
Test Result Document

Project Name	Fuzz Testing을 통한 위성 SW 분석
-----------------	---------------------------

05 조

202002473 김승혁

201902733 이정윤

202002699 조민기


지도교수: 이성호 교수님  (서명)

Table of Contents

1.	INTRODUCTION.....	3
1.1.	OBJECTIVE.....	3
2.	EXPERIMENT RESULT REPORT.....	4
3.	AI 도구 활용 정보.....	8

1. Introduction

1.1. Objective

본 보고서는 NASA의 오픈소스 비행 소프트웨어 프레임워크인 F' (F Prime)의 CmdDispatcher 모듈에 대해 수행한 퍼징(Fuzzing) 테스트 결과를 정리한 것이다. CmdDispatcher는 위성 소프트웨어에서 수신된 명령을 해당 기능 컴포넌트로 전달하고, 그 실행 결과를 처리하는 핵심 모듈이다. 본 테스트의 목적은 이 모듈의 안정성과 예외 입력 처리 능력을 검증하고, F' 환경에서 효과적인 퍼징 기법을 탐색하는 데 있다. 이를 위해 세 가지 퍼저(libFuzzer, AFL++, libprotobuf-mutator 기반 퍼저)를 활용하였으며, 각 퍼저의 성능 차이와 테스트를 통해 확인된 주요 결과를 분석하였다.

2. Experiment Result Report

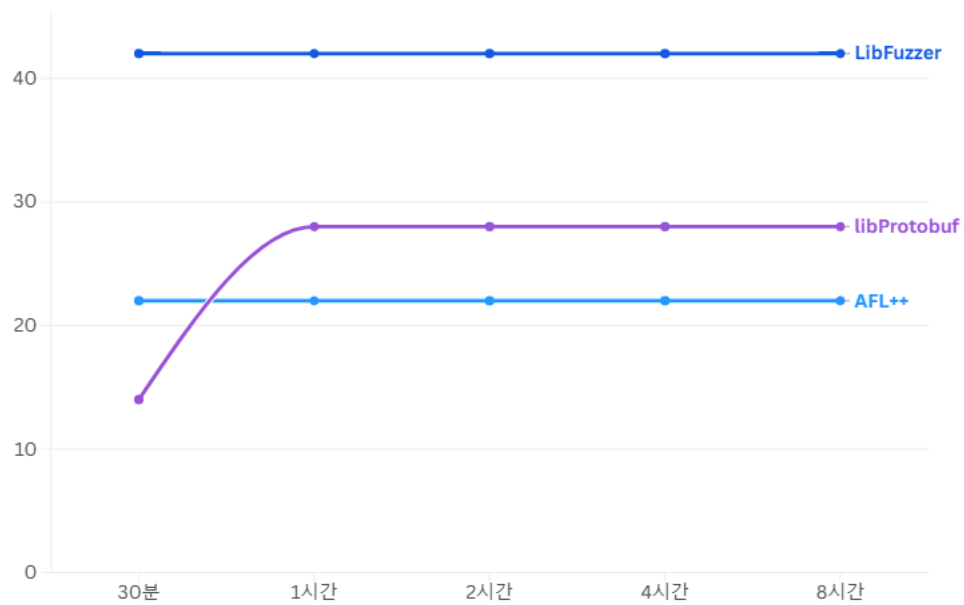
1. 서론
1.1 실험 개요
<p>Cmd Dispatcher 모듈을 대상으로 여러 종류의 퍼저를 적용해, 각 퍼저의 성능 및 결과를 비교하고, 원인을 분석해 Fprime에 최적화된 퍼저의 구조 및 기능을 제시한다. Fprime의 내부에서 사용하는 고유 통신 구조가 있으므로, Generation-Based(Structure-aware) 퍼저가 다른 유형의 퍼저보다 더 높은 성능을 보일 것이라는 가설을 가지고 실험을 진행하였다.</p> <p>입력 데이터는 실존 명령 패킷을 구현한 바이트 파일 11 종을 시드로 사용하였다. 실험 환경은 Ubuntu 22.04 환경에서 fprime 3.6.3 버전을 대상으로 진행하였으며, libFuzzer/LLVM, AFL++, libprotobuf-mutator 등의 퍼저 도구는 apt에 제시된 안정화 버전을 사용하였다. 각 퍼저 환경은 fprime 베이스 이미지에 대한 별도 docker 컨테이너로 관리해, 독립성을 마련했다.</p>
1.2 실험 방법
<p>CmdDispatcher 모듈과 연결되어 모듈의 함수를 호출하는 하네스 코드를 작성한다. 그 다음에, 각 퍼저별로 stdin 또는 자체 입력을 하네스로 전달하는 어댑터를 작성한 뒤, 각 퍼저를 위해 개발된 컴파일러를 사용해 빌드하여, 커버리지와 Basic Box, 타임아웃 및 크래시 등을 실시간으로 계측한다.</p> <p>퍼저 실행에 앞서, CmdDispatcher의 명세를 기반으로 유효한 명령어를 바이트 데이터로 10개 준비 한 뒤, 퍼저에 삽입하여 탐색 속도를 높인다. 이러한 데이터를 시드 코퍼스라 부르며, 각 퍼저는 코퍼스를 기반으로 입력을 생성하기 시작한다.</p> <p>테스트에 사용된 퍼저는, LLVM의 libFuzzer, 커버리지 기반 퍼저의 대표적인 AFL++, Google의 libProtobuf-mutator 를 사용하였다. libFuzzer는 입력 바이트 배열을 시드 데이터를 기반으로 무작위로 변형 및 생성하여 입력을 수행하며, 실행속도가 매우 빠르다. AFL++는 외부에서 stdin을 통해 입력을 수행하며, 코퍼스를 기반으로 비트 플립, 바이트 값 치환, 산술 연산 등의 다양한 변이 전략을 사용한다. libProtobuf-mutator는 사전에 정의된 프로토콜 버퍼(proto) 구조에 따라 입력을 생성 시키는 도구로, 미리 약속된 포맷의 데이터를 만들어 내도록 한다. 본 테스트에서는 CmdDispatcher의 명령어 구조를 proto로 정의하고, 하네스에서 proto 구조체에서 값을 뽑아내 바이트 데이터를 완성하도록 진행하였다.</p> <p>세 퍼저는 Docker 컨테이너를 통해 환경을 분리하였다. F Prime 필수 요소가 설치된 베이스 이미지를 만들고, 그 이미지를 기반으로 각 퍼저가 빌드 및 실행이 되도록 환경을 설정하였다. 실험 진행 중에는 docker logs 명령어를 통해 각 퍼저의 출력 로그를 확인해 기록하였다.</p>

평가 지표는 측정된 코드 커버리지(분기, edge 개수)와 FT (Function trace), 계측된 크래시 회수 및 타임아웃 회수로 결정하였으며, 각 퍼저의 로그 값을 확인해 비교 분석하였다.

2. 테스트 결과 상세

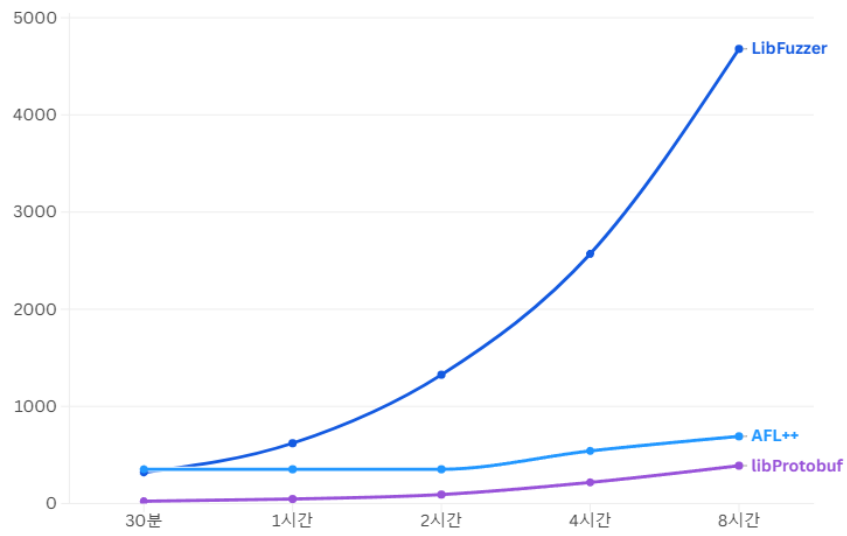
2.1 테스트 결과 개요

edge count(coverage)



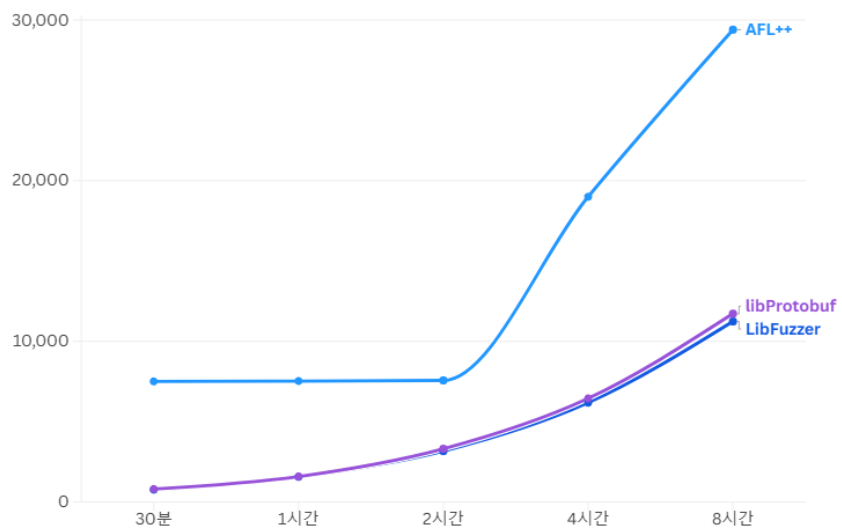
<i>edge count (coverage)</i>	30분	1시간	2시간	4시간	8시간
<i>LibFuzzer</i>	42	42	42	42	42
<i>AFL++</i>	22	22	22	22	22
<i>libProtobuf</i>	14	28	28	28	28

crashes



<i>crashes</i>	30분	1시간	2시간	4시간	8시간
<i>LibFuzzer</i>	321	621	1326	2570	4679
<i>AFL++</i>	353	353	353	542	692
<i>libProtobuf</i>	24	47	93	218	390

timeout



<i>timeout</i>	30분	1시간	2시간	4시간	8시간
<i>LibFuzzer</i>	764	1562	3166	6176	11236
<i>AFL++</i>	7503	7526	7565	19000	29400
<i>libProtobuf</i>	803	1583	3308	6446	11721

2.2 테스트 결과 상세 분석

예상과 달리 libFuzzer가 가장 높은 커버리지(Edge Count 42)를 기록했고, AFL++와 libprotobuf-mutator 조합은 그보다 낮은 값에서 서로 비슷하게 머물렀다. 세 퍼저 모두 1시간 이후에는 커버리지가 거의 늘지 않았는데, 이는 CmdDispatcher의 3단계 입력 검증 절차를 넘어서기 어렵기 때문이다. CmdDispatcher 모듈은 처음 입력을 받으면 ① 패킷 타입과 길이 필드를 검사해 형식이 맞지 않으면 즉시 거부하고, ② 등록-테이블에 존재하지 않는 opcode(operation code) 역시 곧바로 “잘못된 명령” 처리로 빠지며, ③ 남은 인자(byte 버퍼)가 명령 정의와 일치하지 않으면 대상 컴포넌트가 다시 실패를 반환하는, 3단계 검증 시스템을 가지고 있다.

시드 코퍼스에는 이미 유효 opcode가 포함돼 있었다. libFuzzer는 바이트 단위로 조금씩 변형하는 과정에서도 길이와 opcode 범위를 크게 벗어나지 않는 경우가 많아 ①·② 단계를 수월하게 통과해 실제 분기로 진입했다. 반면 AFL++는 초기 단계에서 비트·바이트를 크게 뒤섞는 변이를 우선 적용하면서 opcode 값이 자주 깨졌고, 그 결과 대부분이 ② 단계에서 걸려져 추가 분기를 열지 못했다. libprotobuf-mutator는 “opcode + raw args” 수준의 단순 proto 정의만 사용해 인자 길이까지 정확히 맞추지 못했기 때문에 ③ 단계에서 반복적으로 실패해 커버리지가 낮게 정체됐다.

로그를 보면 타입아웃은 형식이 잘못된 입력을 CmdDispatcher가 무시한 뒤 호출자가 응답을 기다리다 발생한 경우가 대부분이며, 크래시는 이미 등록된 opcode를 중복 등록하려다 의도적으로 abort()를 호출해 생긴 것이 주된 원인이었다. 결국 CmdDispatcher가 opcode별로 다르게 동작하는 준-상태(State-like) 구조를 갖고 있음에도, 세 퍼저 모두 현재 상태에 맞춰 입력을 조정하는 기능이 없어 초반 이후 커버리지가 더 이상 증가하지 않았다.

2.3 실험 결과의 한계와 위협 요인

이번 실험 결과를 그대로 다른 모듈에 적용하기에는 몇 가지 제약이 존재한다. 우선 범용성의 한계가 있다. F Prime의 대부분 모듈이 CmdDispatcher와 유사하게 opcode 등록 테이블을 두고 포트를 통해 통신하지만, 각 모듈마다 명령 형식과 인자 구성이 조금씩 다르다. 따라서 모듈이 바뀌면 새로운 시드 코퍼스를 수집하고, 모듈 전용 하네스 코드를 작성해 연결·빌드해야 하며, 이 과정이 적지 않은 개발 시간을 요구한다.

또 하나의 제약은 설계상 가정이다. 이번 퍼저는 “하나의 입력을 처리할 때마다 CmdDispatcher의 내부 상태가 항상 초기화된단” 전제하에 진행되었다. 그러나 실제 CmdDispatcher는 opcode 등록 순서, 미처리 명령의 유무, 중복 등록 시도 등과 같은 순차적 이벤트에 따라 내부 테이블 내용이 달라진다. 퍼저가 이러한 상태 변화를 인식하거나 조작하지 못했기 때문에, 테이블이 거의 가득 찬 상황이나 특정 오류 플래그가 이미 서 있는 상황처럼 조건부로만 드러나는 코드 경로는 탐색되지 않았다. 결국 입력 변형에만 집중한 기존 퍼저 방식으로는 CmdDispatcher의 상태-의존 로직을 충분히 검증하지 못한다는 한계를 확인할 수 있었다.

3. 결론

이번 실험으로 확인한 핵심 사실은 입력 형식만 알고 돌연변이를 수행하는 퍼저로는 CmdDispatcher의 모든 코드 경로를 충분히 탐색하기 어렵다는 점이다. 명령 등록 테이블과 미완료 명령 목록이 만들어 내는 상태 의존 로직 때문에, 초기에는 빠르게 커버리지를 넓히더라도 일정 시점 이후 성장이 멈춘다. 특히 libFuzzer가 가장 높은 커버리지를 기록한 이유는 실행 속도가 빠르고, 시드에 포함된 유효 opcode 덕분에 CmdDispatcher의 1, 2 단계 검증(패킷 형식, opcode 유효성)을 쉽게 통과했기 때문이다.

이번 결과는 F Prime의 내부 통신 구조를 이해할 뿐 아니라, 실행 중인 모듈의 상태를 읽고 거기에 맞게 입력을 조정할 수 있는 퍼저가 필요함을 보여준다. 즉, 단순히 올바른 형식의 명령을 생성하는 것에서 한 걸음 더 나아가, 현재 등록된 opcode 테이블·미완료 명령 수 등 CmdDispatcher의 실시간 상태를 파악하고 그 정보를 바탕으로 다음 입력을 변형·생성하는 방식이어야 커버리지 한계를 돌파할 수 있다. 이는 웹 프로토콜 분야에서 쓰이는 ‘상태 인식 퍼징’ 전략과 유사한 발상으로, F Prime 환경에 특화된 State-aware 구조적 퍼저 개발이 후속 연구 과제로 떠올랐다.

3.AI 도구 활용 정보

사용 도구	GPT-o3
사용 목적	문장 흐름 정리, 사례 리서치 보조
프롬프트	<ul style="list-style-type: none"> 문장의 흐름을 다듬어줘.
반영 위치	1. 테스트 결과 상세 분석 (p.6)
수작업	있음(논리 보강, 그래프 시각화, F Prime 구조 등)
수정	