

LLM과 함께하는 테스팅과 디버깅

김윤희

한양대학교 소프트웨어공학 연구실

- Logic-based property specification (Linear Temporal Logic, Computation Tree Logic)
- Applying symbolic model checking to embedded systems

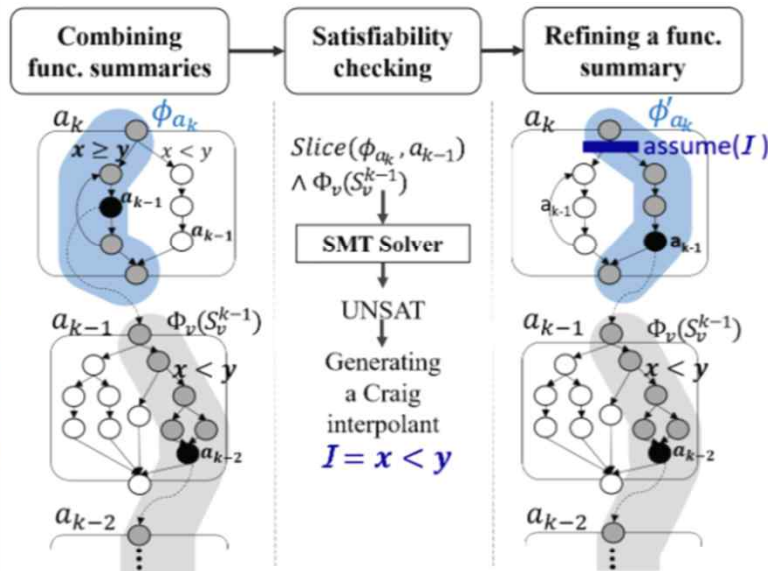


Figure 4: FSR using the Craig interpolants

3.5.3 Generation of Symbolic Path Formulas by Combining Function Summaries. To generate a SPF Φ_v to validate a failure at v , FOCAL combines the summaries of the functions in a failure-context $\langle \text{main}, \dots, a_1 \rangle$ of v and ψ_v in a backward order. If the combined SPF is satisfiable, FOCAL uses a solution of the formula obtained by an SMT solver as a system-level test input to validate a failure at v .

For Fig. 2 where $f_v = f_5$, FOCAL generates SPFs as follows:

1. Suppose that $r(f_3, f_5) > r(f_1, f_5)$. FOCAL generates $\Phi_v(\langle f_3 \rangle) = \text{Slice}(\phi_{f_3}, f_5) \wedge \psi_v$. If $\Phi_v(\langle f_3 \rangle)$ is satisfiable and $r(f_1, f_3) > r(f_2, f_3)$, FOCAL increases a failure-context of v to $\langle f_1, f_3 \rangle$.
2. It generates $\Phi_v(\langle f_1, f_3 \rangle) = \text{Slice}(\phi_{f_1}, f_3) \wedge \Phi_v(\langle f_3 \rangle)$ and checks if $\Phi_v(\langle f_1, f_3 \rangle)$ is satisfiable. If yes, FOCAL increase a failure-context of v to $\langle \text{main}, f_1, f_3 \rangle$.

THEOREM 1 (CRAIG, 1957 [9]). Suppose $A \rightarrow C$ is a valid implication in first-order logic (i.e., $\models A \rightarrow C$). Then, there is a *Craig interpolant* I such that $\models A \rightarrow I$ and $\models I \rightarrow C$.

COROLLARY 1. Suppose that $A \wedge B$ is unsatisfiable in first-order logic (i.e., $\models A \rightarrow \neg B$). Then, by Thm. 1, there is a Craig interpolant I such that $\models A \rightarrow I$ and $I \wedge B$ is unsatisfiable.³

Suppose that A is $\Phi_v(S_v^{k-1})$, B is $\text{Slice}(\phi_{a_k}, a_{k-1})$, and $\Phi_v(S_v^{k-1}) \wedge \text{Slice}(\phi_{a_k}, a_{k-1})$ is unsatisfiable. Then, by Corollary 1, there exists a Craig interpolant I of $\Phi_v(S_v^{k-1})$ and $\text{Slice}(\phi_{a_k}, a_{k-1})$ such that $I \wedge \text{Slice}(\phi_{a_k}, a_{k-1})$ is unsatisfiable. Note that $\text{Slice}(\phi_{a_k}, a_{k-1})$ represents the already explored paths in a_k . Thus, Craig interpolant I can work as a guide in concolic unit testing of a_k to avoid revisiting already explored paths (i.e., $I \rightarrow \neg \text{Slice}(\phi_{a_k}, a_{k-1})$). And at the same time, I can lead the concolic unit testing to explore paths compatible with $\Phi_v(S_v^{k-1})$ (i.e., $\Phi_v(S_v^{k-1}) \rightarrow I$).

Now we propose the following heuristic to build ϕ'_{a_k} such that $\Phi_v(\langle a_k, \dots, a_1 \rangle)$ is satisfiable.

- When FOCAL generates ϕ'_{a_k} using concolic testing, FOCAL enforces a Craig interpolant I of $\Phi_v(S_v^{k-1})$ and $\text{Slice}(\phi_{a_k}, a_{k-1})$ as a FS refining constraint so that ϕ'_{a_k} can be different from ϕ_{a_k} (and, thus, ϕ'_{a_k} may not conflict with $\Phi_v(S_v^{k-1})$).

This strategy constructs a new FS ϕ'_{a_k} that can be compatible with $\Phi_v(S_v^{k-1})$. This is because I guides concolic testing to make ϕ'_{a_k} contain symbolic paths *different from* the ones in ϕ_{a_k} by pruning $\text{Slice}(\phi_{a_k}, a_{k-1})$ (because $I \wedge \text{Slice}(\phi_{a_k}, a_{k-1})$ is unsatisfiable).

FOCAL implements this strategy by inserting `assume(I)` at the beginning of the body of a_k , which guides concolic execution to explore paths that satisfy I by terminating an execution of a_k immediately if I is violated.

Suppose that ϕ'_{a_k} does not resolve the conflict in the first function

첫 시작: Rust 퍼징

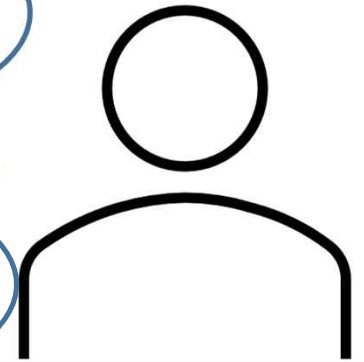


자네 나와 같이
Rust 퍼징 한 번
해보지 않겠나?

C/C++ 퍼징은 너무
레드오션인데
Rust는 뭔가 좀
새롭잖아?

네? 왜 Rust요????

어려워 보이는데..



M

- 결과: 약 6개월 진행 후 포기
- 문제점:
 - 메모리 안전하지 않은 영역에 집중하기 위한 **많은 분석** 필요
 - 전반적으로 품질 좋은 Rust 소프트웨어 생태계

LLM이 쓸만하다고 생각한 계기

- Kang et al., ICSE 2023
- LLM을 활용하여 버그 리포트로부터 버그 재현 테스트 생성
- 소감: 아 LLM이 진짜 괜찮은 테스트 코드를 생성할 수 있을 정도로 발전했구나

Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction

Sungmin Kang*
School of Computing
KAIST
Daejeon, Republic of Korea
sungmin.kang@kaist.ac.kr

Juyeon Yoon*
School of Computing
KAIST
Daejeon, Republic of Korea
juyeon.yoon@kaist.ac.kr

Shin Yoo
School of Computing
KAIST
Daejeon, Republic of Korea
shin.yoo@kaist.ac.kr

arXiv:2209.11515v2 [cs.SE] 22 Dec 2022

Abstract—Many automated test generation techniques have been developed to aid developers with writing tests. To facilitate full automation, most existing techniques aim to either increase coverage, or generate exploratory inputs. However, existing test generation techniques largely fall short of achieving more semantic objectives, such as generating tests to reproduce a given bug report. Reproducing bugs is nonetheless important, as our empirical study shows that the number of tests added in open source repositories due to issues was about 28% of the corresponding project test suite size. Meanwhile, due to the difficulties of transforming the expected program semantics in bug reports into test oracles, existing failure reproduction techniques tend to deal exclusively with program crashes, a small subset of all bug reports. To automate test generation from general bug reports, we propose LIBRO, a framework that uses Large Language Models (LLMs), which have been shown to be capable of performing code-related tasks. Since LLMs themselves cannot execute the target buggy code, we focus on post-processing steps that help us discern when LLMs are effective, and rank the produced tests according to their validity. Our evaluation of LIBRO shows that, on the widely studied Defects4J benchmark, LIBRO can generate failure reproducing test cases for 33% of all studied cases (251 out of 750), while suggesting a bug reproducing test in first place for 149 bugs. To mitigate data contamination (i.e., the possibility of the LLM simply remembering the test code either partially or in whole), we also evaluate LIBRO against 31 bug reports submitted after the collection of the LLM training data terminated: LIBRO produces bug reproducing tests for 32% of the studied bug reports. Overall, our results show LIBRO has the potential to significantly enhance developer efficiency by automatically generating tests from bug reports.

Index Terms—test generation, natural language processing, software engineering

I. INTRODUCTION

Software testing is the practice of confirming that software meets specification criteria by executing tests on the software under test (SUT). Due to the importance and safety-critical nature of many software projects, software testing is one of the most important practices in the software development process. Despite this, it is widely acknowledged that software testing is tedious due to the significant human effort required [1]. To fill this gap, automated test generation techniques have been studied for almost half a century [2], resulting in a number of tools [3], [4] that use implicit oracles (regressions or crash

detection) to guide the automated process. They are useful when new features are being added, as they can generate novel tests with high coverage for a focal class.

However, not all tests are added immediately along with their focal class. In fact, we find that a significant number of tests originate from *bug reports*, i.e., are created in order to prevent future regressions for the bug reported. This suggests that the generation of *bug reproducing tests from bug reports* is an under-appreciated yet impactful way of automatically writing tests for developers. Our claim is based on the analysis of a sample of 300 open source projects using JUnit: the number of tests added as a result of bug reports was on median 28% of the size of the overall test suite. Thus, the bug report-to-test problem is regularly dealt with by developers, and a problem in which an automated technique could provide significant help. Previous work in bug reproduction mostly deals with crashes [5], [6]; as many bug reports deal with semantic issues, their scope is limited in practice.

The general report-to-test problem is of significant importance to the software engineering community, as solving this problem would allow developers use a greater number of automated debugging techniques, equipped with test cases that reproduce the reported bug. Koyuncu et al. [7] note that in the widely used Defects4J [8] bug benchmark, bug-revealing tests *did not exist* prior to the bug report being filed in 96% of the cases. Consequently, it may be difficult to utilize the state-of-the-art automated debugging techniques, which are often evaluated on Defects4J, when a bug is first reported because they rely on bug reproducing tests [9], [10]. Conversely, alongside a technique that automatically generates bug-revealing tests, a wide range of automated debugging techniques would become usable.

As an initial attempt to solve this problem, we propose prompting Large Language Models (LLMs) to generate tests. Our use of LLMs is based on their impressive performance on a wide range of natural language processing tasks [11] and programming tasks [12]. In this work, we explore whether their capabilities can be extended to generating test cases from bug reports. More importantly, we argue that the performance of LLMs when applied to this problem has to be studied along with the issue of *when* we can rely on the tests that LLMs produce. Such questions are crucial for actual

*: these authors contributed equally.

왜 LLM을 써보려고 했을까?

- (2022년 11월기준) AI의 시대가 이미 성큼 다가옴을 느낌
 - GPT-3.5 기반 ChatGPT 의 공개
 - 기존 CodeBERT 등의 코드 모델과는 차원이 다른 성능
 - 지금이라도 흐름을 타야한다는 본능적인 감각
- 진실된 이유: 솔직히 만만해보였습니다
 - 아이디어 구현에 복잡한 배경 지식이 필요하지 않음
 - C 프로그램 정적 분석: Clang 파서로 AST 순회하는데만 1~4주
 - C 프로그램 포인터 분석: SVF로 기본적인 내용 확인하는데 2주
 - 그럼에도 불구하고 꽤 괜찮은 결과를 보여줌

LLM을 활용한 퍼징 하네스 생성 (202301)

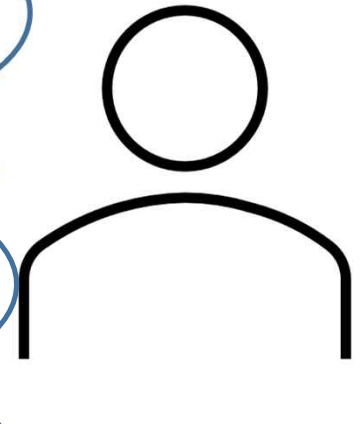


LLM! LLM을
하자!

네? 그건 또
왜요???

쉽잖아?

음...



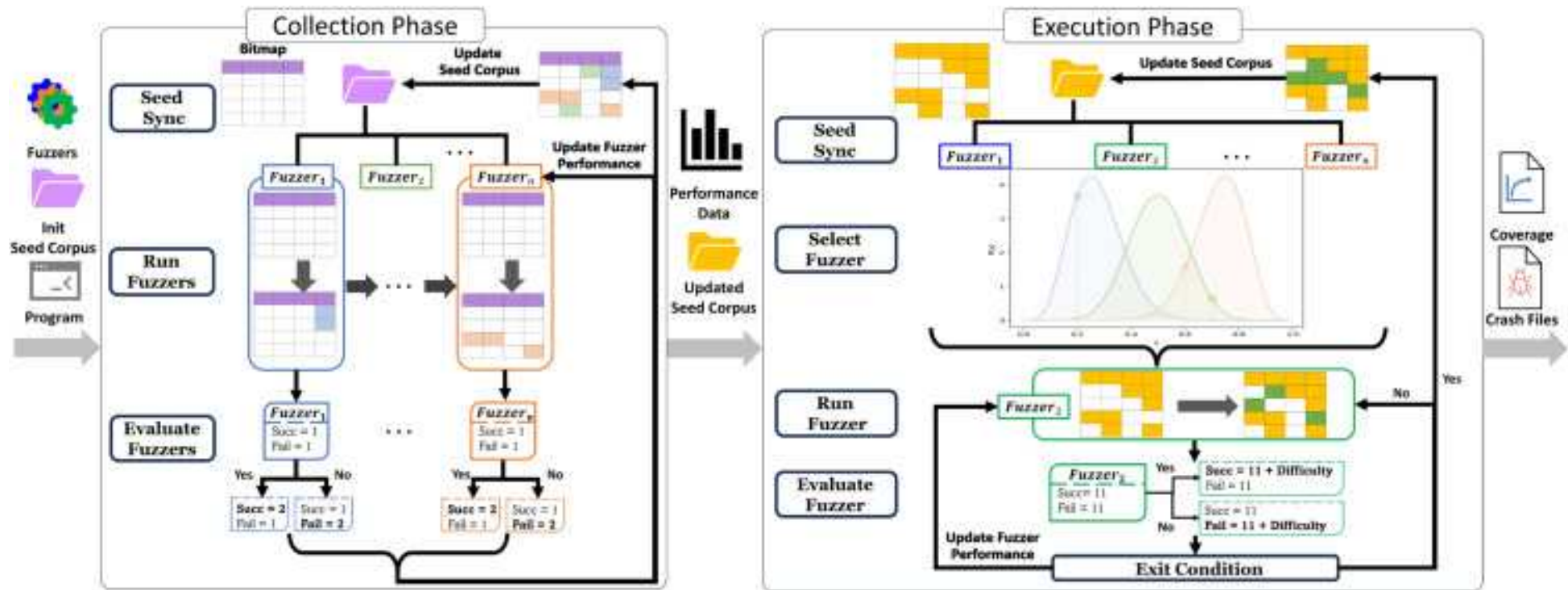
M

- 결과: 약 8개월 수행 후 포기
- 참고: 2023년 8월 Google OSS-fuzz-gen 공개

왜 실패했을까?

- LLM을 어떻게 사용해야 하는지에 대한 감이 전혀 없었음
 - 하나하나 **자세한 명령** 기재 필요
 - 당시 사용 프롬프트 길이: 약 10줄 내외
 - Google OSS-Fuzz-Gen 프롬프트 길이: C++ 타겟 기준 약 250~300 줄
 - 좋은 예제 코드의 중요성
 - 프롬프트 엔지니어링(Chain-of-Thought 등)
- LLM이 잘하는 것과 잘 하지 못하는 것을 파악하지 못함
 - 잘하는 것: 대략적인 구조 보여주기, 힌트 제시하기
 - 잘 못하는 것: 꼼꼼한 작업, 동일한 반복 작업
 - 예: 모르는 API를 LLM에게 물어보면 해당 API의 사용 패턴을 그럴싸하게 보여주나 컴파일은 안 됨

LLM은 나랑 안 맞나봐요..



- 강화학습 기반 퍼저 스케줄링 [Mo et al., JSS25]

LLM을 활용한 퍼징 하네스 생성 (2024)

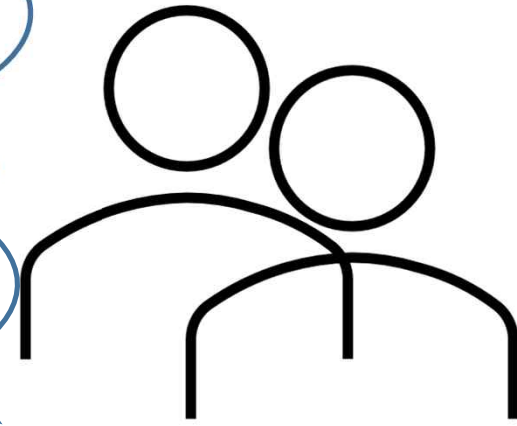


LLM! LLM을
하자!

퍼징 하네스
만들거지?

네 좋아용!

아니용!

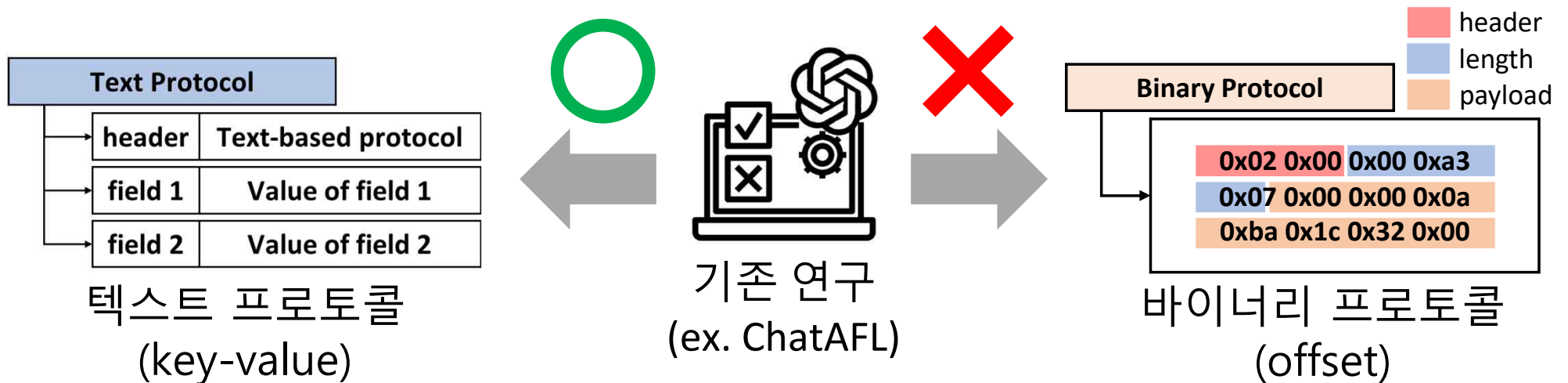


학부생 Y, J

LLM을 활용한 바이너리 프로토콜 퍼징

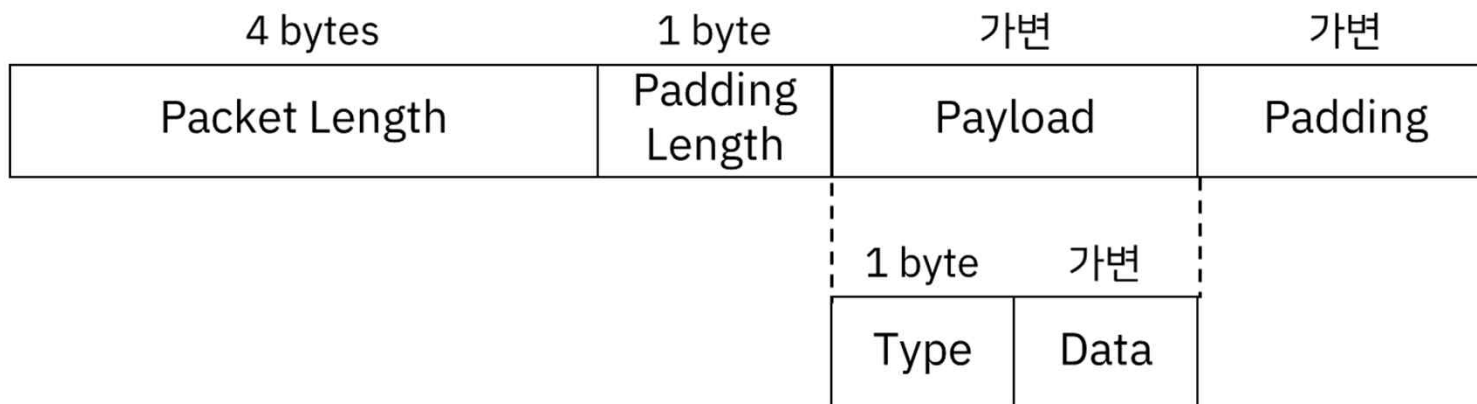
텍스트 프로토콜 위주의 기존 연구

- 기존의 LLM 기반 프로토콜 퍼징 성능 향상 연구는 LLM이 다루기 쉬운 텍스트 프로토콜에 집중.
 - 텍스트 프로토콜: key-value 형식 사용
 - 바이너리 프로토콜: 고정/가변형의 필드별 바이트 수, 오프셋을 사용



바이너리 프로토콜 구조 특징

- 필드 간의 구분이 어려움
 - 텍스트 프로토콜: 라인 단위 구분 혹은 JSON 등 구조화된 포맷 사용
 - 바이너리 프로토콜: 고정 길이 필드 혹은 타 필드에 지정된 길이만큼의 필드
- 필드간의 의존성 파악이 어려움
 - 앞 필드에 지정된 데이터 값이 뒷 필드의 길이와 내용을 결정

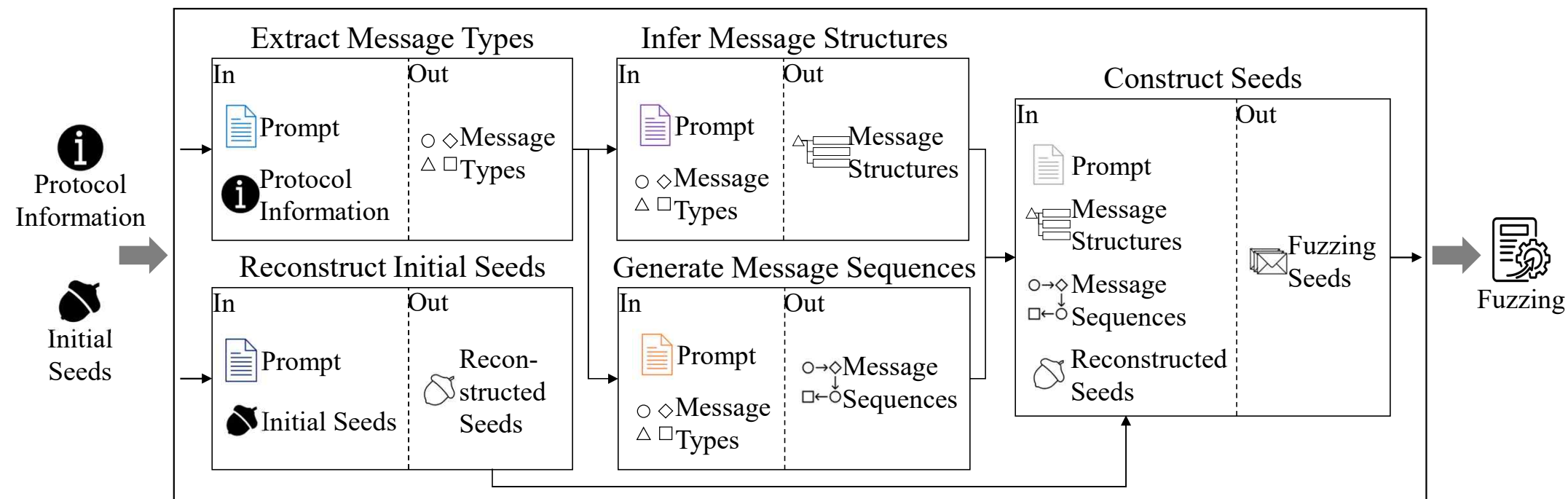


SSH 프로토콜 구조

Structure-aware LLM-assisted Seed Generation

- LLM이 추론한 프로토콜 정보를 활용하여 valid seed 생성
 - 각 타입 별 올바른 메시지 구조를 파악하고
 - 파악한 메시지들의 올바른 순서를 파악한다

SteLLaFuzz Framework



프로토콜 메시지 구조 추론 예시

You are a network protocol expert for [PROTOCOL]. Your task is to extract the message structure for the client-to-server message type [TYPE].

Instructions:

1) Extract the Message Structure:

- List all fields as defined in the [PROTOCOL] documentation.
- Include common headers if applicable, but focus on fields unique to [TYPE].

2) Specify Field Details:

- name – exact name of the field as defined in the protocol specification
- fixed_byte_length – size of the field in bytes; set to null if variable
- data_type – data type (e.g., string, int, bytes, boolean)
- description – brief explanation of the field's purpose or meaning
- details – additional info such as encoding, length constraints, etc.

3) Provide Structured Reasoning:

- Step-by-step explanation
- Cite documentation, RFCs
- Note assumptions or ambiguities



```
"QUERY": {
  "protocol": "DNS",
  "message_type": "QUERY",
  "code": "0",
  "type_description": "A standard query
    message sent by a client to request
    information from a DNS server.",
  "fields": [
    {
      "name": "Transaction_ID",
      "fixed_byte_length": 2,
      "data_type": "uint16",
      "description": "A 16-bit identifier
        assigned by the program that
        generates any kind of query.",
      "details": "Used to match responses to
        queries."
    },
    {
      "name": "Flags",
      "fixed_byte_length": 2
    }
  ]
}
```

DNS 쿼리 메시지 추론 결과

메시지 구조 추론 프롬프트

실험 설정

- 구현: AFLNet 기반으로 파이썬 코드 550여 줄 구현
- 타겟: ProFuzzBench의 텍스트 프로토콜 5개 + 바이너리 프로토콜 8개 구현
- 비교 대상: AFLNet[ICST20], ChatAFL[NDSS24]
- 설정:
 - 64-core AMD EPYC 7763 , 512 GB RAM,
 - 24시간 퍼징 수행, 10번 반복
- LLM 모델: GPT-4o-mini 모델 사용

실험 결과

- AFLNet, ChatAFL 대비 바이너리 프로토콜 상태 커버리지 39.6%, 148.2% 향상
- 텍스트 프로토콜에서도 준수한 상태 커버리지 달성

Subject	State Coverage					State Transition Coverage				
	SteLLaFuzz	AFLNet	Improve	ChatAFL	Improve	SteLLaFuzz	AFLNet	Improve	ChatAFL	Improve
Dcmtk	4.0	4.0	0.0%	4.0	0.0%	3.0	3.0	0.0%	3.0	0.0%
Dnsmasq	116.4	100.5	15.9%	81.6	42.6%	367.6	321.9	14.2%	257.8	42.6%
TinyDTLS	10.0	9.0	11.1%	9.7	3.1%	36.0	34.0	5.9%	37.0	-2.7%
OpenSSH	57.0	32.6	74.8%	19.5	192.3%	115.8	54.1	114.0%	30.5	279.7%
OpenSSL	58.3	29.7	96.3%	9.7	503.1%	64.3	34.3	87.5%	13.6	374.3%
Binary avg.	–	–	39.6%	–	148.2%	–	–	44.3%	–	138.8%
forked-daapd	8.1	8.1	0.0%	7.9	2.5%	23.2	20.7	12.1%	19.9	16.6%
Bftpd	24.0	24.0	0.0%	24.0	0.0%	227.2	169.1	34.4%	179.9	26.3%
LightFTP	23.7	23.4	1.3%	23.7	0.0%	221.1	197.7	11.8%	198.2	11.6%
ProFTPD	28.8	21.9	31.8%	27.8	3.8%	273.8	170.0	61.1%	185.9	47.3%
Pure-FTPd	28.8	27.0	6.4%	29.2	-1.6%	286.8	227.0	26.3%	252.2	13.7%
Live555	14.4	12.7	13.8%	14.2	1.4%	161.9	109.3	48.1%	136.6	18.5%
Exim	18.0	16.1	12.1%	17.7	1.7%	96.6	66.8	44.6%	87.0	11.0%
Kamailio	13.8	14.9	-7.7%	15.1	-8.8%	93.5	102.9	-9.1%	100.9	-7.3%
Text avg.	–	–	7.2%	–	-0.1%	–	–	28.7%	–	17.2%
Total avg.	–	–	19.7%	–	56.9%	–	–	34.7%	–	64.0%

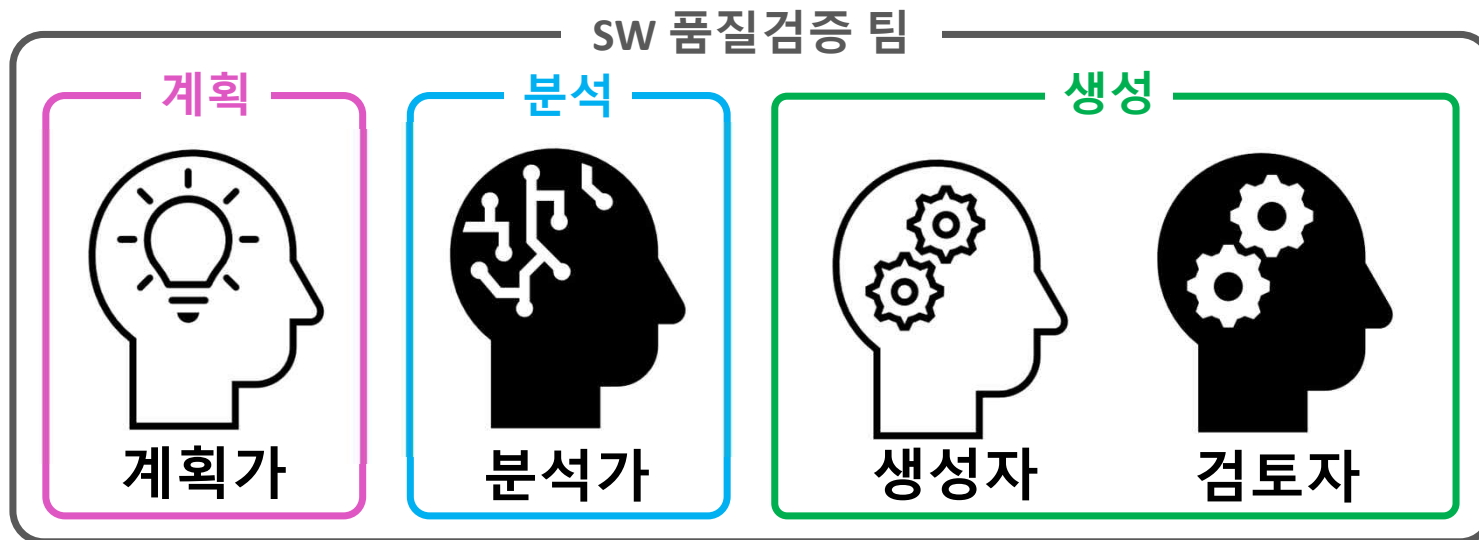
LLM을 활용한 유닛 테스트 생성

단독 에이전트의 테스트 생성 한계

- LLM 등장으로 자동 단위 테스트 생성이 개선됐지만, LLM의 오탐과 환각현상이 발생.
 - 한 측면에서 이해하고 처리하기 때문에, 특정 측면을 간과하거나 예기치 못한 오류를 포함.
- 하나의 에이전트가 완벽하지 않다면 서로 돕고 살아야 한다
 - 검증 과정을 강화하여 여러 에이전트가 상호 보완적으로 작동하도록 설계.
 - 추론 모델 O1과 언어모델 GPT4의 교차 사용으로 장점을 극대화 하여 정교한 문제해결 가능.

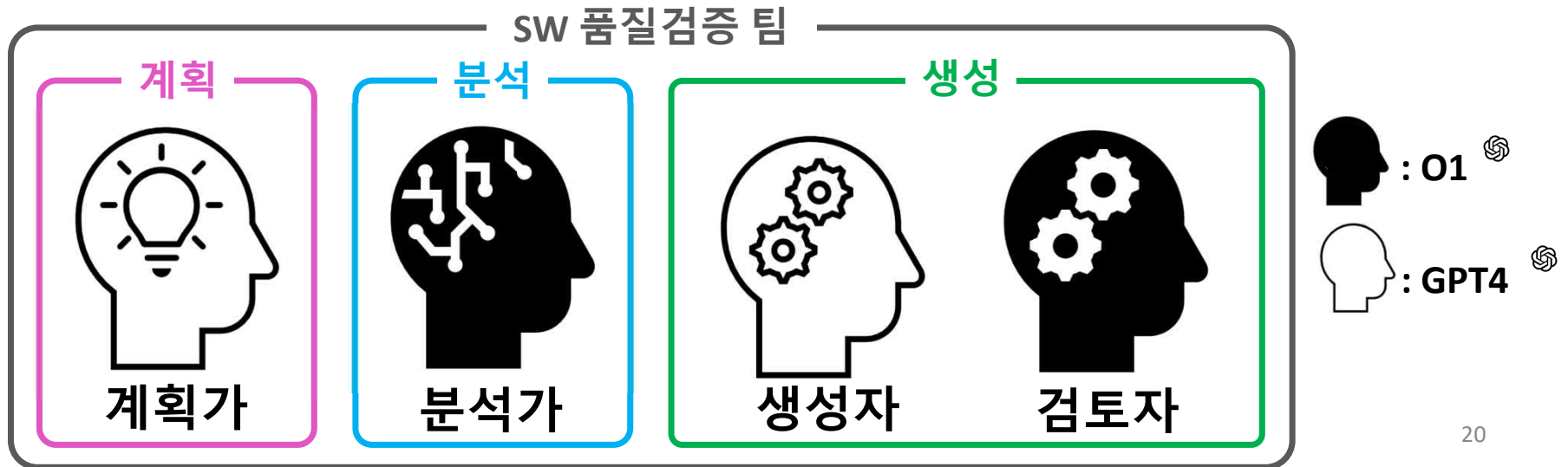
다중 LLM 에이전트의 역할 분배

- 역할 분담을 통한 단위 테스트 생성
 - 각 에이전트는 계획, 분석, 생성, 검토의 역할 전문화를 통해 복잡한 작업을 분배.
 - 에이전트 간의 상호 검증 체계를 통해서 생성된 단위 테스트의 구조적, 논리적 오류를 최소화.
 - 역할 별 전문성을 결합하여 소프트웨어 테스트 작업에서 시너지를 냄.

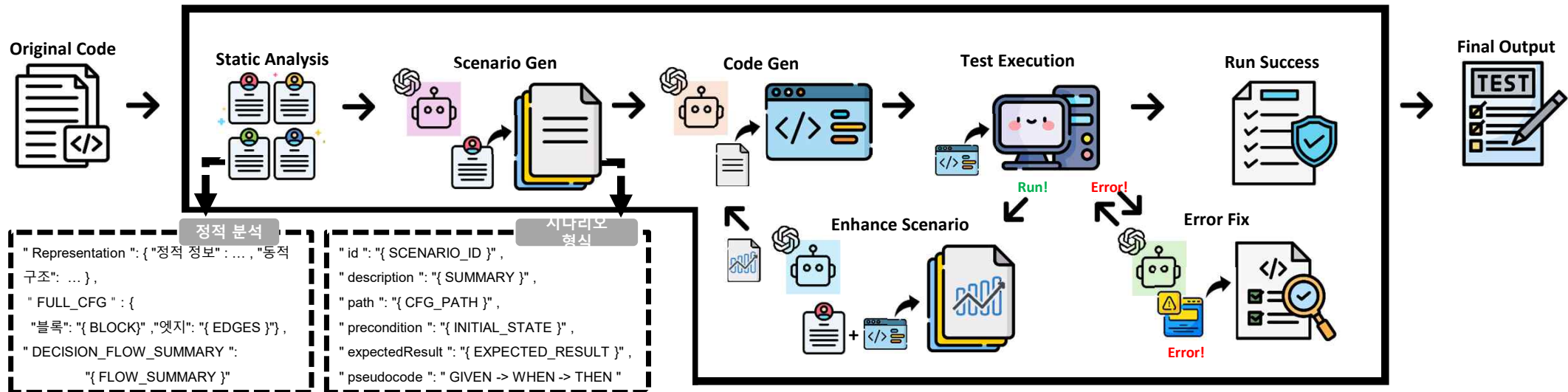


LLM 에이전트의 모델 교차 사용

- Open AI LLM의 추론 모델 o1과 언어 모델 GPT4를 역할에 맞게 사용
 - 추론 모델 o1은 코드 분석, 오류 검토, 생성 모델 GPT4는 계획 생성, 코드 생성 역할.
 - 추론 모델이 더 좋은 거 아니가요? 그것만 쓰면 안되나요?
 - 추론 모델은 비싸다.
 - (1M 토큰당) o1: 입력 \$15, 출력 \$60 vs gpt-4o: 입력 \$2.5, 출력 \$10
 - 추론 모델이 모든 걸 더 잘 하는 건 아니다
 - 컴파일 오류 수정은 GPT4가 더 우수



LLM 멀티 에이전트 기반의 자동 단위 테스트 생성



테스트 시나리오 기반 유닛 테스트 생성

- 테스트 코드 직접 생성보다 시나리오 생성 후 테스트 코드 생성 성능이 좋음
- 테스트 코드 직접 개선 보다 시나리오 개선 후 코드 생성 성능이 더 좋음

Commons-lang DateUtils#iterator TC 05



```
"description": " iterator with rangeStyle = RANGE_WEEK_RELATIVE and calendar on Thursday ",
"path": " B0 -> B3 -> B6 -> B10 -> B11 -> B12 -> B14 -> B16 -> B18 -> B20 -> B22 ",
"precondition": " Calendar is set to a Thursday ",
"expectedResult": " DateIterator adjusted based on the current day of the week ",
"pseudocode": [ " TEST TC05 iterator RANGE_WEEK_RELATIVE on Thursday ",
  " GIVEN ",
  "   Calendar calendar = set to a Thursday ;" ,
  "   int rangeStyle = RANGE_WEEK_RELATIVE ;" ,
  " WHEN ",
  "   Iterator < Date > iterator = DateUtils . iterator ( calendar , rangeStyle ) ;" ,
  " THEN ",
  "   assert DateIterator is adjusted relative to Thursday ;" ] }
```

▶ 시나리오 예제

실험 설정

- 구현: Python 1.3K LoC
- 타겟:
 - Defects4J의 Maven 빌드 시스템 사용 프로젝트 14개 중
 - 순환복잡도 10 이상인 메소드
- 비교 대상: EvoSuite 1.2.0 [FSE11] ChatUnitTest[FSE24]
- LLM 모델: GPT-4o 생성 모델 + o1 기본 모델

실험 결과

- Evosuite 대비 평균 32.6%p분기 커버리지 향상 .
- ChatUniTest 대비 평균 35.6%p 분기 커버리지 향상 .

	Branch Coverage(%)		
Abbr.	<i>Evo</i>	<i>Chat</i>	<i>M ANTIS</i>
COD	94.8	18.2	87.4
COL	10.5	0.0	80.6
COM	16.7	28.1	32.3
CSV	97.0	56.1	97.0
LAN	44.7	0.0	80.9
MAT	1.9	19.3	53.5
GSO	45.0	40.0	72.5
TIM	18.2	46.7	73.4
CHA	0.0	33.1	50.3
COR	42.9	24.5	46.4
DAB	3.3	1.8	47.0
DAF	0.0	29.6	39.9
JSO	44.8	36.1	70.4
JXP	10.0	54.2	54.5
AVG	30.7	27.7	63.3
MED	17.5	28.9	62.4

LLM 실험의 어려움

- 과거 결과 재현이 어려움
 - 내가 쓰는 GPT-3.5-Turbo가 논문에서 사용한 GPT-3.5-Turbo와 같은 모델일까?
 - 지난 주 GPT-4o와 오늘의 GPT-4o가 같은 모델일까?
- 모델의 발전이 나머지 요소를 압도
 - GPT-3.5-Turbo를 잘 쓰기 위한 많은 노력들이 GPT-4 출시 후 무용지물
- Over-fitting 문제
 - GPT-4o: 2023년 10월 1일까지의 데이터 학습, GPT-4.1: 2024년 6월 1일까지 학습
 - 2023년 10월 1일 이후에 완전히 새로 릴리즈된, 벤치마크 하기 적합한 프로젝트의 수?
- 왜 되는데 & 왜 안되는데?
 - 잘 되는데 기여하는 요소와 잘 안되는데 기여하는 요소 파악이 어려움