위성 SW를 위한 효율적인 Fuzzer 개발

202002473 김승혁
201902733 이정윤
202002699 조민기

배경

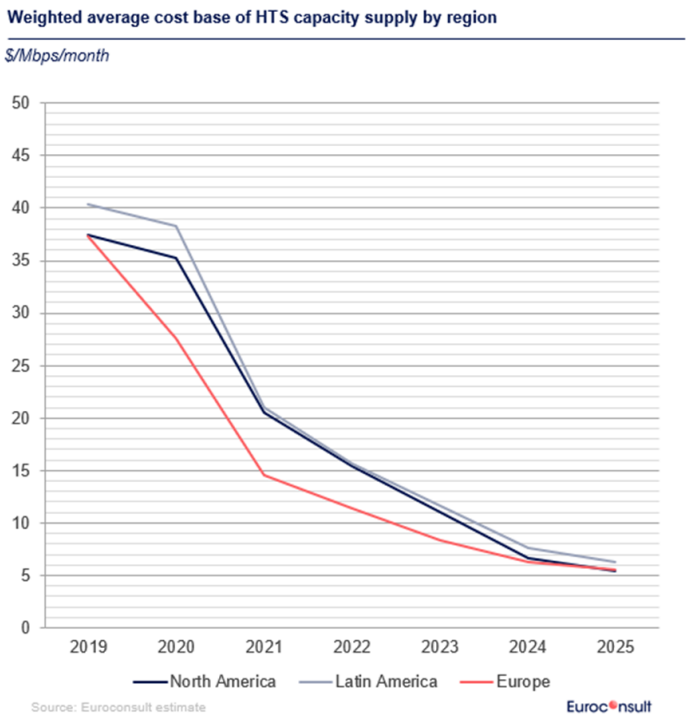


Figure 1. 무게당 발사 가격 추이

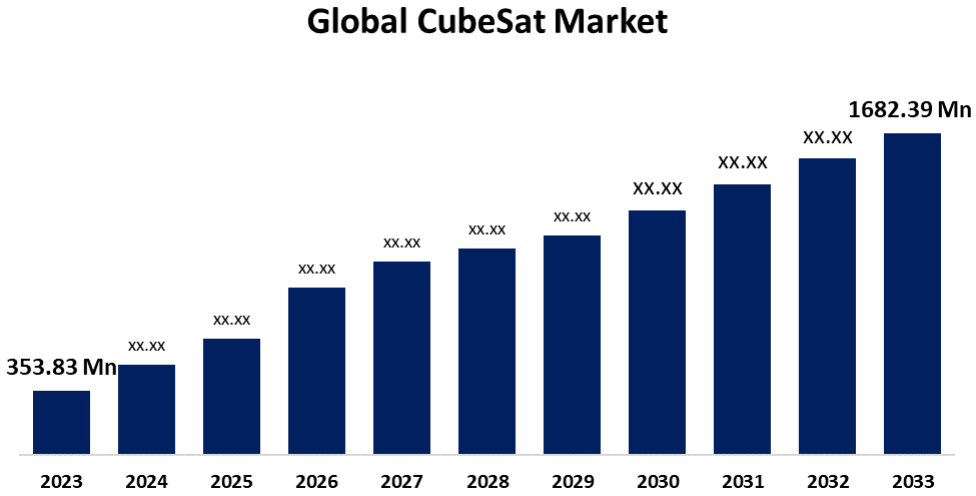


Figure 2. 소형 위성 시장 전망

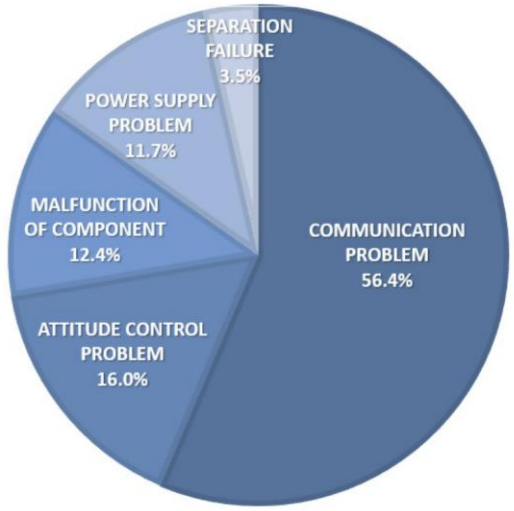


Figure 3. 실패 원인 분석

필요성

- 소형 위성 소프트웨어 검증 방법론의 필요성 대두

  (특히 fprime)

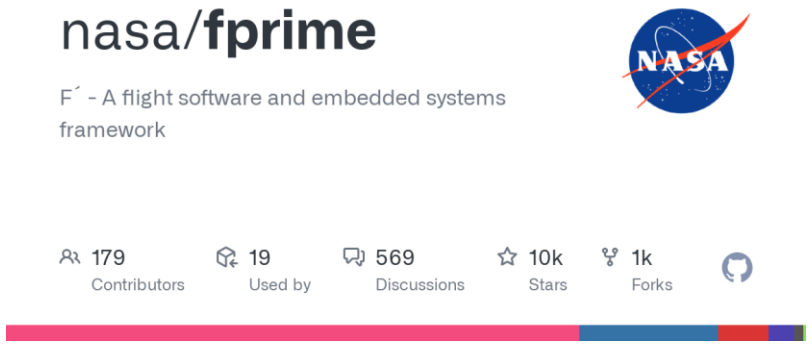


Figure 4. NASA의 F Prime

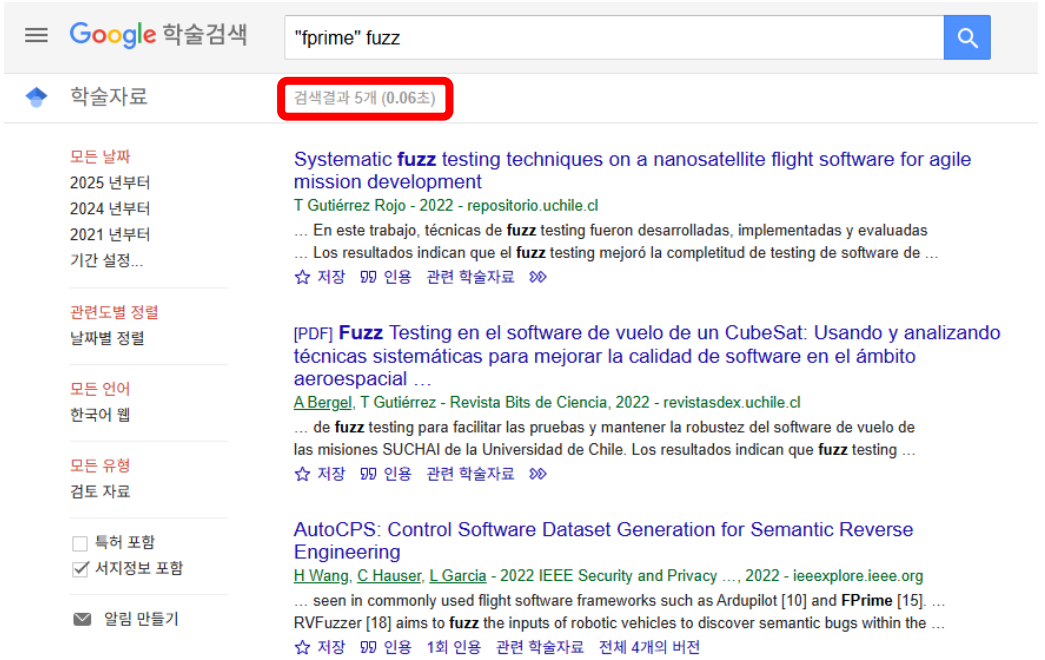


Figure 5. F Prime Fuzzing 연구

목표

- 따라서 Fuzzer를 통한 오류 검증 방식이 소형 위성에 적합한지 판단하고, 위성 검증에 적합한 Fuzzer 개발.
- Fprime의 내부 구조와 특징을 반영한 Fuzzer 개발.

-〉 발사 전 지상 테스트 단계에서 결함을 미리 잡아내고, 실제 위성 운영 중 발생할 수 있는 문제를 사전에 차단하는 것이 목표

연구 계획 타임라인

Fuzzer 개발

TIMELINE

3월	4월	5월	6~7월	8~9월	10~11월
프로젝트 시작	대상 위성 소프트웨어 분석	Fuzzer 적용 연구	Fuzzer 개발 설계	Fuzzer 개발	Fuzzer 적용 테스트 / 프로젝트 종료

팀원 소개 / 역할 분담 / 협업 방식

김승혁

이정윤

조민기

이해 관계자 인터뷰

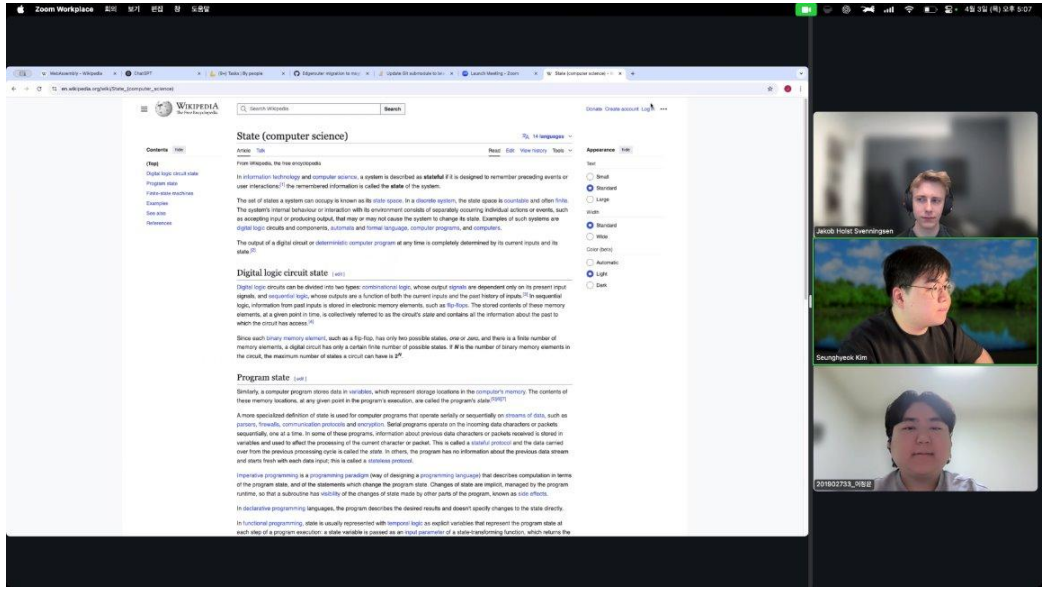


Figure 6. Jakob과의 인터뷰

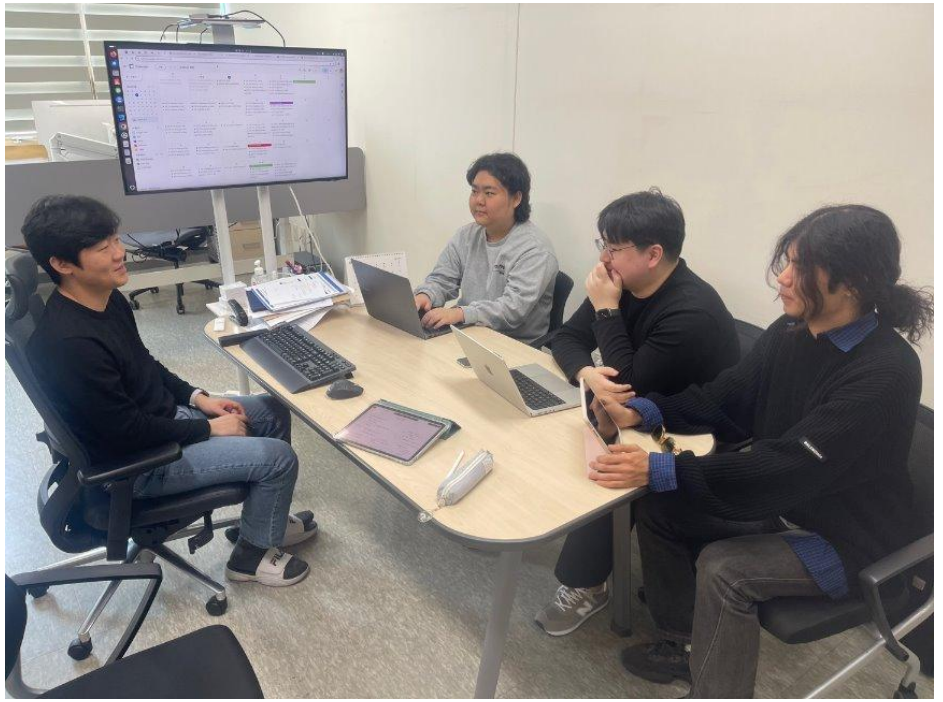


Figure 7. 이성호 교수님과의 인터뷰

사용자 분석

이해 관계자 인터뷰

	이성호 교수님	김형신 교수님	Jakob
인터뷰 내용	위성 SW 맞춤형 테스팅 기법 부족. Fuzzer 성능 비교 후 결함 탐색 시도할 것. 기존 Fuzzer는 위성 SW의 특성을 고려하지 못해 커버리지 확보 및 모듈 간 상호 작용 결함 탐지에 한계가 있을 수 있음.	위성 SW의 오류는 기능 상실 또는 임무 실패로 이어짐. 위성 시스템은 비정상적인 상태에서 탐지 및 복구가 어려움. 따라서 발사 전 지상에서의 철저한 테스트가 매우 중요함. 대학의 소형 위성 같은 경우는 비용, 시간 문제 때문에 테스트가 비교적 적은 경향이 있음.	상태 기반 시스템 대상 Fuzzer 개발 경험 있음. fprime은 상태 기반 시스템이므로 Generation 기반 Fuzzer가 더 적합할 수 있음. Fuzzing 연구는 대상 프로토콜에 대한 깊은 이해가 중요함. (fprime의 fpp 등)
인사이트	대상 SW들 실제 구동해보고 단계적으로 Fuzzer들 적용해보는 것이 가장 중요함. 정량적 평가 지표 설정을 통해 분석해야함.	위성 SW는 특수성을 가지고 있으며, 테스트에서도 이러한 특성을 반영한 효과적인 테스트가 필요하다.	대상 SW의 특성을 잘 분석한 뒤, 이에 맞는 Fuzzing 전략을 찾아 적용하는 것이 중요하다.

핵심 아이디어

제안 방법

: 여러 유형의 Fuzzer를 소형 위성 SW에 적용하여 분석. 이를 바탕으로 Fuzz Testing의 적합성 검증. 위성 SW에 적합한 Fuzzer 유형을 분석하고 제작

기존 방법 개선점

: 기존 테스트 방법에 더불어 새로운 동적 분석 방법을 제시하고 테스트를 자동화하여 검증 효율성을 높이고자 함. 또한 테스트를 자동화하여 검증 효율성을 높이고 신뢰도를 높이고자 함.

Fuzzer 유형 별 구분

입력 형식 기반 Generation-Based Fuzzer

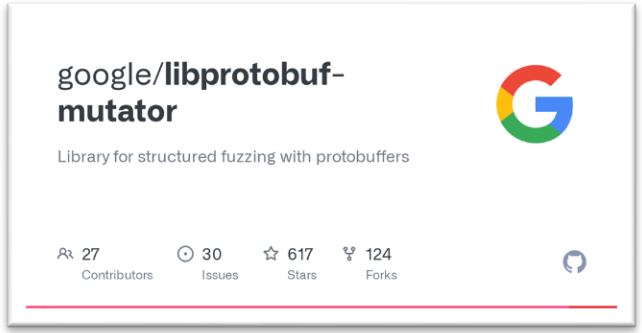


Figure 8. Google의 libProtobuf-mutator

입력 변이 기반 Mutation-Based Fuzzer

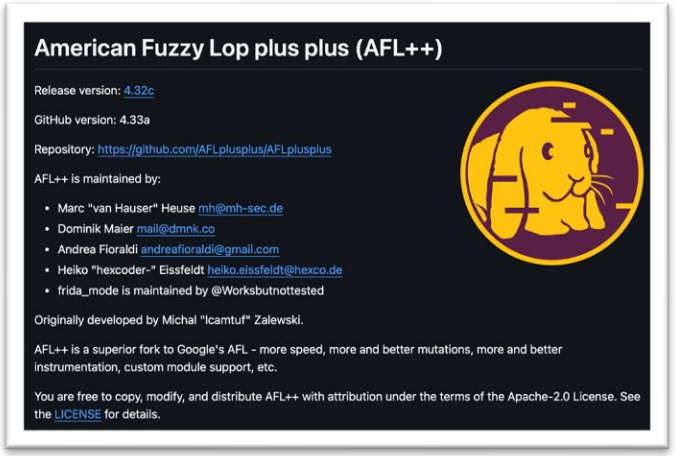


Figure 9. AFL++

실행 경로 기반 Coverage-guided Fuzzer

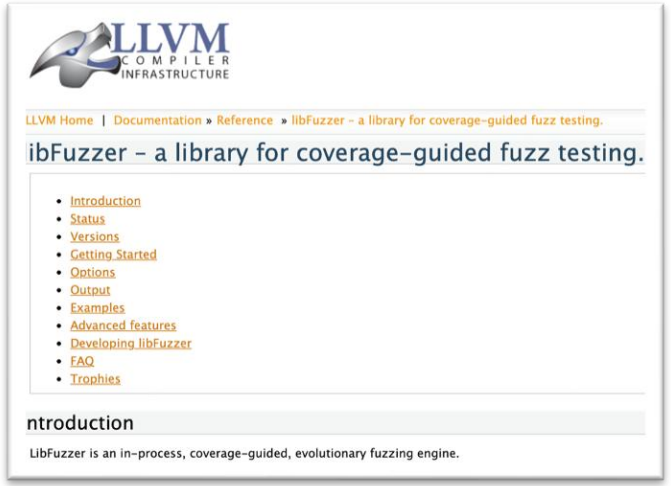


Figure 9. LLVM의 libFuzzer

진행 상황

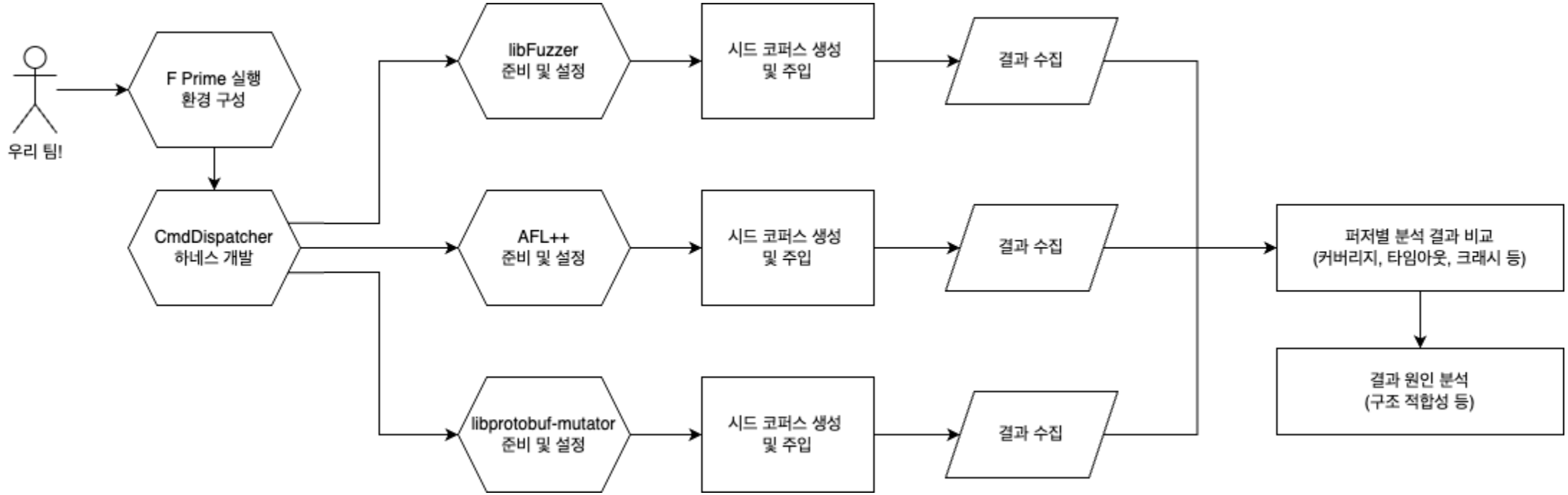
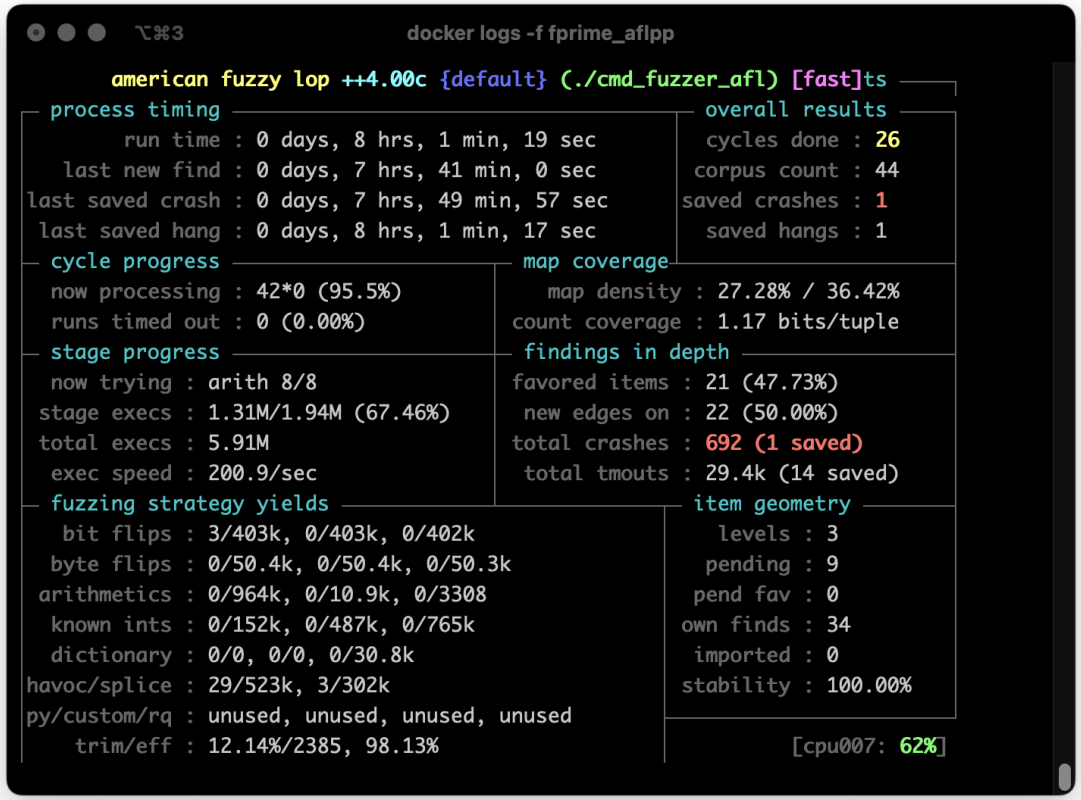
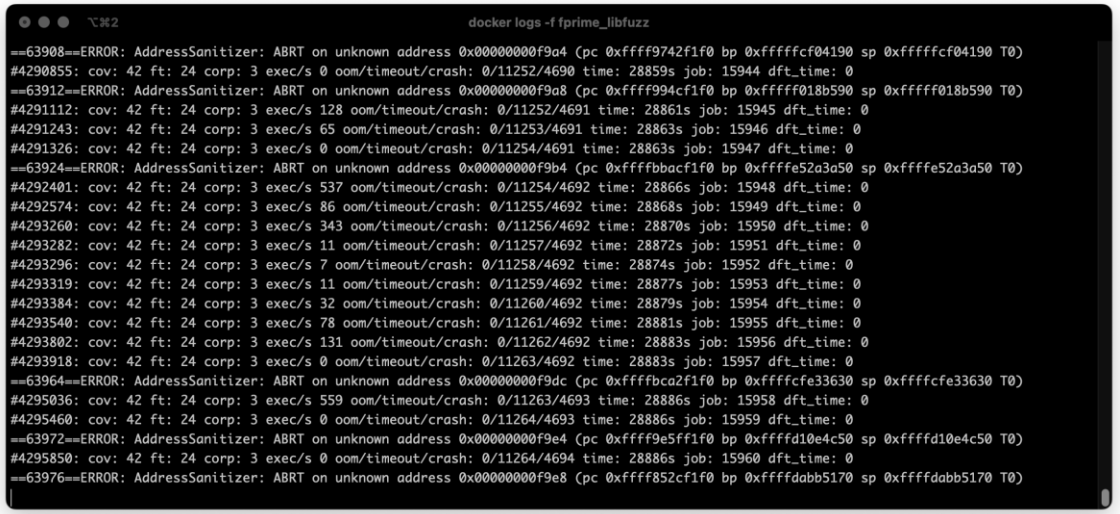


Figure 10. Fuzzer 적용 연구 순서도

진행 상황





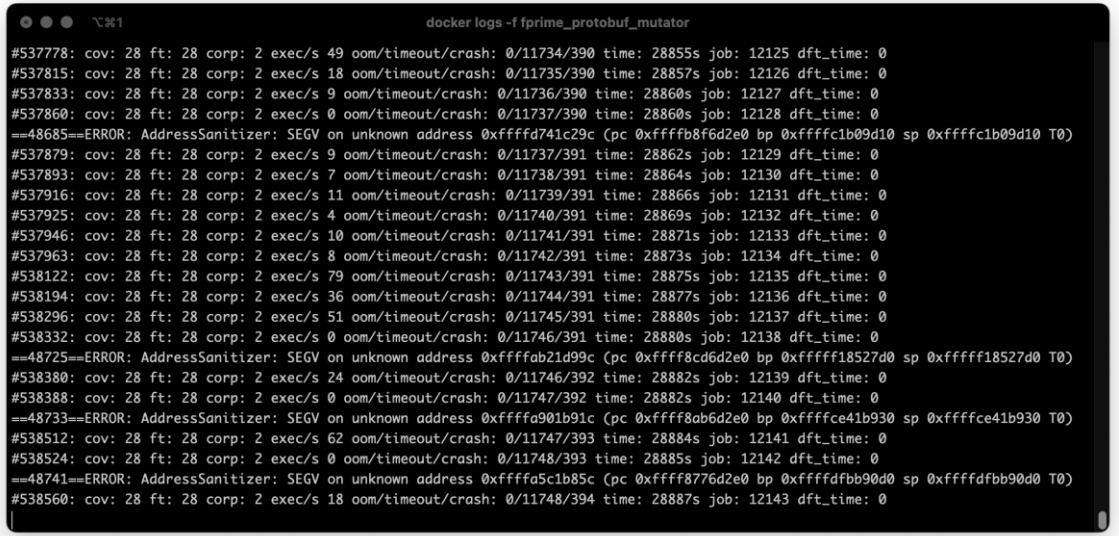


Figure 11,12,13. Fuzzer 실행 사진

진행 상황

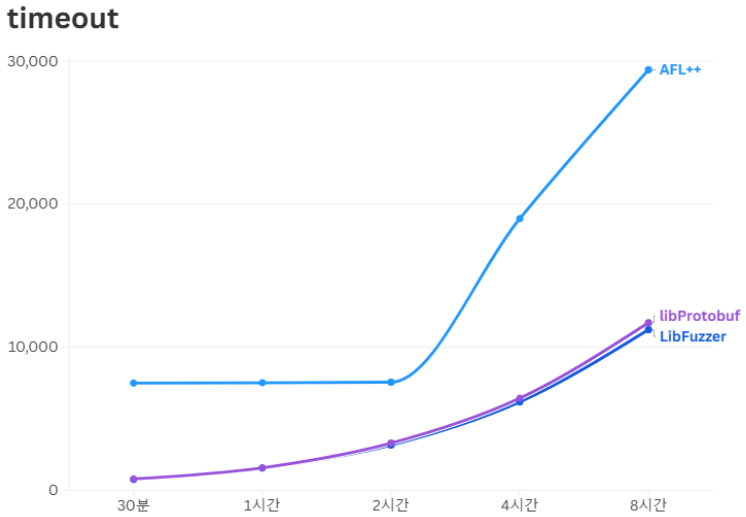


Figure 14. TimeOut 추이 그래프

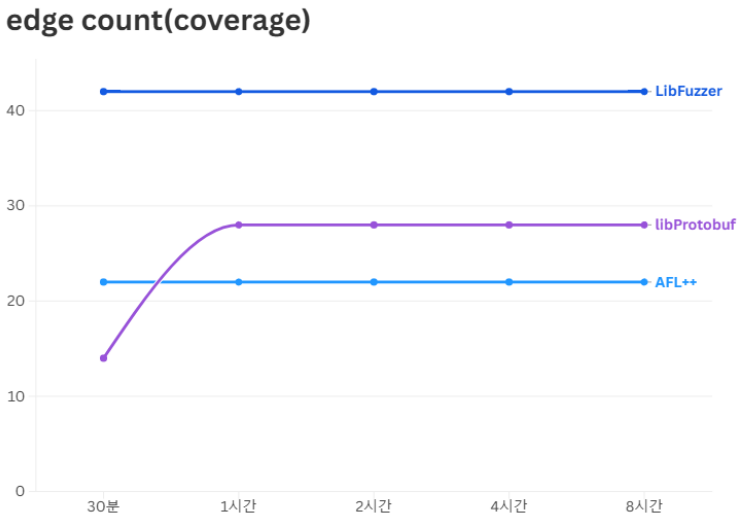


Figure 15. Code Coverage 추이 그래프

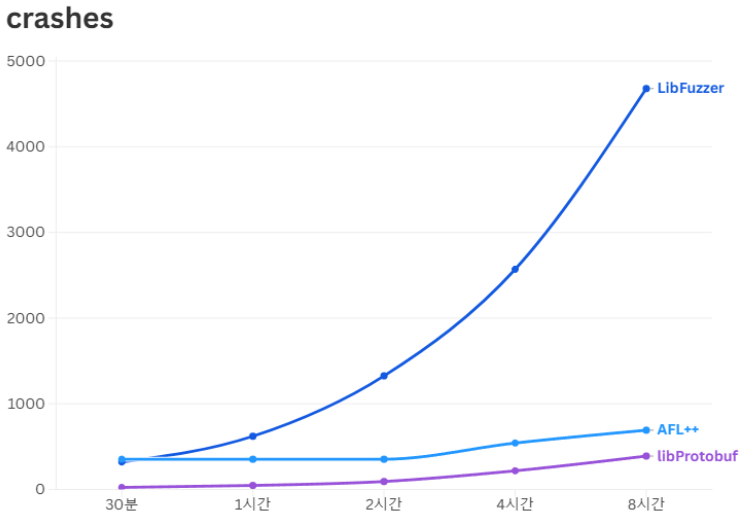


Figure 16. Crashes 추이 그래프

원인 분석

```
void CommandDispatcherImpl::compCmdReg_handler(NATIVE_INT_TYPE portNum, FwOpcodeType opCode) {
    // search for an empty slot
    bool slotFound = false;
    for (U32 slot = 0; slot < FW_NUM_ARRAY_ELEMENTS(this->m_entryTable); slot++) {
        if ((not this->m_entryTable[slot].used) and (not slotFound)) {
            this->m_entryTable[slot].opcode = opCode;
            this->m_entryTable[slot].port = portNum;
            this->m_entryTable[slot].used = true;
            this->log_DIAGNOSTIC_OpCodeRegistered(opCode,portNum,static_cast<I32>(slot));
            slotFound = true;
        } else if ((this->m_entryTable[slot].used) &&
            (this->m_entryTable[slot].opcode == opCode) &&
            (this->m_entryTable[slot].port == portNum) &&
            (not slotFound)) {
                slotFound = true;
                this->log_DIAGNOSTIC_OpCodeReregistered(opCode,portNum);
        } else if (this->m_entryTable[slot].used) { // make sure no duplicates
            FW_ASSERT(this->m_entryTable[slot].opcode != opCode, static_cast<FwAssertArgType>(opCode));
        }
    }
    FW_ASSERT(slotFound,static_cast<FwAssertArgType>(opCode));
}
```

Figure 17. CmdDispatcherImpl.cpp 소스 코드 일부

원인 분석

```
QueueInterface::Status Queue::receive(Fw::SerializeBufferBase& destination,
                                      QueueInterface::BlockingType blockType,
                                      PlatformIntType& priority) {
    FwSizeType actualSize = 0;
    destination.resetSer();  // Reset the buffer
    QueueInterface::Status status =
        this->receive(destination.getBuffAddrSer(), destination.getBuffCapacity(), blockType, actualSize, priority);
    if (status == QueueInterface::Status::OP_OK) {
        Fw::SerializeStatus serializeStatus =
            destination.setBuffLen(static_cast<Fw::Serializable::SizeType>(actualSize));
        if (serializeStatus != Fw::SerializeStatus::FW_SERIALIZE_OK) {
            status = QueueInterface::Status::SIZE_MISMATCH;
        }
    }
    return status;
}
```

```
Fw::QueuedComponentBase::MsgDispatchStatus CommandDispatcherComponentBase ::
  doDispatch()
{
  ComponentIpcSerializableBuffer msg;
  FwQueuePriorityType priority = 0;

  Os::Queue::Status msgStatus = this->m_queue.receive(
    msg,
    Os::Queue::BLOCKING,
    priority
  );
  FW_ASSERT(
    msgStatus == Os::Queue::OP_OK,
    static_cast<FwAssertArgType>(msgStatus)
  );

  // Reset to beginning of buffer
  msg.resetDeser();
```

Figure 18, 19. OS::Queue 소스 코드 일부

원인 분석

- Fprime의 cmddispatcher 모듈은 내부 상태 정보를 유지하는 준-상태 구조를 갖고 있음.
- 그러나 연구에서 사용한 상용 Fuzzer들은 이러한 상태를 고려하지 않고 입력만 무작위로 바꾸는 방식을 사용해서, 새로운 경로를 탐색하지 못하는 한계가 존재

-〉 상용 Fuzzer의 한계점을 보완한 위성 소프트웨어 전용 Fuzzer 개발이 목표

향후 계획

- 해당 연구를 바탕으로 fprime에 적합한 fuzzing 기법 고안
- 테스트 범위 확장하여 fprime 이외의 다른 오픈 소스 위성 소프트웨어에 도 적용 가능한지 검토

- 소형 위성의 신뢰성을 높이고 성공률 높이는데 기여하는 것이 목표

Thank You