# LLM-Based Invariant Testing for Software Functional Bugs

Ruogu Yang[‡§]
*Northeastern University*
yang.r4@northeastern.edu

Yifeng He[§]
*University of California, Davis*
yfhe@ucdavis.edu

Yundi Xu
*University of California, Davis*
xydxu@ucdavis.edu

Yuqing Wei
*Southern University of Science and Technology*
12311043@mail.sustech.edu.cn

Hao Chen
*The University of Hong Kong*
chenho@hku.hk

***Abstract*—Manually writing unit tests to uncover functional bugs in software libraries is not only time-consuming but also requires a deep understanding of the intended semantics of the APIs. Heuristic-based test generation methods suffer from low usability because they cannot reason about program semantics or interpret source code and documentation as humans do. Traditional fuzzing techniques like OSS-Fuzz often rely on crashes to detect bugs, but functional bugs do not always cause crashes. To overcome these limitations, we present LISA, a novel LLM-based invariant testing framework for software functional bugs. LISA iteratively generates API sequences and program invariants guided by API $n$-gram feedback, achieving higher bug-detection rates and competitive code coverage compared with both fuzzing and prior LLM-based test generation approaches, and reporting each finding as a high-confidence bug candidate for developer confirmation.**

***Index Terms*—Large language models, software testing, test generation**

## I. INTRODUCTION

Software defects (bugs) cause security vulnerabilities and unintended behaviors, and testing grows harder as software scales [1–3]. Among automated techniques, coverage-guided grey-box fuzzing is the most widely adopted [4–6]: it mutates inputs to explore deep paths for crashes and hangs, and has uncovered vulnerabilities in thousands of real-world projects [7].

However, fuzzing primarily targets *implementation* bugs that crash the program, which sanitizers can catch [4]. *Functional* bugs, by contrast, stem from incorrect logic and produce wrong results without crashing; they are hard for dynamic testing to detect and have been called "machine un-auditable" [8], as catching them requires domain knowledge of the API's *intended semantics* that current automated techniques lack.

Unit testing is the primary method developers use to identify functional bugs [9]. Unit testing requires developers to manually specify three elements: inputs, expected outputs, and the testing semantics that sequence the calls; we refer to the tested functions as the library's APIs and their invocation order as an API-sequence. Writing tests by hand is costly and often neglected [10, 11], while heuristic-based generators [12, 13] cannot reason about API source or documentation, generalize poorly to new libraries, and yield low-diversity, low-coverage tests [14, 15].

Prior automated approaches share common limitations. Coverage-guided fuzzers waste effort exploring shallow error-handling paths. Meanwhile, although several LLM-based unit test generation (LLM-UT) methods have been proposed recently [16–19], these approaches frequently produce one-shot tests with trivial oracles such as `assert(ptr != NULL)` [18, 19]. End-to-end coding agents suffer from search-space explosion when planning, invoking tools, and generating tests jointly [20].

Prior LLM-UT work has evolved from generating only assertions for simple functions [21] to producing complete test functions and full test files [16–18], and typically evaluates them by pass rate and coverage [22]. He et al. [18] note that this setup lacks a reliable oracle, the long-standing *oracle problem* [23–25]: when a generated test fails, a reader cannot tell whether the LLM's predicted input–output pair is wrong or the API is, so these methods cannot reliably distinguish incorrect oracles from genuine faults. We define LISA's invariant-based alternative and its scope in Section III-F.

To mitigate this oracle ambiguity, we propose decoupling LLM-UT into two independent components: 1) sequences of API calls representing the testing semantics, 2) assertions that verify execution results. To maximize code coverage in the generated API sequences, we follow prior work [12, 26] by using feedback to guide the LLM in selecting diverse API combinations and permuting their orders. Because we generate only API sequences rather than fuzzing drivers, as in PROMPTFUZZ [26], we introduce a novel $n$-gram API combination coverage as a guidance mechanism (Section III-E). To improve the reliability of the generated assertions, we relax the requirement for LLMs to predict exact input-output pairs and instead ask them to infer program invariants [8, 27]. We refer to this testing paradigm as invariant testing. To construct valid unit tests from these components, we propose chunk-invariant reasoning (Section III-G2), which partitions the finalized API sequence into multiple semantically coherent chunks and instructs the LLM to insert invariants at the end of

[‡]Work done while interning at UC Davis. [§]Equal contribution.
This is the authors' extended version of the paper accepted at ISSRE 2026.

each chunk as critical program points. In this setting, invariants serve as oracles that are weaker *as specifications* yet more robust *as oracles* than those required by formal verification (two distinct axes, elaborated in Section III-F), enabling us to check useful correctness properties without requiring precise input-output specifications. By decoupling API-sequence exploration from invariant construction, this two-stage design diversifies generated tests and improves their practicality. It achieves competitive code coverage and enables automated detection of functional bugs without precise input-output specifications. LISA provides an end-to-end automated pipeline for unit test generation. While finding unknown bugs is a significant benefit, the generated test suites are themselves valuable for regression testing. This invariant-based design mitigates, but does not fully solve, the oracle problem; accordingly, LISA reports *high-confidence bug candidates* that a developer confirms rather than acting as a fully autonomous bug detector.

We present LISA, a novel LLM-based invariant testing framework for software functional bug detection. LISA generates unit test functions iteratively, guided by $n$-gram API coverage, and inserts program invariants between API chunks to detect functional bugs in the target APIs. We evaluated LISA on 7 real-world C/C++ libraries. Compared with state-of-the-art fuzzing tools (OSS-Fuzz [7]), the unit tests generated by LISA achieve higher average branch coverage at the library level. Moreover, when evaluated on re-introduced historical functional bugs, LISA detects nine more bugs than the state-of-the-art LLM-based unit testing framework CITYWALK [19]. We make the following contributions:

- We present LISA, a novel LLM-based invariant testing framework that detects functional bugs in software libraries. To the best of our knowledge, LISA is the first to recast functional-bug detection as a *decoupled* two-stage problem, feedback-guided API-sequence synthesis followed by documentation-grounded invariant insertion at chunk boundaries, so that the contribution is the decomposition mechanism rather than the integration of existing components, enabling automated detection of functional bugs without precise input-output specifications.
- We introduce chunk-invariant reasoning and $n$-gram API coverage feedback to decouple API-sequence exploration from invariant construction, enabling accurate and diverse unit tests.
- We present LISA-BENCH, the first benchmark for evaluating functional bug detectors. We empirically validate LISA on LISA-BENCH, and its generated tests achieve higher average branch coverage than existing fuzzing approaches and detect more functional bugs than LLM-UT methods.

## II. BACKGROUND

### A. Bugs in Software Supply-Chain

Bugs in the software supply chain (*i.e.*, upstream libraries) are generally categorized into two classes:

- *Implementation bugs*: These occur when developers make mistakes in the code implementation, leading to incorrect handling of types, memory, or system resources. Implementation bugs typically exhibit universal rather than domain-specific patterns, such as software crashes or hangs. Owing to their universality, such bugs can often be detected by static code analysis tools with predefined rules [28], or by automated dynamic testing techniques targeting behaviors like crashes, such as fuzzing [4] and symbolic execution [29].
- *Functional bugs*: These arise when developers make logical errors in the program. When triggered, the API does not crash or terminate abnormally; instead, it produces incorrect outputs that deviate from the expected behavior described in the documentation. Detecting functional bugs in software libraries typically requires developers to manually craft expected input-output pairs as test cases [9], or to apply formal verification techniques [30] to prove correctness.

Due to their context-specific nature, testing for functional bugs is difficult to automate. Manually crafting high-quality test cases requires developers or domain experts to spend substantial time writing non-feature testing code. Moreover, there is no direct way to optimize these tests for exploring deep program states, except by investing additional effort in writing more tests. Formal verification also requires manual setup of theorem provers, and such tools are not as easily integrated into the software development lifecycle (*i.e.*, continuous development and continuous integration) as fuzzing and unit testing. A middle ground for detecting functional bugs automatically is property-based testing [31], which allows developers to specify the expected properties of APIs and then generates random (or even coverage-guided [32]) inputs to test these APIs against their specified properties. However, no fully automated testing approach yet exists that can detect functional bugs while remaining as easy to integrate into modern development pipelines as fuzzing tools for implementation bugs.

### B. Program Invariants

Program invariants are conditions in code that must hold for the program to proceed to the next stage of execution [34]. As the name suggests, these conditions remain invariant with respect to the program's state: for all internal states and variable configurations, they must be satisfied. In modern large-scale software systems and libraries, developers often *assert* such invariants at critical program points, *i.e.*, locations in the code where certain conditions must hold for the function to produce correct outputs.

Listing 1 shows an example implementation from the LLVM [33, 35] library. This example contains two critical program points, where developers use assertions to ensure the desired properties of the API. The first critical point occurs after initialization, where developers use invariants to verify that all variables are successfully initialized before proceeding to further optimization. The second occurs after compiler optimization, where developers ensure that the computed results satisfy necessary correctness properties. In general, if the appropriate program invariants hold at their corresponding critical program points, it is less likely that the program contains functional bugs.

```
template <class GroupT>
std::vector<Matcher *>
llvm::gi::optimizeRules(
    ArrayRef<Matcher *> Rules,
    std::vector<std::unique_ptr<Matcher>> &
        MatcherStorage)
{
  // setting up
  std::vector<Matcher *> OptRules;
  std::unique_ptr<GroupT> CurrentGroup = std::
      make_unique<GroupT>();

  // invariant to check initialization property
  assert(CurrentGroup->empty() && "Newly created group
       isn't empty!");
  unsigned NumGroups = 0;

  // core logic
  auto ProcessCurrentGroup = [&]() {...};
  for (Matcher *Rule : Rules) {...}
  ProcessCurrentGroup();

  LLVM_DEBUG(dbgs() << "NumGroups: " << NumGroups << "
       \n");
  (void)NumGroups;

  // invariant to check result of core logic
  assert(
    CurrentGroup->empty() && "The last group wasn't
        properly processed"
  );
  return OptRules;
}
```

Listing 1: Example of program invariants in LLVM [33].

```
void Purchase_succeeds_when_enough_inventory() {
    // arrange
    Store store; Store_Init(&store);
    Store_AddInventory(&store, Product_Shampoo, 10);
    Customer customer = {0};

    // act
    bool success = Customer_Purchase(&customer, &store
        , Product_Shampoo, 5);

    // assert
    assert(success == true);
    assert(Store_GetInventory(&store, Product_Shampoo)
        == 5);
}

void Purchase_fails_when_not_enough_inventory() {
    // arrange
    Store store; Store_Init(&store);
    Store_AddInventory(&store, Product_Shampoo, 10);
    Customer customer = {0};

    // act
    bool success = Customer_Purchase(&customer, &store
        , Product_Shampoo, 15);

    // assert
    assert(success == false);
    assert(Store_GetInventory(&store, Product_Shampoo)
        == 10);
}
```

Listing 2: Examples of AAA-style unit test functions [9].

### C. Unit Testing

Unit testing is the primary method developers use to assess functional correctness during software development [9]. A unit test provides a measure of *correctness* [9, 36], verifying that an API produces the expected results for given inputs. Otherwise, a logical error is present in the implementation. Because upstream libraries in the software supply chain are not standalone applications deployed in production environments, where testing often requires mocking, their unit tests typically follow the classical style: arrange, act, and assert (AAA) [9]. Previous work on constructing unit test generation datasets [17] has also adopted this paradigm.

Listing 2 shows two examples of classical AAA-style unit test functions. Both test functions begin with the *arrange* stage, where variables, memories, and objects are initialized. The tests then proceed to the *act* stage, where the APIs under test are invoked with specified inputs. Together, the arrange and act stages provide the *testing semantics*. The examples in Listing 2 demonstrate invocations of a single API for simplicity; however, in practice, the act stage often involves multiple APIs composed in sequence [12]. After invoking the APIs, the final *assert* stage verifies whether the results match the expected outputs. A failed assertion indicates the presence of a logical error in the code.

## III. Methodology

### A. Overview

LISA is a framework for automated unit test generation via LLM-based API reasoning, comprising two modules: API-sequence generation, which produces reasonable API invocation sequences for the library under test, and invariant insertion, which converts a structured API knowledge base into assertions inserted into the API-sequence. As depicted in Figure 1, LISA first extracts library API and type definitions (Section III-B), generates API sequences from selected APIs, project rules, and successful examples (Section III-C), repairs erroneous sequences (Section III-D), and iteratively refines them using API $n$-gram feedback [37] (Section III-E). It then partitions each sequence into semantically meaningful segments [38], converts knowledge-base information into invariants inserted per segment, and executes each segment to validate the invariants (repairing on failure), concatenating the validated segments into the final invariant-enriched test.

### B. API Information Extraction

We build this component on top of PROMPTFUZZ [26], a prior LLM-guided API fuzzing framework for C/C++ libraries. PROMPTFUZZ provides a practical front-end for library analysis and prompt construction, including Clang-based AST parsing and basic LLM invocation utilities, making it a suitable foundation for our pipeline. Importantly, LISA reuses only these front-end components for API extraction and prompt orchestration; we do not use PROMPTFUZZ's fuzzing loop, mutation strategy, or coverage-guided driver generation.

Following PROMPTFUZZ, we use the abstract syntax tree (AST) to extract function and struct definitions from the target library. LISA extracts structured API metadata by statically analyzing header files using Clang's AST dump facility, identifying function declarations, parameter and return types, and definitions of structs, enums, and typedef aliases by

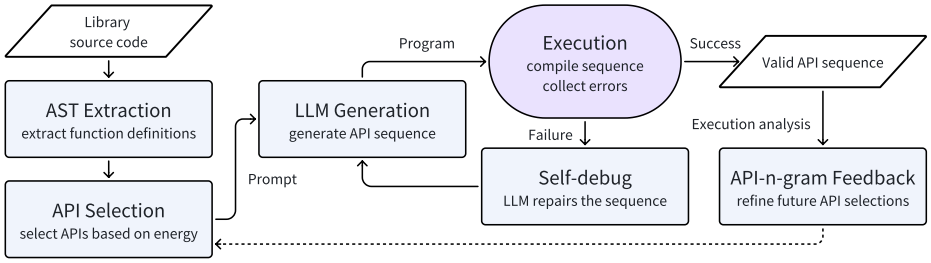


Fig. 1: Overview of LISA's iterative feedback-guided API sequence generation.

parsing the generated JSON-formatted AST. All types are then canonicalized into a standardized representation that preserves pointer mutability and array size information. This metadata is serialized into structured schemas that serve as the foundation for API-sequence generation, not fuzzing-driver synthesis.

### C. API Sequence Generation

LISA adopts a self-adaptive few-shot approach [39] to generate API sequences using a generative large language model (LLM), leveraging its capability to understand and produce valid code for open-source libraries. We use `gpt-5-mini` for its favorable cost and latency in this exploration-heavy phase, and switch to the stronger `gpt-5.1` for $\text{LISA}_{\text{inv}}$ (Section III-G), where invariant reasoning benefits from a more capable model. Because LISA's gains stem from its architecture rather than the base model, as our Vanilla-LLM ablation indicates (Table IV), these component gains are largely model-agnostic, and LISA can be paired with a stronger model when the budget allows. To generate syntactically and logically correct code, we construct a comprehensive prompt (Figure 5 in Section A) that restricts the search space using ground-truth `Library Context`, `Project Rules` (e.g., closing opened handles), and specific `API Combinations`, while `Code Requirements` enforce a straight-line execution policy. Augmented by `Successful Examples` and a structured `Execution Schema (Initialize` $\rightarrow$ `Cleanup)`, the prompt guides LISA to generate deterministic, high-quality API-sequence that reflect a complete usage lifecycle within the library's actual implementation boundaries.

### D. Error Detection and Repair

After generating the default number of API sequences, LISA attempts to identify and correct errors in these sequences in two steps: detection and repair.

*1) Detection:* LISA applies two filters to validate API-sequences. *Static* checks use `clang` to compile and link the program, rejecting any sequence that fails. *Dynamic* checks run the program in an isolated Docker sandbox with a 30-second timeout and catch segmentation faults, assertion failures, unhandled exceptions, and hangs. Recurring error patterns feed back into project-specific rules for later generation rounds; for `zlib`, for example, repeated segfaults from calls to `inflate()` on uninitialized streams caused LISA to add the rule "ensure the stream is initialized with `inflateInit()` before use," which the ablation study (Table V) shows is one of the recurring rules whose removal degrades execution success on stateful libraries. The `Successful Examples` slot in the prompt (Section III-C) is populated the same way: LISA retains sequences that compiled and executed in previous rounds and injects them as in-context exemplars for the next round, so the few-shot pool grows automatically.

*2) Repair:* Rather than discarding erroneous seeds, LISA performs automatic repair by invoking a large language model (LLM) with an error-guided prompt. When the generated API sequence fails either during compilation or execution, LISA extracts the corresponding error message, including the error type, code, and diagnostic details to construct a verbal feedback [40, 41]. These fields are substituted into our repair template (Figure 6 in Section A), which instructs the LLM to regenerate a fixed version of the same program without altering its overall logic or structure. The template explicitly restricts the model from redefining the `main` function, changing function names or parameters, or introducing control-flow constructs such as loops or branches. This constraint ensures the semantic equivalence between the repaired code and the original while correcting low-level implementation errors (e.g., undeclared variables, mismatched types, or segmentation faults).

If the error type indicates an execution failure without detailed diagnostics, LISA interprets it as a potential segmentation fault and prompts the LLM to strengthen pointer and memory safety. LISA recompiles and re-executes each repaired candidate to verify the fix. If the repaired sequence is still invalid, LISA abandons the seed to prevent unnecessary API invocations and computation overhead. In practice, this *sequence-repair* loop performs at most one repair attempt per erroneous program: a single LLM-guided fix recovers most compilation and runtime errors, whereas further attempts on a still-failing sequence rarely succeed and only add cost, so the seed is discarded rather than retried.

### E. API N-Gram Feedback

To statistically capture the local co-occurrence patterns of API calls in real-world programs, we adopt an *API-$n$-gram* model. Similar to $n$-gram models in natural language processing, an API-$n$-gram represents a contiguous subsequence of $N$ API invocations extracted from program traces or source code. For example, the sequence `{open, read, close}` constitutes an API-3-gram, while `{malloc, memcpy, free,`

TABLE I: Sensitivity to the $n$-gram order $N$ on `zlib` (3-hour budget). #Valid: successfully generated programs; coverage measured with `llvm-cov`.

| $N$ | #Valid | Line Cov. | Branch Cov. |
|---|---|---|---|
| 2 | 512 | 71.48% | 59.22% |
| **3** | **818** | **73.01%** | **60.01%** |
| 4 | 705 | 69.89% | 57.02% |

`printf}` forms an API-4-gram. By modeling which APIs co-occur and in what order, the API-$n$-gram serves as a lightweight, data-driven dependency model: it captures usage and ordering dependencies among APIs without constructing an explicit API-dependency or interaction graph, as used by dependency-aware approaches such as Hopper [42] and CITYWALK [19]. Integrating such richer dependency models to further improve sequence validity is a promising extension that we leave to future work. We set $N = 3$ for LISA. Table I reports a sensitivity analysis on `zlib` under an identical 3-hour budget: $N = 3$ attains the highest line and branch coverage and yields the most valid programs, whereas $N = 2$ under-constrains the sampled API combinations and $N = 4$ makes them too sparse to satisfy, lowering both validity and coverage. We therefore adopt $N = 3$ throughout; the full sweep is also available in our artifact.

*1) API Scheduling:* Let $E$ denote the energy of an API $a$. We assign a baseline energy to each API $a_i$ to initialize the feedback loop: $E(a_i) = 1$. When a successful program is generated, we extract all consecutive API 3-grams (triples) from its execution trace $\mathcal{T} = \{(a_i, a_{i+1}, a_{i+2}) \mid i = 1, 2, \ldots, L-2\}$, where $L$ is the length of the API call sequence. For each newly discovered 3-gram $(a_i, a_j, a_k) \in \mathcal{T}$ that has not been observed before, we update the energies of all three constituent APIs:

$$E(a) \leftarrow E(a) + 1, \text{for each } a \in (a_i, a_j, a_k) \tag{1}$$

This additive reward scheme ensures that APIs appearing in successful execution patterns accumulate higher energy over time, with each API starting from the same baseline value.

*2) Adaptive Condensed Normalization of Energy (*ACNE*):* We assign energy to successive $n$-grams in the API call sequence. However, assigning sampling probabilities proportional to raw energy skews the distribution heavily: as sampling progresses, a few high-energy $n$-grams dominate the probabilities and cause premature convergence, leaving other API functions under-explored. To encourage continued exploration of low-energy, potentially under-explored API functions, we propose ACNE, which dynamically adjusts energy values based on their distribution to balance exploration and exploitation.

*a) Normalization:* Recall from Equation 1 that our energy assignment is additive: each time a successful $n$-gram is observed, the energies of its constituent APIs are incremented by 1. The first step of ACNE normalizes the energy values into a probability-friendly range. Let $E_{\min}$ and $E_{\max}$ denote the minimum and maximum energy values among all API functions. We normalize the energy of a given API function $a$ with min-max normalization, and ensure every API function keeps a non-zero selection probability by adding a small constant $\varepsilon > 0$:

$$\hat{E}(a) = \varepsilon + (1-\varepsilon)\bar{E}(a), \quad \text{where } \bar{E}(a) = \frac{E(a) - E_{\min}}{E_{\max} - E_{\min}}. \tag{2}$$

We use $\varepsilon = 0.01$ in our implementation, so $\hat{E}(a) \in [\varepsilon, 1]$. At the first iteration every API shares the same energy ($E_{\min} = E_{\max}$); in this degenerate case we skip normalization and sample APIs uniformly.

**Definition III.1** (Condensation transformation)**.** *Let $I = [0, 1]$ denote the interval containing the (normalized) energies of all API functions. A function $f : I \to [0, \infty)$ is a* condense transformation *if it satisfies:*

1) *Strictly monotone increasing (order-preserving):* $\forall x, y \in I,\ x < y \implies f(x) < f(y)$.
2) *Strictly concave: for $\lambda \in (0, 1)$ and $\forall x, y \in I$ with $x \neq y$,* $f(\lambda x + (1-\lambda)y) > \lambda f(x) + (1-\lambda) f(y)$.
3) *Endpoint preserving:* $f(0) = 0$.

**Proposition 1** (Range compression)**.** *Any condense transformation $f$ compresses the range of input values: for any $x, y \in I$ with $0 < x < y \leq 1$,* $1 < {}^{f(y)}/_{f(x)} < {}^{y}/_{x}$.

We prove Proposition 1 in Section B-A.

*b) Condensation:* To mitigate the skewness of the energy distribution, we apply a condense transformation to the normalized energy values. Its properties ensure that 1) API functions with higher energy *always* have a higher probability of being selected by LISA, and 2) the marginal effect of increasing energy decreases as energy grows. A valid condense transformation always reduces the relative differences between energy values (Proposition 1). In LISA, we use *power condensation*: for a given API function $a$, with a parameter $\alpha$ controlling the degree of condensation,

$$C(a) = \hat{E}(a)^{\alpha}, \quad \alpha \in (0, 1). \tag{3}$$

We show that power condensation is a valid condense transformation in Section B-A.

*c) Distribution-aware adaptive condensation:* The parameter $\alpha$ controls the degree of condensation: a small $\alpha$ largely boosts the probabilities of low-energy API functions (encouraging exploration), whereas $\alpha$ close to 1 makes the effect mild (favoring exploitation). A fixed $\alpha$ is suboptimal because different stages of LISA require different exploration–exploitation trade-offs: early on we want to identify likely-useful API functions quickly (exploitation), and once such a set is identified and sufficiently tested we want to explore other potentially useful API functions (exploration). We quantify the skewness of the energy distribution at iteration $t$ with the coefficient of variation $s_t = {}^{\sigma_t}/_{\mu_t}$, the ratio of the standard deviation $\sigma_t$ to the mean $\mu_t$ of the normalized energies at iteration $t$, and adaptively set

$$\alpha_t = \alpha_{\min} + (1 - \alpha_{\min})\, e^{-s_t}. \tag{4}$$

Thus, when the distribution is highly skewed (large $s_t$), $\alpha_t$ is automatically lowered to encourage exploration. Here $\alpha_{\min}$

controls the maximum degree of condensation; we set $\alpha_{\min} = 0.5$ (derived in Section B-B).

*d) Sampling new API functions:* Finally, we use the condensed energy values to sample new API functions. At iteration $t$, the probability of selecting an API function $a$ is

$$P_t(a) = \frac{C_t(a)}{\sum_{i=1}^{N} C_t(a_i)}, \quad \text{where } C_t(a) = \hat{E}_t(a)^{\alpha_t}. \quad (5)$$

We use these probabilities to sample a subset of APIs as the `API Combinations` for API-sequence generation, thereby guiding the LLM's attention.

### F. The Invariants LISA Generates

An invariant in LISA is a Boolean predicate $\varphi$ over program state that must hold every time execution reaches a designated program point [8, 27, 34]. Concretely, for a generated API-sequence partitioned into chunks $C_1 \| \cdots \| C_N$ (Section III-G2), each invariant is a pair $\langle \varphi, \ell \rangle$ in which $\ell$ is the end of some chunk $C_i$ and $\varphi$ ranges over the variables, return values, and reachable struct fields that are live at $\ell$. LISA compiles each such $\varphi$ into an executable `assert(`$\varphi$`)`; the assertion is *satisfied* on a given build when it never fires and *violated* otherwise. An invariant is therefore a *partial* specification: it constrains a property the program must preserve, such as a size relation, a state flag, or a return-code contract, without pinning down the exact input–output mapping. That partiality is what makes invariants usable as oracles for functional bugs, sidestepping the input–output ambiguity long recognized as the obstacle to testing *non-testable* programs [24, 25].

The predicates LISA emits fall into four kinds of increasing semantic depth. *Structural* invariants assert that a handle or state object is well-formed after a constructor-like call. *Value-range* invariants keep a field within its documented domain. *Relational and conservation* invariants link live values across a call: the bytes a call consumes plus the bytes that remain equal the input size, for example. *Documentation-derived semantic contracts* encode behavioral intent that exact input–output oracles cannot easily express, such as the numeric identity of a checksum API or the guarantee that a destructor resets a state object. The first two kinds are largely structural; the last two carry the intent that lets LISA detect silent logic errors. LISA produces these predicates by prompting the LLM with the chunk code and the relevant API documentation from the knowledge base (Section III-G1), seeded by documentation-mined contracts and Daikon candidates; the algorithm and its verification-and-repair loop are the subject of Section III-G2. A documented *read-only* guarantee, for instance, becomes `assert(memcmp(buf, saved, n) == 0)` on the relevant input buffer.

A LISA invariant is *semantically valid* when it does not contradict the documented contract of the APIs in scope and holds on the reference build of the library, so that a later violation signals a behavioral deviation rather than an over-strong assertion. LISA enforces the first condition during knowledge-base construction (Section III-G1) and the second automatically, by executing each candidate against the reference build before retaining it. This is a deliberately weaker guarantee than soundness in formal verification: LISA does not prove $\varphi$ over all inputs, only that $\varphi$ is consistent with the documented contract and with observed reference behavior. In that sense LISA's invariants are weaker as specifications, since they are partial and unproven, yet more robust as oracles, since asserting a stable property is less brittle than predicting an exact output [18, 24, 25]. Because LISA does not prove its invariants, a violation is only a *high-confidence bug candidate*: an assertion that survives the verification-and-repair loop yet still fails on the target build, and a developer confirms every candidate before we count it as a detected bug. We revisit the residual threats this raises in Section VI-B.

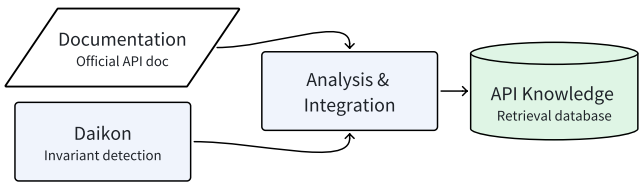


Fig. 2: API knowledge database preparation.

### G. Invariant Generation

Existing studies demonstrate that LLM-based invariant generation without structured guidance struggles to produce effective and reliable results [27]. To overcome these limitations, LISA first constructs an API contract documentation knowledge base to guide invariant generation, as shown in Figure 2 and detailed in Section III-G1.

As shown in Figure 3, LISA adopts a segment-based invariant generation strategy to incrementally augment API-level programs with assertions. The target program is first split into multiple segments. For the first segment, the LLM takes the raw code as input and outputs the code augmented with inferred assertions. For each subsequent segment, the input consists of the previously processed code and assertions, the current code segment, and relevant API documentation. The model then generates assertions for the current segment by leveraging both the program context and external knowledge. All processed segments are concatenated to produce the final unit tests enriched with invariants.

Notably, LLM infers likely invariants based on the constructed API contract documentation. When an execution failure occurs, LISA starts a repair loop and retries up to five times to eliminate transient test bugs, such as mistakes in the generated test code or invariants. If the failure persists after all retries, LISA marks it as a high-confidence bug candidate, since repeated failures are unlikely to be caused by random or temporary test errors. Nevertheless, because semantic reasoning is inherently complex, some persistent failures may still result from subtle test-side inaccuracies rather than actual implementation defects. These candidates are forwarded to human reviewers for final verification.

*1) Building API Knowledge Database:* We build the API contract knowledge base through a hybrid approach: 1) **Auto-**

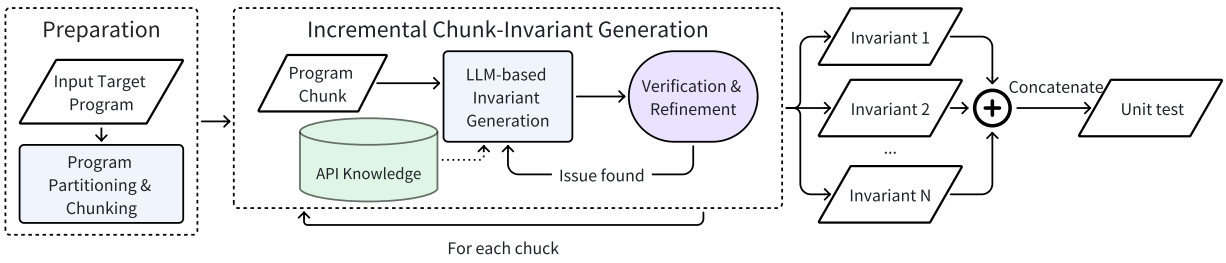


Fig. 3: Overview of LISA's incremental invariant generation pipeline.

**mated Extraction:** Daikon [43] mines *likely invariants* from executable API sequences, and an LLM extracts intended invariants from official API documentation [44] (e.g., "read-only" or "no internal state modified"). 2) **Manual Filtering:** We keep a Daikon candidate only if it does not contradict the documentation and does not cause assertion failures when added to the program. 3) **Lightweight Manual Curation:** We supplement explicitly documented invariants missed by Daikon. This curation can also be performed incrementally on demand, rather than exhaustively documenting all APIs upfront. To quantify this manual cost, two annotators with 3+ years of C/C++ experience, after aligning on the annotation protocol, independently curated the knowledge-base entries for 20 APIs, resolving the few disagreements on which invariants to retain by discussion. They spent 5687 and 6231 seconds in total, i.e., 4.7 and 5.2 minutes per API respectively (overall mean 5.0 minutes per API), indicating that on-demand curation costs only a few minutes per API. We detail how Daikon-mined and documentation-derived invariants are reconciled in Section E.

*2) Chunk-level Invariant Generation:* To keep reasoning local and stable, we partition each generated program into smaller chunks [45]: during API-sequence construction the LLM emits inline step markers (`// STEP1`, `// STEP2`, ...) that we use to split the program by regular expressions. We then process the $N$ chunks $C_1, \ldots, C_N$ left-to-right. For chunk $C_i$, the prompt contains the already-accepted code and assertions from $C_1 \ldots C_{i-1}$, the raw code of $C_i$, and the documentation for the APIs it uses; the LLM returns $C_i$ augmented with candidate assertions. This preserves prior state while keeping the context bounded.

*a) Verification-and-repair loop:* Each set of candidate assertions for $C_i$ is immediately checked: we compile and execute $C_1 \| \cdots \| C_i$ and accept the assertions if no failure occurs. When failures arise, we enter a bounded repair loop (the *invariant-repair* loop, up to $K{=}5$ attempts by default). In contrast to the single-attempt sequence-repair loop (Section III-D), a failing assertion is often salvageable by weakening or correcting an over-strong invariant, so a few extra attempts recover many otherwise-discarded tests; we cap at $K{=}5$ because gains plateau beyond that. At each attempt, the LLM receives the failing assertion, triggering input (if available), and error trace, and is asked to weaken, repair, or remove the assertion. If the failure persists after $K$ attempts, we flag the seed as a potential unknown bug for human inspection, discard the assertion, and proceed. After all chunks are processed, the accumulated program forms the final invariant-enriched test. Compared with one-shot inference over the whole program, this incremental scheme shortens each prompt, localizes mistakes, and enables early detection and targeted repair of over-strong assertions via immediate execution feedback.

## IV. Experimental Setup

### A. Research Questions

To evaluate LISA, we pose the following three research questions. For convenience, we refer to the API sequence generation stage as $\text{LISA}_{\text{API}}$ and the invariant generation stage as $\text{LISA}_{\text{inv}}$.

**RQ1** *How does* LISA *perform in C/C++ unit test generation, and what advantages does it offer compared to existing approaches?* We evaluate LISA's effectiveness in generating valid, meaningful C/C++ unit tests and compare it against state-of-the-art methods.

**RQ2** *How does each component impact the performance of* LISA*?* LISA consists of two stages, each comprising multiple components that contribute to its overall performance. We conduct ablation studies to understand the contribution of each component to LISA's overall effectiveness.

**RQ3** *How effective is* LISA *at detecting real-world functional bugs?* We evaluate LISA's effectiveness in detecting bugs. Specifically, we focus on previously reported functional bugs in real-world projects, as these cases have been manually verified and confirmed by actual developers.

*a) Baseline:* To address these questions, we compare LISA with a state-of-the-art unit test generation framework and relevant fuzzing techniques. 1) CITYWALK [19] is a recent C/C++ unit test generation framework that uses project-dependency awareness and language-specific knowledge to improve test reliability and quality. 2) OSS-Fuzz [7] is Google's continuous fuzzing platform for open-source software. It combines coverage-guided fuzzing with continuous integration to detect bugs and security vulnerabilities in widely used libraries. CITYWALK represents the state-of-the-art in LLM-based unit test generation, making it a natural benchmark for assessing LISA's ability to generate valid and high-quality unit tests. OSS-Fuzz serves as a widely adopted industrial standard

TABLE II: Benchmark for evaluation. LoC: line of code. #APIs: number of public APIs.

| Library | Commit ID | LoC | #APIs |
|---|---|---|---|
| **zlib** | `5a82f71` | 30k | 87 |
| **libpng** | `0f07f70` | 57k | 246 |
| **libpcap** | `d81c01c` | 58k | 84 |
| **sqlite3** | `7efded5` | 413k | 289 |
| **lcms** | `8888d84` | 45k | 286 |
| **cJSON** | `c859d25` | 10k | 76 |
| **re2** | `a4b2aee` | 28k | 70 |

for fuzzing-based bug detection, providing a strong baseline for evaluating LISA's effectiveness in code coverage and bug detection. For OSS-Fuzz, we use the official fuzzing harnesses from the OSS-Fuzz GitHub repository and run `libFuzzer` for 3 hours on the same library version (identical commit) as LISA. We measure branch and line coverage for all tools independently using `llvm-cov` with identical compiler flags, following Fuzzbench's recommendation [46] to use Clang source-based coverage rather than fuzzer-reported metrics. Section C records the remaining design and measurement details (harness handling, library-level coverage, and the comparison with HOPPER).

*b) Scope of baselines:* We considered agentic SE workflows (SWE-agent [47], OpenHands [48]) and recent LLM-UT methods (IntUT [49]) and excluded them for three reasons. *Language*: SWE-agent and OpenHands target Python [50] and IntUT targets Java; none supports C/C++ test generation, whereas LISA targets C/C++ library APIs. *Toolchain coupling*: their pipelines rely on language-specific infrastructure (Python execution environments; IntUT's Java static analysis for test-intention derivation), so a fair comparison would require us to re-implement each system for C/C++ rather than run an existing tool. *Task and oracle mismatch*: SWE-agent and OpenHands resolve a *given* issue by patching a known defect rather than discovering unknown functional bugs, whereas IntUT relies on the exact-output oracle that LISA explicitly avoids. Our Vanilla-LLM baseline and component ablations (Table IV, Table VI) address the orthogonal concern that LISA's gains might come from its iterative generate-and-repair loop rather than the invariant oracle: they disable LISA's components while keeping the same underlying LLM. We ground key parameters similarly: $N{=}3$ follows from the sensitivity analysis in Table I, and the maximum condensation factor $\alpha_{\min}{=}0.5$ follows from the analytical derivation of ACNE in Section III-E and Section B-B.

## B. LISA-BENCH

Existing benchmarks are insufficient for functional-bug-oriented test generation. Fuzzing benchmarks such as Magma [51] and FuzzBench [46] curate only crash-triggering bugs (e.g., overflows, use-after-free) that manifest as memory errors, not the silent logic errors functional bugs produce. LLM-UT benchmarks such as TestGenEval [22] rely on pass rate and coverage, which are insensitive to the *oracle problem*: an assertion-free suite trivially achieves high pass rate and coverage yet detects nothing. Neither family, therefore, evaluates whether generated tests can expose functional defects.

To fill this gap, we introduce LISA-BENCH, a benchmark specifically designed for evaluating functional-bug-oriented unit test generation for C/C++ libraries. We select 7 real-world open-source C/C++ projects from GitHub, five of which include historical bugs. The projects are selected according to four criteria: (1) the library exposes a public C/C++ API; (2) it builds and executes in our environment with reasonable effort; (3) it has sufficient functional complexity for invariant-based testing; and (4) historical bug reports or commits are available, enabling confirmed functional-bug cases. We also intentionally include projects from different application domains and codebase sizes to avoid overfitting the evaluation to a single library style. As shown in Table II, these projects span diverse application domains: data compression (zlib [52]), PNG image processing (libpng [53]), network packet capture (libpcap [54]), an embedded SQL database engine (sqlite3 [55]), color management (lcms [56]), structured-format parsing (cJSON [57]), and regular-expression matching (re2 [58]). Spanning small to very large code bases, these libraries let us curate historically reported, developer-confirmed functional bugs and directly measure whether generated tests detect real-world defects.

## C. Metrics

*1) Evaluating* $\text{LISA}_{API}$*:* We evaluate $\text{LISA}_{\text{API}}$ using four standard metrics [59, 60]: Compilation Success Rate (CSR) and Execution Success Rate (ESR) measure the fractions of generated API-sequences that compile and run without runtime errors, respectively; Line Coverage (LC) and Branch Coverage (BC) are measured with `llvm-cov`.

*2) Evaluating* $\text{LISA}_{inv}$*:* We evaluate $\text{LISA}_{\text{inv}}$ using two metrics that reflect both its ability to generate assertions and their quality.

*a) Average Unique Verifications Count (AUVC):* AUVC measures the expected number of valid, unique verifications produced per input seed. Let $\mathcal{T}_{total}$ be the valid seeds from $\text{LISA}_{\text{API}}$ and $\mathcal{T}_{pass} \subseteq \mathcal{T}_{total}$ those that still compile and execute after invariant insertion; for $t \in \mathcal{T}_{pass}$, let $U(t)$ be its number of unique verifications. Then AUVC $= \sum_{t\in\mathcal{T}_{pass}} U(t) \,/\, |\mathcal{T}_{total}|$. Test cases whose invariants are invalid or violate intended behavior are excluded from $\mathcal{T}_{pass}$ and thus contribute zero, penalizing failed generation. Uniqueness is semantic, not syntactic: two annotators with 3+ years of C/C++ experience label each assertion, treating repeated queries over an evolving stream state as distinct while collapsing syntactically different but equivalent predicates such as `x >= 1` and `x > 0` for $x \in \mathbb{Z}$. Disagreements are resolved by discussion to consensus (full protocol in Section D-A).

*b) Historical Bug Detection Rate (HBDR):* We measure fault detection via *historical bug re-introduction*: for five libraries, we revert five previously fixed bugs in each (25 bugs in total). A bug counts as detected when the generated test passes on the patched branch but fails on the buggy one.

TABLE III: Comparison of Compilation and Execution Success Rates between Vanilla LLM and LISA.

| Library | Vanilla LLM | | LISA | |
|---|---|---|---|---|
| | Comp. | Exec. | Comp. | Exec. |
| **libpcap** | 80.0% | 45.2% | **99.3%** | **90.4%** |
| **lcms** | 92.0% | 30.6% | **95.0%** | **77.5%** |
| **zlib** | 94.3% | 92.6% | **99.5%** | **98.6%** |
| **sqlite3** | 88.3% | 28.6% | **98.3%** | **91.5%** |
| **re2** | 96.4% | 73.0% | **100%** | **99.9%** |
| **cJSON** | 56.8% | 56.1% | **100%** | **99.4%** |
| **libpng** | 65.1% | 38.2% | **93.5%** | **79.2%** |
| **average** | 81.8% | 52.0% | **97.9%** | **90.9%** |

TABLE IV: Code Coverage Comparison: LISA vs. Vanilla LLM vs. OSS-Fuzz. All tools were run for 3 hours. **Bold** indicates the best performance.

| Library | Vanilla LLM | | OSS-Fuzz | | LISA | |
|---|---|---|---|---|---|---|
| | Line | Branch | Line | Branch | Line | Branch |
| **libpcap** | 21.06% | 24.60% | 25.79% | 27.51% | **27.48%** | **30.32%** |
| **lcms** | 30.28% | 20.82% | 44.74% | **34.44%** | **49.18%** | 27.59% |
| **zlib** | 53.32% | 41.43% | 65.99% | 52.81% | **73.01%** | **60.01%** |
| **sqlite3** | 14.40% | 10.54% | **35.38%** | **26.60%** | 31.06% | 22.46% |
| **re2** | 36.09% | 41.03% | 29.77% | 31.62% | **39.77%** | **46.54%** |
| **cJSON** | 51.01% | 47.53% | 42.28% | 45.52% | **76.80%** | **69.07%** |
| **libpng** | 13.01% | 8.92% | **22.57%** | **17.94%** | 20.21% | 14.92% |
| **average** | 31.31% | 27.84% | 38.07% | 33.78% | **45.36%** | **38.70%** |

To draw the five bugs for each library, we scan developer-fixed commits in reverse chronological order and keep the first five whose fix message flags a non-crashing functional defect and whose buggy revision reproduces deterministically on our toolchain; we skip commits that fail either filter and examine the next candidate. This protocol fixes the corpus rather than hand-picking it. We prefer this design to mutation score, the alternative common in test-generation studies [61]: synthetic mutants often diverge from the fault distribution of real software [62], whereas a developer-verified historical fix bounds each bug's semantics and repair on evidence that a human maintainer already accepted.

## V. RESULTS AND ANALYSIS

### A. Coverage and Success Rate

To ensure a fair and rigorous comparison, we enforced a strict 3-hour time budget for all dynamic execution experiments, including LISA, the Vanilla LLM baseline, and OSS-Fuzz. We present our results in Table III and Table IV. LISA significantly outperforms both the Vanilla LLM and OSS-Fuzz across multiple dimensions.

*a) Superiority over Vanilla LLM:* Our Vanilla LLM baseline disables all of LISA's key components (feedback, repair, chunking, rules), retaining only basic correctness checks and a minimal prompt with API knowledge. The results demonstrate that LISA's superior performance stems from its architectural design rather than the underlying LLM capability. LISA achieves a 38% higher average execution rate and more than doubles the line coverage in complex libraries such as `sqlite3` (31.06% vs. 14.40%). This confirms LISA's $n$-gram feedback and repair mechanisms are essential for guiding LLM beyond shallow happy paths to explore deep, unseen code.

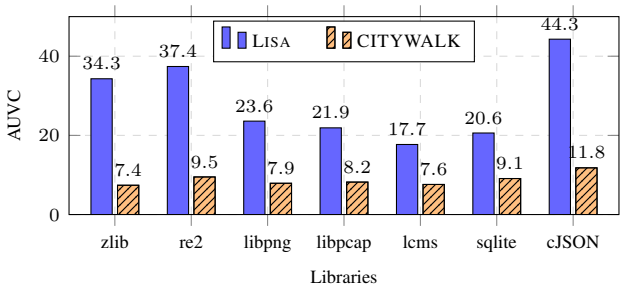


Fig. 4: Comparison of AUVC between LISA and CITYWALK.

*b) Comparison with Fuzzing:* Compared to the industrial standard OSS-Fuzz, LISA demonstrates remarkable efficiency, achieving higher coverage on the majority of libraries. LISA leads OSS-Fuzz in line coverage on five of the seven libraries, with gains ranging from +1.7% on `libpcap` to +34.5% on `cJSON` (e.g., +7.0% on `zlib` and +10.0% on `re2`). For libraries requiring structured inputs (`cJSON`, `re2`) or complex state transitions (`zlib`), the LLM's inherent knowledge of syntax and state flows bypasses the grammar barriers and state-space traps that hinder stochastic fuzzers in short-duration runs. Conversely, OSS-Fuzz leads on the two libraries with extensive error-handling branches for malformed inputs (`sqlite3`, `libpng`), where input mutation excels; however, the coverage gained from these error paths rarely translates into the discovery of functional bugs.

### B. Quality of Generated Oracles: AUVC

While code coverage measures how much code is executed, it does not reflect whether the execution is properly verified. We use AUVC (Section IV-C2) to evaluate the semantic richness of the generated tests. We compare LISA against CITYWALK, the state-of-the-art LLM-based unit test generation framework.

*a) High Density of Verifications:* As illustrated in Figure 4, LISA demonstrates a substantial advantage in assertion density across all evaluated libraries. On average, LISA generates **3.25**× more unique verifications per test than CITYWALK. For example, in `cJSON`, LISA achieves an AUVC of 44.3 compared to CITYWALK's 11.8.

*b) Semantic Depth:* This performance gap stems from fundamental differences in generation strategies. CITYWALK generates tests in a single pass ("one-test-per-method"), often producing simple checks such as `assert(ptr != NULL)` to ensure compilability. In contrast, LISA's dual-stage design allows $\text{LISA}_{\text{inv}}$ to focus exclusively on logic verification. By leveraging the constructed API knowledge base, LISA injects rich, state-aware invariants, for example, validating complex struct fields or return states, that CITYWALK misses. This high AUVC confirms that LISA not only executes code but rigorously verifies its correctness, directly contributing to its superior bug-finding capability.

TABLE V: Ablation study of LISA$_{\text{API}}$ components. w/o Feedback: the variant with only random selection. Comp.: compile rate. Exec.: execution rate.

| Library | Variant | Comp. | Exec. | Line Cov. | Branch Cov. |
|---|---|---|---|---|---|
| **zlib** | w/o Rules | 99.3% | 98.1% | 66.48% | 52.44% |
| | w/o Feedback | 99.0% | 98.4% | 65.58% | 52.48% |
| | w/o Examples | 99.5% | 98.3% | 67.31% | 53.92% |
| | w/o Repair | 98.8% | 95.7% | 67.58% | 54.30% |
| | **LISA$_{\text{API}}$** | **99.5%** | **98.6%** | **73.01%** | **60.01%** |
| **cJSON** | w/o Rules | 99.9% | 98.8% | 72.54% | 66.22% |
| | w/o Feedback | 100% | 98.5% | 71.80% | 63.95% |
| | w/o Examples | 99.3% | 98.2% | 73.20% | 66.60% |
| | w/o Repair | 99.8% | 97.0% | 72.72% | 65.37% |
| | **LISA$_{\text{API}}$** | **100%** | **99.4%** | **76.80%** | **69.07%** |
| **sqlite3** | w/o Rules | 97.6% | 83.2% | 26.88% | 19.37% |
| | w/o Feedback | 98.0% | 70.4% | 21.40% | 15.22% |
| | w/o Examples | 96.2% | 74.8% | 27.27% | 19.68% |
| | w/o Repair | 90.9% | 51.4% | 25.53% | 17.87% |
| | **LISA$_{\text{API}}$** | **98.3%** | **91.5%** | **31.06%** | **22.46%** |

### C. Ablation Study

We perform an ablation study on LISA's components, denoted LISA$_{\text{API}}$ and LISA$_{\text{inv}}$.

*1) Components of* LISA$_{API}$*:* To rigorously assess the contribution of each component within LISA$_{\text{API}}$, we conduct an ablation study on three representative libraries: `sqlite3`, `cJSON`, and `zlib`. We choose these three for three reasons: they span the complexity spectrum of our benchmark, ranging from a small structured-format parser (`cJSON`), through a stateful streaming codec (`zlib`), to a large and highly stateful database engine (`sqlite3`); they all belong to the bug-finding benchmark and thus reflect the settings where LISA is ultimately evaluated; and limiting the ablation to three subjects keeps the cost of running every component variant tractable. We derive four variants by selectively disabling specific modules from the full framework: • **w/o Rules:** Removes project-specific constraints and headers from the prompt. • **w/o Feedback:** Replaces the $n$-gram guided feedback mechanism with random selection. • **w/o Repair:** Disables the error detection and repair loop. • **w/o Examples:** Removes the few-shot successful examples from the prompt context. We evaluate these variants using the same metrics as RQ1. As shown in Table V, every component within LISA$_{\text{API}}$ is indispensable for effective API sequence generation.

For simpler libraries (`cJSON`, `zlib`), execution stays above 95% even under ablation, yet the full framework still yields the best coverage (e.g., +7.43% line coverage over random selection on `zlib`). For complex, state-sensitive libraries such as `sqlite3`, the components become critical: removing repair (*w/o Repair*) drops execution to 51.4% and line coverage to 25.53%, as LLMs frequently hallucinate invalid states. The $n$-gram feedback is the single most important component for coverage: its removal yields the lowest coverage across all libraries (e.g., 21.40% line coverage on `sqlite3`), since it otherwise steers generation toward under-explored API combinations.

*2) Component of* LISA$_{inv}$*:* Unlike LISA$_{\text{API}}$, whose primary goal is to produce valid and diverse execution paths (measured by coverage), the contribution of the invariant generation component (LISA$_{\text{inv}}$) is intrinsically linked to the ability to detect faults. A simple count of assertions (like AUVC) in ablation variants does not fully capture the *correctness* or *utility* of those assertions. Therefore, to provide a realistic evaluation, we defer the detailed ablation study of LISA$_{\text{inv}}$ (analyzing the impact of *Repair*, *Chunking*, and *Knowledge*) to **RQ3**. There, we evaluate how each component contributes to functional-bug detection, including two knowledge-source variants: one using only Daikon outputs and one using only official documentation.

Although several LISA$_{\text{API}}$ components are implemented through prompts, LISA is not simply a prompt-engineering variant. Its main contribution is the two-stage design: it first explores executable API sequences, then performs invariant generation as a separate stage grounded in documentation and chunk-level reasoning. This decomposition changes both the exploration target and how test oracles are constructed. The benefit is evident in the LISA$_{\text{API}}$ ablations (Table V) and in the bug-finding results, where removing LISA$_{\text{inv}}$ components consistently reduces detection effectiveness. These results suggest the gains come from the overall framework design, with prompting serving as merely one implementation mechanism.

### D. Bug-Finding Ability

We evaluate the practical bug-finding ability of LISA on LISA-BENCH, targeting 25 reproduced historical functional bugs across five libraries. To ensure a fair comparison, we aligned the execution constraints with the operational nature of each tool: • OSS-Fuzz: Guaranteed a continuous 6-hour fuzzing window to maximize mutation depth. • CITYWALK: Allowed to run to completion, processing the entire set of focal methods without time truncation. • LISA: Operated within a strict 3 hours pipeline for LISA$_{\text{API}}$ seed generation and generated final unit tests through LISA$_{\text{inv}}$ with the total runtime capped at 6 hours.

As shown in Table VI, LISA (Full) achieved a total detection rate of 48% (12/25), significantly outperforming both CITYWALK (12%) and OSS-Fuzz (8%).

Fisher's exact test confirms that these differences are statistically significant: LISA vs. OSS-Fuzz ($p = 0.0036$, odds ratio $= 10.6$) and LISA vs. CITYWALK ($p = 0.0121$, odds ratio $= 6.77$). Both $p$-values are below 0.05.

The baselines' poor performance reveals their structural limitations. OSS-Fuzz (2/25) detects only crash-inducing faults (e.g., memory corruption) and misses functional logic errors that do not trigger runtime aborts. Tellingly, the single libpng defect that OSS-Fuzz caught but LISA missed was a use-after-free that aborts at runtime rather than a silent logic error, underscoring the complementary scopes of the two tools. CITYWALK (3/25), despite using LLMs, achieves a low detection rate because its "one-test-per-method" strategy prioritizes compilability through extensive mocking and isolated testing [19]. While this yields good coverage for focal functions,

TABLE VI: Bug-finding results.

| Library | Total | Baselines | | Ours | Ablation of LISA$_{inv}$ | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | CITYWALK | OSS-Fuzz | LISA (Full) | w/o Repair | w/o Chunk | w/o Knowledge | Daikon Only | Doc Only |
| **cJSON** | 5 | 1 | 0 | **3** | 1 | 2 | 1 | 2 | 2 |
| **lcms** | 5 | 1 | 0 | **3** | 1 | 1 | 1 | 1 | 1 |
| **zlib** | 5 | 0 | 1 | **2** | 1 | 0 | 0 | 1 | 0 |
| **sqlite3** | 5 | 1 | 0 | **2** | 1 | 1 | 1 | 1 | 1 |
| **libpng** | 5 | 0 | 1 | **2** | 0 | 0 | 0 | 0 | 0 |
| **Total** | 25 | 3 (12%) | 2 (8%) | **12 (48%)** | 4 (16%) | 4 (16%) | 3 (12%) | 5 (20%) | 4 (16%) |

it cannot construct the complex, multi-step API sequences necessary to trigger state-related functional bugs.

In contrast, LISA's dual-stage design enables superior detection. LISA$_{API}$'s feedback-guided exploration constructs diverse, valid API sequences that penetrate deep program states, while LISA$_{inv}$ injects invariants that reveal silent logic errors, helping LISA find nine more bugs than the best baseline.

The ablation study validates the necessity of LISA$_{inv}$'s three core components. First, repair is critical for test survivability: without it (*w/o Repair*, 4 bugs), many tests terminate prematurely due to minor assertion failures or syntax errors and never reach the API calls that trigger bugs. Second, chunking is necessary for long sequences (*w/o Chunk*, 4 bugs): without it, the model struggles to maintain attention across the entire program, missing intermediate states and failing to place assertions at key points. Third, the knowledge base provides essential semantic guidance (*w/o Knowledge*, 3 bugs). Neither source alone suffices: Daikon-only introduces substantial noise (5 bugs), while documentation-only covers basic requirements but misses semantic constraints (4 bugs). Combining both gives LISA broader semantic coverage and stronger filtering of spurious invariants.

## VI. Discussion and Future Work

### A. Case Study: Silent Logic Error in SQLite

```
1 sqlite3 *db = nullptr;
2 sqlite3_stmt *stmt = nullptr;
3 sqlite3_open(":memory:", &db);
4 sqlite3_prepare_v2(db, "SELECT 0 OR 2", -1, &stmt,
      nullptr);
5 sqlite3_step(stmt);
6 int val = sqlite3_column_int(stmt, 0);
7 assert(val == 1);
```

Listing 3: Simplified PoC derived from LISA's generated test that triggers the SQLite logic bug.

To illustrate LISA's ability to find functional bugs, we analyze a logic regression in SQLite (commit d443f0a). As described in the commit message, a regression in the query optimizer caused the SQL expression **"0 OR 2"** to be erroneously evaluated as **2** (the integer value of the second operand) instead of 1 (Boolean TRUE). This is a classic logic error: the program executes valid CPU instructions and manages memory correctly, but the mathematical result is wrong. As shown in Listing 3, LISA exposed this defect as follows:

- **API Sequence Construction:** LISA first generated a valid call sequence to prepare and execute the specific query `"SELECT 0 OR 2"` using the `sqlite3_prepare_v2` and `sqlite3_step` APIs.
- **Invariant Assertion:** Crucially, instead of merely checking for a successful return code (e.g., `SQLITE_ROW`), LISA inferred the semantic invariant of the SQL operation. It generated the strict assertion `assert(val == 1)`, enforcing that the logical OR operation must yield a Boolean True (normalized to 1 by LISA).

Consequently, although the buggy version returned 2 due to an incorrect bitwise optimization and would pass crash-based fuzzing, LISA's test triggered an assertion failure.

### B. Threats to Validity and Limitations

We organize the residual threats along the standard three axes. *Internal validity*: LLM non-determinism (*Reproducibility*) and hyperparameter choices (*Hyperparameter Settings*) could bias the measured effect; the variance study (Table VII) shows the residual noise is well below LISA's margin over the baselines. *External validity*: the 25-bug, five-library, C/C++-only corpus limits generalization (*Benchmark scale*); we mitigate this with a fixed selection protocol (Section IV-C) and consistent per-library gains rather than aggregate-only claims. *Construct validity*: a violated invariant is a *high-confidence bug candidate*, not a proven defect (*Final oracle verification*), and LISA's recall is bounded by documentation completeness (Section III-F).

*a) Benchmark scale and representativeness:* LISA-BENCH currently comprises 25 developer-confirmed historical functional bugs spanning five libraries from distinct domains. While modest in absolute size, the set is assembled by a fixed protocol rather than hand-picked: each bug is drawn from a previously fixed historical commit, included only if it reproduces deterministically on our toolchain and exhibits a non-crashing functional symptom, and excluded otherwise. LISA's advantage is consistent across the per-library breakdown (Table VI) rather than driven by any single library, and its margin over both baselines remains statistically significant (Fisher's exact test, $p < 0.05$). Nonetheless, the absolute scale limits the precision of the point estimates, and evaluation on a larger and more varied bug corpus is an important direction for future work.

*b) Reproducibility and non-determinism:* Although the underlying LLM is non-deterministic, this affects only *which* tests LISA generates, not their stability once generated: every

TABLE VII: Run-to-run variance of LISA over three repetitions on three representative libraries. Comp./Exec.: compilation/execution success rate (mean±std, %); Cov. in %; Bugs: number of the 5 historical bugs detected in each of the three runs.

| Library | Comp. | Exec. | Line Cov. | Branch Cov. | Bugs |
|---|---|---|---|---|---|
| `zlib` | 99.8±0.3 | 98.3±0.5 | 70.5±2.1 | 57.1±2.5 | 2, 2, 2 |
| `cJSON` | 100±0.0 | 98.9±0.5 | 74.6±3.5 | 67.5±2.9 | 3, 3, 2 |
| `sqlite3` | 98.5±0.9 | 90.0±3.3 | 31.4±1.8 | 22.8±1.5 | 2, 2, 2 |

generated test is a fixed C/C++ program that compiles and executes identically on every run, so LISA does not produce flaky tests. Non-determinism during generation is further dampened by the two repair loops, which execute each candidate against the reference build and discard transient or unstable assertions, retaining only invariants that pass stably. To quantify the residual variance, we repeat the full pipeline three times on the three representative libraries (Table VII). Compilation and execution success rates are highly stable, while line and branch coverage vary by only a few percentage points (standard deviation $\leq 3.5$), far smaller than LISA's margin over the baselines; in every run LISA exceeds the Vanilla baseline on all three libraries and its ranking against OSS-Fuzz is unchanged. Bug detection is likewise stable (Table VII, last column): all but one of the detected bugs recurs in every run. The main tables report one representative run from this set.

*c) Hyperparameter Settings:* The choice of hyperparameters, such as the value of $n$ in $n$-gram and the repair attempt limit (5), along with the selected LLM, relies on preliminary tuning and prior studies. We acknowledge that these settings may not be optimal. However, to mitigate bias, we maintained fixed configurations across all comparative methods. Future work could investigate automated hyperparameter optimization or sensitivity analysis to better understand their impact on LISA's performance.

*d) Final oracle verification and documentation coverage:* Although LISA uses invariants and iterative repair to mitigate the oracle problem, final verification still requires human inspection. In particular, a failing assertion may indicate either a genuine implementation defect or a misinterpretation of the specification by the LLM. By design, LISA's chunk-invariant reasoning and verification-and-repair loop produce high-confidence assertions; nevertheless, residual false positives cannot be fully eliminated. A second, orthogonal limit is recall: LISA can only assert what documentation (plus filtered Daikon candidates) makes explicit, so conventional-but-undocumented contracts that experienced developers take for granted yield no invariant, and LISA can miss bugs that violate only such unwritten conventions. Grounding invariants in documentation is a deliberate trade-off, keeping them semantically valid and low-noise at the cost of leaving undocumented intent to future work on mining usage patterns and existing test suites. More broadly, the test oracle problem remains a fundamental open challenge in software testing [25], and resolving it more fully is an important direction for future work.

*e) Automation:* Constructing the API knowledge database requires one-time manual effort, so the overall pipeline is not yet fully automated. In principle, this step could be delegated to an AI agent to further reduce manual effort. However, it remains unclear whether such automation can match the quality and reliability of human curation. Developing and evaluating agent-based alternatives for this stage is an important direction for future work.

### C. Future Work

Beyond automating knowledge-base curation and mining undocumented conventions, as discussed above, two extensions look promising. First, integrating richer API-dependency models, such as those used by Hopper [42] and CITYWALK [19], could further raise the validity of generated sequences beyond what the $n$-gram model captures. Second, extending LISA to managed-language libraries (Python, Java) would test the portability of the two-stage decomposition beyond the C/C++ setting we studied here.

## VII. RELATED WORK

### A. Automated Software Testing for Libraries

*a) Library fuzzing:* Library fuzzing typically uses LLVM's libFuzzer [4], for which developers write *fuzz drivers* that parse inputs and invoke APIs; OSS-Fuzz [7] maintains such drivers for over 1000 projects. As writing drivers requires expert knowledge of library semantics, recent work automates their generation [26, 42, 63]: Hopper [42] models API calls with a lightweight interpreter and grammar to explore valid API compositions (*API sequences* in this paper), and PROMPTFUZZ [26] prompts an LLM to generate fuzz drivers, iteratively mutating the API combinations under coverage feedback to reach deep implementation bugs.

*b) Automated unit testing:* Heuristic-based unit test generation has been studied for decades but targets mainly object-oriented software (Java, C#), generalizing poorly to C/C++ because it focuses on object internal state rather than library-level API interactions [12, 13]. RANDOOP [12] mutates method-call sequences and checks language-level *oracle* properties, while $\mu$TEST and EvoSuite generate finer-grained oracles via mutation analysis; however, the injected artificial bugs are often unrealistic [15, 16], leaving these heuristics with low maintainability and usability [15].

### B. LLM-Based Unit Test Generation

To overcome the limitations of heuristic-based unit test generation, recent studies have proposed using the code under test as prompts to LLMs to automatically generate unit tests [16–19, 21, 38, 64, 65]. This line of work has evolved from generating only assertions [21, 64, 65] to producing complete test functions and even full test files [16–19].

Several benchmarks have been proposed to evaluate LLM-based software testing [22, 66, 67]. TestEval [66] and SWT-Bench [67] focus on generating single test functions, whereas TestGenEval [22] evaluates the generation of complete test suites, similar to EvoSuite. SWT-Bench measures bug detection

capability by generating tests for bug-fixing pull requests (PRs), where the generated tests should fail before the PR is merged and pass afterward. TestGenEval incorporates mutation score to assess whether the generated tests can detect behavioral changes in the code. However, no existing benchmark provides a ground-truth oracle for test generation, which is necessary for evaluating whether generated tests can detect real-world functional bugs in the current versions of software libraries. Moreover, all existing benchmarks support only Python, while more widely used languages in the software supply chain, such as C and C++, remain largely neglected.

### C. Invariant Generation

Program invariant generation methods can be broadly categorized into dynamic execution-based [34], search-based [68], and learning-based approaches [8, 27, 68]. Execution-based methods, such as Daikon [34], capture invariants of the *current* version of the software and are therefore better suited for generating regression tests rather than detecting functional bugs. LLM-based approaches, such as SmartInv [8], infer invariants that *should hold* according to the developer's intent by reasoning over both source code semantics and corresponding natural language descriptions [69]. Such invariants can be used to detect functional bugs in software libraries. LISA is a generic framework capable of integrating any style of invariant generation method based on user preference, supporting both regression testing with Daikon and functional bug detection using LLM-based invariant generalization. LISA reconciles the two lines: Daikon supplies candidate relations that LISA filters against documentation, while the LLM supplies intent-level contracts from the same documentation, checked in turn against the reference build. Neither source alone flags functional bugs, but their documentation-grounded reconciliation does.

### D. Positioning of LISA

LISA shares individual building blocks with prior work, but its contribution lies in how it *decomposes* functional-bug detection rather than in any single component. PROMPTFUZZ [26] generates *fuzz drivers* for *crash* detection and steers exploration by mutating API combinations under code-coverage feedback; LISA instead synthesizes *API sequences* (not drivers), targets *functional* bugs, steers exploration with a new $n$-gram API-combination coverage (Section III-E), and supplies the invariant oracle that PROMPTFUZZ lacks, reusing only PROMPTFUZZ's front-end API extraction, not its fuzzing loop. SmartInv [8] infers invariants for *smart contracts* from code and intent; LISA instead targets *C/C++ library APIs*, derives invariants from API *documentation* as executable test oracles, and, unlike monolithic inference, *decouples* oracle construction from sequence generation, inserting invariants incrementally at chunk boundaries under a verify-and-repair loop (Section III-G2). LLM-UT methods such as CITYWALK [19] follow a one-test-per-method scheme whose oracles are frequently trivial (e.g., `assert(ptr != NULL)`) and cannot construct the multi-step API sequences required to reach stateful functional bugs. The novelty of LISA is therefore not any single technique it borrows, but this decomposition, separating reachability (sequence synthesis) from oracle construction, guiding the former with $n$-gram API-combination coverage and grounding the latter in documentation-derived chunk invariants, which no prior approach provides.

## VIII. CONCLUSION

We presented LISA, a novel LLM-based unit testing framework for detecting functional bugs in software libraries. By decoupling API-sequence exploration from invariant construction in unit tests, LISA leverages $n$-gram API-combination coverage feedback and chunk-invariant reasoning to generate diverse and semantically meaningful test functions. Across 7 C/C++ libraries, it attains higher average branch coverage than OSS-Fuzz, and on 25 historical functional bugs it detects 12 (48%), versus 3 for CITYWALK and 2 for OSS-Fuzz. Its novelty lies in this decomposition rather than in any single borrowed component, and it mitigates rather than solves the oracle problem, emitting high-confidence bug candidates for developer confirmation.

## DATA AVAILABILITY

Our artifacts and source code to reproduce the data are available at [70] and on GitHub.[1] This public digital repository includes: (1) the LISA implementation and prompts, (2) the full list of the 25 re-introduced historical bugs (IDs, links, and corresponding patches/commits), (3) the generated test programs for the bugs found by LISA, (4) all scripts/configurations used for coverage measurement and reporting, and (5) the rationale and sensitivity data for choosing the $n$-gram parameter $N$.

## REFERENCES


[1] N.E. Fenton and N. Ohlsson. "Quantitative analysis of faults and failures in a complex software system". In: *IEEE Transactions on Software Engineering* 26.8 (2000), pp. 797–814.

[2] Andy Chou et al. "An empirical study of operating systems errors". In: *SIGOPS Oper. Syst. Rev.* 35.5 (Oct. 2001), pp. 73–88.

[3] Hao-Nan Zhu et al. *From Bugs to Benchmarks: A Comprehensive Survey of Software Defect Datasets*. 2025. arXiv: 2504.17977 [cs.SE].

[4] Kosta Serebryany. "Continuous Fuzzing with libFuzzer and AddressSanitizer". In: *2016 IEEE Cybersecurity Development (SecDev)*. 2016, pp. 157–157.

[5] Andrea Fioraldi et al. "AFL++ combining incremental steps of fuzzing research". In: *Proceedings of the 14th USENIX Conference on Offensive Technologies*. 2020, pp. 10–10.

[6] Marcel Böhme et al. "Directed Greybox Fuzzing". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2329–2344.


[1] https://github.com/SecurityLab-UCD/CNTG and https://github.com/SecurityLab-UCD/CGNTG

[7] Kostya Serebryany. “OSS-Fuzz - Googles continuous fuzzing service for open source software”. In: Vancouver, BC: USENIX Association, Aug. 2017.

[8] Sally Junsong Wang, Kexin Pei, and Junfeng Yang. “ SmartInv: Multimodal Learning for Smart Contract Invariant Inference”. In: *2024 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2024, pp. 2217–2235.

[9] Vladimir Khorikov. *Unit Testing Principles, Practices, and Patterns*. Simon and Schuster, 2020.

[10] Yangyang Zhao et al. “The impact of continuous integration on other software development practices: A large-scale empirical study”. In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2017.

[11] Moritz Beller et al. “When, How, and Why Developers (Do Not) Test in Their IDEs”. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: Association for Computing Machinery, 2015, pp. 179–190.

[12] Carlos Pacheco et al. “Feedback-Directed Random Test Generation”. In: *29th International Conference on Software Engineering (ICSE’07)*. 2007, pp. 75–84.

[13] Gordon Fraser and Andrea 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*. ESEC/FSE ’11. Szeged, Hungary: Association for Computing Machinery, 2011, pp. 416–419.

[14] Sina Shamshiri et al. “How Do Automatically Generated Unit Tests Influence Software Maintenance?” In: *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 2018, pp. 250–261.

[15] Annibale Panichella et al. “Revisiting test smells in automatically generated tests: limitations, pitfalls, and opportunities”. In: *2020 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE. 2020, pp. 523–533.

[16] Nikitha Rao et al. “CAT-LM Training Language Models on Aligned Code and Tests”. In: *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*. ASE ’23. Echternach, Luxembourg: IEEE Press, 2024, pp. 409–420.

[17] Yifeng He et al. “UniTSyn: A Large-Scale Dataset Capable of Enhancing the Prowess of Large Language Models for Program Testing”. In: *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2024. Vienna, Austria: Association for Computing Machinery, 2024, pp. 1061–1072.

[18] Yifeng He et al. “FuzzAug: Data Augmentation by Coverage-guided Fuzzing for Neural Test Generation”. In: *Findings of the Association for Computational Linguistics: EMNLP 2025*. Suzhou, China: Association for Computational Linguistics, Nov. 2025, pp. 15642–15655.

[19] Yuwei Zhang et al. “CITYWALK: Enhancing LLM-Based C++ Unit Test Generation via Project-Dependency Awareness and Language-Specific Knowledge”. In: *ACM Trans. Softw. Eng. Methodol.* (Aug. 2025). Just Accepted.

[20] Chunqiu Steven Xia et al. “Demystifying LLM-Based Software Engineering Agents”. In: *Proc. ACM Softw. Eng.* 2.FSE (June 2025).

[21] Pengyu Nie et al. “Learning Deep Semantics for Test Completion”. In: *Proceedings of the 45th International Conference on Software Engineering*. ICSE ’23. Melbourne, Victoria, Australia: IEEE Press, 2023, pp. 2111–2123.

[22] Kush Jain, Gabriel Synnaeve, and Baptiste Roziere. “TestGenEval: A Real World Unit Test Generation and Test Completion Benchmark”. In: *The Thirteenth International Conference on Learning Representations*. 2025.

[23] William E. Howden. “Theoretical and Empirical Studies of Program Testing”. In: *Proceedings of the 3rd International Conference on Software Engineering (ICSE)*. ICSE ’78. Atlanta, Georgia, USA: IEEE Press, 1978, pp. 305–311.

[24] Elaine J. Weyuker. “On Testing Non-testable Programs”. In: *The Computer Journal* 25.4 (1982), pp. 465–470.

[25] Earl T. Barr et al. “The Oracle Problem in Software Testing: A Survey”. In: *IEEE Transactions on Software Engineering* 41.5 (2015), pp. 507–525.

[26] Yunlong Lyu et al. “Prompt Fuzzing for Fuzz Driver Generation”. In: *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. CCS ’24. Salt Lake City, UT, USA: Association for Computing Machinery, 2024, pp. 3793–3807.

[27] Kexin Pei et al. “Can Large Language Models Reason about Program Invariants?” In: *Proceedings of the 40th International Conference on Machine Learning*. Vol. 202. Proceedings of Machine Learning Research. PMLR, 23–29 Jul 2023, pp. 27496–27520.

[28] Al Bessey et al. “A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World”. In: *Communications of the ACM* 53.2 (2010), pp. 66–75.

[29] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs”. In: *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2008, pp. 209–224.

[30] K. Rustan M. Leino. “Dafny: An Automatic Program Verifier for Functional Correctness”. In: *16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*. Springer, 2010, pp. 348–370.

[31] Koen Claessen and John Hughes. “QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs”. In: *5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, 2000, pp. 268–279.

[32] Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. "Coverage Guided, Property Based Testing". In: *Proceedings of the ACM on Programming Languages (OOPSLA)* 3 (2019), pp. 1–29.

[33] *The LLVM Compiler Infrastructure*. https://github.com/llvm/llvm-project, accessed 2025-10-12.

[34] M.D. Ernst et al. "Dynamically discovering likely program invariants to support program evolution". In: *IEEE Transactions on Software Engineering* 27.2 (2001), pp. 99–123.

[35] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2004, pp. 75–86.

[36] Edsger W. Dijkstra. "Notes on Structured Programming". In: *Structured Programming*. Academic Press, 1972, pp. 1–82.

[37] André L. Santos et al. "Stepwise API usage assistance using n-gram language models". In: *Journal of Systems and Software* 131 (2017), pp. 461–474.

[38] Gabriel Ryan et al. "Code-Aware Prompting: A Study of Coverage-Guided Test Generation in Regression Setting using LLM". In: *Proc. ACM Softw. Eng.* 1.FSE (July 2024).

[39] Xingchen Wan et al. "Better Zero-Shot Reasoning with Self-Adaptive Prompting". In: *Findings of the Association for Computational Linguistics: ACL 2023*. Toronto, Canada: Association for Computational Linguistics, July 2023, pp. 3493–3514.

[40] Noah Shinn et al. "Reflexion: language agents with verbal reinforcement learning". In: *Thirty-seventh Conference on Neural Information Processing Systems*. 2023.

[41] Jicheng Wang, Yifeng He, and Hao Chen. *RepoGenReflex: Enhancing Repository-Level Code Completion with Verbal Reinforcement and Retrieval-Augmented Generation*. 2024. arXiv: 2409.13122 [cs.SE].

[42] Peng Chen et al. "Hopper: Interpretative Fuzzing for Libraries". In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. CCS '23. Copenhagen, Denmark: Association for Computing Machinery, 2023, pp. 1600–1614.

[43] Michael D. Ernst et al. "Quickly detecting relevant program invariants". In: *Proceedings of the 22nd International Conference on Software Engineering*. ICSE '00. Limerick, Ireland: Association for Computing Machinery, 2000, pp. 449–458.

[44] Danning Xie et al. "DocTer: documentation-guided fuzzing for testing deep learning API functions". In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2022. Virtual, South Korea: Association for Computing Machinery, 2022, pp. 176–188.

[45] Nelson F. Liu et al. "Lost in the Middle: How Language Models Use Long Contexts". In: *Transactions of the Association for Computational Linguistics* 12 (2024), pp. 157–173.

[46] Jonathan Metzman et al. "FuzzBench: an open fuzzer benchmarking platform and service". In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2021. Athens, Greece: Association for Computing Machinery, 2021, pp. 1393–1403.

[47] John Yang et al. "SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2024.

[48] Xingyao Wang et al. "OpenHands: An Open Platform for AI Software Developers as Generalist Agents". In: *The Thirteenth International Conference on Learning Representations (ICLR)*. 2025.

[49] Zifan Nan et al. "Test Intention Guided LLM-based Unit Test Generation". In: *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 2025.

[50] Carlos E. Jimenez et al. "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?" In: *The Twelfth International Conference on Learning Representations (ICLR)*. 2024.

[51] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. "Magma: A Ground-Truth Fuzzing Benchmark". In: vol. 4. 3. New York, NY, USA: Association for Computing Machinery, Nov. 2020.

[52] Jean-loup Gailly and Mark Adler. *ZLIB DATA COMPRESSION LIBRARY*. https://zlib.net/. 2025.

[53] PNG Development Group. *LIBPNG: Portable Network Graphics Reference Library*. https://github.com/pnggroup/libpng. 2025.

[54] The Tcpdump Group. *libpcap: Portable Packet Capture Library*. https://github.com/the-tcpdump-group/libpcap. 2025.

[55] SQLite Development Team. *SQLite*. https://github.com/sqlite/sqlite. 2025.

[56] Marti Maria. *Little CMS: A Free Color Management Engine*. https://github.com/mm2/Little-CMS. 2025.

[57] Dave Gamble. *cJSON: Ultralightweight JSON Parser in ANSI C*. https://github.com/DaveGamble/cJSON. 2025.

[58] Google team. *RE2: Efficient Regular Expression Matching*. https://github.com/google/re2. 2025.

[59] Max Schäfer et al. "An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation". In: *IEEE Transactions on Software Engineering* 50.1 (2024), pp. 85–105.

[60] Runlin Liu et al. *LLM-based Unit Test Generation for Dynamically-Typed Programs*. 2025. arXiv: 2503.14000 [cs.SE].

[61] J. H. Andrews, L. C. Briand, and Y. Labiche. "Is mutation an appropriate tool for testing experiments?" In: *Proceedings of the 27th International Conference on Software Engineering*. ICSE '05. St. Louis, MO,

USA: Association for Computing Machinery, 2005, pp. 402–411.

[62] Mike Papadakis et al. “Are Mutation Scores Correlated with Real Fault Detection? A Large Scale Empirical Study on the Relationship Between Mutants and Real Faults”. In: *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 2018, pp. 537–548.

[63] Hongxiang Zhang et al. “LLAMAFUZZ: Large Language Model Enhanced Greybox Fuzzing”. In: *ACM/IEEE International Conference on Automation of Software Test (AST)*. 2026.

[64] Cody Watson et al. “On learning meaningful assert statements for unit test cases”. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE ’20. Seoul, South Korea: Association for Computing Machinery, 2020, pp. 1398–1409.

[65] Elizabeth Dinella et al. “TOGA: A Neural Method for Test Oracle Generation”. In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE ’22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 2130–2141.

[66] Wenhan Wang et al. “TestEval: Benchmarking Large Language Models for Test Case Generation”. In: *Findings of the Association for Computational Linguistics: NAACL 2025*. Albuquerque, New Mexico: Association for Computational Linguistics, Apr. 2025, pp. 3547–3562.

[67] Niels Mündler et al. “SWT-Bench: Testing and Validating Real-World Bug-Fixes with Code Agents”. In: *Advances in Neural Information Processing Systems*. Vol. 37. Curran Associates, Inc., 2024, pp. 81857–81887.

[68] Xujie Si et al. “Code2Inv: A Deep Learning Framework for Program Verification”. In: *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 2124, 2020, Proceedings, Part II*. Los Angeles, CA, USA: Springer-Verlag, 2020, pp. 151–164.

[69] Yifeng He et al. “Evaluating Program Semantics Reasoning with Type Inference in System $F$”. In: *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*. 2025.

[70] Ruogu Yang et al. *Lisa Artifacts*. Zenodo, https://doi.org/10.5281/zenodo.20839021. Jan. 2026.

## APPENDIX A
## PROMPT TEMPLATES

Your task is to write a complete, logically correct C++ function named "int test_{project}_api_sequence()" using the {project} library.
The API sequence should focus on the usage of the {project} library, and several essential aspects of the library are provided below. {headers} {APIs} {context}
Please use the following API functions in your function {combinations}
Here is a successful example function: {successful_example}
Below is basic function requirements: {Function Requirements} and project's specific rules:{project_rules}
Code Quality Rules:

- The API sequence should follow a realistic and complete usage pattern:
  - Initialize → Configure → Operate → Validate → Cleanup
- Ensure data flows meaningfully between API calls (no dummy or unused variables).
- Do not use placeholders like '// your code here'.
- If you have to write any helper functions, begin them with static
- Only output the function body 'int test_{project}_api_sequence()' { ... }'

Example output:

```
int test_{project}_api_sequence() {
// Step 1: Initializations
// Step ...
// Step ... : Cleanup
return 66;
}
```

Fig. 5: The structured prompt template employed in our framework. Dynamic fields are denoted by brackets.

The previous attempt to generate code failed with the following error:
Error code:{error_code}
Error Type: {error_type}
Error Details:{error_details}
Please regenerate a new program to repair the error without changing the logic, do not redefine main function and any other parameters even if the error is not defined, and do not change the function name.
If error type is execution error without error details, you can regard it as Segmentation fault (core dumped).
Do not include if branches or loops; the function should be a straight-line sequence of API calls.

Fig. 6: The structured repair prompt template for correcting erroneous API sequences. Dynamic fields are denoted by brackets.

## APPENDIX B
## DETAILS OF API N-GRAM FEEDBACK

### A. Proofs of Condense Transformation

*Proof of Proposition 1.* Let $x, y \in I$ such that $0 < x < y \leq 1$. Let $\lambda = x/y \in (0,1)$, then $x = \lambda y + (1-\lambda)\cdot 0$. By strict concavity,

$$\lambda f(y) = \lambda f(y) + (1-\lambda) f(0) < f(\lambda y + (1-\lambda)\cdot 0) = f(x).$$

Rearranging the above inequality gives

$$\lambda f(y) < f(x) \implies f(y) < \frac{f(x)}{\lambda} = \frac{y}{x} f(x) \implies \frac{f(y)}{f(x)} < \frac{y}{x}.$$

Since $f$ is strictly monotone increasing, we have $0 < f(x) < f(y)$, thus

$$1 < \frac{f(y)}{f(x)} < \frac{y}{x}.$$

We conclude that any strictly monotone increasing concave function with left-endpoint preserved compresses the range of input values. □

*Proof that power condense is a condense transformation.* We prove that power condense is a condense transformation. We write the power condense $C(a) = \hat{E}(a)^\alpha$ as $f(x) = x^\alpha$ for $\alpha \in (0,1)$ for simplification.

- *Strictly monotone increasing:* We take the first derivative of $f$ with respect to $x$,

  $$f'(x) = \frac{\mathrm{d}x^\alpha}{\mathrm{d}x} = \alpha x^{\alpha-1}.$$

  Since $\alpha > 0$, $f'(x) > 0$ for all $x > 0$. Thus, $f$ is strictly monotone increasing.
- *Strictly concave:* For concavity, we take the second derivative of $f$ with respect to $x$,

  $$f''(x) = \frac{\mathrm{d}^2 x^\alpha}{\mathrm{d}x^2} = \frac{\mathrm{d}\alpha x^{\alpha-1}}{\mathrm{d}x} = \alpha(\alpha-1)x^{\alpha-2}.$$

  Since $\alpha \in (0,1)$, we have $\alpha - 1 < 0$, thus $f''(x) < 0$ for all $x > 0$. Therefore, we conclude that $f$ is strictly concave.
- *Left-endpoint preserved:* We have $f(0) = 0^\alpha = 0$.

Thus, we conclude that power condense satisfies Definition III.1. □

### B. Choice of Maximum Condensation Factor

In this section, we illustrate how we choose the maximum condensation factor $\alpha_{\min}$ in Equation 4. Let $x, y \in I$ be the energies of two API functions such that $x < y$, then their relative ratio after power condense is given by

$$R = \frac{y^\alpha}{x^\alpha} = \left(\frac{y}{x}\right)^\alpha.$$

Let us consider the extreme case after normalization, where $x = \varepsilon$ and $y = 1$, so $y/x = 1/\varepsilon = 100$. Since $R = (y/x)^\alpha$ is increasing in $\alpha$, the *strongest* condensation ($\alpha = \alpha_{\min}$) yields the *smallest* selection ratio between the highest- and lowest-energy API function:

$$R_{\alpha_{\min}} = \left(\frac{1}{\varepsilon}\right)^{\alpha_{\min}}.$$

We choose $\alpha_{\min}$ so that, even under this strongest condensation, the highest-energy API function stays a useful factor more likely to be selected than the lowest-energy one, preventing over-condensation. For example, to keep that factor at $10\times$,

$$R_{\alpha_{\min}} = \left(\frac{1}{\varepsilon}\right)^{\alpha_{\min}} = 10,$$

$$\alpha_{\min} \log\left(\frac{1}{\varepsilon}\right) = \log(10) \implies \alpha_{\min} = \frac{\log(10)}{\log(1/\varepsilon)} = 0.5.$$

Through this feedback loop, the API energy distribution dynamically evolves based on the success and novelty of generated 3-grams, enabling the model to balance correctness reinforcement with exploratory diversity.

## APPENDIX C
## DESIGN AND MEASUREMENT NOTES

### A. Comparison with HOPPER

LISA builds on Hopper's [42] distinction between Intra-API (argument validity) and Inter-API (call dependencies). LISA's advantage lies in its "decoupling" strategy, which delegates resolution of Inter-API constraints entirely to $\text{LISA}_{\text{API}}$. $\text{LISA}_{\text{API}}$ focuses on generating valid and high-quality API sequences (solving the "reachability" problem) through LLM reasoning and API $n$-gram feedback before invariant generation. This process drastically reduces the search space complexity, allowing the subsequent stage to concentrate entirely on verifying Intra-API behaviors through invariants, thereby ensuring high-quality assertions.

### B. Straight-Line Execution Policy

We acknowledge that straight-line programs cannot explicitly encode branches. However, the library under test can still take different internal branches based on internal state, call order, and implicit conditions, even when the generated driver itself is straight-line. This is an intentional trade-off to prioritize correctness and executability while still enabling substantial state evolution and control-flow coverage.

### C. Measurement Setup

For OSS-Fuzz, we used the official fuzzing harnesses from the OSS-Fuzz project for each library and did not modify them. All coverage metrics reported in the main text are measured at the library level. That is, the coverage only reflects the execution of the target library code, not the fuzzing harness or the test driver. In our implementation, we used `gpt-5-mini` for the API sequence generation phase ($\text{LISA}_{\text{API}}$), since this phase focuses on fast, large-scale exploration of API sequences. We used `gpt-5.1` for the invariant generation phase ($\text{LISA}_{\text{inv}}$), since this phase requires stronger reasoning to produce correct and meaningful invariants.

## APPENDIX D
## METRIC DEFINITIONS AND ANNOTATION PROTOCOL

This appendix supplements the metric definitions in Section IV-C with the annotation protocols used for AUVC and HBDR; the core $\text{LISA}_{\text{API}}$ metrics (CSR, ESR, LC, BC) are defined there.

### A. AUVC Uniqueness Annotation Protocol

Multiple invariant statements may be considered unique even if they query the same program state. For example, repeated queries over the same zlib stream state following a sequence of API operations are considered distinct. Conversely, syntactically different invariants may be non-unique if they express equivalent conditions (e.g., `x >= 1` and `x > 0` for $x \in \mathbb{Z}$). The uniqueness of verifications is determined through manual analysis by two authors with more than three years of C/C++ development experience, who independently inspected and categorized the generated invariants based on their semantic meaning. Their results are discussed and confirmed by a group of senior software researchers and library maintainers. Disagreements were resolved through discussion to reach a consensus.

### B. HBDR vs. Mutation Score

While mutation score is a widely adopted metric in evaluating unit test generation techniques [61], synthetic mutants may not fully represent the complexity of real-world software faults [62]. We therefore prefer historical bug re-introduction as a more faithful proxy for real-world fault detection.

## APPENDIX E
## API CONTRACT DOCUMENTATION DETAILS

We construct API contract documentation through two complementary ways: dynamic analysis and natural-language documentation analysis.

*a) Invariants generated by Daikon:* Daikon [43] is a dynamic analysis tool designed to detect *likely invariants* in programs by observing their runtime behavior. It reports properties that hold during the observed executions. However, the properties extracted from *observed* executions cannot be directly used to detect functional bugs, since they only capture the snapshot of the current program development stage. Even more critically, if researchers mistakenly treat incorrect or spurious properties as true invariants, correct code with correct logic may be falsely flagged as buggy, leading to misleading or invalid conclusions. Nevertheless, if these Daikon-generated invariants are further manually reviewed and validated, they can still serve as valuable references for understanding program behavior and guiding subsequent analysis, such as refining test oracles or identifying potential correctness properties.

*b) Inferring Invariants From Documentation:* In open-source libraries, developers provide official documentation that describes each API's purpose, parameters, return values, and expected behavior. These expert-crafted manuals include statements such as "this method is read-only" or "this API does not modify internal state", thereby implicitly defining the behavior invariants of the API. Because the documentation already tells what the API does and what it guarantees, we leverage these descriptions to manually infer invariants (i.e., properties that should always hold when the API is used correctly). For example, if the documentation states that an API is read-only, then a concrete invariant in natural language might be "no internal state is modified". Prior work [44] provides evidence that leveraging documentation can effectively support program analysis.

Daikon captures empirical invariants based on runtime behavior, while API documentation provides intended invariants that reflect the developer's design. In practice, we first run Daikon on automatically generated executable API sequences to obtain candidate properties. We use a filtering step to keep invariant candidates that match the official documentation and do not break the program when added as assertions. In our current method, checking if candidates match the documentation requires only a small, one-time human check per API. This step needs no training data or model tuning,

and the effort does not grow with the number of tests. This combination improves both the accuracy and coverage of the inferred invariants.