Fuzzing: State of the Art

Hongliang Liang, Member, IEEE, Xiaoxiao Pei, Xiaodong Jia, Wuㄴwei Shen, Member, IEEE, and Jian Zhang, Senior Member, IEEE

Abstract - 가장 널리 사용되는 소프트웨어 테스팅 기술 중 하나인 퍼징은 수많은 테스트 입력을 생성함으로써 소프트웨어 버그 및 취약성과 같은 프로그램의 다양한 약점을 찾아 낼 수 있습니다. 이러한 효과 때문에, 퍼징은 의미 있는 버그 탐지 방법으로 간주됩니다. 이 논문에서는 일반적인 장애와 분류에 초점을 둔 퍼징 (fuzzing)에 대한 개요를 제시하고 주요 장애물과 이러한 장애물을 극복하거나 완화하기 위한 최첨단 기술에 대해 자세히 논의합니다. 우리는 광범위하게 사용되는 몇 가지 퍼징 도구를 추가로 조사하고 분류합니다. 우리의 주된 목표는 이 분야의 종사자들에게 소프트웨어 테스팅 및 보안 분야에서 퍼징 방법을 개선하기 위해 퍼징 및 잠재적 해결책에 대한 더 나은 이해를 갖추는 것입니다. 미래의 연구에 영감을 주기 위해, 우리는 또한 퍼징과 관련하여 향후 방향을 예측합니다.

I. INTRODUCTION

FUZZING은 "파일, 네트워크 프로토콜, 응용 프로그래밍 인터페이스 (API) 호출 및 기타 대상에서 비롯된 무효 데이터를 응용 프로그램 입력으로 사용하여 수많은 경계 사례를 다루는 자동 테스트 기술로 악용 될 수 있는 취약점이 없는지 확인합니다." [1]. Fuzzing은 Miller et al. [2] 로부터 1988년 처음으로 제안되었으며, 그 이후로 소프트웨어에서 버그를 찾는 효과적이고 빠르고 실용적인 방법으로 발전해왔다. 퍼징(fuzzing)의 핵심 아이디어는 소프트웨어 오류를 유발할 수 있는 많은 테스트 사례로 대상 프로그램을 생성하고 공급하는 것입니다. 퍼징의 가장 창의적인 부분은 적절한 테스트 케이스를 생성하는 방법이며, 현재 널리 사용되는 방법에는 적용 범위 가이드 전략, 유전 알고리즘, 기호 실행, taint analysis 등이 있습니다. 이러한 방법을 기반으로 현대의 퍼징 기술은 숨겨진 버그를 발견하는데 있어 매우 지능적입니다. 따라서 테스트 성공이 의미 있는 소프트웨어 품질 용어로 정량화 될 수 있는 독창적인 테스트 방법으로 퍼징은 중요한 이론적 및 실험적 역할을 합니다. 다른 방법을 평가해야하는 비교 표준으로 사용됩니다 [6]. 또한, 퍼징은 점차적으로 목표 기술의 정적 정보와 동적 정보를 시너지 효과 있게 결합하는 합성 기술로 진화하여 더 나은 테스트 케이스가 생성되고 더 많은 버그를 탐지해낼 수 있습니다.

퍼징은 대상 프로그램에 잘못된 형식 또는 semivalid(반유효한) 테스트 사례를 지속적으로 보내 공격을 시뮬레이션합니다. 이러한 불규칙한 입력 덕분에 fuzzing 도구라고도 불리는 fuzzer는 이전에 알려지지 않은 취약점을 발견 할 수 있습니다 [8] - [11]. 그리고 이것이, 퍼징이 소프트웨어 테스팅에서 중요한 역할을 하는 주요 이유 중 하나입니다. 그러나 테스트 케이스를 생성하는 과정에서 코드 커버리지[1)]가 낮아지는 것은 퍼징(fuzzing)이 극복하려는 주요 단점입니다. 위에서 언급했듯이, 이 문제를 완화하기 위해 몇 가지 방법이 사용되고 있으며

1) 코드 커버리지: 전체 코드가 테스트 된 정도. 소프트웨어의 테스트를 논할 때 얼마나 테스트가 충분한가를 나타내는 지표중 하나. 코드 커버리지가 90%라면 90%의 코드가 테스트에 사용된 것.

퍼징 (fuzzing)은 인상적인 발전을 이루었습니다. 요즘 fuzzing은 컴파일러, 어플리케이션, 네트워크 프로토콜, 커널 등을 포함한 다양한 소프트웨어와 syntax error recovery [12] 및 fault localization [13]과 같은 여러 어플리케이션 영역을 테스트하는 데 널리 사용됩니다.

A. Motivation

이 개요를 작성하도록 동기를 부여하는 데는 두 가지 이유가 있습니다.

1) 퍼징(fuzzing)은 버그를 발견 할 수 있는 효과적인 능력 때문에 소프트웨어 보안 및 안정성 영역에서 점점 더 많은 주목을 받고 있습니다. 구글이나 마이크로소프트와 같은 많은 IT 기업들이 퍼징 기술을 연구하고 대상 프로그램의 버그를 찾기 위해 퍼지 도구 (예 : SAGE [14], Syzkaller [15], SunDew [16] 등)를 개발하고 있습니다.

2) 지난 몇 년 동안 퍼징에 대한 체계적인 검토가 없다. 일부 논문은 퍼지의 개요를 제시하지만, 일반적으로 특정 기사 [1], [17] 또는 특정 시험 주제에 대한 설문 조사 [18] [19]에 대한 검토일 뿐입니다. 따라서 우리는 이 분야에서 최신 방법과 새로운 연구 결과를 요약하기 위해 포괄적인 리뷰를 작성해야한다고 생각했습니다. 따라서 이 백서를 통해 초보자가 퍼징에 대한 전반적 이해를 얻을 수 있을 뿐 아니라 전문가들 또한 퍼징에 대한 철저한 검토를 할 수 있기를 바랍니다.

B. Outline

이 논문의 나머지 부분은 다음과 같이 구성되어있습니다 : 섹션 II는 설문 조사에서 사용 된 검토 방법과 일부 선정 된 논문의 간략한 요약 및 분석을 제공합니다. 섹션 III에서는 퍼징 (fuzzing)의 일반적인 프로세스를 설명합니다. 섹션 IV는 퍼징 방법의 분류를 소개합니다. 섹션 V는 퍼지 (fuzzing)의 최신 기술을 설명합니다. 섹션 VI는 응용 분야 및 문제 영역별로 분류 된 몇 가지 인기 있는 퍼저를 제공합니다. 설문 조사 결과, 제 7 장의 미래 연구를 위한 배경으로서 많은 연구 과제가 확인되었습니다. 마지막으로, VIII장에서 논문을 결론짓습니다.

II. REVIEW METHOD

퍼징에 대한 포괄적인 서베이를 수행하기 위해 우리는 Kitchenham [20]과 Webster and Watson [21]의 가이드라인에서 영감을 받은 체계적이고 구조화 된 방법을 따랐습니다. 다음 섹션에서는 연구 방법, 수집 된 데이터 및 분석에 대해 자세히 소개합니다.

A. Research Questions

이 조사는 주로 퍼지에 관한 다음 연구 질문에 답하는 것을 목표로 합니다.

1) RQ1 : 연구 문제를 해결하는 데있어 핵심적인 문제와 해당 기법은 무엇입니까?

2) RQ2 : 사용 가능한 fuzzers와 알려진 응용 프로그램 도메인은 무엇입니까?
3) RQ3 : 향후 연구 기회 또는 방향은 무엇입니까?

V(5) 장에서 답을 얻은 RQ1은 원래 소개된 이 분야의 최첨단 기술을 요약 한 퍼징(fuzzing)에 대한 심층적 견해를 탐구 할 수 있게 해줍니다. 섹션 VI(6) 에서 논의되는 RQ2는 퍼징(fuzzing)의 범위와 다른 영역에 대한 적용 가능성에 대한 통찰력을 제공하기 위해 제안되었습니다. 마지막으로, 이전 질문에 대한 답을 토대로, 우리는 VII(7) 절에서 답하는 RQ3에 대한 미해결 문제 및 연구 기회를 파악할 것으로 기대합니다.

B. Inclusion and Exclusion Criteria

방법, 도구, 특정 테스트 문제에 대한 적용, 경험적 평가 및 조사와 같은 퍼징의 모든 측면과 관련된 문서를 찾기 위해 기존 문헌을 면밀히 조사했습니다. 비슷한 내용의 동일한 저자가 쓴 기사는 의도적으로 분류되어 보다 엄격한 분석을 위해 별도의 기여로 평가되었습니다. 그런 다음 결과 발표에 큰 차이가 없는 이 기사들을 그룹화 했습니다. 다음 기준에 따라 해당 논문을 제외했습니다.

1) 컴퓨터 과학 분야와 관련이 없다.
2) 영어로 쓰여지지 않았다.
3) 평판이 좋은 출판사에 의해 출판되지 않았다.
4) 평판이 좋은 출판사에 의해 출판되었지만 6 페이지 미만이다.
5) 웹을 통해 액세스 할 수 없다.

예를 들어, Wiley InterScience 웹 사이트의 검색 인터페이스를 사용하여 fuzzing / fuzz testing / fuzzer와 같은 키워드가 포함된 논문들을 찾았고, 결과로 얻은 32 개의 논문들 중 7 개는 초록에 따라 컴퓨터 과학 분야에만 관련되어있었습니다.

C. Source Material and Search Strategy

퍼징 관련 모든 출판물을 대상으로 한 전체 설문 조사를 제공하기 위해 1990 년 1 월부터 2017 년 6 월까지 3 단계를 통해 350 개가 넘는 퍼징 관련 논문을 포함하는 레포지토리(저장소)를 만들었습니다. 먼저 IEEE XPlore, ACM Digital Library, Springer Online Library, Wiley InterScience, USENIX 및 Elsevier ScienceDirect Online Library와 같은 주요 온라인 저장소를 검색하고 "fuzz testing", "fuzzing", "fuzzer", "random testing" 또는 "swarm [2]testing"를 제목, 초록 또는 키워드의 키워드로 사용합니다. 두 번째로, 수집된 논문들의 초록을 바탕으로 일정 기준에 따라 일부를 제외했습니다. 초록으로 결정할 수 없는 경우 직접 논문을 읽고 결정했습니다. 이 단계는 두 명의 다른 저자가 수행했습니다. 후보 논문들은 서베이 범위 내에서 171개로 축소되었습니다. 이 논문들을 1차 연구라고 부르도록 하겠습니다 [20]. 아래 표 1은 각 출처에서 추출한 1차 연구의 수를 나타냅니다.

---

2) swarm: “군중”, “떼” 라는 뜻으로 해킹 분야에서 사용될 때는 광범위한 공격을 동시다발적으로 수행하는 것을 말함

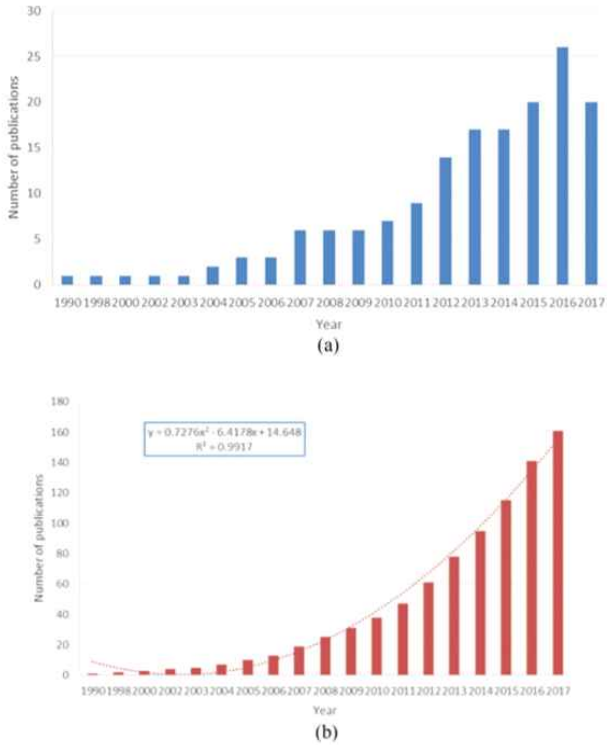
<표 1> 출판사별 퍼징 주요연구 수

출판사 (Publisher)	주요 연구(Primary Studies)
ACM digital library	33
Elsevier Science Direct	10
IEEEXplore digital library	87
Springer online library	17
Wiley InterScience	7
USENIX	11
Semantic Scholar	6
Total	**171**

평판이 좋은 출판사의 일부에 초점을 맞추기 때문에 검색이 모든 관련 논문을 완전히 포함하지는 못합니다. 그러나 우리는 이 백서의 전반적인 경향이 정확하고 퍼징에 대한 최신 기술에 대한 적절한 그림을 제공한다고 확신합니다.

D. Summary of Results

다음 섹션에서는 퍼블리싱 동향, 장소, 저자 및 퍼징에 대한 지리적 분포와 관련하여 당사의 주요 연구를 요약합니다.



[그림 1] 1990년 1월부터 1일부터 2017년 6월 30일 사이 출판된 퍼징 논문. (a) 연도별 출판 수. (b) 연도별 출판 누적 수.

1) 출판 동향 : 그림 1 (a)는 1990 년 1 월부터 2017 년 6 월 30 일 사이에 퍼징에 관한 출판물 수를 나타낸 것 입니다. 이 그래프에서 논문 수는 2004년 이후, 특히 2009년 이후 지속적으로 증가한 것으로 나타났습니다. 논문의 누적 수는 그림 1 (b)에 제시되어있다. 우리는 높은 결정 계수[3] (R2 = 0.992)를 가진 2차 함수에 가까운 산식을 계산했는데, 이것은 강력한 다항식 성장과 건강에 대한 지속적인 신호와 이 주제에 대한 관심을 나타낸다. 추세가 계속된다면, 이 기법을 처음 도입 한 후 2018 년 말까지 유명한 출판사가 발행 한 200여 가지의 퍼지 논문이 나옵니다.

2) 출판 장소 : 171 개의 주요 연구가 78 개의 별개의 장소에서 출판되었습니다. 그것은 퍼징 관련 문헌에 의해 다루어지는 영역이 매우 넓음을 의미합니다. 이 기술은 매우 실용적이며 여

러 테스트, 안정성 및 보안 도메인에 적용되었기 때문일 수 있습니다. 개최지 유형과 관련하여 회의 및 심포지엄 (73%), 저널 (15%), 워크샵 (9%), 기술 보고서 (3%) 순으로 대부분의 논문이 발표되었습니다. 표 II는 적어도 세 개의 퍼지 논문이 제시된 장소를 나열합니다.

<표 2> 퍼징 관련 출판이 가장 많았던 학회

장소 (Venue)	논문 수 (Papers)
Int. Conf. on Programming Language Design and Implementation (PLDI)	12
Int. Symposium on Software Testing and Analysis (ISSTA)	9
Int. Conf. on Software Engineering (ICSE)	8
Int. Conf. on Automated Software Engineering (ASE)	7
Int. Conf. on Software Testing, Verification & Validation (STVV)	6
USENIX Security Symposium (USENIX SEC.)	5
Workshop on Offensive Technologies (WOOT)	5
IEEE Transaction on Reliability (TR-IEEE)	4
Network and Distributed System Security Symposium (NDSS)	4
ACM Conf. on Computer and Communications Security (CSS)	4
IEEE Symposium on Security and Privacy (S&P)	3
Communication of ACM (CACM)	3
ESEC/ACM Foundations of Software Engineering (FSE)	3
Security and Communcation Networks (SCN)	3

3) 결정 계수(R제곱): 통계학에서 추정한 선형 모형이 주어진 자료에 적합한 정도를 재는 척도. 선형 회귀모델에서 독립변수 X로 설명되는 응답변수 Y의 비례적 변동량을 나타냄. 결정계수가 클수록 변동성 증가.

3) 출판물의 지리적 분포 : 각 주요 연구의 지리적 기원은 1저자의 소속 국가와 관련시켰다. 흥미롭게도 우리는 표 III에 제시된 바와 같이 미국, 중국, 독일이 상위 3 위인 171 개 기본 연구가 모두 22 개국에서 시작된 것으로 나타났습니다 (4개 이상의 논문을 출판한 국가만 해당). 대륙별로는 43 %가 미국, 32 %가 유럽, 20 %가 아시아, 5 %가 오세아니아 출신이다. 이는 퍼징 커뮤니티가 소수의 국가에 의해 형성되었지만 전 세계에 꽤 분포되어 있음을 의미합니다.

<표 3> 출판 논문의 지역 분산

국가 (Country)	논문 수 (Papers)
United States	71
China	17
Germany	12
United Kindom	9
France	9
Austria	95
Switzerland	6
Singapore	5
Netherland	5
Italy	4
Korea	4

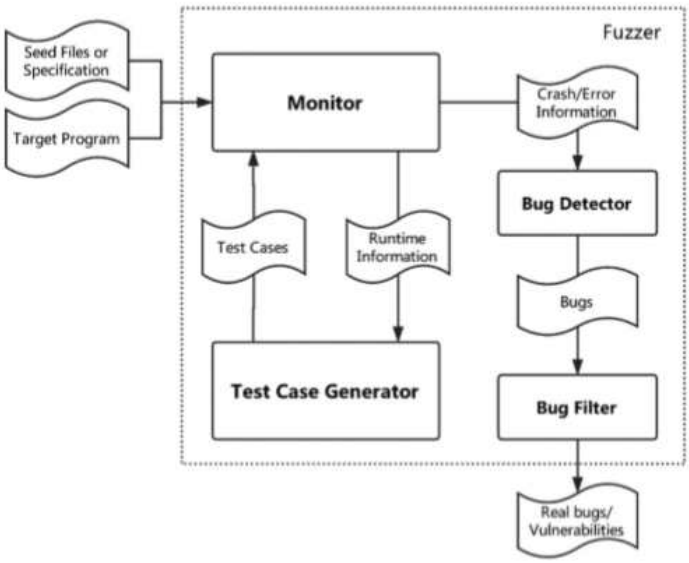
<표 4> 퍼징 관련 논문 저자 상위 10

저자 (Author)	연구기관 (Institution)	논문 수 (Papers)
P. Godefroid	Microsoft Research	9
S. Rawat	Vrije Universiteit Amsterdam	6
T. Y. Chen	Swinburne University of Technology	6
A. Arcuri	Simula Research Laboratory	4
C. Cadar	Standford University	4
A. Groce	Oregon State University	4
B. P. Miller	University of Wisconsin	4
Y. Chen	University of Utah	3
V. T. Pham	National University of Singapore	3
K. Sen	University of California	3

4) 연구원 및 단체 : 우리는 검토 중인 171 개의 1차 연구에서 125 명의 서로 다른 연구자를 확인했습니다. 표 4는 퍼징 논문 수에 대한 상위 저자들과 소속을 제시합니다.

III. GENERAL PROCESS OF FUZZING

Fuzzing은 테스트 사례를 자동으로 생성 할 수 있는 소프트웨어 테스트 기술입니다. 테스트 케이스들을 대상 프로그램에서 실행 한 다음 동작을 관찰하여 프로그램에 버그 또는 취약점이 있는지 여부를 확인하는 퍼징(fuzzing)의 일반적인 프로세스가 그림 2에 나와 있습니다.



[그림 3] 퍼징의 일반적인 프로세스

대상 프로그램 (Target Program) : 대상 프로그램은 테스트중인 프로그램입니다. 바이너리 또는 소스 코드 일 수 있습니다. 그러나 실제 소프트웨어의 소스 코드는 일반적으로 쉽게 액세스 할 수 없기 때문에 fuzzer는 대부분의 경우, 바이너리 코드를 대상으로 합니다.

모니터 (Monitor) :이 구성 요소는 일반적으로 흰색 상자 또는 회색 상자의 fuzzer에 내장되어 있습니다. 모니터는 코드 계측, 오염 분석 등과 같은 기술을 활용하여 코드 적용 범위, 데이터 흐름 또는 대상 프로그램의 기타 유용한 런타임 정보를 수집합니다. black-box fuzzer에서는 모니터가 필요 없습니다.

테스트 케이스 생성기 (Test case generator) : fuzzer가 테스트 케이스를 생성하는 주요 방법은 돌연변이 기반 및 문법 기반 [1] 메소드 두 가지가 있습니다. 첫 번째 방법은 well-formed seed 파일을 무작위로 돌연변이화하거나 런타임 중에 수집 된 target-program-oriented 정보를 기반으로 조정할 수 있는 입력을 생성합니다. 첫 번째 방법과는 반대로 두 번째 방법은 시드 파일이 필요하지 않습니다. 문법과 같은 사항들로부터 입력을 생성하기 때문입니다. 대부분의 경우, 퍼징의 테스트 케이스는 일반적으로 초기 구문 분석 단계를 통과할 만큼 유효하고 대상 프로그램의 심층 논리에서 버그를 유발할 만큼 무효한 “반유효” 입력 값들입니다.

버그 탐지기 (Bug detector) : 사용자가 대상 프로그램에서 잠재적 버그를 발견 할 수 있도록 버그 감지기 모듈이 설계되고 구현됩니다. 대상 프로그램이 충돌하거나 오류를 보고하면 버그 감지기 모듈은 관련 정보 (예 : 스택 추적 [22])를 수집하고 분석하여 버그가 있는지 여부를 결정합니다. 경우에 따라 디버거를 수동으로 사용하여 이 모듈의 대안으로 예외 정보 [23] - [25]를 기록 할 수 있습니다.

버그 필터(Bug filter) : 테스터는 일반적으로 정확성이나 보안 관련 버그에 중점을 둡니다. 따라서 보고 된 모든 버그로부터 악용 가능한 버그 (즉, 취약성)를 제거하는 것은 중요한 작업이며 일반적으로 수동으로 수행 됩니다 [23]. 이는 시간이 많이 소요될 뿐만 아니라 해결하기도 어렵습니다. 현재, 일부 연구 [26]에서 문제를 완화하기위한 다양한 접근법을 제안했습니다. 예를 들어, fuzzer의 결과 (버그 유도 테스트 케이스)를 정렬하여 다양하고 흥미로운 테스

트 케이스가 우선순위가 지정되며, 테스터는 지푸라기 더미에서 바늘을 찾듯 원하는 버그를 찾기 위해 수동적으로 프로세스를 처리할 필요가 없습니다.

fuzzing의 과정을 보다 명확하게 설명하기 위해, 예를 들어, 파일 변환기 인 png2swf를 테스트하기 위해 돌연변이[4] 기반 커버리지 가이드 fuzzer 인 American fuzzy lop (AFL) [27]을 사용합니다. 이 예제에서는 AFL에 대한 png2swf 실행 파일을 대상 프로그램으로 제공합니다. 첫째, 돌연변이 기반 기술을 채택하기 때문에 AFL을 위한 몇 가지 시드 파일 (이상적인 시드는 잘 형성되어있고 크기가 작아야 함)을 제공합니다. 둘째로, 우리는 AFL을 간단한 명령으로 실행합니다 (예 : "afl-fuzz -i [input_directory] -o [output_directory] -Q -[target_directory] (@@)", 대상 프로그램이 파일에서 입력을 받으면 " @@" 필수적이다). 테스트 과정에서 AFL의 "*모니터*" 구성 요소는 이진 계측을 통해 특정 런타임 정보 (이 경우 경로 커버리지 정보)를 수집 한 다음, 이 정보를 "*테스트 케이스 생성기*" 구성 요소에 전달하여 테스트 케이스 생성 프로세스를 안내합니다. 일반적인 전략은 다음 번 돌연변이를 위한 새로운 프로그램 경로를 커버 할 수 있는 테스트 케이스를 저장하고, 그렇지 않은 경우를 버리는 것입니다. 셋째, 새로 생성 된 테스트 케이스는 대상 프로그램의 입력으로 "모니터"로 다시 전달됩니다. 이 프로세스는 AFL 인스턴스를 중지하거나 주어진 제한 시간에 도달 할 때까지 계속됩니다. 게다가 AFL은 실행 시간, 고유 한 충돌 횟수, 실행 속도 등과 같은 런타임 동안 화면에 유용한 정보를 인쇄합니다. 따라서 대상 프로그램을 손상시킬 수있는 일련의 테스트 사례를 얻을 수 있습니다. 마지막으로 테스트 케이스를 분석하고 대상 프로그램을 수동으로 또는 다른 도구를 사용하여 충돌시키는 버그를 식별합니다. AFL 발견은 주로 버퍼 오버플로, 액세스 위반, 스택 스매시 등과 같은 메모리 연산과 관련이 있습니다. 대개 프로그램이 중단되거나 크래커가 악용 할 수 있습니다.

IV. BLACK, WHITE, OR GRAY?

Fuzzing의 기술은 실시간으로 대상 프로그램에서 필요한 정보의 양에 따라 blackbox, white box 및 gray box로 구분됩니다. 이 정보는 코드 커버리지, 데이터 커버리지, 프로그램의 메모리 사용량, CPU 사용률 또는 테스트 케이스 생성을 안내하는 기타 정보 일 수 있습니다.

A. Black-Box Fuzzing

전통적인 블랙 박스 퍼징은 "블랙 박스 무작위 테스트"라고도합니다. 대상 프로그램이나 입력 형식에서 정보를 요구하는 대신, 블랙 박스 랜덤 테스트는 잘못된 형식의 시드 파일을 무작위로 정해 무효 입력을 만듭니다. 돌연변이 규칙은 비트 플립, 바이트 카피, 바이트 제거 등이 될 수있다 [28]. 최근의 블랙 박스 퍼징은 semivalid(반유효) 입력을 생성하기 위해 문법 [29]이나 입력 및 특정 지식 [30]을 사용한다. 퍼지 (fuzz) [31]와 트리니티 (Trinity) [32]와 같은 블랙박스 퍼저(black-box fuzzers)는 소프트웨어 산업에서 널리 사용되는데, 그 이유는 버그를 발견하고 사용하기 쉽기 때문입니다. 예를 들어, Trinity는 Linux 커널의 시스템 호출

4) 돌연변이 기반 테스트 (mutant based test) : 원본 프로그램에 의도적인 변경을 가해 돌연변이 프로그램을 만들어 테스트 하는 것. 테스트 케이스의 적합성을 판단하기 위해 수행하며 테스트 결과의 신뢰성 확보가 목적임.

인터페이스를 모호하게 만드는 것을 목표로 합니다. 테스터는 제공된 템플릿을 먼저 사용하여 입력 유형을 설명해야합니다.

```
1 void test(char input[4]) {
2  int count = 0;
3  if (input[0] == 'o' ) count++;
4  if (input[1] == 'o' ) count++;
5  if (input[2] == 'p' ) count++;
6  if (input[3] == 's' ) count++;
7  if (count == 4) abort(); //error
8 }
```

[그림 4] 예시 함수

그런 다음 Trinity는 더 높은 범위에 도달 할 수있는 더 유효한 입력을 생성 할 수 있습니다. 이 fuzzer는 많은 버그를 발견했습니다.

그러나 이 기술의 단점도 분명합니다. 그림 3에서 보이고 있는 함수를 고려하면, 라인 7의 abort () 함수는 라인 1의 입력 파라미터가 랜덤하게 할당 될 때 도달 할 수 있는 기회가 1/232에 불과합니다. 이 예제는 대상 프로그램에서 많은 수의 경로를 커버하는 테스트 케이스를 생성하는 블랙 박스 퍼지 (black box fuzzing)가 왜 어려운지 직관적으로 설명합니다. 이러한 맹점으로 인해 블랙 박스 퍼징은 종종 실제로 코드 커버리지가 낮습니다 [33]. 이것이 최근 fuzzer 개발자가 주로 리버스 엔지니어링 [34], 코드 계측 [35], 테인트 분석 [23], [36], [37] 및 기타 기술을 중점적으로 고려하여 fuzzer를 "더 똑똑하다"고 여기는 이유이며, 화이트 박스 및 그레이 박스 퍼징이 최근에 더 많은 주목을 받고 있는 이유이기도 합니다.

B. White-Box Fuzzing

화이트 박스 퍼징은 대상 프로그램의 내부 논리 지식을 기반으로 합니다 [17]. 이론적으로는 대상 프로그램의 모든 실행 경로를 탐색 할 수있는 방법을 사용합니다. Godefroid 등이 처음 제안한 바 있습니다. [38]. 블랙 박스 퍼징의 맹점을 극복하기 위해 그들은 대체 방법을 연구하고 화이트 박스 퍼징 (white-box fuzzing)이라고 불렀습니다. 화이트 박스 퍼징은 동적 기호 실행 (concolic execution [39]이라고도 함) 및 커버리지 최대화 검색 알고리즘을 사용하여 대상 프로그램을 철저하고 빠르게 검색 할 수 있습니다.

블랙 박스 퍼징과 달리 화이트 박스 퍼징은 대상 프로그램의 정보를 필요로하며 필요한 정보를 사용하여 테스트 케이스 생성을 안내합니다. 구체적으로, 주어진 구체적인 입력으로 실행을 시작하는 화이트 박스 fuzzer는 입력 아래의 실행 경로를 따라 모든 조건문에서 기호 제약을 수집합니다. 따라서 한 번 실행 한 후 화이트 박스 fuzzer는 논리 제약을 사용하여 모든 기호 제약 조건을 결합하여 경로 제약 조건 (path constraint, PC)을 형성합니다. 그러면 화이트 박스 fuzzer가 제약 조건 중 하나를 체계적으로 무효화하고 새로운 PC를 해결합니다. 새로운 테스트 케이스는 프로그램이 다른 실행 경로를 실행하도록 유도합니다. 커버리지 최대

화 휴리스틱 검색 알고리즘을 사용하여 화이트 박스 퍼지는 대상 프로그램의 버그를 가능한 빨리 찾아 낼 수 있습니다 [38].

이론상으로, 화이트 박스 퍼징은 모든 프로그램 경로를 커버하는 테스트 케이스를 생성 할 수 있습니다. 그러나 실제로는 실제 소프트웨어 시스템의 수많은 실행 경로와 심볼릭 실행 (symbolic execution) 중에 제약 조건을 해결하는 부정확성과 같은 많은 문제로 인해 (섹션 5-B 참조), 화이트 박스 퍼징의 코드 커버리지 범위는 100 %를 달성할 수 없습니다. 가장 유명한 화이트 박스 퍼지 중 하나는 SAGE [14]입니다. SAGE는 대규모 Windows 응용 프로그램을 대상으로 하며 실행 최적화를 통해 많은 수의 실행 추적을 처리합니다. 자동으로 소프트웨어 버그를 발견하고 인상적인 결과를 얻을 수 있습니다.

C. Gray-Box Fuzzing

그레이 박스 퍼징은 블랙 박스 퍼징 (fuzzing)과 화이트 박스 퍼징 (white-box fuzzing)의 중간에 서서 대상 프로그램에 대한 부분 지식으로 소프트웨어 오류를 효과적으로 나타냅니다. 그레이 박스 퍼징에서 흔히 사용되는 방법은 코드 계측[5](code instrumentation)입니다 [40, 41]. 이 방법으로, 그레이 박스 fuzzer는 런타임에 대상 프로그램의 코드 적용 범위를 얻을 수 있습니다. 그런 다음, 이 정보를 이용하여 더 많은 실행 경로를 커버하거나 버그를 더 빨리 발견 할 수 있는 테스트 사례를 생성하기 위해 변이 전략을 조정 (예 : 유전 알고리즘 [28], [42] 사용)합니다. Gray-box fuzzing에 사용 된 또 다른 방법은 오염 데이터 흐름을 추적하기위한 코드 계측을 확장하는 테인트 분석(taint analysis) [23, 43] - [45]입니다. 따라서 fuzzer는 대상 프로그램의 잠재적 공격 지점에 영향을 줄 수 있는 특정 입력 필드를 변경하는 데 중점을 둘 수 있습니다.

그레이 박스 및 화이트 박스 퍼징은 두 가지 방법 모두 테스트 프로그램 생성을 이끌어내기 위해 대상 프로그램의 정보를 사용한다는 점에서 매우 유사합니다. 그러나 그레이 박스 퍼징은 대상 프로그램의 런타임 정보 (예 : 코드 커버리지(code coverage), 오염 데이터 흐름 (taint data flow) 등)를 사용하여 어떤 경로를 탐색했는지 결정합니다 [28]. 또한 그레이 박스 퍼징은 획득한 정보를 사용하여 테스트 케이스 생성을 이끌어내지만, 이 정보를 사용하면 새로운 경로를 다루거나 특정 버그를 유발할 수 있는 더 나은 테스트 케이스가 반드시 생성된다고 보장 할 수 없습니다. 반대로 화이트 박스 퍼징은 대상 프로그램의 소스 코드 또는 바이너리 코드를 사용하여 모든 실행 경로를 체계적으로 탐색합니다. concolic 실행[6](conclic execution)과 제약 조건 해결자를 사용하여 생성 된 테스트 사례가 대상 프로그램에서 새로운 실행 경로를 탐색하도록 유도합니다. 따라서 화이트 박스 퍼징은 퍼징 프로세스 중 맹목(blindess)을 보다 철저하게 감소시키는데 도움이 됩니다. 요약하면, 두 가지 방법 모두 블랙 박스 모호성의 실명을 완화하기 위해 대상 프로그램의 정보를 사용하지만 사용하는 정도는 다릅니다.

BuzzFuzz [46]는 BuzzFuzz의 개발자가 white-box fuzzer라고 했지만 회색 박스 fuzzer가

---

5) 코드 계측 (code instrumentation): 컴퓨터 프로그래밍에서 “인스트루먼데이션”은 오류를 진단하거나 추적 정보를 쓰기 위해 제품의 성능 정도를 모니터하거나 측정하는 기능을 의미한다.

6) concolic 실행 (concolic execution): concrete execution과 symbolic processing의 혼합 개념. concrete execution은 실제로 코드를 실행하는 실제수행으로 구체적인 입력 값이 있고, 이 입력값을 기반으로 연산을 수행하여 출력값 도출. symbolic execution은 입력 값을 구체적인 값이 아닌 일종의 기호로 주어 프로그램을 실행.

작동하는 방식을 보여주는 좋은 예 입니다. 이 도구는 대상 프로그램의 부분 지식 (흐린 데이터 흐름) 만 가져 오기 때문에 이 도구를 회색 상자의 fuzzer로 간주합니다. BuzzFuzz는 다음과 같이 작동합니다. 먼저 대상 프로그램에서 수집 된 오염 정보를 기반으로 입력 된 데이터의 어느 부분이 사전 정의 된 공격 포인트에 영향을 미칠 수 있는지 계산합니다. (BuzzFuzz는 lib 호출을 잠재적 공격 지점으로 간주합니다.) 그런 다음 입력 데이터의 중요한 부분을 변경하여 새로운 테스트 케이스를 만듭니다. 마지막으로 새 테스트 사례를 실행하고 대상 프로그램이 충돌하는지 여부를 관찰합니다. 이를 통해 BuzzFuzz는 보다 타깃 지향적이고 효과적인 방식으로 버그를 찾아 낼 수 있습니다. 더 중요한 것은 공격 지점은 개발자의 관심에 따라 특정 라이브러리 호출 또는 취약성 패턴 등으로 정의 할 수 있습니다.

D. How to Choose?

트리거 되거나 발견 된 가능성에 따라 버그는 "얕은(shallow)" 버그와 "숨겨진(hidden)" 버그의 두 가지 범주로 분류 할 수 있습니다. 실행 초기에 타겟 프로그램이 충돌하게 하는 버그는 "얕은" 버그, 예를 들어 선행 조건부 분기 없이 잠재적으로 0으로 나누기 연산으로 간주됩니다. 반대로 프로그램 논리에 깊이 존재하고 트리거하기 어려운 버그는 복잡한 조건부 분기에 있는 버그와 같은 "숨겨진" 버그로 간주됩니다. "얕은" 및 "숨겨진" 버그를 식별하는 표준 방법은 없습니다. 따라서 일반적으로 사용되는 fuzzer의 평가 기준은 코드 적용 범위, 발견되는 버그의 수와 악용 가능성입니다. 일반적으로 임의의 돌연변이 방법(random mutation method)을 사용하여 테스트 케이스를 생성하는 전통적인 블랙 박스 fuzzer는 높은 코드 커버리지에 도달 할 수 없으므로 일반적으로 얕은 버그를 찾습니다. 그러나 가볍고 빠르며 사용하기 쉽습니다. 비교해 보면, 화이트 박스 또는 그레이 박스 퍼지는 더 높은 코드 커버리지를 달성 할 수 있으며 일반적으로 블랙 박스 퍼지보다 숨겨진 버그를 발견 할 수 있습니다. 그러나 이러한 퍼지는 더 복잡하게 구성되며 퍼지 프로세스는 블랙 박스 퍼지보다 시간이 오래 걸립니다.

단순한 돌연변이 방법만을 사용하는 퍼지 기법은 일반적으로 형식을 알지 못하고 입력 파일의 일부 바이트를 무작위로 변형시키는 Charlie Miller가 사용한 유명한 "5 행의 Python"방법과 같은 "멍청한" 퍼징으로 간주됩니다. 대조적으로, 테스트 케이스 생성을 안내하기 위해 입력의 사양 또는 다른 지식을 이용하거나 런타임 정보 (예를 들어, 경로 커버리지)를 채택하는 기술은 일반적으로 "똑똑한" 퍼징으로 간주된다. 일반적으로 "벙어리 (dumb)"및 "스마트 (smart)"퍼징 방법은 서로 다른 비용 및 정밀한 절충점을 제공하며 서로 다른 상황에 적합합니다. 테스터의 경우, 어떤 종류의 fuzzer는 주로 1) 대상 프로그램의 유형과 2) 테스트 요구 사항 (시간 / 비용 등)에 따라 달라집니다.

대상 프로그램 (예 : 컴파일러, 시스템 호출, 네트워크 프로토콜 등)의 입력 형식이 특수하거나 엄격한 경우 문법 기반의 fuzzer를 선택하는 것이 더 효과적입니다 (이러한 종류의 fuzzers는 대부분 블랙 박스 방식입니다. (예 : Trinity [32]) 개발자는 최근 Syzkaller와 같은 소프트웨어를 타겟팅하는 그레이 박스 fuzzer를 만들었습니다. 다른 경우 테스터는 테스트 요구 사항에 대해 더 자세히 고려해야합니다. 테스트의 주된 목표가 정밀도 또는 고품질 출력보다는 효율성이라면 블랙 박스 퍼징 (black-box fuzzing)이 좋은 선택이 될 것입니다. 예를 들어 소프트웨어 시스템이 전에 테스트되지 않았고 테스터가 가능한 한 빨리 "얕은(shallow)" 버그를 찾아 제거하려는 경우 블랙 박스 퍼징이 좋은 시작입니다.

대상 프로그램 (예 : 컴파일러, 시스템 호출, 네트워크 프로토콜 등)의 입력 형식이 특수하거나 엄격한 경우 문법 기반의 fuzzer를 선택하는 것이 더 효과적입니다 (이러한 종류의 fuzzers는 대부분 검은 색입니다 -box (예 : Trinity [32]) 개발자는 최근 Syzkaller와 같은 소프트웨어를 타겟팅하는 회색 상자 fuzzer를 만들었습니다. 다른 경우 테스터는 테스트 요구 사항에 대해 더 자세히 고려해야합니다. 테스트의 주된 목표가 정밀도 또는 고품질 출력보다는 효율성이라면 블랙 박스 퍼징 (black-box fuzzing)이 좋은 선택이 될 것입니다. 예를 들어 소프트웨어 시스템이 전에 테스트되지 않았고 테스터가 가능한 한 빨리 "얕은"버그를 찾아 제거하려는 경우 블랙 박스 퍼징이 좋은 시작입니다.

단순한 돌연변이 방법만을 사용하는 퍼징 기법은 일반적으로 형식을 알지 못하고 입력 파일의 일부 바이트를 무작위로 변형시키는 Charlie Miller가 사용한 유명한 "5 행의 Python"방법과 같은 "멍청한" 퍼징으로 간주됩니다. 대조적으로, 테스트 케이스 생성을 안내하기 위해 입력의 사양 또는 다른 지식을 이용하거나 런타임 정보 (예를 들어, 경로 커버리지)를 채택하는 기술은 일반적으로 "똑똑한" 퍼징으로 간주됩니다. 일반적으로 "벙어리 (dumb)" 및 "스마트 (smart)" 퍼징 방법은 서로 다른 비용과 정확도에 대한 절충을 제공하며 서로 다른 상황에 적합합니다. 테스터의 경우, 선택할 fuzzer의 종류는 주로 두 가지 요소에 따라 다릅니다. :
1) 대상 프로그램의 유형 및 2) 테스트 요구 사항 (시간 / 비용 등). 대상 프로그램 (예 : 컴파일러, 시스템 호출, 네트워크 프로토콜 등)의 입력 형식이 특수하거나 엄격한 경우 문법 기반의 fuzzer를 선택하는 것이 더 효과적입니다 (이러한 종류의 fuzzers는 대부분 블랙박스입니다. (예 : Trinity [32]) 개발자는 최근 Syzkaller와 같은 소프트웨어를 타겟팅하는 그레이 박스 fuzzer를 만들었습니다. 다른 경우 테스터는 테스트 요구 사항에 대해 더 자세히 고려해야합니다. 테스트의 주된 목표가 정밀도 또는 고품질 출력보다는 효율성이라면 블랙 박스 퍼징 (black-box fuzzing)이 좋은 선택이 될 것입니다. 예를 들어 소프트웨어 시스템이 전에 테스트되지 않았고 테스터가 가능한 한 빨리 "얕은" 버그를 찾아 제거하려는 경우 블랙 박스 퍼징이 좋은 시작입니다. 반대로 테스터가 출력의 품질 (즉, 다양성, 악용 가능성 및 발견 된 버그 수)에 더 집중하고 더 높은 코드 적용 범위를 달성하려는 경우, 그레이 박스 (때때로 화이트 박스) 퍼징이 일반적으로 적합합니다 [14]. 그레이 박스 퍼징과 비교할 때, 화이트 박스 퍼징은 매우 비싸고 (시간 / 리소스 소비), 많은 도전에 직면하기 때문에 (예를 들어, SAGE는 시장에서 유명한 화이트 박스 퍼지기임에도 불구하고) , 메모리 모델링, 제약 해결 등). 그러나 white-box fuzzing은 많은 잠재력을 가진 인기 있는 연구 방향입니다.

V. STATE OF THE ART IN FUZZING

섹션 III에 설명 된 일반적인 퍼징 프로세스에 따라 fuzzer 구축시 다음 질문을 고려해야합니다.
1) 시드 및 기타 테스트 케이스를 생성하거나 선택하는 방법;
2) 목표 프로그램의 세부 사항에 대해 인풋 값들을 검증하는 방법;
3) 충돌 유발 시험 사례를 다루는 방법;
4) 런타임 정보를 활용하는 방법;
5) 퍼징의 확장성을 향상시키는 방법.

이 절에서는 문헌에서 fuzzing의 위의 다섯 가지 문제에 대한 주요 기여를 요약하여 RQ1에 대해 설명합니다. 우리는 섹션 V-A의 시드 생성 및 선택, 섹션 V-B의 입력 값 유효성 검사

및 적용 범위, 섹션 V-C의 런타임 정보 및 효율성, 섹션 V-D의 테스트 사례 처리 충돌 및 섹션 V-E의 퍼징 확장성과 관련된 문서를 검토합니다.

A. Seeds Generation and Selection

퍼징은 (fuzz) 대상 프로그램이 주어지면, 테스터는 먼저 대상 프로그램이 읽을 수 있도록 입력 인터페이스 (예 : 표준 파일)를 찾아야합니다
그런 다음 대상 프로그램에서 허용 할 수 있는 파일 형식을 결정한 다음 수집 된 시드 파일의 하위 집합을 선택하여 대상 프로그램을 바꿉니다. seed 파일의 품질은 퍼징 결과에 매우 영향을 줄 수 있습니다. 따라서 더 많은 버그를 발견하기 위해 적합한 시드 파일을 생성하거나 선택하는 방법이 중요한 문제입니다.

퍼징 대상 프로그램이 주어지면, 테스터는 먼저 대상 프로그램이 읽을 수 있도록 입력 인터페이스 (예 : 표준 파일)를 찾아야합니다. 그런 다음 대상 프로그램에서 허용 할 수 있는 파일 형식을 결정한 다음 수집 된 시드 파일의 하위 집합을 선택하여 대상 프로그램을 바꿉니다. seed 파일의 품질은 퍼징 결과에 매우 영향을 줄 수 있습니다. 따라서 더 많은 버그를 발견하기 위해 적합한 시드 파일을 생성하거나 선택하는 방법이 중요한 문제입니다 퍼징 (fuzz) 대상 프로그램이 주어지면, 테스터는 먼저 대상 프로그램이 읽을 수 있도록 입력 인터페이스 (예 : 표준 파일)를 찾아야합니다.

이 문제를 해결하기 위해 일부 연구가 수행되었습니다. Rebert et al. [47] 6 가지 종류의 선택 알고리즘을 테스트했다 :
1) Peach에서 커버 알고리즘을 설정;
2) 랜덤 시드 선택 알고리즘;
3) 최소한의 세트 커버 (욕심쟁이 알고리즘에 의해 계산 될 수있는 분과 동일 함);
4) 크기에 의해 가중 된 최소 세트 커버;
5) 실행 시간으로 가중 된 최소 세트 커버;
6) 핫셋 알고리즘 (t 초 동안 각 시드 파일을 퍼즈 (fuzz)하고, 발견 된 고유 버그의 수에 따라 순위를 정하고, 목록에 상위 몇 개의 시드 파일을 반환하는 알고리즘)

Amazon Elastic Compute Cloud에서 10 개의 응용 프로그램을 처리하기 위해 650 CPU 일을 소비함으로써 다음 결론을 이끌어 냈습니다.
1) 휴리스틱을 사용하는 알고리즘은 완전히 무작위 추출보다 잘 수행됩니다.
2) unweighted minset 알고리즘은이 6 가지 알고리즘 중에서 가장 잘 수행됩니다.
3) 감소 된 일련의 시드 파일은 실제로 원본 세트보다 효율적으로 수행됩니다.
4) 동일한 파일 형식을 구문 분석하는 여러 응용 프로그램에 축소 된 시드 집합을 적용 할 수 있습니다.

Karg'en과 Shahmehri [48]는 생성 된 프로그램의 기계 코드에서 올바른 형식의 입력 대신 직접 변이를 수행함으로써 테스트 결과가 테스트중인 프로그램에서 기대하는 형식에 더 가깝고 더 나은 코드를 생성한다고 주장했다 적용 범위. Liang 등은 일반적으로 여러 객체 (예 : 글꼴, 그림)가 포함 된 다양한 입력을받는 복잡한 소프트웨어 (예 : PDF 리더)를 테스트합니다. [49] 레버리지

유형이 다른 글꼴들 중에서 시드 파일을 선택하기위한 글꼴 파일의 구조 정보. Skyfire [50]는 코퍼스와 문법을 입력으로 사용하여 방대한 입력을 처리하는 퍼지 프로그램에 대한 잘 분산 된 시드 입력을 생성하기 위해 방대한 수의 기존 샘플에서 지식을 사용했습니다.
프로그램과 시드 입력이 주어지면 black-box mutational fuzzing에서 발견 된 버그의 수를 최대화하기 위해 [51]에 알고리즘이 제시되었습니다. 이 배후의 주요 아이디어는 주어진 프로그램 시드 쌍에 대한 실행 추적에 대한 화이트 박스 기호 분석을 사용하여 입력의 비트 위치 간의 종속성을 감지 한 다음이 종속 관계를 사용하여이 프로그램에 대해 확률 적으로 최적의 돌연변이 비율을 계산하는 것입니다. 씨앗 쌍. 또한, 확률 분석에서 쿠폰 수집기의 문제의 예로 퍼징을 고려하면, Arcuri et al. [52]는 미리 정의 된 목표를 다루기 위해 무작위 테스트를 통해 샘플링 된 예상 테스트 케이스 수에 대해 중요하지 않은 최적의 하한을 제안하고 입증했지만 실제 목표 범위 달성 방법은 제시되지 않았습니다. 실제로 목표는 주어지지 않는다.

객체 지향 프로그램의 단위 테스트를 무작위로 생성하기 위해 Pacheco et al. [53]은 시퀀스가 실행되는 동안 얻은 피드백을 순서대로 사용하도록 제안했다.
새롭고 합법적 인 객체 상태를 생성하는 시퀀스로 검색을 안내합니다. 따라서 중복되거나 불법적 인 상태를 만드는 입력은 결코 확장되지 않습니다. 그러나 Yatoh et al. [54]는 피드백 지침이 세대를 지나치게 지시하고
생성 된 테스트의 다양성, 피드백을 적응 적으로 제어하는 피드백 제어 무작위 테스트라는 알고리즘을 제안했습니다.

어떻게하면 원래 시드 파일을 얻을 수 있습니까? 일부 오픈 소스 프로젝트의 경우, 테스트를 위해 방대한 입력 데이터를 사용하여 애플리케이션을 게시하므로 자유롭게 얻을 수 있습니다
퍼징을위한 품질의 씨앗으로. 예를 들어, FFmpeg 자동화 테스트 환경 (FATE) [55]은 테스터의 힘으로 수집하기 어려운 다양한 테스트 케이스를 제공합니다. 때로는 테스트 데이터를 공개적으로 사용할 수 없지만 개발자는 프로그램 버그를보고 할 사람과 테스트 데이터를 교환하려고합니다. 일부 다른 오픈 소스 프로젝트도 형식 변환기를 제공합니다. 따라서 특정 형식의 다양한 파일 세트를 사용하여 테스터는 형식 변환기를 사용하여 퍼징을위한 알맞은 종자를 얻을 수 있습니다. 예를 들어 cwebp [56]는 TIFF / JPEG / PNG를 WEBP 이미지로 변환 할 수 있습니다. 또한 리버스 엔지니어링
퍼지에 대한 시드 입력을 제공하는 데 도움이됩니다. Prospex [57]는 프로토콜 상태 머신을 포함하여 네트워크 프로토콜 스펙을 추출 할 수 있으며,이를 사용하여 자동으로 스테이트 풀 (stateful) fuzzer에 대한 입력을 생성합니다.Adaptive random test (ART) [58]는 테스트 공간을 샘플링하여 무작위 테스트를 수정하고 이전에 실행 된 모든 테스트에서 입력에 대한 거리 메트릭에 의해 결정되는 가장 먼 거리 ART는 복잡한 실제 프로그램 [59]에 대해 항상 효과적이지는 않으며, 대부분 숫자 입력 프로그램에 적용되었습니다.

위에서 언급 한 접근 방식과 비교할 때,
인터넷을 크롤링하여 파일이 더 일반적입니다. 테스터는 특정 문자 (예 : 파일 확장자, 매직 바이트 등)를 기반으로 필요한 시드 파일을 다운로드 할 수 있습니다. 수집 물이 거대하면 저장 비용이 저렴하고 등가 코드 커버리지에 도달하는 동안 코퍼스를 더 작은 크기로 압축 할 수 있으므로 심각한 문제는 아닙니다. 잘못 삽입 된 파일의 수를 줄이고 최대 테스트 케이스 커버리지를 유지하기 위해 Kim et al. [61] 추적 및 분석을 통한 이진 파일의 필드 분석 제안 스택 프레임, 어셈블리 코드 및 레지스터를 대상 소프트웨어로 파싱하여 파일을 구문 분석합

니다.

B. Input Validation and Coverage

타겟 프로그램의 예기치 않은 동작을 트리거하기 위해 자동으로 수많은 테스트 케이스를 생성하는 기능은 퍼징의 중요한 장점입니다. 그러나 대상 프로그램 입력 검증 메커니즘을 가지고 있기 때문에 이러한 테스트 케이스는 실행 초기 단계에서 거부 될 가능성이 큽니다. 따라서 이러한 장애물을 극복하는 방법은 테스터가 이러한 메커니즘을 사용하여 프로그램을 퍼지기 시작할 때 필요한 고려 사항입니다.

1) 무결성 검증 : 전송 및 저장 중에 원본 데이터에 오류가 발생할 수 있습니다. 이러한 "왜곡 된"데이터를 검출하기 위해, 체크섬 메커니즘은 일부 파일 포맷 (예 : PNG) 및 네트워크 프로토콜 (예 : TCP / IP)를 사용하여 입력 데이터의 무결성을 확인하십시오. 체크섬 알고리즘 (예를 들어, 해시 함수)을 사용하여, 원래의 데이터는 고유 한 체크섬 값으로 첨부된다. 데이터 수신기의 경우, 수신 된 데이터의 무결성은 재 계산
같은 알고리즘으로 체크섬 값을 계산하고이를 첨부 된 값과 비교합니다. 이러한 종류의 시스템을 모호하게하려면 새로 추가 된 로직을 fuzzer에 추가하여 새로 생성 된 테스트 케이스의 올바른 체크섬 값을 계산해야합니다. 그렇지 않은 경우, 개발자는 이 장애물을 제거하기 위해 다른 방법을 사용해야합니다. [36], [62]는이 문제를 해결할 수있는 새로운 방법을 제시하고 TaintScope라는 이름의 fuzzer를 개발했다. TaintScope는 동적 테인 분석과 사전 정의 된 규칙을 사용하여 중요한 체크섬 지점과 중요한 입력 바이트를 감지하여 중요한 애플리케이션 프로그래밍 인터페이스를 오염시킬 수 있습니다. (API)를 대상 프로그램에 포함시킵니다. 그런 다음 핫 바이트를 새로운 테스트 케이스를 생성하고 체크섬 포인트를 변경하여 생성 된 모든 테스트 케이스가 무결성 검증을 통과하도록합니다. 마지막으로 심볼 실행 및 제약 조건을 사용하여 대상 프로그램을 충돌시킬 수있는 테스트 케이스의 체크섬 값을 수정합니다
해결. 이를 통해 무결성 검증을 통과 할 수 있고 프로그램이 충돌 할 수있는 테스트 케이스를 생성 할 수 있습니다.

샘플 입력 집합이 주어지면, H¨oschele과 Zeller [63]는 각 입력 문자의 데이터 흐름을 추적하기 위해 동적 인 손상을 사용하고 입력 조각을 어휘 및 구문 실체로 집계합니다. 결과는 유효한 입력 구조를 반영하는 컨텍스트 - 프리 문법이며, 이는 나중에 퍼징 프로세스에 도움이됩니다. Steelix [64]는 커버리지 기반의 fuzzers의 한계를 줄이기 위해 경량 정적 분석과 바이너리 계측을 활용하여 커버리지 정보뿐만 아니라 비교 진행 정보도 fuzzer에 제공했다. 이러한 프로그램 상태 정보는 fuzzer에게 테스트 바이트에서 매직 바이트가 어디에 위치 하는지를 알려주고 매직 바이트를 효율적으로 매치시키기 위해 변이를 수행하는 방법을 알려줍니다. 이 문제를 완화 시키는데 진전을 이룩한 몇 가지 다른 노력이 있다 [65], [66].

2) 형식 검증 : 네트워크 프로토콜, 컴파일러, 인터프리터,
입력 형식에 대한 엄격한 요구 사항이 있습니다. 형식 요구 사항을 충족하지 않는 입력은 프로그램 실행 시작시 거부됩니다. 따라서 이러한 종류의 타겟 시스템을 퍼징하는 것은 형식 유효성 검사를 통과 할 수있는 테스트 케이스를 생성하는 추가 기술을 필요로합니다. 이 문제의 대부분의 해결책은 입력 특정 지식이나 문법을 활용하는 것입니다. Ruiter and Poll [30]은 상태 머신 학습과 함께 블랙 박스 퍼징 (blackbox fuzzing)을 사용하여 일반적으로 사용되는 전송 계층 보안 (TLS) 프로토콜 구현 9 가지를 평가했습니다. 그들은 테스트 하니스에 의해

테스트중인 시스템에 전송 된 구체적인 메시지로 변환 될 수있는 추상 메시지 목록 (입력 알 파벳으로도 알려짐)을 제공했습니다. Dewey et al. [67], [68]은 CLP (Constraint Logic Programing)를 사용하여 복잡한 유형의 시스템을 사용하는 잘 유형화 된 프로그램을 생성하 고 녹 또는 자바 스크립트 프로그램을 생성하는 방법을 적용한 새로운 방법을 제안했다. Cao et al. [69] 처음에는 안드로이드 시스템 서비스의 입력 검증 상황을 조사했고 안드로이드 장 치의 경우 입력 검증 취약점 스캐너를 구축했습니다. 이 스캐너는 대상 시스템 서비스 메소드 에 의해 구현 된 예비 검사를 통과 할 수있는 semivalid 인수를 생성 할 수 있습니다. 이 문 제를 해결하는 데 중점을 둔 [24], [25], [70], [71]과 같은 다른 작품이 있습니다.

3) 환경 검증 : 특정 환경 (특정 구성, 특정 런타임 상태 / 조건 등) 하에서 만 많은 소프트 웨어 취약점이 드러날 것입니다. 일반적인 퍼지는 입력의 구문 론적 의미 적 유효성 또는 탐 색 된 입력 공간의 비율을 보장 할 수 없습니다. 이러한 문제를 완화하기 위해 Dai et al. [72]는 특정 조건에서만 발생하는 취약점을 검사하기 위해 실행중인 응용 프로그램의 구성이 특정 실행 지점에서 변경되는 구성 퍼징 기술을 제안했습니다. FuzzDroid [73]는 또한 앱이 악의적 인 행동을 노출하는 Android 실행 환경을 자동으로 생성 할 수 있습니다. FuzzDroid 는 검색 기반 알고리즘을 통해 정적 분석과 동적 분석의 확장 가능한 세트를 결합합니다.
앱을 구성 가능한 대상 위치로 안내합니다.

4) 입력 범위 : Tsankov et al. [74] 정의 된 반자동 입력 범위 (SVCov)는 퍼지 테스트의 첫 번째 적용 범위 기준입니다. 유효한 입력이 유한 제한 집합에 의해 정의 될 수있을 때마다 기준이 적용됩니다. 적용 범위를 늘림으로써
SVCov에서 이전에 알려지지 않은 취약점을 발견했습니다.
성숙한 인터넷 키 교환 (IKE) 구현 심각하게 느리고 지나치게 일반화 된 기존의 문법 추론 알 고리즘의 단점을 보완하기 위해 Bastani et al. [75]는 일련의 입력 예제와 프로그램에 대한 블랙 박스 액세스로부터 유효한 프로그램 입력의 언어를 인코딩하는 문맥 자유 문법을 합성하 기위한 알고리즘을 제시했다. ArtFuzz [76]는 입력이 효과적인지를 결정하기 위해 목표 프로 그램의 충돌을 일으키는 많은 방법과 달리 noncrash 버퍼 오버 플로우 취약점을 포착하는 것을 목표로합니다. 유형 정보를 활용하고 동적으로 발견합니다.
가능성이있는 메모리 레이아웃이 퍼징 프로세스를 돕습니다. 메모리 레이아웃에서 식별 된 버 퍼 경계를 초과하면 오류가 보고 됩니다.