

목차

Fuzzing like the Legendary Super Saiyan	2
1. Fuzzing Overview	2
2. American Fuzzy Lop	3
3. Useful strategies	10
AFL+ + : Combining Incremental Steps of Fuzzing Research	12
1. Introduction	12
2. State-of-the-Art	12
3. A New Baseline for Fuzzing	14
4. Evaluation Use Cases	18
5. Future Work	18
6. Conclusion	19
Breaking Through Binaries: Compiler-quality Instrumentation for Better Binary-only Fuzzing	20
1. Introduction	20
2. Background on Fuzzing	20
3. Compiler-based Fuzzing Enhancements	22
4. Binary-only Fuzzing : the Bad & the Ugly	23
	27

Fuzzing like the Legendary Super Saiyan

Agenda

1. Fuzzing Overview
2. American Fuzzy Lop
3. Useful strategies

1. Fuzzing Overview

Fuzzing

- Fuzzing : Find an input that triggers a bug.
- 퍼징은 손으로 만들 수 없는 많은 input을 실행하는 데 유용하다.
- => 복잡한 입력을 처리해야하는 프로그램에게 매우 효과적
- ex) parser가 있는 프로그램, image libraries, PDF readers, compilers ...

fuzzer	description
White-Box Fuzzer	<ul style="list-style-type: none">- 최대한 많은 경로를 탐색하기 위해 프로그램 분석을 사용한다. (More code coverage)- symbolic execution
Black-Box Fuzzer	<ul style="list-style-type: none">- dumb bash fuzzed나 Radamsa와 같이 프로그램의 구조를 인식할 수 없다.- (output과 연관된) input을 기반으로 한다.
Grey-Box Fuzzer	<ul style="list-style-type: none">- <u>code coverage 측정과 taint tracking 등을 위해 instrumentation을 사용한다.</u>[*] 참고로 이 자료에서 grey box coverage-guided fuzzer 중에 American Fuzzy Lop에 다룬다.

Input structure awareness

- fuzzer는 입력 구조를 인식할 수도 있고 인식을 못할 수도 있다.
- Peach : XML 모델을 사용하여 입력 구조를 설명하면 fuzzer는 testcase를 생성한다.
- AFLSmart : American Fuzzy Lop + Peach

Other notable fuzzers

- LLVM libFuzzer : LLVM toolchain에서 제공하는 fuzzer
- Google honggfuzz : instructions counting뿐만 아니라 coverage도 포함하는 fuzzer

Funky hybrid stuffs

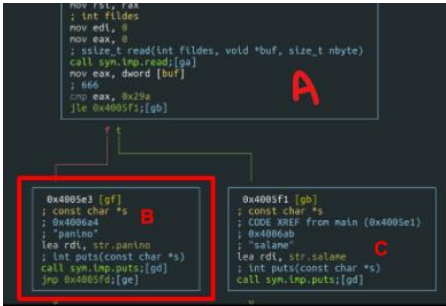
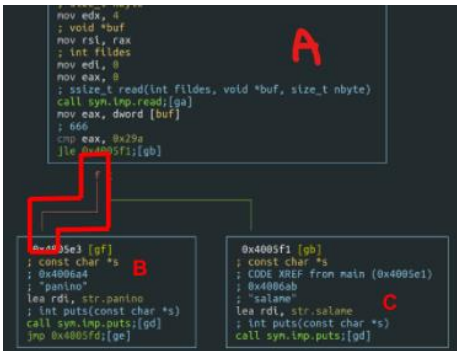
- hybrid fuzzing : 퍼저가 멈췄을 때, symbolic tracing을 가능하게 하는 것을 목표로 한다.
 1. input이 흘러가는 path와 관련된 제약 조건(constraints)를 수집
 2. 새로운 상태를 생성할 수 있는 분기(branch)에 도달하면 기존의 제약 조건(constraints) 대신, 분기를 우회할 수 있는 새로운 input을 얻음
- ex) Georgia Tech의 QSYM

2. American Fuzzy Lop

AFL

- OpenSSH, PHP, MySQL, Firefox 등의 소프트웨어에서 치명적인 취약점을 발견하는데 사용되었다.
- edge coverage에 대한 피드백을 사용한다.

code coverage	description
---------------	-------------

<p>Standard code coverage</p>	<ul style="list-style-type: none"> - 기본 블록이 실행되는 것을 기록함 - e.g. A→B→C 
<p>Edge code coverage</p>	<ul style="list-style-type: none"> - 기본 블록이 전환(transition)되는 것을 기록함 - e.g. A→C A→B - <u>전환(transition) = 튜플(tuple)</u> e.g. A→C is (A,C) 

AFL Instrumentation

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

- shared_mem : fuzzer와 instrumented target program 간에 공유되는 64kb SMB이다.
- 충돌이 가능하다.
- 컴파일 타임에 instrumentation을 삽입하기 위해 afl-gcc 및 afl-g++ 을 사용하여 프로그램을 컴파일할 수 있다.
- LLVM 모드의 AFL을 사용하면 다른 clang 기능도 사용할 수 있다.

- ex) ASAN (data tracking and validation)

standard gcc	afl gcc
<pre> 0x4005f1 [gb] ; const char *s ; CODE XREF from main (0x4005e1) ; 0x4006ab ; "salame" lea rdi, str.salame ; int puts(const char *s) call sym.imp.puts;[gd] </pre>	<pre> 0x89b [gf] lea rsp, [rsp - 0x98] mov qword [rsp], rdx mov qword [local_8h], rcx mov qword [local_10h], rax mov rcx, 0xb71e call loc.__afl_maybe_log;[ga] ; [0x10:8]=0x1003e0003 mov rax, qword [local_10h] ; [0x8:8]=0 mov rcx, qword [local_8h] mov rdx, qword [rsp] ; 0x98 lea rsp, [arg_98h] ; const char *s ; 0xf9b ; "salame" lea rdi, str.salame; a.c:6 ; int puts(const char *s) call sym.imp.puts;[ge] </pre> <p>save registers</p> <p>restore registers</p> <p>else puts("salame");</p>

- afl-gcc로 컴파일을 하면 standard gcc로 컴파일했을 때(노란색)의 코드와 함께 call __afl_maybe_log 라는 과정이 더 추가된다.

AFL Heuristics

```

EXP_ST u8* trace_bits; /* SHM with instrumentation bitmap */
EXP_ST u8 virgin_bits[MAP_SIZE], /* Regions yet untouched by fuzzing */
        virgin_tmout[MAP_SIZE], /* Bits we haven't seen in tmouts */
        virgin_crash[MAP_SIZE]; /* Bits we haven't seen in crashes */

```

- fuzzer는 target program의 모든 실행에서 볼 수 있는 tuple 모음을 유지, 관리한다.
- input은 새로운 tuple을 trace_bits에 등록한다.
- input은 새로운 tuple을 발견할 때마다, hit_count를 증가시킨다.
(shared_mem[...]++)
- 하지만 input이 들어올 때마다 hit_count를 증가시키면 path explosion이 발생할 수 있기 때문에 이것을 방지하기 위해 같은 bucket 내에서 발견되는 tuple은 무시된다.

- 참고로 `hit_count`는 tuple별로 존재한다.

AFL Queue & Evaluation

- input queue는 항상 증가한다.
 - 새로운 input은 대체되는 것이 아니라 추가된다.
- 따라서 가장 최근 input에서의 edge coverage는 이전의 input coverage의 상위 집합이 될 수 있다.
- AFL은 주기적으로 queue에 있는 testcase를 평가한다.
- 실행 지연 시간(latency)와 파일 크기에 비례하여 점수를 매기게 되는데, 점수가 가장 낮은 testcase를 선택한다.

AFL Mutator

- mutation이 제대로 되지 않으면 fuzzer는 좋은 coverage에 도달할 수 없다.
- 하지만 mutation이 많이 될 경우, fuzzer는 처음부터 돌아가지않는 많은 testcases를 생성할 것이다.
- AFL은 target input에 대해 deterministic 단계를 거친 후, 이후 단계에서는 not-deterministic 퍼징 및 다른 입력의 재조합을 한다.

stage	description
deterministic stage	<ul style="list-style-type: none"> - 자동 추가 감지를 포함한 walking bit flips - 간단한 산술 - 0, 1, INT_MAX 등과 같은 정수
not-deterministic stage (havoc)	<ul style="list-style-type: none"> - bit flip - 정수 삽입 - 랜덤 엔디안 덧셈/뺄셈 - 임의의 값으로 설정된 단일 바이트 - 블록 삭제/복제/memset

last resort stage (splice)	<ul style="list-style-type: none"> - 이전의 모든 단계가 새로운 input을 찾지 못했을 때 호출되는 최후의 수단 - 두 입력을 재조합한다. - 재조합을 한 후, havoc 단계로 전달된다.
----------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------

AFL Forksever

- execve의 로더 호출은 fuzzer에게 있어서 오버헤드이다.
- AFL은 어떤 코드를 ELF에 삽입하여 자식 프로세스가 main에서 중지되게 한다.
- 파이프를 이용하여 요청된 각 실행은 자식 프로세스의 분기이다.
- LD_BIND_NOW

AFL Command line

```

andrea@malweisse ~/Desktop
$ afl-gcc program.c -o program
afl-cc 2.52b by <lcamtuf@google.com>
afl-as 2.52b by <lcamtuf@google.com>
[+] Instrumented 1 locations (64-bit, non-hardened mode, ratio 100%).

```

- \$ afl-fuzz -i input_directory -o afl_out -- ./program

AFL Status Screen

no bug	bug
<pre> american fuzzy lop 2.52b (boh) process timing run time : 0 days, 0 hrs, 0 min, 4 sec last new path : 0 days, 0 hrs, 0 min, 0 sec last uniq crash : none seen yet last uniq hang : none seen yet cycle progress new processing : 0 (0.00%) paths timed out : 0 (0.00%) stage progress now trying : user extras (insert) stage execs : 560/2784 (20.11%) total execs : 7988 exec speed : 1773/sec fuzzing strategy yields bit flips : 18/232, 0/231, 0/229 byte flips : 0/29, 0/28, 0/26 arithmetics : 4/1612, 0/105, 0/0 known ints : 0/151, 0/774, 0/1144 dictionary : 6/2571, 0/0, 0/0 havoc : 0/0, 0/0 trim : 0.00%/7, 0.00% overall results cycles done : 0 total paths : 36 uniq crashes : 0 uniq hangs : 0 map coverage map density : 0.93% / 1.35% count coverage : 1.87 bits/tuple findings in depth favored paths : 5 (13.89%) new edges on : 18 (50.00%) total crashes : 0 (0 unique) total tnouts : 0 (0 unique) path geometry levels : 2 pending : 36 pend fav : 5 own finds : 31 imported : n/a stability : 100.00% </pre>	<pre> american fuzzy lop 2.52b (boh) process timing run time : 0 days, 0 hrs, 1 min, 11 sec last new path : 0 days, 0 hrs, 0 min, 9 sec last uniq crash : 0 days, 0 hrs, 0 min, 9 sec last uniq hang : none seen yet cycle progress new processing : 16 (13.01%) paths timed out : 0 (0.00%) stage progress now trying : interest 32/8 stage execs : 340/1329 (25.58%) total execs : 120k exec speed : 1735/sec fuzzing strategy yields bit flips : 24/2000, 3/1993, 0/1979 byte flips : 0/250, 0/243, 0/229 arithmetics : 8/13.9k, 0/919, 0/0 known ints : 1/1273, 0/6712, 0/8932 dictionary : 18/19.9k, 42/21.2k, 0/1489 havoc : 25/37.9k, 0/0 trim : 2.34%/60, 0.00% overall results cycles done : 0 total paths : 123 uniq crashes : 3 uniq hangs : 0 map coverage map density : 0.93% / 1.44% count coverage : 3.24 bits/tuple findings in depth favored paths : 29 (23.58%) new edges on : 39 (31.71%) total crashes : 3 (3 unique) total tnouts : 3 (3 unique) path geometry levels : 2 pending : 117 pend fav : 28 own finds : 118 imported : n/a stability : 100.00% </pre>

AFL .. mode

mode	description
AFL Qemu mode	<ul style="list-style-type: none"> - 소스코드를 사용할 수 없을 때? - TCG - QEMU 모드로 실행하려면 -Q 옵션을 추가한다.
AFL LLVM mode	<ul style="list-style-type: none"> - afl-gcc를 사용하여 생성된 asm 파일에 instrumentation code가 삽입된다. 이 경우 컴파일러 최적화를 적용하기 어렵기 때문에 속도가 느려진다. - /llvm_mode/README.llvm - 관련 컴파일러 : afl-clang-fast - LLVM 모드를 사용하면 아래와 같이 target 위치 코드에 이 스니핑을 삽입한 후 지연된 진입점을 정의할 수 있다. <pre> #ifdef __AFL_HAVE_MANUAL_CONTROL __AFL_INIT(); #endif </pre>
AFL Persistent mode	<ul style="list-style-type: none"> - fork()가 무거우면 afl-clang-fast를 사용하면 된다. (다만, 퍼징 중인 어플리케이션이 stateless여야 한다.) - 수명이 긴 single process를 재사용하여 여러 input을 테스트할 수 있다.

Trimming

- 파일 크기는 퍼징에 큰 영향을 미친다.
- AFL은 queue에서 가져온 input을 퍼징할 때, input의 일부를 삭제하려고 시도한다.
- 그리고 이 삭제하려는 시도가 instrumentation output에 영향을 미치는지 확인한다.
- 삭제가 trace_bits 체크섬에 영향을 미치지 않으면 디스크에 commit된다.

Parallel fuzzing

- AFL은 single core에서 작동한다.

- CPU의 모든 성능을 활용하려면 여러개의 AFL 인스턴스를 실행해야 한다.
- 그렇게 하면 동일한 input을 여러번 퍼징하는 경우가 생기지 않을까?
 - AFL이 퍼징을 하는 동안 다른 인스턴스 사이의 queue에 있는 testcase를 동기화한다.
 - 동일한 output 디렉터리(-o afl_out)를 지정하고 각 퍼저에 대한 역할과 이름을 지정해야 한다.
- 마스터 인스턴스는 앞에서 설명한 모든 단계(deterministic, havoc, splice stage)를 모두 수행한다.
- 하지만, 슬레이브 인스턴스는 deterministic 단계를 거치지 않고 havoc과 splice 단계만 수행한다.
- `$ afl-fuzz -i initial_dir -o afl_out -M afl_master -- ./program`

AFL Output directory

```

afl_out /
    [FUZZER_1] /
        crashes /
        hangs /
        queue /
        fuzz_bitmap (copy of virgin_bits)
        fuzzer_stats
        plot_data
    [FUZZER_2] /
        ...

```

3. Useful strategies

.. minimization

minimization	description
Testcase minimization	- 퍼저 외부에서 afl-tmin 도구를 사용하여 testcase를 최소화할 수 있다.
Corpus minimization	- afl-cmin은 queue에 있는 모든 testcase의 최소화 유형에 사용된다. - queue에서 더이상 사용되지 않는 모든 testcase를 제거한다. - afl-cmin을 실행하는 방법? - fuzzer를 중지, queue에서 afl-cmin 실행, 모든 파일에서 afl-tmin을 실행

Dictionary

- 사용자는 퍼징 중에 사용할 사전 키워드를 지정할 수 있다.
- 옵션 -x를 사용한다.

LibTokenCap

- 위의 키워드를 알지 못한다면?
- LibTokenCap을 사용하여 strncmp(), memcmp() 및 관련 함수를 instrumentation하여 syntax token을 자동으로 추출할 수 있다.
- if (strcmp(input, "pippo") == 0) goto new_path;
- “pippo”는 read-only 섹션에 있기 때문에 logging된다.
- LD_PRELOAD=./libctoken.so
- 출력 파일 : AFL_TOKEN_FILE

ASAN / MSAN

- crash와 security issue는 관련이 없을 수도 있다.
- ex) metadata를 1 byte overflow했지만 사실은 metadata 영역이 아닌 dummy memory였다면? (어쨌든 1 byte overflow)
- 이러한 crash를 감지하는 유용한 도구는 memcheck이라는 플러그인이다.
- 하지만 이 플러그인은 비싸기 때문에 LLVM tool chain은 dynamic instrumentation 대신 compile-time instrumentation을 기반으로 하는 유사한 기능을 제공한다.

	description
AddressSanitizer(ASAN)	<ul style="list-style-type: none"> - 메모리 오류 감지기 - AFL_USE_ASAN=1
MemorySanitizer(MSAN)	<ul style="list-style-type: none"> - 초기화되지 메모리 감지기 - AFL_USE_MSAN=1

LibDislocator

- AFL은 힙 기반 메모리 감지를 위해 라이브러리를 제공한다.
- LD_PRELOAD를 통해 malloc(), calloc(), free()와 같은 libc 함수를 제공한다.

AFL++: Combining Incremental Steps of Fuzzing

Research

1. Introduction

1. 최근 퍼징 연구 동향을 하나로 통합하여 사용 가능한 도구로써 AFL++를 구축하였다.
2. AFL++ 특유의 Custom Mutator API를 개방함으로써 향후 연구에 활용할 수 있도록 방안을 제시했다.
3. AFL++에서 사용된 다양한 기술들은 함께 적용하거나 또는 각각 따로 구분하여 사용할 수 있으므로, 각 기법이나 연구의 상관관계를 비교해볼 수 있으며 이는 잠재적으로 향후 연구 주제에 대한 귀감을 제공할 것이다.

2. State-of-the-Art

2.1 American Fuzzy Lop

- **AFL : Coverage-guided fuzzer**
- testcase를 mutate하는 방식으로, 기존에 탐색하지 않는 프로그램 경로를 따르도록 한다.
- 만약 새로운 coverage를 달성하게 되는 경우, 해당 testcase를 queue에 저장해두고 활용한다.

	description
Coverage Guided Feedback	<ul style="list-style-type: none">- hybrid metric- bucket : 한 번의 실행에서 해당 edge가 실행된 횟수- path explosion을 방지하기 위해 edge가 얼마나 많이

	<p>실행되었는지 횟수를 측정한다.</p> <ul style="list-style-type: none"> - bucket -> hitcounts : bitmap 방식으로 카운트되며 각각의 바이트들이 하나의 edge를 표현한다. - trimming : testcase size를 줄이고 실행 속도를 향상시키기 위해하는 작업
Mutations	<ul style="list-style-type: none"> - deterministic : bit flips, ... - havoc(non-deterministic) : mutations이 랜덤으로 쌓이고 testcase의 크기가 변경됨 - splicing stage : 두 개의 testcase를 하나로 병합한 후 havoc를 적용하는 방식
Forkserver	<ul style="list-style-type: none"> - execve()의 오버헤드를 피하기 위해 AFL은 fork server를 삽입한다.
Persistent Mode	<ul style="list-style-type: none"> - fork()는 Persistent Mode의 경우 병목 현상(bottleneck)이 발생한다. - 각각의 testcase마다 fork를 수행하지 않고, loop로 패치하는 방식으로 대체할 수 있다.

2.2 Smart Scheduling

- 전체 coverage를 최대한 확보하면서 버그 탐지율을 높이는 것

	description
AFLFast	<ul style="list-style-type: none"> - 더 많은 분기를 탐색하고 더 많은 것을 찾기 위해 low-frequency path(희귀한 path)를 찾을 필요가 있다.
MOpt	<ul style="list-style-type: none"> - mutation scheduling - 퍼징 단계를 Pilot과 Code 모듈로 나누는데, 이 모듈들을 통해 확률 분석 기반 뮤테이션을 수행할 수 있다.

2.3 Bypassing Roadblocks

- coverage-guided fuzzer는 일반적으로 roadblock이라는 문제를 갖는다.
- roadblock : 복잡한 루틴 때문에 그 뒤에 있는 코드에 도달하기 힘들다.
 - ex) 문자열 및 체크섬 검사와 같은 비교
- roadblock 문제를 해결하기 위해 아래와 같은 연구가 진행되었다.

	description
LAF-Intel	<ul style="list-style-type: none"> - 멀티바이트 비교를 여러 단일바이트 비교로 <u>분할</u>하여 coverage guided fuzzer에게 전달한다. - LLVM에서의 작동을 염두하고 설계되었지만 인수 중 하나를 알고 있는 경우, 컴파일 타임에 strcmp 함수를 호출할 수도 있다. - ex) <ol style="list-style-type: none"> 1. >=을 >과 ==으로 나눈다. 2. unsigned integer → signed integer 3. 64,32 혹은 16비트인 정수를 8비트로 나눈다.
RedQueen	<ul style="list-style-type: none"> - I2S(Input-To-State)라는 방법으로 소스코드가 없는 환경에서도 <u>roadblock을 해결</u>할 수 있도록 제안되었다.

2.4 Mutate Structured

- 퍼저는 유효하지 않은 input을 마구 생성한다.
- 효과적이고 효율적인 input을 생성하는 모델링 기술이 필요하다.

	description
AFLSmart	<ul style="list-style-type: none"> - 프로토콜 퍼저인 PEACH fuzzer의 input model을 AFL에 결합한 것 - 샘플로 주어진 testcase의 구조를 분석하여 이에 알맞은 구조로 다른 input을 생성하도록 한다.

3. A New Baseline for Fuzzing

- AFL++에 적용된 기술들에 대한 내용을 다룬다.
- AFL++ : AFL을 fork한 버전

3.1 Seed Scheduling

- Seed Scheduling은 AFLFast 논문의 방법을 기반으로 한다.
- AFLFast 논문 : Markov Chain을 이용한 커버리지 기반의 그레이박스 퍼징을 제안
 - Power Schedules라는 개념을 소개
 - schedule 함수는 아래와 같은 인자를 가진다.
 1. queue에서 seed가 선택되는 시간
 2. 해당 seed에서 동일한 커버리지 input 값이 생성되는 수
 3. 동일한 커버리지로부터 생성되는 testcase 수
 - default schedule 함수는 explore이다.

3.2 Mutators

- AFL++은 AFL의 deterministic, havoc 파이프라인보다 더 많은 mutator를 통합한다.
- mutator는 다른 것들과 조합하여 사용할 수 있다.

3.2.1 Custom Mutator API

- AFL++는 AFL 연구를 위해 API를 제공한다.
- afl_custm_(de)init
- afl_custom_queue_get
- afl_custom_fuzz
- afl_custom_havoc_mutation
- afl_custom_post_process
- afl_custom_queue_new_entry
- afl_custom_init_trim
- afl_custom_trim
- afl_custom_post_trim

3.2.2 Input-To-State Mutator

- AFL++은 RedQueen의 Input-To-State(I2S)를 기반으로 한 mutator를 구현했다.

1. colorization

2. 각 비교의 확실적인 퍼징

- 각 비교에서 fuzzer가 새로운 입력을 생성하지 못하면 다음 실행에서 이 비교는 실행되지 않을 확률이 높다.
- I2S처럼 보이지만 아니다.
- CmpLog Instrumentation : LLVM이나 QEMU instrumentation에서 사용할 수 있다.

3.2.3 MOpt Mutator

- AFL++은 MOpt의 Core 및 pilot이 구현되어 있다.
- 참고로 MOpt에서 제안한 모듈은 PSO Initialization Module, Pilot Fuzzing Module, Core Fuzzing Module, PSO Updating Module)이 있다.

3.3 Instrumentations

- AFL++은 Instrumentation을 위해 여러 백엔드(LLVM, GCC, QEMU, Unicorn 및 QBDI)를 지원한다.
- 기존 AFL의 hitcount 매커니즘은 overflow되는 문제가 있다.
- 그래서 AFL++은 이를 개선하기 위해 NeverZero와 Saturated Counters라는 방법을 제안했다.
- 비트맵 항목에서 carry flag를 추가하여 overflow를 방지하는 NeverZero는 어느 방면에 있어서나 후자의 방법보다 효과적이라고 한다.
- 따라서 AFL++의 기본값을 NeverZero로 설정했다.

3.3.1 LLVM

- **커버리지 피드백을 측정한다.**
- LLVM 모드에서 사용자는 특정 소스 모듈을 기기에 지정할 수 있다.
 - ex) 많은 input 형식을 처리한다.
 - ex) persistent mode에서 AFL(stdin or file)의 대상 프로그램에 input을 전달하는 방법 외에도 AFL++은 공유 메모리를 통해 새로운 testcase를 전달할 수도 있다.

3.3.2 GCC

- old afl-gcc wrapper와 함께, AFL++ 은 GCC plugin을 제공한다.
- AFL LLVM 모드와 같은 persistent mode를 지원한다.
- 지원되는 기능은 LLVM과 완전히 똑같지는 않지만, feature parity에 도달하는 것을 목표로 AFL++ 에 추가 기능이 계획되어 있다.

3.3.3 QEMU

- 바이너리 전용 퍼징을 위한 버전 2.1 AFL QEMU는 AFL++ 에서 QEMU 3.1.1을 기반으로 대체되었다.
- QEMU 모드는 emulation time에 instrumentation을 추가한다.
- Compare Coverage
- Persistent Mode

3.3.4 Unicornfl

- 펌웨어와 같은 fuzzing blob 바이너리를 위해 AFL++ 는 Voss의 afl-unicorn을 통합하여 Unicornfl이라고 하는 것을 제공한다.
- AFL++ 의 unicornfl은 AFL++ 와 직접 상호 작용하기 위해 C API, Rust 및 Python 바인딩을 추가한다.
- AFL++ 전용 API를 사용하면 herness가 언제든지 빠른 persistent mode를 시작할 수 있다.

3.3.5 QBDI

- AFL++ 은 LLVM을 사용하여 컴파일러 instrumentation으로 Android 라이브러리를 퍼징할 수 있다.
- 만약 closed-source 라이브러리라면 QBDI Dynamic Binary Instrumentation 프레임워크도 AFL++ 에서 지원한다.

3.4 Platform Support

- GNU/Linux뿐만 아니라 Android, iOS, macOS, FreeBSD, OpenBDS, NetBSD에서 지원
- OS 기준으로, Debian, Ubuntu, NixOS, Arch Linux, FreeBSD, Kali Linux에서 지원
- QEMU를 사용하거나 Wine mode를 사용하면 GNU/Linux 상에서 Win32 바이너리에 대한 퍼징도 가능하다.

3.5 Snapshot LKM

- fork()는 Persistent Mode의 경우 병목 현상(bottleneck)이 발생한다.
- AFL++ 는 다른 논문에서 리눅스 커널 모듈 기반으로 Perffuzz 도구를 제시했다.
- 실제로 이는 fork 방식에 비해 2배의 성능 차이가 있었다.

4. Evaluation Use Cases

- AFL++ 을 사용했을 때 얼마나 성능이 좋을까?
- default AFL과 비교하여 MOpt, Ngram4, RedQueen, 등등을 각각 적용한 경우와 Ngram4+ Rare 또는 MOpt + RedQueen을 적용한 경우 등을 비교한다.

4.1 AFL++ Optimal

- 그래서 어떤 조합을 쓸 것인가?
- 논문에 정답은 나와있지 않다. 그저 각 상황에 따라 다르다.

5. Future Work

- AFL++ 개발은 현재 진행형이다.

5.1 Scaling

- testcase 마다 fork를 호출하는 것에 대한 문제가 있다.
- 멀티 스레딩을 지원하여 병렬 퍼저 간의 동기화 오버헤드를 최소화하는 연구 중이다.

5.2 Collision-Free instrumentation

- 현재 AFL의 해시 방식은 collision(충돌)이 발생한다.
- 속도? 정확성?
- source code와 emulation을 기반의 instrumentation을 고민 중이다.

5.3 Static Analysis for Optimal Fuzz Settings

- 현재 목표는 가장 좋은 instrumentation, mutation 및 scheduling을 사용하는 것이다.
- 사실 이것은 바이너리에 따라 달라진다.
- target에 대한 정적 분석을 통해 최적화된 솔루션을 제안할 예정이다.

5.4 Plug-in System

- Custom Mutator는 개발이 되었지만, 목표는 scheduler, executor, queue 기능을 플러그인 방식으로 제공할 예정이다.

6. Conclusion

- AFL++로 취약점을 많이 찾았다.
- ex) VLC, Vim, Tcpdump ...
- AFL++ 연구는 현재진행형이며, 많은 커뮤니티의 참여를 바란다.

Breaking Through Binaries: Compiler-quality Instrumentation for Better Binary-only Fuzzing

1. Introduction

- 크고 무거운 프로그램에서 symbolic execution과 같은 분석기술을 수동적으로 진행한다는 것은 힘들다.
- 그래서 fuzzing이 등장하였다.
- **Fuzzing** : mutation을 통해 대량의 testcase를 생성하고, target 프로그램에 미치는 영향을 관찰하는 것으로 구성됨
 - 최종 목표 : bug를 trigger하는 취약점을 찾는 것
- **Ceverage-guided grey-box fuzzing** : feedback loop을 추가하여 새로운 code path에 도달하는 testcase만 유지
- coverage는 target 프로그램의 기본 block에 삽입된 instrumentation을 통해 수집된다.
- ex) AFL, libFuzzer, honggfuzz
- 대부분의 최신 fuzzer는 target 프로그램의 소스 코드에 대한 액세스가 필요하다.
- 소스코드가 아닌 바이너리를 대상으로 한 퍼징은 컴파일러의 속도 등을 지원할 수 없어 효율성이 떨어진다.
- 이 문제를 해결하기 위해 **ZAFL**을 제안한다..

2. Background on Fuzzing

- Coverage-guided grey-box fuzzing : testcase를 자주 변경하여 코드 적용 범위를 늘리고 도달한 path를 수집하기 위해 instrumentation을 사용한다.
- Coverage-guided grey-box fuzzing의 기본 구성 요소에 대해 설명한다.

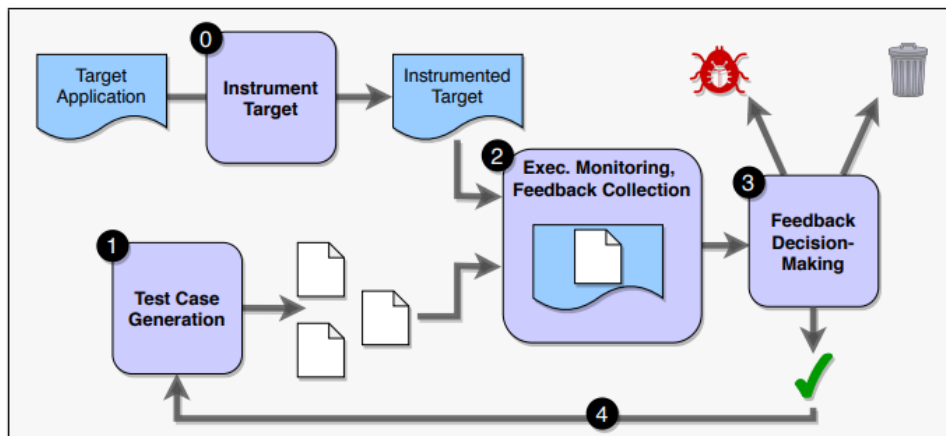


Figure 1: A high-level overview of the basic fuzzing workflow.

2.1 An Overview of Fuzzing

0. **Instrumentation** : code coverage를 tracking하기 위해 target 프로그램을 원하는대로 수정

1. **Test Case Generation** : seed를 선택하고 mutate하여 testcase를 생성

2. **Execution Monitoring and Feedback Collection** : testcase를 실행하고 instrumentation을 통해 feedback을 수집하기 위해 target 프로그램의 실행을 모니터링

3. **Feedback Decision-making** : 지정된 제약 조건(constraint)와 일치하는 실행 동작이 있는 testcase만 유지

4. Return to step 1.

2.2 Coverage-guided Grey-box Fuzzing

- Coverage-guided grey-box 퍼징은 가장 인기 있는 퍼징 기법
 - ex) aFL, honggfuzz, libFuzzes
- **Coverage-guided grey-box fuzzer** : 버그를 찾기 위해 가능한 많은 target

프로그램의 기능을 테스트하는 것을 목표로 code coverage를 증가시키는 testcase에 주목한다.

- **instrumentation** : target 프로그램에 대해 testcase가 어디까지 도달했는지 추적하는 데 사용된다.
- “컴파일러 기반/바이너리 전용”
- 대부분의 실제 퍼징은 소스가 없는 상태, 즉 바이너리 전용으로 수행된다.

3. Compiler-based Fuzzing Enhancements

- Coverage-guided fuzzing은 “컴파일러 기반/바이너리 전용”으로 나뉜다.
- 두 방법 모두 instrumentation을 사용하여 testcase가 어디까지 도달했는지 파악한다.
- “컴파일러 기반” 퍼징은 대부분 좋은 결과를 얻을 수 있다. (high throughput)
- 최근 피드백된 퍼저는 실행 시간만 줄인다.

Focus	Category	Effect on Fuzzing
Performance	Instrumentation	Overhead reduction from fewer blocks instrumented
	Pruning	
	Instrumentation	Overhead reduction from lighter-weight instrumentation
	Downgrading	
Feedback	Sub-instruction	Incremental coverage to guide code penetration
	Profiling	
	Extra-coverage	Ability to consider finer-grained execution behavior
	Behavior	

Table 1: Popular compiler-based fuzzing-enhancing program transformations, listed by category and effect.

- 따라서 우리는 소스코드가 없을 때, mutation 성능을 향상시키는 데에 초점을 둔다.

3.1 Instrumentation Pruning

- Graph reducibility 기술은 퍼징할 때 일부 기본 block을 instrumentation하지 않는다.
- 따라서 이 경우에는 전체 런타임 오버헤드를 줄일 수 있다.

3.2 Instrumentation Downgrading

- 오늘날 대부분의 퍼저는 testcase가 어느 edge를 거쳤는지 추적한다.
- edge는 일반적으로 퍼저에 의해 시작과 끝 블록만 기록된다.
- 이때, 연산이 많아질 수 있기 때문에 CollAFL의 컴파일러 instrumentation은 먼저 수행되는 블록을 쪼개어서(downgrading)하여 최적화를 한다.

3.3 Sub-instruction Profiling

- 최근 컴파일러 기반 퍼저는 멀티 바이트 조건문 단일 바이트 조건문으로 분해하는 profiling을 적용한다.
- ex) CmpCov, honggfuzz, laf-Intel

3.4 Extra-coverage Behavior Tracking

- 퍼징에 대한 연구는 기존 code path 경로 범위를 넘는 경로를 포함하는것이다.

	description
Context insensitive coverage	<ul style="list-style-type: none">- $A \rightarrow B \rightarrow C$ and $B \rightarrow A \rightarrow C$- 두 번째 경로는 새로운 edge를 제공하지 않아 놓치게 된다.
Context sensitive coverage	<ul style="list-style-type: none">- $A \rightarrow B \rightarrow C$ and $B \rightarrow A \rightarrow C$ 에서 $B \rightarrow C$ and $A \rightarrow C$- 두 가지 별개의 경로를 나타낼 수 있다.

4. Binary-only Fuzzing : the Bad & the Ugly

- 컴파일러 기반 퍼저는 프로그램 변환을 한다.
- ex) AFL++, CollAFL, laf-Intel
- 이것은 퍼징의 성능을 매우 좋게 만들지만 이를 지원하는 플랫폼은 존재하지 않는다.
- 기존의 바이너리 기반 instrumentation과 효과적인 바이너리 기반 instrumentation에 방해가 되는 제약 사항을 알아본다.

4.1 Limitations of Existing Platforms

- coverage-guided fuzzer는 빠른 컴파일러 instrumentation을 통해 testcase의 code coverage를 측정한다.
- 바이너리 전용 퍼징에서 code coverage는 아래 세가지 메커니즘 중 하나로 정의된다.

	description
Hardware-assisted Tracing	<ul style="list-style-type: none">- ex) Intel PT- 오버헤드
Dynamic Binary Translators	<ul style="list-style-type: none">- target 프로그램이 실행되고 있을 때 coverage를 바로 측정한다.- ex) DynamoRIO, PIN, QEMU- 오버헤드
Static Binary Rewriters	<ul style="list-style-type: none">- 런타임 전에 바이너리를 수정하여 성능을 향상시킨다.- ex) Dyninst- 오버헤드

4.2 Fundamental Design Considerations

- 컴파일러가 성능이 뛰어난 프로그램 변환을 지원하는 방법에 대한 분석은 아래의 설계를 포함한다.

	description
1. Rewriting versus Translation	<ul style="list-style-type: none">- dynamic(동적) 변환은 일반적으로 emulation을 통해 실행되는 target 바이너리의 source instruction stream을 처리한다. (런타임 오버헤드)

	<ul style="list-style-type: none"> - 컴파일러와 마찬가지로 static(정적) rewriting은 실행하기 전에 분석(control-flow 복구, instrumentation) 등을 수행하여 오버헤드가 낮다. - 따라서 static rewriting은 바이너리 전용 펌웨어에서 컴파일러 속도를 빠르게 하는데에 가장 적합하다. <p>* Instrumentation added via static rewriting</p>
2. Inlining versus Trampolining	<ul style="list-style-type: none"> - instrumentation code(coverage tracing)이 호출되는 방식 - trampolining : 점프를 위해 두 번의 전송이 필요(redirection) - inlining : redirection이 아닌 연속적인 명령을 실행 (오버헤드가 낮음) <p>* Instrumentation is invoked via inlining</p>
3. Register Allocation	<ul style="list-style-type: none"> - x86과 같은 CPU 레지스터가 있는 아키텍처에서 빠른 코드를 생성하는데에 가장 중요한 것은 플래그와 같은 condition code 레지스터를 수정하는 것이다. - saving/restoring하려면 스택에 push해야하는데, 이는 청나게 느리다. - 따라서 컴파일러는 이를 위해 레지스터가 사용되는지 확인한다. <p>* Must facilitate register liveness tracking</p>
4. Real-world Scalability	<ul style="list-style-type: none"> - DynamoRIO, QEMU, PIN과 같은 dynamic translators는 emulation 기술에 의존하기 때문에 기존의 static rewriting은 신뢰성이 많이 떨어진다. - 개발자들은 C++을 더 많이 사용하는 추세지만 바이너리는 여전히 C로 작성된다. - 많은 사람들이 strip되지 않은 바이너리를 디버깅하고 싶어하지만 실제로는 거의 불가능하다. - 그리고 대부분 리눅스에서 지원이 되고, Windows 64비트 중 일부는 지원이 되지 않는다. - 따라서 컴파일러 품질의 바이너리 전용 펌웨어 instrumenter는 이런 다양한 종류의 closed-source 바이너리를 지원해야 한다. <p>* Support common binary formats and platforms</p>

- Hardware-assisted tracing은 위의 1~ 3을 위반한다.
- DynamoRIO, PIN, QEMU는 static rewriting(dynamic translation(1))을 채택하므로 성능 저하가 발생한다.
- Dyninst와 RetroWrite는 static rewriting(1)을 수용하지만 trampoline 호출(2)에 의존하고 common format(4)을 지원하지 않는다.
- 따라서 바이너리 전용의 컴파일러 성능을 위한 instrumentation은 새로운 접근 방식을 요구한다.

****피드백 환영****