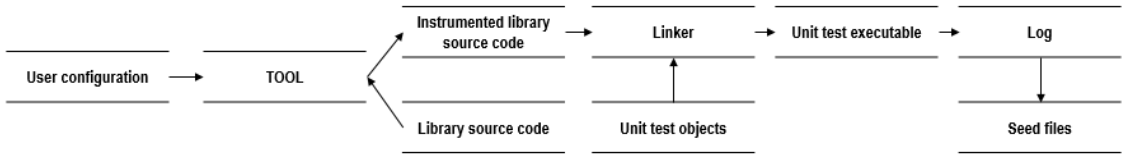

Fig. 5. The overall process to generate input seeds

것은 하나의 테스트 함수가 다른 테스트 함수에 영향을 미치지 않는다고 가정하였으므로, 전체 프로그램에 영향을 미치지 않는다고 볼 수 있다.

함수의 삭제는 구체적으로 테스트 함수를 호출하는 명령어를 삭제하거나 혹은 테스트 함수의 body를 삭제하여 바로 return 되도록 하는 방법이 있다.

3.5.6 modify

위의 과정을 통해 수정된 시작지점 함수와 테스트 함수들은 기존의 비트코드에 저장된다. 이렇게 저장된 비트코드는 빌드 프로세스에 따라 빌드되어 최종적으로 바이너리의 형태로 만들어진다.

생성된 실행파일은 기존의 퍼징 도구의 입력값을 전달받을 수 있는 인터페이스를 가지고 있으며, 해당 인터페이스로 입력된 값이 지정한 FA를 통해 라이브러리가 사용할 메모리에 적재된다. FA를 호출하는 테스트 함수들이 한 번의 실행에 동시에 테스트되며 각 테스트 함수들은 FA와 관련된 다양한 함수 시퀀스를 보유하므로 유닛테스트의 코드 커버리지와 비례하여 커버리지를 달성할 수 있다.

위에서 소개한 과정은 하나의 유닛테스트 실행파일이 빌드될 때 동작하는 것이다. 한 프로젝트에는 여러 개의 유닛테스트를 위한 실행파일이 존재할 수 있으므로 자동으로 생성되는 실행파일도 여러 개가 생성될 수 있다.

3.6 시드 자동 생성

시드는 유닛테스트에서 FA의 입력값으로 활용되는 미리 정의된 값들로부터 추출할 수 있다. 다양한 프로젝트의 유닛테스트 분석 결과 미리 정의된 값들은 대개 아래와 같은 방법으로 저장되어 있다.

- 소스 코드
- 유닛 테스트를 실행하는 스크립트 파일
- 저장소내의 별도의 파일

위와 같이 저장된 입력값들을 추출하기 위해서는 다양한 분석기가 필요하다. 예를 들어, 유닛테스트를 실행하는 스크립트 파일 내에 저장된 경우 ruby, python과 같이 다수의 스크립트 분석기가 필요한데 이러한 분석기를 모두 준비하는 것을 비효율적이므로 우리는 더욱 간단한 방법을 제시하고자 한다.

사용자 설정을 통해 FA를 명시하였기 때문에, 라이브러리 코드 내의 FA 함수에 입력값을 특정 파일에 로깅하도록 자동으로 코드를 삽입한다. 이렇게 삽입되어 빌드된 라이브러리 코드에 대해 유닛테스트가 동작하면, 유닛테스트가 보유한 다양한 입력값이 특정 파일에 취합된다. 취합된 입력값은 서로 다른 파일로 분류되어 향후, 특정 FA를 기준으로 생성한 실행파일을 퍼징할 때 활용된다. 이 전체적인 과정은 Fig.5.와 같다. 이 방법은 유닛테스트의 입력값이 어떻게 저장되어 있든 모두 출력이 가능하다는 점에서 실용적이다.

IV. 실 험

실험을 통해 우리는 해당 도구를 이용하여 자동으로 생성된 실행파일을 퍼징했을 때와 자동으로 생성한 시드의 효율성을 보여주고자 한다. 이를 위하여 ossfuzz에 등록된 프로젝트 중 일부를 선정하여 실험을 진행하였다. ossfuzz에 등록된 프로젝트에는 퍼징을 위한 실행파일들과 시드들이 모두 존재한다. 대부분은 프로그램의 이해를 바탕으로 작성되고 준비된 것들이지만 도구를 이용하는 경우는 그러한 준비 과정이 필요 없으므로 자동으로 생성한 실행파일과 시드파일이 ossfuzz에 준하는 성능을 보여준다면 효율적이라 볼 수 있다.

생성한 실행파일의 퍼징을 위해 가장 잘 알려진 그레이박스 퍼징 도구인 AFL 2.52b와 Address Sanitizer[11]를 함께 사용하였다. AFL은 64bit

Table 4. Tested projects and commit number

project	commit
ossfuzz	a55a1276d9e0c453f588160b7e3581cdf6236013
c-ares	a9c2068e25a107bf535b1fc988eec47384b86dc6
expat	39e487da353b20bb3a724311d179ba0fddffc65b
boringssl	d2a0ffdfa781dd6fde482ccb924b4a756731f238
yara	a3784d3855029bd0ad24071e72746cc0c31b8cba

실행파일에 대한 Address Sanitizer를 지원하지 않기 때문에, 실행파일은 모두 32bit으로 빌드하였다. 단, ossfuzz는 64bit 빌드를 기본으로 하기 때문에 32bit으로 빌드가 되지 않는 일부 프로젝트는 테스트 대상에서 제외하였다.

ossfuzz와의 비교를 위하여 ossfuzz가 각 프로젝트에 선정한 FA와 동일하게 실행파일을 생성했다. 단 유닛테스트에 FA와 관련된 테스트케이스가 없는 경우 본 도구를 통한 자동생성을 할 수 없으므로 테스트 대상에서 제외하였다.

이런 기준으로 ossfuzz로부터 expat, c-ares, boringssl, yara, 4개의 프로젝트를 선정하였으며, Table 4.에 각각의 GitHub 주소와 테스트에 사용한 commit 번호를 기술하였다.

성능 측정을 위해서 라인 커버리지를 활용하였고, gcov를 통해 측정하였다. 여기서 측정한 커버리지는 테스트 코드는 제외한 실제 대상 라이브러리 코드만 측정했다.

실험은 Intel(R) Core(TM) i7-9700K CPU, Debian GNU/Linux 9.5 OS 상에서 진행하였고, 비교를 위해 우리가 사용한 접근법은 "Automated"라 명명하였다.

4.1 자동생성 시드

생성된 시드의 효율성을 검증하기 위하여 프로젝트별 선정된 FA들에 대하여 개발한 도구를 활용하여 자동으로 시드를 생성한 뒤, 생성한 시드와 ossfuzz에 존재하는 시드들의 라인 커버리지를 비교

Table 5. Tested binaries and their ID and FA

project	ID	executable file	FA
c-ares	c1	ares_create_query_fuzzer	ares_create_query
	c2	ares_parse_reply_fuzzer	ares_parse_*
expat	e1	parse_ISO_8859_1_fuzzer	XML_Parse
	e2	parse_US_ASCII_fuzzer	XML_Parse
	e3	parse_UTF_16BE_fuzzer	XML_Parse
	e4	parse_UTF_16_fuzzer	XML_Parse
	e5	parse_UTF_16LE_fuzzer	XML_Parse
	e6	parse_UTF_8_fuzzer	XML_Parse
boringssl	b1	bn_div	CBS_init
	b2	bn_mod_exp	CBS_init
	b3	client	CBS_init
	b4	dtls_client	CBS_init
	b5	dtls_server	CBS_init
	b6	pkcs12	CBS_init
	b7	pkcs8	CBS_init
	b8	read_pem	BIO_new_mem_buf
	b9	server	CBS_init
	b10	session	SSL_SESSION_from_bytes
	b11	spki	CBS_init
	b12	ssl_ctx_api	CBS_init
yara	y1	dex_fuzzer	yr_rules_scan_mem
	y2	dotnet_fuzzer	yr_rules_scan_mem
	y3	elf_fuzzer	yr_rules_scan_mem
	y4	pe_fuzzer	yr_rules_scan_mem
	y5	rules_fuzzer	yr_compiler_add_string

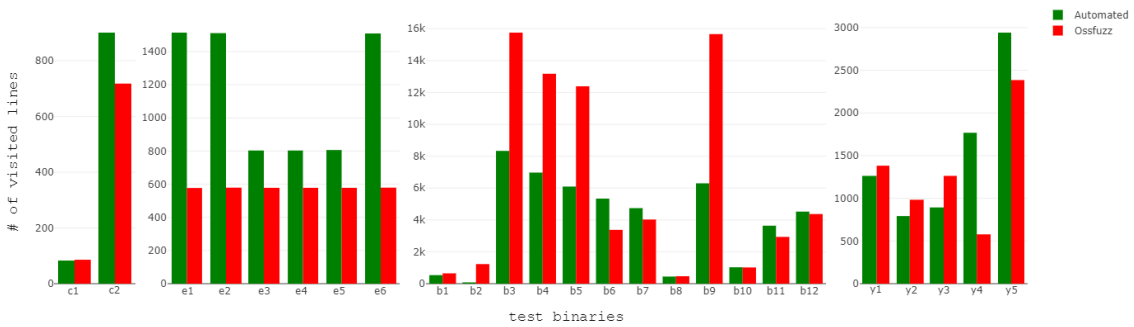

Fig. 6. Initial line coverage comparison between Automated and Ossfuzz

하였다.

비교에 앞서, ossfuzz에 존재하는 시드와 자동으로 생성된 시드는 afl-cmin을 통해 중복되는 시드를 제거하였다.

Table 5.는 실험에 사용한 실행파일에 대한 정보이다. 여기서 executable file 열은 선정한 4개의 프로젝트를 빌드했을 때 퍼징을 위해 빌드된 실행파일들의 이름이며 각각 대상으로 삼고 있는 함수들을 FA열에 표기하였다. 실행파일이 많고 그 이름이 길어서 각각을 쉽게 언급하기 위해 ID를 부여하였다. 단 c2의 FA인 ares_parse_*의 경우 ares_parse_로 시작하는 10개의 FA를 줄여서 표현한 것이다.

Fig.6.은 자동으로 생성된 시드파일과 ossfuzz가 제공하는 시드파일만을 가지고 측정된 최초 라인 커버리지이다. 총 25개의 퍼징 대상 중 14개의 바이너리에서 더 높은 결과가 나타났으며, 6개에서는 근소한 차이로 코드 커버리지가 낮은 것을 확인할 수 있었다. 즉, 유닛테스트를 통해 자동으로 생성한 시드파일이 실제 프로젝트를 대상으로도 충분히 효율적임을 알 수 있다.

유닛테스트는 퍼징테스트보다 더 정립된 테스트이기 때문에 상대적으로 테스트에 대한 이해 수준과 이를 위한 개발자의 지원이 풍부하다. 따라서, 대상으로 선정한 FA 및 이와 관련된 함수 시퀀스가 유닛테스트에 구현된 경우 이를 테스트하기 위한 입력값이 퍼징테스트에 비해 상대적으로 다양하며 그 종류가 더 많이 존재한다. 그 결과 Table 6.과 같이 더 높은 커버리지를 달성한 프로그램에 대해서 제안하는 방법은 유닛테스트로부터 더 많은 시드파일을 생성할 수 있었다. afl-cmin을 통해 같은 경로를 탐색하는 중복 시드가 제거된 것이므로 더 많은 시드의 개수가 더 높은 커버리지로 이어졌다고 해석할 수 있다.

물론, c1과 같이 시드파일이 많다고 해서 항상 높은 라인 커버리지를 갖는 것은 아니다. afl-cmin이 중복을 확인하기 위해 고려하는 개념은 베이직 블록을 바탕으로 하고 라인 커버리지는 소스코드의 라인을 바탕으로 하기 때문이다.

반면, b2, b3, b4, b5, b9와 같이 차이가 큰 차이로 라인 커버리지가 낮은 경우도 존재한다. 이 결과는 해당 실행파일의 FA의 특성과 FA를 활용하는 유닛테스트와 관련되어 있다. CBS_init은 라이브러리가 정의한 특정한 메모리구조에 값을 저장하는 역

Table 6. Comparison of # of seed files with ossfuzz

program	# of seed files	
	automated	ossfuzz
c1	16	6
c2	57	22
e1, e4, e6	15	1
e2, e3, e5	14	1
b1	30	43
b2	7	49
b3	10	281
b4	8	80
b5	8	78
b6	73	4
b7	51	13
b8	12	41
b9	9	263
b10	13	2
b11	56	8
b12	176	86
y1	15	6
y2	15	3
y3	15	15
y4	15	8
y5	139	3

할을 하며, 이후에 어떤 함수가 불리느냐에 따라서 해당 데이터의 사용법이 결정된다. 문제가 된 다섯 개의 바이너리의 경우 해당 실행파일이 가지고 있는 함수 시퀀스가 유닛테스트에는 구현이 되어있지 않고 따라서 그와 관련된 입력 파일이 존재하지 않는다. 이것은 커버리지를 높이면 유리한 유닛테스트의 성격을 고려했을 때, 유닛테스트가 잘 작성될수록 해결될 수 있는 문제이다.

4.2 자동생성 실행파일

마찬가지로, Table 7.에 자동으로 생성된 실행파일에 할당한 ID를 확인할 수 있다. target unit test 열은 프로젝트별 빌드된 유닛테스트 실행파일의 이름이다.

fb1, fb2의 경우, 하나의 프로젝트에 여러 개의 유닛테스트 실행파일이 존재하여 여러 개의 실행파일이 생성된 것이다. 또한 fb1, fb3의 경우, 하나의 유닛테스트 실행파일에 대해 어떤 FA를 선택했느냐에 따라 서로 다른 실행파일이 생성된 것이다.

실행파일의 생성기준은 FA이기 때문에 비교를 위하여 FA별로 테스트 셋을 만들어 테스트를 진행하였다. Table 8.은 FA기준으로 만들어진 테스트 셋에 어떤 실행파일이 비교되었는지를 명시한 것이다. 각 실행파일별로 기본적으로 6시간의 퍼징을 수행했다. 단 T3과 같이 실행파일의 개수에 차이가 있는

Table 7. Generated executable files and its ID and FA

project	ID	target unit test	FA
c-ares	fc1	ares_test	ares_create_query
	fc2	ares_test	ares_parse_*
expat	fe1	parse_ISO_8859_1_fuzzer	XML_Parse
boringssl	fb1	crypto_test	CBS_init
	fb2	ssl_test	BIO_new_mem_buf
	fb3	crypto_test	BIO_new_mem_buf
	fb4	ssl_test	SSL_SESSION_from_bytes
yara	fy1	test-api	yr_rules_scan_mem
	fy2	test-api	yr_compiler_add_string

Table 8. Test sets for line coverage comparison of executable files based on the FA

test set	automated	ossfuzz	FA
T1	fc1	c1	ares_create_query
T2	fc2	c2	ares_parse_*
T3	fe1(36)	e1, e2, e3, e4, e5, e6	XML_Parse
T4	fb1(60)	b1, b2, b3, b4, b5, b6, b7, b9, b11, b12	CBS_init
T5	fb2, fb3	b10(12)	BIO_new_mem_buf
T6	fb4	b8	SSL_SESSION_from_bytes
T7	fy1(24)	y1, y2, y3, y4	yr_rules_scan_mem
T8	fy2	y5	yr_compiler_add_string

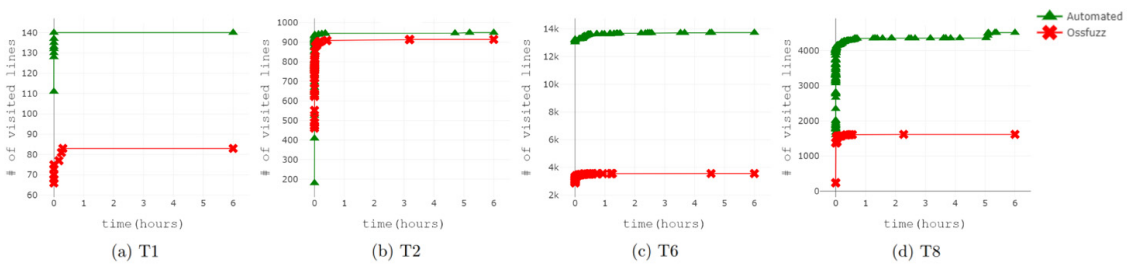

Fig. 7. Line coverage comparison between automated and ossfuzz over 6 hours fuzzing

경우에는 사실상 자동으로 생성된 실행파일에 6시간, ossfuzz 실행파일에 60시간을 부여한 것이 되어 공정한 결과라 보기 어렵다. 따라서, 공정한 결과를 위해 fb1에는 60시간의 퍼징 시간을 부여한다. 이것은 함수 시퀀스를 고려했을 때, fb1에는 여러 개의 함수 시퀀스가 하나의 실행파일에 존재하고 ossfuzz는 여러 개의 실행파일에 나누어져 있다고볼 수 있으므로 합리적인 시간 배분이라 볼 수 있다. 이렇게 6시간 이상을 할당받은 실행파일은 이름 옆에 괄호로 시간을 얼마나 부여했는지 시간(hour) 단위로 명시하였다.

Fig.7.은 자동으로 생성한 실행파일과 ossfuzz의 실행파일이 1개씩인 T1, T2, T6, T8에 대해 6시간 퍼징을 수행하면서 라인 커버리지가 어떻게 변하는지 보여준다. 이 경우 본 연구에서 제안한 방법으로 생성한 실행파일이 더 높은 커버리지를 갖는다는 것을 보이며 이것은 퍼징을 통해 더 많은 코드 영역을 테스트할 수 있다는 의미로 해석할 수 있다.

Fig.8.은 서로 다른 시간 동안 퍼징을 수행한 T3, T4, T5, T7에 관한 결과이다. T3, T7의 경우 비록 커버리지가 생성한 실행파일에서 다소 낮게 나오긴 했지만, 우리의 목표인 유사한 수치를 달성하는 것에는 만족함을 보인다. 다시 말하지만, 해당 라이브러리에 대한 이해 없이 자동으로 생성한 실행파일이라는 점과 유닛테스트 코드가 더욱 잘 작성되면 될수록 성능이 향상하는 비례적인 구조라는 점을 고려했을 때 나쁜 결과라 볼 수 없다.

T4의 경우는 4.1 실험 결과의 CBS_init 관련 테스트 결과와 같게 설명될 수 있다. 반면, T5의 경우에는 자동으로 생성한 실행파일이 더 높은 커버리지를 달성하였다.

Fig. 8. Line coverage comparison between automated and ossfuzz over various time fuzzing based on the number of executable files

4.3 자동생성 실행파일을 통해 발견한 취약점

우리는 github에서 파싱관련 라이브러리 중 Most stars로 정렬한 뒤, 우리의 조건에 맞는 프로젝트 cJSON, mpc, 2개를 선정하여 실제 버그를 찾을 수 있는지 확인하기 위한 추가 실험을 진행하였다. 우리가 선정한 프로젝트와 commit 번호는 Table 9.와 같다.

결과적으로 두 개의 프로젝트에 대해서 3개의 버그를 찾았으며, ossfuzz를 포함하여 총 찾은 버그에 대하여 Table 10.에 정리하였다.

expat의 heap buffer overflow은 normal_up-datePosition 함수에서 할당된 메모리를 넘어서는 읽기가 발생하는 버그로 특정 상황에서 메모리 내의 정보가 유출되는 문제가 발생할 수 있다.

mpc의 stack buffer overflow는 mpca_gra-mmar_find_parser 함수에서 stack에 할당된 배열에 저장된 값을 특정 메모리로 복사할 때 해당 배열에 접근하기 위한 인덱스를 입력값을 통해 조작하여 할당된 배열을 넘어서는 값을 읽고 이를 특정 공간에 쓸 수 있는 버그이다. 이 버그는 특정 상황에서 메모리 정보 유출 및 제어 흐름 조작이 발생할 수 있다.

mpc의 heap buffer overflow는 mpcf_strtri-mr 함수에서 heap에 저장된 사용자 입력값을 순회하면서 해당 영역에 0을 할당할 때, 경계 검사를 하지 않아 할당된 공간을 넘어선 메모리에 0을 저장하는 버그이다. 이 버그는 다른 세 개의

Table 9. Tested projects from GitHub

project	commit
cJSON	08103f048e5f54c8f60aaefda16761faf37114f2
mpc	b31e02e427f55d4ce69c33ed9936a1b396628440

Table 10. Found bugs via automatically generated executable files

project	bug type
expat	heap buffer overflow
cJSON	null dereference
mpc	stack buffer overflow
mpc	heap buffer overflow

버그를 포함하여 특정 상황에서 프로세스가 강제 종료되거나 이상한 값이 참조되면서 서비스 실패를 일으킬 수 있다.

무엇보다도 expat에서 발견한 heap buffer overflow는 ossfuzz가 막대한 자원을 가지고 퍼징을 하고 있음에도 찾지 못한 버그라는 점에서 의미가 있다. 해당 버그는 ossfuzz가 사용한 실행파일에 정의되어 있지 않은 함수 시퀀스가 우리의 접근법을 바탕으로 생성한 실행파일에 존재하기 때문에 찾은 버그이다. 또한, 그 외의 버그들은 본 접근법을 활용한 실행파일을 통해 기존에는 적용하기 어려운 프로젝트에 적용함으로써 찾은 버그라는 점에서 의미가 있다.

V. 결 론

본 연구에서는 소스코드가 공개된 개발단계에서 라이브러리를 대상으로 그레이박스 퍼징을 적용할 수 있는 방법을 제시하고, 이를 도구로 구현하였다. 이 도구를 통해 개발자들은 퍼징 코드 작성으로부터 자유로우며, 테스터들은 프로그램에 대한 이해 없이도 그레이박스 퍼징을 적용하기 위한 실행파일과 시드파일을 자동으로 생성할 수 있기에, 개발 프로세스의 효율을 증가시킬 수 있다. 이 방법은 유닛테스트 분석을 통해 이루어지므로 유닛테스트가 더 많은 커버리지를 가지면 가질수록 생성되는 실행파일이나 시드도 더 높은 커버리지를 갖는 비례관계가 있으므로 유닛테스트 코드가 잘 작성되면 작성될수록, 더 좋은 실행파일 및 시드가 생성될 수 있다.

VI. 향후 연구

아래는 본 연구를 토대로 더 정교하게 실행파일을 생성하기 위하여 필요한 요건들을 정리한 것이다.

6.1 FA 자동 선정

본 연구에서 FA의 선택은 수동적으로 이루어진다. 물론, 특정 데이터를 입력받기 위해 사용되는 API는 라이브러리에 한정되어 있고 명시적이기 때문에 이를 선별하는 것은 어려운 일은 아니지만, 더 높은 수준의 자동화를 위해서는 이 기능이 필요하다.

6.2 실행파일 최적화

현재 생성된 실행파일은 함수의 형태로 구현된 테스트케이스 중 FA를 호출하는 대상에 한정해서만 동작하고, 그렇지 않은 것은 삭제하도록 구현되어 있다. 하지만 이외에도 유닛테스트 코드에는 로깅이나 입력값에 관한 결과를 비교하는 등의 퍼징과는 무관한 명령어들이 포함되어 있으며, 이러한 명령어들이 포함되면 실행속도가 느려진다. 결과적으로 퍼징의 효율성을 낮출 수 있으므로, 실행파일을 더욱 최적화하여 관련된 명령어들만 남길 수 있도록 하는 방안이 필요하다.

6.3 False Alarm

테스트케이스를 구현한 함수에는 예측하지 못한 입력값에 대해서 프로그램을 특정 신호와 함께 강제 종료시키는 assert 및 abort 같은 함수가 사용될 수 있다. 이러한 코드는 그레이박스 퍼징 도구의 false alarm을 일으킬 수 있다.

또한, 퍼징의 입력값은 기존 유닛 테스트케이스 입장에서는 대부분 예측 불가능한 오류인데, 이를 검출하는 코드가 구현되어 있지 않을 수 있다. 그 이유는 유닛테스트의 입력값은 대부분 정상적이거나 예측 가능한 수준의 오류로 구성되기 때문이다. 이로 인해 발생하는 버그는 테스트 코드의 버그이지 실제 대상 프로그램의 버그라 볼 수 없다. 따라서, 이를 구별할 방법 역시 자동화될 필요가 있다.

6.4 Input Type 확장

현재 입력값을 받기 위한 매개변수가 char 포인터의 형태로 한정되어 구현되어 있다. 하지만 c++의 경우에는 string type을 이용하여 입력값을 전달받거나 구조체와 같은 사용자 정의 타입이 활용될 수 있다. 그뿐만 아니라 퍼징의 대상을 넓혀서 int, float과 같은 데이터 타입도 퍼징하여 라이브러리의 더 많은 영역을 탐색할 수 있는 실행파일을 생성할 필요가 있다.

References

〔1〕 Amerian Fuzzy Lob, http://lcamtuf.coredump.cx/afl, Accessed : June, 2019

〔2〕 Böhme, Marcel, Van-Thuan Pham, and Abhik Roychoudhury. "Coverage-based greybox fuzzing as markov chain." IEEE Transactions on Software Engineering, vol. 45, no. 5, pp. 489-506, Dec. 2017.
〔3〕 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.
〔4〕 Hui Peng, Yan Shoshitaishvili, Mathias Payer, "T-Fuzz: fuzzing by program transformation", 2018 IEEE Symposium on Security and Privacy (SP), pp. 697-710, May. 2018.
〔5〕 Peng Chen, Hao Chen, "Angora: Efficient fuzzing by principled search", 2018 IEEE Symposium on Security and Privacy (SP), pp. 711-725, May. 2018.
〔6〕 Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, Zuoning Chen, "Collafl: Path sensitive fuzzing", 2018 IEEE Symposium on Security and Privacy (SP), pp. 679-696, May. 2018.
〔7〕 Mike Aizatsky, Kostya Serebryany, Oliver Chang, Abhishek Arya and Meredith Whittaker, "Announcing oss-fuzz: Continuous fuzzing for open source software", Google Open Source Blog, 2016.
〔8〕 Chris Lattner and Vikram Adve, "LLVM: A compilation framework for lifelong program analysis & transformation", International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization, pp. 75, Mar. 2004.
〔9〕 Libfuzzer, https://llvm.org/docs/LibFuzzer.html, Accessed : June, 2019
〔10〕 Google Test, https://github.com/google/googletest, Accessed : June, 2019
〔11〕 Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov, "AddressSanitizer: A fast address sanity checker", as part of the 2012 USENIX Annual Technical Conference, pp. 309-318, Jun. 2012.
〔12〕 lafintel, https://lafintel.wordpress.com, Accessed : June, 2019

〈저자소개〉

장 준 언 (Joon Un Jang) 학생회원
2013년 8월: 성균관대학교 컴퓨터공학과 학사
2018년 3월~현재: 고려대학교 정보보호대학원 석사과정
2013년 7월~현재: ㈜삼성전자 Samsung Research 연구원
〈관심분야〉 온라인게임 보안, 소프트웨어 보안, 보안 테스트

김 휘 강 (Huy Kang Kim) 종신회원
1998년 2월: KAIST 산업경영학과 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~현재: 고려대학교 정보보호대학원 부교수
〈관심분야〉 온라인게임 보안, 네트워크 보안, 네트워크 포렌식