LLM Based Input Space Partitioning Testing for Library APIs

Jiageng Li
Fudan University
jgli22@m.fudan.edu.cn

Zhen Dong*
Fudan University
zhendong@fudan.edu.cn

Chong Wang
Nanyang Technological University
chong.wang@ntu.edu.sg

Haozhen You
Fudan University
hzyou23@m.fudan.edu.cn

Cen Zhang
Nanyang Technological University
cen001@e.ntu.edu.sg

Yang Liu
Nanyang Technological University
yangliu@ntu.edu.sg

Xin Peng
Fudan University
pengxin@fudan.edu.cn

***Abstract*—Automated library APIs testing is difficult as it requires exploring a vast space of parameter inputs that may involve objects with complex data types. Existing search based approaches, with limited knowledge of relations between object states and program branches, often suffer from the low efficiency issue, *i.e.,* tending to generate invalid inputs. Symbolic execution based approaches can effectively identify such relations, but fail to scale to large programs.**

In this work, we present an LLM-based input space partitioning testing approach, LISP, for library APIs. The approach leverages LLMs to understand the code of a library API under test and perform input space partitioning based on its understanding and rich common knowledge. Specifically, we provide the signature and code of the API under test to LLMs, with the expectation of obtaining a text description of each input space partition of the API under test. Then, we generate inputs through employing the generated text description to sample inputs from each partition, ultimately resulting in test suites that systematically explore the program behavior of the API.

We evaluate LISP on more than 2,205 library API methods taken from 10 popular open-source Java libraries (*e.g.,* apache/commons-lang with 2.6k stars, guava with 48.8k stars on GitHub). Our experiment results show that LISP is effective in library API testing. It significantly outperforms state-of-the-art tool EvoSuite in terms of edge coverage. On average, LISP achieves 67.82% branch coverage, surpassing EvoSuite by 1.21 times. In total, LISP triggers 404 exceptions or errors in the experiments, and discovers 13 previously unknown vulnerabilities during evaluation, which have been assigned CVE IDs.

***Index Terms*—Input Space Partitioning Testing, Large Language Models, Symbolic Execution, API testing.**

I. Introduction

The third party libraries, as an essential part in software ecosystems, have become one of the most significant contributors to fast development of today's software system. According to a recent study [1], a Java project directly relies on different 14 third party libraries. Vulnerabilities within these libraries can pose significant risks to numerous software systems. Consequently, testing libraries is imperative to ensure system security.

However, testing library APIs is notoriously challenging as it entails exploring a vast input space of multiple parameters, particularly when these parameters involve objects with complex data types. The behavior of the libraries can be constrained by a specific state of one or more input objects. Triggering such a state involves generating input values satisfying *relevant* conditions as well as generating statements to instantiate these objects.

This poses numerous challenges for existing automated test generation techniques: (1) *Search based testing:* Most existing techniques [2]–[7] frame automated test generation as an optimization problem over the input space with the goal of generating inputs to achieve maximal code coverage, for instance, EvoSuite [3], a widely used automated test generation tool adopts a genetic algorithm to generate tests. The problem with this type of techniques is the low efficiency issue when tackling the expansive space of inputs involving multiple objects with complex data types within library APIs; (2) *Symbolic execution:* Symbolic execution is an effective testing technique that can generate inputs that cover desired program paths. Yannic Noller et al. leverages symbolic execution to guide fuzzing to generate inputs that cover deep program behavior [8]. Despite significant efficiency improvement, these techniques face difficulties in scaling to large programs due to inherent limitations of symbolic execution, *e.g.,* SPF [9] has limited support for heap input. SUSHI [10] proposed by Pietro Braione et al. can be only applied to Java classes.

In this paper, we view automated test generation as a program input space sampling problem. The ideal way to sample is to compute *input space partitions*, and then choose inputs from each partition so as to cover all possible program behavior. In this perspective, search-based approaches kind of leverage heuristics to guide search, aiming to sample inputs from as many partitions as possible. Symbolic execution based approaches attempt to compute input space partitions by solving program path conditions and then sample inputs from each partition. Both type of approaches come at a cost. The former requires executing a large amount of inputs that go through redundant program paths, the latter requires heavy computation resources to solve path conditions.

In this work, we propose an Large Language Model (LLM) based input space partitioning testing approach for library

* Corresponding author.

APIs. Specifically, we leverage LLMs to infer the input space partitions of a library API under test and then sample inputs from each partition so as to generate test suites with high quality. Recently, LLMs have demonstrated promising capabilities in understanding programs and common knowledge reasoning, leading to their widespread adoption in the software engineering domain [11], [12]. *Motivated by these capabilities, we explore using LLMs to automate input space partitioning, achieving the objectives of symbolic execution without explicitly performing it*. To this end, we propose a framework that interacts with LLMs to compute input space partitions for a given library API and generate input values based on textual descriptions of each partition, resulting in high-quality test inputs. Subsequently, the framework takes those inputs to generate test suites for library API testing.

We evaluated LISP on 2,205 APIs from 10 widely used libraries, including Apache Commons-lang3 and Google Guava. The results show LISP is highly effective in testing library APIs, achieving exceptionally high code coverage with a minimal number of generated inputs. In the comparison experiments, LISP outperformed the state-of-the-art technique EvoSuite, achieving 1.21 times higher edge coverage. Furthermore, LISP identified 404 exceptions across the 10 libraries, including 13 previously undiscovered vulnerabilities, which have been assigned CVE IDs. To support future research, we make our the experimental data and results publicly available at the following link: https://github.com/FudanSELab/LISP [13].

II. Motivating Examples

```
1  // ApcomplexMath.java
2  public static Apcomplex pow(Apcomplex z,Apcomplex w)
3  throws ApfloatRuntimeException {
4    Apcomplex result = ApfloatHelper.checkPow(
5      z, w, Math.min(z.precision(), w.precision()));
6    if (result != null) {
7      return result;
8    } else if (z.real().signum() >= 0 &&
9               z.imag().signum() == 0) {
10     Apfloat x = z.real();
11     Apfloat one = new Apfloat(
12       1L, Long.MAX_VALUE, x.radix());
13     x = // ignore some code
14     return exp(w.multiply(ApfloatMath.log(x)));
15   } else {
16     return exp(w.multiply(log(z)));
17   }
18 }
```

Listing 1. org.apfloat.ApcomplexMath::pow

A. Importance of Code Understanding and Common Knowledge

Listing 1 presents an API method named `pow` within the `class ApcomplexMath` from the *apfloat*. This method, which takes two parameters named `z` and `w`, exhibits distinct behaviors based on the content of `z` and `w`, which means that each input space can be represented by the states of `z` and `w`. Specifically, when `result != null`, the API returns the `result` directly (line 7); when `z.real().signum()` is greater than or equal to 0 and `z.imag().signum()` equals 0 (line 8-9), the API returns the result at line 14. Otherwise, the API engages in a calculation for complex numbers (line 16).

In software testing, precise partitioning of the input space facilitates efficient input generation.

- *Symbolic execution* is the ideal solution to partition the input space. We attempt one of the state-of-the-art tools, SPF. However, it fails to work due to insufficient modeling of native methods when creating an `Apcomplex` object.
- *Search-based testing* is another approach for input space partitioning, which is more scalable compared to symbolic execution. We use the state-of-the-art tool in the SBST field, EvoSuite, with the default configuration and run it for 200s. EvoSuite generates 64 test cases but only achieves 55% coverage. We find that EvoSuite generates a large number of equivalent inputs, none of which can reach line 10-14.

In the context of "exponentiation", awareness of certain corner cases is crucial. For instance, 0^0 is typically undefined; the computation process of $z^w(z \in R)$ is different from that of $z^w(z \notin R)$. Failure to bridge the gap between such background knowledge and software testing, leads to blind exploration of a vast search space for `z` and `w`. Therefore, it is essential to present an approach that effectively understands and navigates the input space while avoiding falling into the trap of generating invalid or single-scenario inputs. This approach should integrate common knowledge in both code-level and semantic-level.

B. Input Space Partitioning with Large Language Models

Recently, large language models (LLMs) have demonstrated considerable capabilities across diverse domains, such as code understanding [14], [15], common knowledge acquisition [16]–[20] and code generation [21], [22], which align with the requirements for library API testing.

Assume that we need to test the `pow` method. We can employ LLMs to partition the input space. Specifically, we can provide the signature and code of `pow` for LLMs and instruct them to partition the input space. Then, we can obtain the text form of the input space partitioning results, such as " (1) `z`: real part is non-negative and imaginary part is 0; `w`: is an Apcomplex number. (2) `z`: real part is negative or imaginary part is non-zero; `w`: is an Apcomplex number". From the above input space partitioning results, we know that LLMs believes that it should generate a complex number with "Real positive and Imaginary zero", which is exactly one of the conditions for entering a block (line 10-14) in Listing 1 (another implicit condition is "`result == null`").

C. Input Generation with Large Language Models

Listing 2 presents two types, `Apcomplex` and `Apfloat`. `Apcomplex` inherits `java.lang.Number` and represents complex numbers [23] in mathematics. `Apfloat` inherits the former type and represents float numbers. In addition, we present two constructors of type `Apcomplex`, and the first constructor requires inputs of type `Apfloat`.

Assume that we intend to construct a corresponding `Apcomplex` instance for the parameter `z`, which complies with the requirements of the text description of this input space partition ("real part is non-negative and imaginary part is 0"). In general, the process can be divided into two necessary steps.

```
public class Apcomplex extends Number {
    private Apfloat real;
    private Apfloat imag;
    public Apcomplex(Apfloat real, Apfloat imag) ..
    public Apcomplex(String value) ..
    // overlook other constructors
}

public class Apfloat extends Apcomplex {
    private ApfloatImpl impl;
    public Apfloat(long value) ..
    public Apfloat(String value, long precision) ..
    // overlook other constructors
}
```

Listing 2. Type `Apcomplex` and Type `Apfloat`

1) Top-down type dependency analysis and constructor selection: To generate inputs for a reference type, we need to acquire (1) all derived classes of that type, and (2) all constructors of any involved reference types. For this example, to generate an input object of `Apcomplex` type representing $1 + 0i$, we first retrieve its available constructors and then identify the appropriate constructors. This process continues recursively until all relevant reference types are addressed, resulting in a sequence of constructors that can be used to generate the target object. Specifically, we provide LLMs with the text description of partition, so as to drive LLMs to select the appropriate constructors. In Listing 2, the first constructor that takes `real` and `imag` as two parameters, is exactly what we need. Since the type of `real` and `imag` is still a reference type, we repeat the previous process to generate two `Apfloat` objects. In this case, we use LLMs to select three constructors, as depicted in the upper half of Listing 3.

```
    // selected constructors
    // after type dependency analysis
    Apfloat real = new Apfloat(/* TODO */);
    Apfloat imag = new Apfloat(/* TODO */);
    Apfloat c1 = new Apcomplex(real, imag);

    // object instantiation statements
    float real_value = 1.0f;
    float imag_value = 0.0f;
    Apfloat real = new Apfloat(real_value);
    Apfloat imag = new Apfloat(imag_value);
    Apfloat c1 = new Apcomplex(real, imag);
```

Listing 3. Selected Constructors and Instantiation Statements

2) Bottom-up object instantiation with concrete values: After obtaining the appropriate constructors, we need to fill in correct values to generate the desired input object. For this example, to instantiate an `Apcomplex` instance representing $1 + 0i$, it is required to construct an `Apfloat` object representing 1 and another `Apfloat` object representing 0, according to the selected constructors in the upper half of Listing 3. Specifically, we can provide LLMs with these selected constructors, supplemented by a text description of the partition with specific values, so as to guide LLMs to generate valid inputs, as shown in the lower half of Listing 3.

Looking at the process of constructing the `z` for `pow`, LLMs can serve as a vital tool in the field of input space partitioning testing. Specifically, we have utilized the code understanding and generation capabilities of LLMs in three place, *i.e.,* input space partitioning, top-down type dependency analysis and bottom-up object instantiation.

III. Approach: LISP

We introduce LISP, a novel workflow designed to strategically guide Large Language Models (LLMs) in understanding the source code of API methods. This approach ultimately generates high-quality inputs and tests drivers for library APIs.

In Figure 1, the lower section (*i.e.,* the gray part) illustrates the common process of existing input generation approaches. We find that search-based approaches typically search and partition the input space at runtime, often neglecting the source code of the APIs [10]. Building on these insights, the upper section of Figure 1 depicts our proposed workflow, which decomposes the API input generation process into three parts: *input space partitioning*, *top-down type dependency analysis*, and *bottom-up object instantiation*.

A. Input Space Partitioning

To systematically generate inputs for a given API under test with the goal of covering all branches and triggering exceptional behaviors efficiently, a thorough understanding of the input search space is crucial.

- *Semantic level*. Inputs often embody concepts within specific domains (*e.g.,* exponentiation), reflecting background knowledge. [24], [25] This semantic-level understanding imposes constraints on input values, effectively narrowing and categorizing the input space into distinct partitions. For example, for the parameter `z` of the `pow` function in Listing 1, a valid value should contain two numbers representing the real part and the imaginary part, respectively.
- *Code level*. When implementing a functionality, the specific implementation is contingent on factors such as project-specific logic, code optimization strategies, and others. Consequently, the input space is further restricted and partitioned at code level. For example, `pow` incorporates a special branch for the parameter `z` to handle the case where `z` represents a positive real number.

We have designed a prompt to harness the capacities of LLMs at the semantic and code levels. The prompt is illustrated in Figure 2.

- *System Instruction*. We highlight "input space partitioning" as the task we expect the LLM to accomplish.
- *Few-shot CoT Examples*.
  - *Question*. We include only the source code of the API under test. We emphasize that the source of the API under test encapsulates both semantic-level and code-level knowledge to understand the input space of its parameters. In addition, we also implement a LISP variant, described in Section IV-A, which includes the called

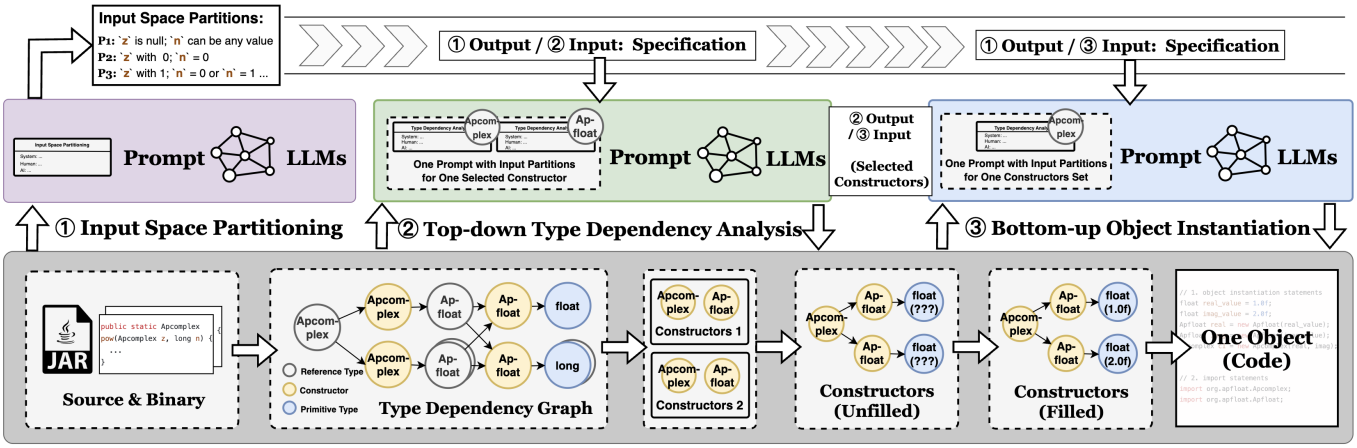

Fig. 1. Approach Overview of LISP

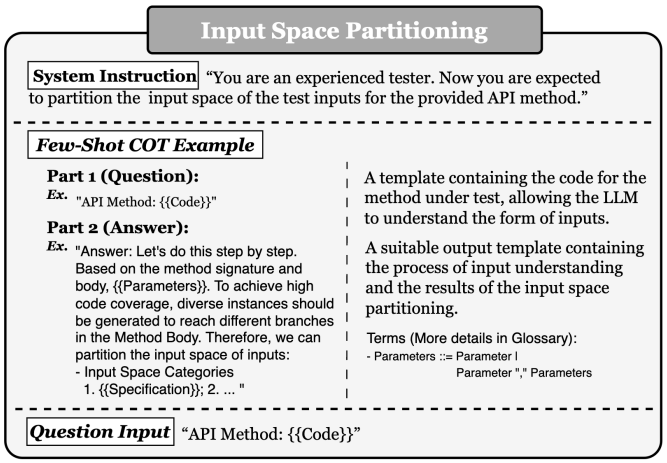

Fig. 2. The prompt for input space partitioning

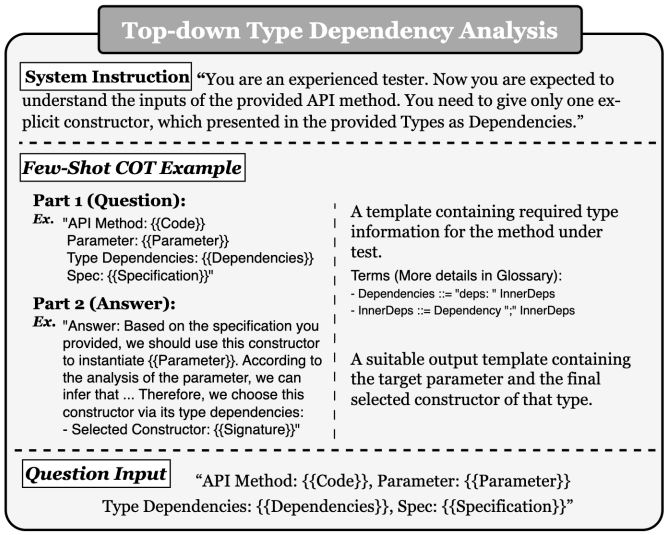

Fig. 3. The prompt for top-down type dependency analysis

methods based on the call graph, in order to provide more contexts. This variant is called LISP-CG.

– *Answer*. We construct a chain of thought that analyzes the code in the order of method signature, body, and parameters. We expect the LLM to understand the code under our guidance and provide partitioning results for achieving high coverage.

For this part, the input is the source *Code* (Table I) of the API under test, and the output is a collection of textual descriptions of the input space partitions, referred to as *Specification*s (Table I).

Example. For the `pow` API presented in Listing 1, LISP can produce 6 partitions of the input space. These partitions are represented in textual form, *e.g.,* "z: real part is non-negative and imaginary part is 0; w: is an Apcomplex number", which covered line 10-14.

B. Top-down Type Dependency Analysis

The type of each parameter in the API under test typically can be classified into the primitive type and the reference type. For the primitive type, we can create them directly with language-specific syntax. However, for the reference type, creating an object is not trivial, and the following two categories of issues arise simultaneously. (1) *Nested Reference Types*. In various common OOP languages, the reference type often involve multiple levels of nesting, which means that constructing an object of a reference type may require multiple calls to constructors. (2) *Multiple Constructor Candidates*. Since a type tends to own multiple constructors, different constructors often yield different construction results.

For the first issue, we construct a "*Type Dependency Graph*" (TDG). In detail, we abstract each reference type as a "node" and view the usage of each reference type during the instantiation of an object as an "edge", and select all reachable types derived from the types of parameters in the directed acyclic graph.

For the second issue, we engage in an interaction with the LLM to select the most appropriate constructor, based

TABLE I
GLOSSARY OF KEYWORDS IN PROMPTS

No	Keyword	Description	Example
1	Code	The source code of the API under test.	"public static Apfloat pow(Apcomplex z, Apcomplex w) { ... }"
2	Type	The fully-qualified name of a type.	"org.apfloat.Apfloat", "org.apfloat.Apcomplex"
3	Parameter	The parameter list of the API method under test.	"Apcomplex z", "Apcomplex w"
4	Constructor	The constructor of a *Type*.	"Apcomplex(Apfloat real, Apfloat imag)"
5	Dependency	The "is-a" and "has-a" relationships between two *Type*s. (represented as text)	"class org.apfloat.Apfloat: Constructors: public Apfloat(float value)"
6	Specification	The constraints on the input space partition.	"z with 1; n = 0 or n = 1"

on the text description of the input space partition (*i.e., Specification*). We design a prompt to drive LLMs to select the most appropriate constructor for each type, along the top-down process. The prompt is illustrated in Figure 3.

- *System Instruction*. We highlight "constructor selection" as the task and expect that the LLM can employ the *Specification* to select only one *Constructor* (Table I) for each type.
- *Few-shot COT Examples*.
  - *Question*. For each parameter in the API under test, we provide a list of *Constructor*s for each type and attach the corresponding *Specification* along with the dependency information of the parameter type recorded in the TDG.
  - *Answer*. We expects the LLM not only to take all provided information into consideration, but also to select only one constructor for each type.

For this part, the inputs are the *Code* of the API under test, and the *Specification*s, while the output is a mappings between *Parameter* (Table I) and its corresponding *Constructor* (Table I) sequence used for instantiation.

Examples. For the constructors presented in Listing 2, LISP can output the selected constructors like the upper half of Listing 3. Specifically, LISP first selects the first constructor for the `Apcomplex` type parameter in the API under test, and then selects the first constructor of `Apfloat` for both "`Apfloat real`" and "`Apfloat imag`", according to one of partitions outputted by input space partitioning. We break down the type dependency analysis task through the TDG into multiple sub-tasks, which increases the number of interactions with the LLM, but reduces the token of a single prompt, which avoids exceeding the token limit and also helps the LLM focus on selecting the appropriate constructor for a single type.

C. Bottom-up Object Instantiation

The ultimate goal of LISP is to generate high-quality inputs. We need to fill in appropriate values into the selected constructors and obtain instantiation statements through interaction with LLMs. We design a prompt to drive LLMs to fill the appropriate values into the selected constructor. The prompt is illustrated in Figure 4.

- *System Instruction*. We highlight two parts of statements that the LLM is supposed to provide (1) the "instantiation statements" about the target inputs, and (2) the "import statements" related to instantiation.

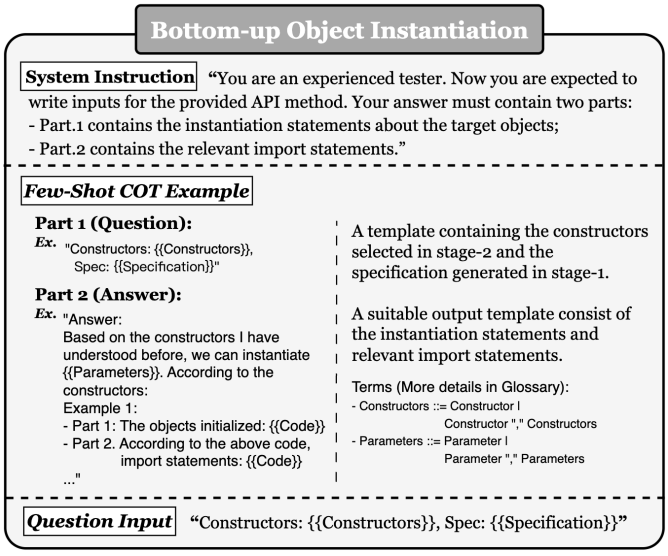

Fig. 4. The prompt for bottom-up object instantiation

- *Few-shot COT Examples*.
  - *Question*. We provide all selected *Constructor*s and the *Specification* to assist the LLM in filling in the appropriate values. Then, we consider the objects instantiated in this way as arguments.
  - *Answer*. We construct a chain of thought and expect the LLM to synthesize the instantiate statements and the relevant import statements.

For this part, the inputs are *Specification*s and selected *Constructor*s, while the outputs are statements that can be used in object instantiation.

Examples. For the selected constructors presented in the upper half of Listing 3, if the specification represents $1 + 0i$, LISP can fill "`1.0f`" and "`0.0f`" into selected constructors and finally generate instantiation statements like in the lower half of Listing 3.

Test Driver Generation. After interacting with the LLM and extracting the statements for constructor invocation, we encapsulates these statements with the necessary class and method declarations (*i.e.,* `class Driver` and `void main(...)`). The generated driver is expected to instantiate objects and invoke the API under test. This process results in the creation of an executable program. The driver template is available [13].

IV. Evaluation

We conduct extensive experiments to evaluate LISP with the following research questions.

- **RQ1 (Code Coverage)**. To what extent can LISP cover the code? Can LISP outperform the state-of-the-art test tools, EvoSuite and SPF, in terms of code coverage?
- **RQ2 (Usefulness)**. Can LISP trigger exceptions? (We only focus on unhandled exceptions and errors, both of which implement `java.lang.Throwable` but used in different scenarios) Can LISP find vulnerabilities previously not discovered?
- **RQ3 (Cost)**. Can LISP outperform EvoSuite in terms of time, while keeping the token consumption within a reasonable range?
- **RQ4 (Ablation Study)**. Are input space partitioning and top-down type dependency analysis of LISP both effective? How do they contribute?

A. Evaluation Setup

Experiment Subjects. To evaluate LISP, we selected 10 Java libraries from previous studies [26], [27] and some awesome lists (*i.e.,* awesome-java, useful-java-links), with the requirement that each selected library is highly starred and has recent code commits. All experimental data and results are available on our site [13].

TABLE II
DETAILS OF 10 JAVA LIBRARIES SELECTED.
$\#LOC$ = THE NUMBER OF LINE OF CODE OF THE LIBRARY;
$\#Stars$ = THE NUMBER OF STARS OF THE GITHUB REPOSITORY;
$\#APIs$ = THE NUMBER OF SELECTED API METHODS;

$\#No$	Library Name	Version	$\#LOC$	$\#Stars$	$\#APIs$
1	commons-lang3	3.13.0	85.6k	2.6k	545
2	guava	32.1.2-jre	163.7k	48.8k	195
3	jfreechart	1.5.4	214.1k	1.1k	169
4	jgrapht	1.5.2	89.8k	2.5k	131
5	joda-time	2.12.5	72.2k	4.9k	185
6	threeten	1.6.8	51.9k	546	106
7	time4j-base	5.9.1	74.3k	407	70
8	iCal4j	4.0.0-rc3	66.1k	705	201
9	SIS-Utility	1.4	783.8k	94	505
10	XChart	3.8.7	36.9k	1.5k	98

We have obtained 2,205 API methods, employing the following strategies for method selection to improve the quality of our datasets.

1) Exclude methods within abstract classes or interfaces, since the classes or interfaces cannot be instantiated directly.
2) Exclude methods that only have one basic block, since 100% edge coverage is guaranteed and meaningless.
3) Exclude methods inherited from `class Object` (*e.g.,* `equals`, `toString`, `hashCode`).

Implementation. To demonstrate the feasibility of LISP, we have implemented it in Java. Specifically, we utilize *JDT* [28] and *Soot* [29] to obtain AST, class hierarchy and call graph. We employ *langchain* [30] to interact with LLMs. It is important to note that while the implementation is specific to Java, the underlying concept of LISP can be applied to common OOP languages and automated testing frameworks in a more general scene.

Baselines & Variant. We have chosen two baselines, in order to better conduct various experiments.

- Search-based baseline. EvoSuite [3], a state-of-the-art tool in the field of SBST, is still actively maintained and has continuously been incorporating new SBST optimization algorithms since its release. It is widely used in both academia and industry. We choose the latest version released in 2021, Evosuite-v1.2.0, and refer to the time budget used in previous studies. [31].
- LLM-based baseline. We only provide the signature and code of the library API under test and expect the LLM to output the same format of results of LISP directly.
- LISP-CG (LISP with Call Graph). A variant that first obtains the call graph of the target library, and then includes the source code of those methods called within the API under test when constructing the prompt, in order to investigate the impact of "the source code of the called method" as mentioned in Section III. We include the source code of only one layer of methods invoked by the API under test. LISP never exceed the token limits during our evaluation, even though there are no prompt trimming or compression tricks in LISP. The prompt of LISP-CG is available at our site [13].
- Symbolic-based baseline. We have tried SPF [9] with lazy initialization, whose performance and scalability are among the best. We initially run SPF on 35 APIs, but only 4 are able to run. The remaining 31 APIs suffer from issues such as path explosion, insufficient support for collections, arrays and interfaces, etc. As a result, we abandon the comparison with symbolic execution tools.

Metrics. We evaluate LISP and baselines based on branch coverage and the number of found exceptions. Specifically, we adopt the branch coverage collection module used in EvoSuite to record coverage during each execution. For exception detection, we employ the same module in JQF [32] to record detected exceptions.

Identifying False Positives. Unlike system testing, API testing may generate false positives. These false positives occur when the generated inputs violate the assumptions of the APIs, leading to exceptions. The API assumptions are typically specified in Javadoc comments. For instance, as shown in Figure 5, API `intArrayToLong` from library commons-lang3 assumes their parameters need to meet `srcPos + nInts > src.length` constraint. During testing, inputs that do not meet such constraints can be generated, resulting in false positives.

We identify such false positives based on a convention used in the Java API specification. Specifically, when an API assumption is violated, the type of exception thrown is often specified in the Javadoc comments or the signature of the API.

1) Javadoc Exceptions. As shown in Figure 5, exception `ArrayIndexOutOfBoundsException` for constraint

TABLE III
DETAILS OF RESULTS IN RQ1.

Metrics	Indicators	Libraries										Overall
		commons-lang3	JFreeChart	JGraphT	guava	joda-time	threeten	time4j	iCal4j	SIS-Utility	XChart	
$\#Input$	LISP	3,409	694	772	1,191	853	505	399	1,033	2,814	503	12,173
	EvoSuite-100s	9,925	2,902	2,342	3,596	3,068	1,776	1,397	3,723	8,813	1,767	39,309
	EvoSuite-150s	14,343	4,155	3,226	5,164	4,645	2,743	1,756	5,209	13,031	2,545	56,817
	EvoSuite-200s	18,628	5,326	4,392	6,597	6,173	3,487	2,288	6,691	16,919	3,289	73,790
	LLM-baseline	1,328	162	123	474	301	190	67	279	1087	130	4,141
	LISP-CG	5,104	663	890	1,811	906	624	342	1,846	3357	618	16,161
$\#Edge$	LISP	7,443	1,135	1,959	1,702	1,407	586	647	757	3,227	237	19,100
	EvoSuite-100s	5,678	1,041	1,569	1,490	476	471	374	673	3,278	183	15,233
	EvoSuite-150s	5,724	1,065	1,754	1,571	460	481	353	626	3,358	179	15,571
	EvoSuite-200s	5,734	1,120	1,838	1,657	527	518	353	532	3,252	194	15,725
	LLM-baseline	4,255	386	570	1,041	829	339	213	417	2,336	93	10,489
	LISP-CG	7,977	1,235	1,864	1,734	1,350	655	749	756	3,224	221	19,765
$\frac{\#Edge}{\#Input}$	LISP	2.183	1.635	2.538	1.429	1.652	1.160	1.622	0.733	1.147	0.471	1.569
	EvoSuite-100s	0.572	0.359	0.670	0.414	0.155	0.265	0.268	0.181	0.372	0.104	0.388
	EvoSuite-150s	0.399	0.256	0.544	0.304	0.099	0.175	0.201	0.120	0.258	0.070	0.274
	EvoSuite-200s	0.308	0.210	0.418	0.251	0.085	0.149	0.154	0.080	0.192	0.059	0.213
	LLM-baseline	3.204	2.383	4.634	2.1962	2.754	1.784	3.179	1.530	2.149	0.715	2.533
	LISP-CG	1.563	1.863	2.094	0.957	1.490	1.050	2.190	0.410	0.959	0.358	1.223
$\#Time$	LISP	19,915	6,799	4,667	5,399	6,694	3,724	2,360	7,234	13,293	3,344	73,429
	EvoSuite-100s	54,500	16,900	13,100	19,500	18,500	10,600	7,000	20,100	50,500	9,800	220,500
	EvoSuite-150s	81,750	25,350	19,650	29,250	27,750	15,900	10,500	30,150	75,750	14,700	330,750
	EvoSuite-200s	109,000	33,800	26,200	39,000	37,000	21,200	14,000	40,200	101,000	19,600	441,000
	LLM-baseline	5,565	2,214	1,431	1,919	2,379	1,809	1,148	3,566	10,520	1,409	31,960
	LISP-CG	20,703	6,621	4,280	7,847	6,264	3,626	2,059	7,906	13,356	4,192	76,854
$\frac{\#Time}{\#Input}$	LISP	5.84	9.80	6.05	4.53	7.85	7.37	5.91	7.00	4.72	6.65	6.03
	EvoSuite-100s	5.49	5.82	5.59	5.42	6.03	5.97	5.01	5.40	5.73	5.55	5.61
	EvoSuite-150s	5.70	6.10	6.09	5.66	5.97	5.80	5.98	5.79	5.81	5.78	5.82
	EvoSuite-200s	5.85	6.35	5.97	5.91	5.99	6.08	6.12	6.01	5.97	5.96	5.98
	LLM-baseline	4.19	13.67	11.63	4.05	7.90	9.52	17.13	12.78	9.68	10.83	7.72
	LISP-CG	4.06	9.99	4.80	4.33	6.91	5.81	6.02	4.28	3.98	6.78	4.76

```
/** ...
 * @throws IllegalArgumentException if {(n-1)*32+dstPos >= 64}
 * @throws ArrayIndexOutOfBoundsException if {srcPos+n > src.length}
 */
 public static long intArrayToLong(final int[] src, final int srcPos,
  final long dstInit, final int dstPos, final int n) { ... }

/** ...
 * @param values the text containing the values to parse, or {null}.
 *               All non-null segments must be parseable as {double}.
 */
 public static double[] parseDoubles(final CharSequence values,
  final char separator) throws NumberFormatException { ... }
```

Fig. 5. False Positive Exception Examples (Javadoc & Signature)

`srcPos+n > src.length` is specified in the Javadoc comments.

2) Signature Exceptions. As shown in Figure 5, exception `NumberFormatException` is specified in its signature.

In our experiments, we take a conservative approach by filtering out all exceptions related to API assumption violations during result reporting. Specifically, for each API under test, we use Soot [29] to extract exceptions declared in Javadoc comments and signatures, and exclude these exceptions from the collected data during testing. The related code can be found in our artifact [13].

Evaluation Environment. Our experiments run on a 64-bit Linux machine (Ubuntu-22.04) with a 1.8GHz 8-Core AMD Ryzen 7 5700U CPU and 16GB RAM and use an OpenAI API key with 500 RPM to run all experiments. We use *gpt-3.5-turbo* with a token limit of 16K. To make the output more consistent, we set the temperature to 0.

B. RQ1: Code Coverage

Test inputs with higher code coverage are usually indicative of more comprehensive execution across the API functionalities. In addition, it is critical to generate high-quality inputs stably for API testing. In this study, we evaluate LISP from three dimensions, (1) average code coverage (for individual libraries and overall average), (2) quality (indicated by the coverage improvement caused by inputs), and (3) efficiency (indicated by the speed of generating valid inputs).

Table III presents the results of our experiment: (1) #Input. (i) For the LISP and LLM-baseline rows, each number represents the total number of valid inputs generated by the experiment. These inputs are generated by the tool for each API in the corresponding Java library executed once. (ii) For the EvoSuite-Xs rows, each number represents the total number of inputs generated by running each API for X seconds using EvoSuite. (2) #Edge. In Section IV-A, for the `LISP`

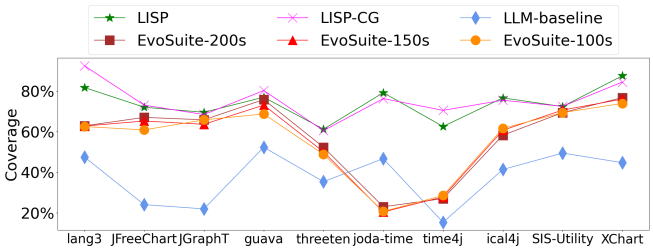

Fig. 6. Average code coverage of the 10 selected libraries.

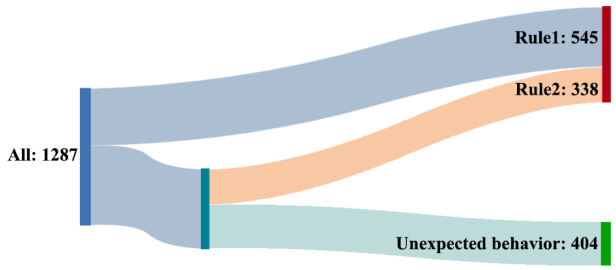

Fig. 7. A sankey diagram illustrating the filtering process that utilizes "Javadoc Exceptions" (Rule 1) and "Signature Exceptions" (Rule 2) to handle all captured exceptions and errors.

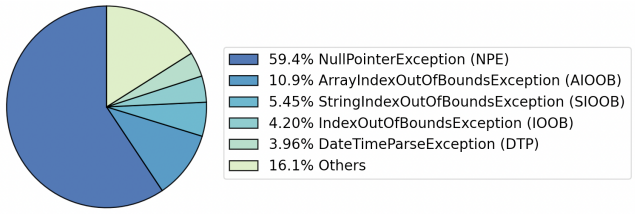

Fig. 8. Top 5 exception types found by LISP.

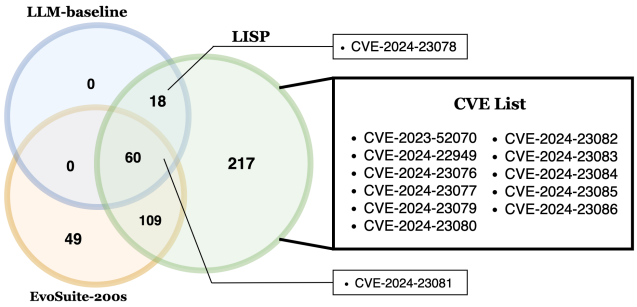

Fig. 9. A Venn diagram representing the distribution of exceptions found by LISP, *EvoSuite-200s*, LLM-baseline as well as the details of CVEs

and `LLM-baseline` rows, each number represents the total number of distinct edges covered by running each API just once in the corresponding library, which is also identical to the numerator of the "average code coverage" metric. (3) #Time. Each number represents the total time to generate inputs for all APIs in the corresponding Java library.

Result & Analysis. (1) Average code coverage. As shown in Figure 6 and Table III, in the selected 10 libraries, LISP overall outperforms both baselines, with an average code coverage that is 1.25 times that of EvoSuite-100s, 1.22 times that of EvoSuite-150s, and 1.21 times that of EvoSuite-200s. LISP-CG achieves similar code coverage with much more inputs. In addition, as shown in Table V, LISP-CG consumes much more tokens. (2) Quality. As shown in Table III, in terms of coverage improvement caused by inputs, LISP still overall outperforms both baselines, with fewer inputs but highest edge coverage, whose code coverage improvement per input is 4.04 times that of EvoSuite-100s, 5.73 times that of EvoSuite-150s, and 7.37 times that of EvoSuite-200s. LLM-based baseline generates the smallest number of inputs and attained less edges than LISP. (3) Efficiency. As shown in Table III, considering the time efficiency for generating valid inputs, EvoSuite outperforms LISP because EvoSuite uses search-based algorithms, making it easier to generate valid inputs. However, the efficiency of LLM-based variants is not unacceptable.

Summaries. (1) In terms of code coverage, LISP outperforms both search-based baseline (*i.e.,* EvoSuite) and LLM-based baseline; (2) In terms of quality, LISP outperforms both baselines and achieves the highest coverage improvement per input. (3) In terms of efficiency, LISP outperforms all EvoSuite variants, but LLM-based approach exhibited less time cost, because LISP requires more interaction with LLMs.

C. RQ2: Usefulness

The ability to find software vulnerabilities is one of the most effective criterion for judging an automated testing tool. In this study, we mainly concern three aspects, (1) statistics (indicated by categories and count of exceptions), (2) differences (indicated by the diversity in exceptions triggered by LISP and baselines), and (3) vulnerabilities (indicated by findings).

Result. (1) Statistics. LISP captures a total of 1287 Java exceptions or errors. As shown in Figure 7, after applying the two filtering rules outlined in Section IV-A wherein Rule 1 ("Javadoc Exceptions") filters out 545 of those and Rule 2 ("Signature Exceptions") subsequently filters out an additional 338, we finally obtain 404 exceptions, totaling 18 types. As shown in Figure 8, it is important to note that "NPE" accounts for the largest proportion, reaching 59.4%, followed by "index out of bound" is the second most common, accounting for a total of 35.75% ("AIOOB" (10.9%) + "SIOOB" (5.45%) + "IOOB" (4.20%)). We also record the exceptions captured by baselines, among which LLM-baseline captures 9 types of exceptions, a total of 78, while EvoSuite with a time budget of 200s (the best in EvoSuite) captures 13 types of exceptions, a total of 217 (but due to space constraints we no longer tabulate). (2) Differences. As depicted in Figure 9, LISP captures all the exceptions identified by LLM-baseline, which is expected given that LISP has extended the capabilities of LLM-baseline through prompt-engineering. In absolute terms, LISP only misses 49 exceptions that EvoSuite detects, while EvoSuite fails to identify 235 exceptions that LISP detects. (3) Vulnerabilities. We conduct a case-by-case study on exceptions found during our evaluation. Then, we identify vulnerabili-

ties and report them. As shown in Figure 9, 13 previously undiscovered vulnerabilities are identified as CVEs to date, all of which are detectable by LISP, with 11 of those being unique findings of our approach. Table IV shows that "NPE" still accounts for the largest proportion in identified CVEs. In addition, a "StackOverflowError" is identified as a CVE.

TABLE IV
THE TYPE AND ID OF FOUND CVES.

Categories	CVE-ID
NullPointerException	CVE-2024-22949 CVE-2024-23076 CVE-2024-23078 CVE-2024-23080 CVE-2024-23081 CVE-2024-23083 CVE-2024-23085
ArrayIndexOutOfBoundsException	CVE-2023-52070 CVE-2024-23077 CVE-2024-23079 CVE-2024-23084
StringIndexOutOfBoundsException	CVE-2024-23082
StackOverflowError	CVE-2024-23086

Case Study: CVE. To better illustrate the role of LISP in triggering exceptions and discovering vulnerabilities, we select one of CVEs that LISP found during our experiments (CVE-2024-23086). As shown in Listing 4, `modPow` is an instance method of `DoubleModMath`, used to calculate the result of "$a^n \mod m$", where m denotes the return value of `getModulus`. When $n = 0$, it naturally returns 1 directly. When $n < 0$, due to *Fermat's little theorem* [33], $a^m \equiv 1(\mod m)$ holds when m is prime, therefore $a^{m-1+n} = a^n(\mod m)$. In `modPow`, it employs recursive calls to gradually transform n into $m - 1 + n$, until $l * (m-1) + n > 0$, where l denotes the number of recursive layers. For power calculation, it is not wrong in mathematics. However, in the field of programming, the size of the stack is limited. If $m - 1$ is excessively small and n is a negative number with an extremely large absolute value (*e.g.,* $p = 2, n = -100,000.0$), too many recursive calls will lead to a stack overflow.

```
// DoubleModMath.java
public final double modPow(double a, double n) {
    if (n ==0) { return 1; }
    else if (n < 0) {
        return modPow(a, getModulus() - 1 + n);
    }
    // ignore some code
    return r;
}

// DoubleElementaryModMath.java
public final double getModulus() {
    return this.modulus;
}
private double modulus;
```

Listing 4. CVE-2024-23086: StackOverflowError due to recursive calls

```
public class TestModPow {
    @Test
    public void testModPow() {
        DoubleModMath dmm = new DoubleModMath();
        // assign "2" to modulus
        dmm.setModulus(2);
        // throw java.lang.StackOverflowError
        dmm.modPow(4, -1000000.0);
    }
}
```

Listing 5. POC of CVE-2024-23086

We attribute the generation of this input originates to LLMs' understanding of code and real-world knowledge. At the code level, LLMs recognize the significance of recursive calls when $n < 0$. At the conceptual-level, LLMs consider that when m is small, `modPow` demands a substantial number of recursive calls to enter the subsequent logic, by combining real-world knowledge from *Fermat's Little Theorem* with the possible reasons of stack overflow.

Case Study: LISP's miss. To better illustrate the limitations of LISP and explore how to further enhance the current LISP capabilities, we conduct a case-by-case analysis on the 49 exceptions that EvoSuite can trigger but LISP misses. Finally, we find that "index out of bound" accounts for 53.0%, while "NPE" accounts for 22.4%.

```
// Strings.java
public static String toString(
        final Class<?> classe,
        final Object... properties) {
    final StringBuilder buffer =
        new StringBuilder(32)
            .append(Classes.getShortName(classe))
            .append('[');
    // ignore some code
    for (int i=0; i<properties.length; i++) {
        final Object value = properties[++i];
        if (value != null) {
            // ignore some code
        }
    }
    return buffer.append(']').toString();
}
```

Listing 6. An exception that EvoSuite detected but LISP failed

As shown in Listing 6, `toString` is a static method that EvoSuite has successfully triggered an exception for, but LISP misses. This method takes a `Class` object and a "varargs" of `Object` as parameters. By reviewing the code, we find that if the number of arguments passed to the `properties` is not even, an `ArrayIndexOutOfBoundsException` will be triggered at Line 11. We have summarized two reasons why LISP fails to trigger this exception. (1) The current prompt design of LISP lacks special treatment for arrays, resulting in not good enough performance in detecting "index-out-of-bound" exceptions. (2) LISP generates fewer inputs and undergoes a certain randomness. In the future, we will further enhance LISP in these aspects.

Summaries. (1) In terms of statistics, LISP triggers 404 exceptions, a total of 18 types. In addition, LISP fully covers the exceptions triggered by LLM-baseline, while also triggering 77.5% of the exceptions triggered by search-based baseline (*i.e.,* EvoSuite-200s). In comparison, the search-based baseline only triggered 41.8% of the exceptions. (2) In terms of vulnerabilities, LISP identifies 13 previously undiscovered CVEs in total, and 11 of them are derived from the exceptions that both baselines fail to trigger.

TABLE V
DETAILS OF PARSING FAILURE, COMPILING FAILURE AND TOKEN CONSUMPTION IN RQ3.

Metrics	Indicators	Libraries										Overall
		commons-lang3	JFreeChart	JGraphT	guava	joda-time	threeten	time4j	iCal4j	SIS-Utility	XChart	
$\#API\ Failed$	LISP	2 / 545	4 / 195	1 / 169	1 / 131	5 / 185	5 / 106	9 / 70	0 / 201	2 / 505	0 / 98	29 / 2,205
	LISP-CG	3 / 545	7 / 195	6 / 169	2 / 131	7 / 185	7 / 106	4 / 70	0 / 201	3 / 505	0 / 98	39 / 2,205
	LLM-baseline	4 / 545	0 / 195	0 / 169	0 / 131	1 / 185	0 / 106	1 / 70	0 / 201	3 / 505	0 / 98	9 / 2,205
$\#Invalid\ Input\ (\%)$	LISP	42.04%	62.57%	57.21%	39.11%	49.91%	43.45%	40.27%	59.74%	50.96%	57.98%	49.82%
	LISP-CG	31.52%	51.32%	35.13%	32.45%	36.60%	41.08%	33.72%	35.41%	42.02%	45.69%	37.02%
	LLM-baseline	38.12%	81.05%	78.15%	29.40%	55.93%	61.54%	80.41%	50.18%	34.44%	49.42%	49.68%
$\#Token\ Input$	LISP	19,877,517	3,020,835	2,341,596	3,485,580	3,306,834	4,247,010	2,804,631	8,053,296	15,889,005	3,926,481	66,952,785
	LISP-CG	20,764,098	5,544,456	4,297,776	6,397,452	6,069,378	5,219,025	3,446,526	9,896,454	17,849,982	4,825,137	84,310,284
	LLM-baseline	2,931,549	909,048	704,646	1,048,902	995,112	570,174	376,530	1,964,694	4,936,167	957,909	15,394,731
$\#Token\ Output$	LISP	1,726,830	324,240	251,334	374,121	354,936	435,843	287,820	826,458	1,255,674	402,951	6,240,207
	LISP-CG	2,127,645	411,198	318,741	474,462	450,129	490,041	323,613	929,232	1,751,994	453,057	7,730,112
	LLM-baseline	540,330	167,553	129,879	193,329	183,414	105,093	69,399	353,664	888,558	172,434	2,803,653
$\#Cost$	LISP	12.529812	1.9969238	1.5479114	2.3041429	2.1859817	2.7774227	1.8341470	5.2666223	9.828611	2.5678059	42.8393807
	LISP-CG	13.574352	3.389278	2.627192	3.9107060	3.710157	3.344790	2.208823	6.3424792	11.553664	3.092353	53.7537942
	LLM-baseline	2.2764086	0.705896	0.547173	0.814495	0.772726	0.442751	0.292383	1.5129235	3.801126	0.737644	11.9035261

D. RQ3: Cost

Two main aspects of cost need to be considered when using LLMs to generate inputs. (1) Failures. How many APIs the LLM cannot correctly provide parsable answers due to hallucinations. Additionally, how much of the generated code is actually not executable. (2) Token consumption. Whether the token cost of interacting with the LLM is within an acceptable range.

Table V presents the results of experiments. (1) #API Failed. Each ratio represents "the number of APIs that failed to generate any inputs" / "the total number of APIs". (2) #Invalid Input. Each number represents the ratio to the total number of inputs generated cannot be run directly. (3) #Token Input and #Token Output. Each number represent the amount of tokens consumed. (4) #Cost. It is obtained according to "#Token Input" and "#Token Output", which is based on the OpenAI billing standard [34] in US dollars.

Result. (1) Failures. As shown in Table V, all the LLM-based variants demonstrate stable output under *gpt-3.5-turbo*. However, nearly 50% inputs generated by both LISP and the baseline cannot be run directly, while LISP-CG has around 10% lower failure rate. (2) Token consumption. The pricing of *gpt-3.5-turbo* we use is "US$0.50 per 1M input tokens" and "US$1.50 / 1M output tokens" [34]. As shown in Table V, after testing 2,205 APIs, LISP incurs a cost of $42.84, with 66.95M tokens as input and 6.24M tokens as output. Also, we run LISP-CG and LLM-baseline.

- For LISP-CG (LISP with deeper functions), on the input side, the token consumption increases significantly, over 80% on libraries like JFreeChart and guava. The overall input token consumption across the libraries increased by more than 20%. On the output side, the token consumption increases slightly, since the output formats are not changed.
- For LLM-baseline, both the input token and output token consumption decrease significantly, because the LLM-baseline refrains from frequently interacting with the LLM on the task of constructor selection.

Analysis. (1) The LISP and LISP-CG can yield parsing failures during the whole workflow depicted in Figure 1. In contrast, the LLM-based baseline only outputs the results directly, which reduces interactions and fewer parsing failures. (2) Actually, nearly 50% failure rate is acceptable [35]. The failure rate is stable across 10 libraries for LISP and LISP-CG. LISP-CG provides the LLM with more context and results in a lower failure rate. However, the baseline exhibits a large variance, possibly due to a lack of task decomposition for input generation.

Summaries. (1) In terms of failures, nearly half of the inputs generated by LISP are not runnable, but it is still acceptable [35]. (2) In terms of token consumption, LISP naturally consumes more tokens than the LLM-based baseline, but overall, it still remains within a reasonable range. Furthermore, LISP-CG consumes much more token than LISP, although it achieves more stable output and higher coverage.

E. RQ4: Ablation Study

In this study, we explore the role of each part within LISP and evaluate their contributions to the overall approach. In Section II, we have summarized the input object generation process and the importance of input space partitioning. Here, we design an ablation study that consists of three parts (no ISP+TDA since we need instantiation statements to generate input objects and test drivers).

- ISP+OI (without top-down type dependency analysis), a variant that cannot select the appropriate constructors step by step, and expects LLMs to generate inputs directly.
- TDA+OI (without input space partitioning), a variant that only simulates the process of input generation solely through top-down type dependency analysis and bottom-up object instantiation.

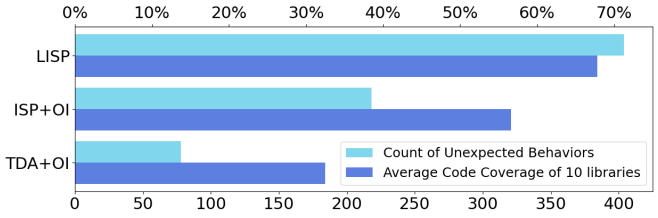

Fig. 10. The experiment results of ablation study.

Result & Analysis. As shown in Figure 10, we can see that LISP overall performs the best, followed by w/o TDA+OI and w/o ISP. (1) ISP+OI brings a certain decline (code coverage of ISP+OI is 56.60%, which is 83.46% of LISP). Based on the results of input space partitioning, the LLM still generates inputs purposefully. However, the absence of TDA in this variant contributes to the generation of invalid objects. (2) TDA+OI brings a significant decrease both in "average code coverage" and in "exceptions" (code coverage of TDA+OI is 32.52%, which is 47.95% of LISP). This further indicates that the LLM lacks essential directives in the process of constructor selection, due to the absence of input space partitioning.

Summaries. Input space partitioning and top-down type dependency analysis are both effective and contribute significantly to LISP. (1) Input space partitioning significantly improves the code coverage of the input objects generated by LLMs. (2) Top-down type dependency analysis assists LLMs in effectively understanding nested reference types and generating valid objects.

V. Limitations

(1) Interpretability challenges. Since the selected LLM used in Section IV is closed-source, we cannot provide a set of state-of-the-art prompts. (2) Complicated API interactions. Currently, LISP cannot generate a sequence of API calls to handle interactions between APIs. This is our future research direction. (3) Currently used drivers. The term "inputs" of a method should also include environment variables, system configurations, and more. Our drivers can merely generate arguments for the API under test. (4) Document-enhanced prompt engineering. Currently, we only include code comments when feeding API code into LLMs. We believe that it would be beneficial to integrate relevant documentation, and we plan to investigate retrieval-augmented generation (RAG) to achieve such an integration in future work.

VI. Related Work

A. Input Generation

Input object generation is a crucial component of automatic test-suite generation that has received significant attention from researchers. Over time, various techniques have been employed in the field of object-oriented input generation [3], [10], [31], [36], [37]. Gordon Fraser et al. developed *EvoSuite* [3], which is considered the state-of-the-art SBST tool. To further improve performance, other research projects promote the search-based approaches through advanced algorithms. [38], [39]. For instance, Yun Lin et al. developed *EvoObj* [31] that constructs an "object construction graph" via static analysis to generate a test seed template. Harrison Green et al. developed *GraphFuzz* [40] that mutates the "dataflow graph" to generate more test templates and unit tests.

B. Large Language Models

Present Large Language Models (LLMs) are typically developed through a two-step process [41]. Initially, they are trained on massive quantities of diverse text data, enabling them to capture the intricacies of language and acquire a wide range of knowledge [16]–[20]. Subsequently, these pre-trained models undergo a fine-tuning phase using additional datasets, further refining their understanding and text generation abilities, allowing them to possess extensive knowledge, language understanding and text generation capabilities [42]–[44]. In addition, they possess the capabilities to generate consistent and appropriate text results for various natural language processing tasks, such as text generation [20], [45], information extraction [46], etc.

Recently, LLMs are applied to various fields of secure software development life-cycle, including implementation [47], [48], maintenance [49], [50] and testing [12], [21], [22]. To interact with LLMs more efficiently [51], [52], prompts with task definitions and demonstrations are typically employed for better performance [42], [44], [53]. In the context of software testing, LLMs are often provided with zero-shot or few-shot prompts to synthesize input generators, method invocations and assertions [21], [51].

VII. Conclusion and Future Work

In this paper, we explore the potential of LLMs in the field of input space partitioning testing. Compared to the existing techniques, we have utilized the information in the code, which plays a crucial role in constructing high-quality inputs. Our experiments show that our approach achieves higher coverage, higher efficiency and stronger ability to find vulnerabilities.

In future work, we aim to build upon and extend these findings. We plan to address the sophisticated challenges of API testing, such as test generation involving multiple APIs and API testing in microservices. Furthermore, we will delve into the combination between software testing and LLM-related emerging technologies (*e.g.,* Agents), in order to explore the boundary of automated software testing. We hope that LLMs can enable the automated design and execution of test cases, the comprehensive analysis of results, and even the suggestion of improvements. In this manner, we will refine not only the efficacy and accuracy of automated tests but also their scalability across diverse and complex software systems.

Acknowledgements

We thank the anonymous reviewers for their insightful comments and suggestions. We also thank Yannic Noller for his valuable discussion and feedback on the symbolic execution tool SPF. This work was supported by National Key R&D Program of China (2023YFB4503805).

REFERENCES

[1] Y. Wang, M. Wen, Z. Liu, R. Wu, R. Wang, B. Yang, H. Yu, Z. Zhu, and S.-C. Cheung, "Do the dependency conflicts in my project matter?" in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 319–330. [Online]. Available: https://doi.org/10.1145/3236024.3236056

[2] R. Meng, Z. Dong, J. Li, I. Beschastnikh, and A. Roychoudhury, "Linear-time temporal logic guided greybox fuzzing," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1343–1355. [Online]. Available: https://doi.org/10.1145/3510003.3510082

[3] G. Fraser and A. Arcuri, "Evosuite: Automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 416–419. [Online]. Available: https://doi.org/10.1145/2025113.2025179

[4] Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury, "Time-travel testing of android apps," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 481–492. [Online]. Available: https://doi.org/10.1145/3377811.3380402

[5] X. Gao, S. H. Tan, Z. Dong, and A. Roychoudhury, "Android testing via synthetic symbolic execution," ser. ASE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 419–429. [Online]. Available: https://doi.org/10.1145/3238147.3238225

[6] J. Sun, T. Su, J. Li, Z. Dong, G. Pu, T. Xie, and Z. Su, "Understanding and finding system setting-related defects in android apps," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 204–215. [Online]. Available: https://doi.org/10.1145/3460319.3464806

[7] W. Guo, Z. Dong, L. Shen, W. Tian, T. Su, and X. Peng, "Detecting and fixing data loss issues in android apps," ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 605–616. [Online]. Available: https://doi.org/10.1145/3533767.3534402

[8] Y. Noller, R. Kersten, and C. S. Păsăreanu, "Badger: complexity analysis with fuzzing and symbolic execution," in *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, 2018, pp. 322–332.

[9] Y. Noller, C. Areanu, A. Fromherz, X. B. D Le, and W. Visser, "Symbolic pathfinder for sv-comp," 03 2019.

[10] P. Braione, G. Denaro, A. Mattavelli, and M. Pezzè, "Sushi: A test generator for programs with complex structured inputs," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 21–24. [Online]. Available: https://doi.org/10.1145/3183440.3183472

[11] Z. Yuan, J. Liu, Q. Zi, M. Liu, X. Peng, and Y. Lou, "Evaluating instruction-tuned large language models on code comprehension and generation," 2023.

[12] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models."

[13] "Artifacts of our work," 2024. [Online]. Available: https://github.com/FudanSELab/LISP

[14] Z. Zeng, H. Tan, H. Zhang, J. Li, Y. Zhang, and L. Zhang, "An extensive study on pre-trained models for program understanding and generation," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 39–51. [Online]. Available: https://doi.org/10.1145/3533767.3534390

[15] C. Niu, C. Li, V. Ng, D. Chen, J. Ge, and B. Luo, "An empirical comparison of pre-trained models of source code," 2023.

[16] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:160025533

[17] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, P. Schuh, K. Shi, S. Tsvyashchenko, J. Maynez, A. Rao, P. Barnes, Y. Tay, N. Shazeer, V. Prabhakaran, E. Reif, N. Du, B. Hutchinson, R. Pope, J. Bradbury, J. Austin, M. Isard, G. Gur-Ari, P. Yin, T. Duke, A. Levskaya, S. Ghemawat, S. Dev, H. Michalewski, X. Garcia, V. Misra, K. Robinson, L. Fedus, D. Zhou, D. Ippolito, D. Luan, H. Lim, B. Zoph, A. Spiridonov, R. Sepassi, D. Dohan, S. Agrawal, M. Omernick, A. M. Dai, T. S. Pillai, M. Pellat, A. Lewkowycz, E. Moreira, R. Child, O. Polozov, K. Lee, Z. Zhou, X. Wang, B. Saeta, M. Diaz, O. Firat, M. Catasta, J. Wei, K. Meier-Hellstern, D. Eck, J. Dean, S. Petrov, and N. Fiedel, "Palm: Scaling language modeling with pathways," 2022.

[18] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima, S. Presser, and C. Leahy, "The pile: An 800gb dataset of diverse text for language modeling," 2020.

[19] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton, "Program synthesis with large language models," 2021.

[20] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," 2023.

[21] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models," 2023.

[22] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang, "Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzgpt," 2023.

[23] Wikipedia contributors, "Complex number — Wikipedia, the free encyclopedia," 2023, [Online; accessed 15-December-2023]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Complex_number&oldid=1187479457

[24] F. Petroni, T. Rocktäschel, P. Lewis, A. Bakhtin, Y. Wu, A. H. Miller, and S. Riedel, "Language models as knowledge bases?" 2019.

[25] W. Huang, P. Abbeel, D. Pathak, and I. Mordatch, "Language models as zero-shot planners: Extracting actionable knowledge for embodied agents," in *International Conference on Machine Learning*. PMLR, 2022, pp. 9118–9147.

[26] A. Blasi, A. Gorla, M. D. Ernst, and M. Pezzè, "Call me maybe: Using nlp to automatically generate unit test cases respecting temporal constraints," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3551349.3556961

[27] Z. Xie, Y. Chen, C. Zhi, S. Deng, and J. Yin, "Chatunitest: a chatgpt-based automated unit test generation tool," 2023.

[28] E. Foundation, "Eclipse java development tools (jdt)," 2023. [Online]. Available: https://github.com/eclipse-jdt/eclipse.jdt.core

[29] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '99. IBM Press, 1999, p. 13.

[30] H. Chase, "LangChain," Oct. 2022. [Online]. Available: https://github.com/langchain-ai/langchain

[31] Y. Lin, Y. S. Ong, J. Sun, G. Fraser, and J. S. Dong, "Graph-based seed object synthesis for search-based unit testing," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1068–1080. [Online]. Available: https://doi.org/10.1145/3468264.3468619

[32] R. Padhye, C. Lemieux, and K. Sen, "Jqf: Coverage-guided property-based testing in java," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 398–401. [Online]. Available: https://doi.org/10.1145/3293882.3339002

[33] Wikipedia contributors, "Fermat's little theorem — Wikipedia, the free encyclopedia," 2024, [Online; accessed 22-March-2024]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Fermat%27s_little_theorem&oldid=1193612930

[34] OpenAI, "Openai api pricing," 2024. [Online]. Available: https://openai.com/api/pricing/

[35] N. Alshahwan, J. Chheda, A. Finegenova, B. Gokkaya, M. Harman, I. Harper, A. Marginean, S. Sengupta, and E. Wang, "Automated unit test improvement using large language models at meta," 2024. [Online]. Available: https://arxiv.org/abs/2402.09171

[36] A. Arcuri and X. Yao, "Search based software testing of object-oriented containers," *Information Sciences*, vol. 178, no. 15, pp. 3075–3095, 2008, nature Inspired Problem-Solving. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0020025507005609

[37] A. Sakti, G. Pesant, and Y.-G. Guéhéneuc, "Instance generator and problem representation to improve object oriented code coverage," *IEEE Transactions on Software Engineering*, vol. 41, no. 3, pp. 294–313, 2015.

[38] M. Harman and P. McMinn, "A theoretical and empirical study of search-based testing: Local, global, and hybrid search," *Software Engineering, IEEE Transactions on*, vol. 36, pp. 226 – 247, 05 2010.

[39] S. K. Gargari and M. R. Keyvanpour, "Sbst challenges from the perspective of the test techniques," in *2021 12th International Conference on Information and Knowledge Technology (IKT)*, 2021, pp. 119–123.

[40] H. Green and T. Avgerinos, "Graphfuzz: Library api fuzzing with lifetime-aware dataflow graphs," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1070–1081. [Online]. Available: https://doi.org/10.1145/3510003.3510228

[41] OpenAI, "Gpt-4 technical report," 2023.

[42] J. Wei, M. Bosma, V. Y. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai, and Q. V. Le, "Finetuned language models are zero-shot learners," 2022.

[43] H. W. Chung, L. Hou, S. Longpre, B. Zoph, Y. Tay, W. Fedus, Y. Li, X. Wang, M. Dehghani, S. Brahma, A. Webson, S. S. Gu, Z. Dai, M. Suzgun, X. Chen, A. Chowdhery, A. Castro-Ros, M. Pellat, K. Robinson, D. Valter, S. Narang, G. Mishra, A. Yu, V. Zhao, Y. Huang, A. Dai, H. Yu, S. Petrov, E. H. Chi, J. Dean, J. Devlin, A. Roberts, D. Zhou, Q. V. Le, and J. Wei, "Scaling instruction-finetuned language models," 2022.

[44] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "Llama: Open and efficient foundation language models," 2023.

[45] W. Jiao, W. Wang, J. tse Huang, X. Wang, S. Shi, and Z. Tu, "Is chatgpt a good translator? yes with gpt-4 as the engine," 2023.

[46] Y. Ma, Y. Cao, Y. Hong, and A. Sun, "Large language model is not a good few-shot information extractor, but a good reranker for hard samples!" 2023.

[47] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021.

[48] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W. tau Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," 2023.

[49] J. A. Prenner and R. Robbes, "Automatic program repair with openai's codex: Evaluating quixbugs," 2021.

[50] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery*, 2023.

[51] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020.

[52] L. Reynolds and K. McDonell, "Prompt programming for large language models: Beyond the few-shot paradigm," 2021.

[53] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," 2023.