# FLAME: Enhancing Functional Coverage in Processor Verification via Large Language Models

Xiaopeng Li, Junjie Chen, Ming Yan, Dong Wang, Xingyu Fan, Zhentao Tang, Shixiong Kai, Jianye Hao, Mingxuan Yuan, *Senior Member, IEEE*, and Zan Wang

***Abstract*—Processor functional verification plays a crucial role in ensuring the quality of processor designs. Traditional techniques like constrained random verification (CRV) struggle to achieve high functional coverage due to the vast instruction space of processors. While large language model (LLM)-based techniques show potential, merely instructing LLMs has notable limitations, especially when addressing functional points (FPs) that require deep semantic understanding. To tackle these challenges, we propose a novel technique, FLAME, which specially designs and integrates retrieval-augmented generation (RAG), chain-of-thought (CoT), and functional-coverage-guided feedback strategies. This technique establishes semantic mappings between FPs and instructions, enabling the iterative generation of valid and effective test cases. Evaluation on four widely used open-source processor designs shows that FLAME outperforms the typical and state-of-the-art baselines in functional coverage improvement with an average of 34.25%~220% while drastically reducing the time required to achieve the same functional coverage by up to 86.13%. Moreover, ablation analysis highlights the vital role of each component in the framework's overall effectiveness. This work demonstrates the superiority of our LLM-based technique FLAME in enhancing processor functional verification.**

***Index Terms*—Functional coverage, large language models (LLMs), processor verification, test generation.**

## I. Introduction

PROCESSOR verification is a vital step in processor development that ensures design correctness, particularly as modern hardware designs grow more complex [1], [2]. Manual verification is quite costly as it requires significant domain-specific expertise. The main automatic verification method involves simulating and functionally verifying the processor design through test cases to ensure all functionalities work as expected [3], [4]. Functional coverage is a key metric for evaluating processor verification effectiveness [5]. Covering a larger number of functional points (FPs) indicates higher test effectiveness, which serves as a crucial indicator of functional verification automation in practice.

In the last decade, various techniques based on simulation have been developed to cover more FPs automatically, such as constrained random verification (CRV) [3], [6] and coverage-guided fuzzing (CGF) methods [7], [8], [9], [10]. However, many FPs remain uncovered due to the inherent complexity of their triggering conditions (often requiring test cases with specific semantics). Due to the vast space of instruction types and parameters, it is difficult for simulation-based techniques to search for effective test cases that satisfy such complex triggering conditions. Particularly, most of these existing techniques require extensive involvement from experienced engineers to write instruction constraints for ensuring test validity, causing significant manual effort in implementing these techniques.

Large language models (LLMs) possess powerful semantic understanding capabilities, and recent advances have brought forth some LLM-based techniques to processor functional verification [11], [12], [13], [14]. In general, they instructed LLMs to understand the semantics of uncovered FPs and then generate test cases to cover them. The results have demonstrated the superiority of such LLM-based techniques over traditional ones. However, simply instructing LLMs still exhibits significant limitations, particularly in addressing the FPs that demand comprehensive semantic understanding. For instance, in a real-world RISC-V processor (i.e., IBEXv2 [15]), the design document specifies an FP named `auto_InstrCategoryCSRIllegal`, defined as *"any instruction attempting a CSR access that is not allowed"* [15]. Achieving this FP requires the generated test cases to include specific combinations of control and status register (CSR)-related instructions (e.g., `csrw` and `csrr`), as well as the proper parameters of these instructions and CSRs' addresses. FPs like this often necessitate the generation of highly specific instructions or instruction combinations to produce valid test cases—an area where the current LLM-based techniques fall short. Indeed, our experiments confirm that the state-of-the-art LLM-based techniques [12] suffer from low test validity (only up to 26.13% on a large-scale processor, i.e., CVA6 [16]) and yields only marginal improvements in covering FPs in practical scenarios (as demonstrated in Section V-B).

To sum up, the main reasons for the significant challenge of covering more FPs are twofold.

1) These FPs represent specific processor scenarios and behaviors under test, requiring special instructions to trigger corresponding test conditions. Notably, mapping abstract FPs to concrete input instructions is challenging, which requires a deep understanding of both the

Received 26 May 2025; revised 23 October 2025; accepted 9 February 2026. This work was supported in part by the National Key Research and Development Program of China under Grant 2024YFB4506300, in part by the National Natural Science Foundation of China under Grant 62322208, in part by the Huawei Project Funding, in part by the National Natural Science Foundation of China under Grant 62232001, and in part by the CCF Young Elite Scientists Sponsorship Program (by CAST). This article was recommended by Associate Editor M. Sonza Reorda. *(Corresponding author: Junjie Chen.)*

Xiaopeng Li, Junjie Chen, Ming Yan, Dong Wang, Xingyu Fan, and Zan Wang are with the School of Computer Software, Tianjin University, Tianjin 300350, China (e-mail: xpl@tju.edu.cn; junjiechen@tju.edu.cn; yanming@tju.edu.cn; dong_w@tju.edu.cn; fxyqaq@tju.edu.cn; wangzan@tju.edu.cn).

Zhentao Tang, Shixiong Kai, Jianye Hao, and Mingxuan Yuan are with the Huawei Noah's Ark Laboratory, Beijing 100085, China (e-mail: tangzhentao1@huawei.com; kaishixiong@huawei.com; haojianye@huawei.com; Yuan.Mingxuan@huawei.com).

Digital Object Identifier 10.1109/TCAD.2026.3666802

instruction set architecture (ISA) and processor-related documentation.

2) Complex interactions between instructions require precise modeling of dependencies and execution order, making it challenging to generate valid instruction sequences as test cases. Existing LLM-based techniques struggle to generate semantically desirable and correct instruction sequences due to their tendency to hallucinate.

In this work, we aim to harness LLMs' capabilities more effectively to enhance functional coverage in processor verification, addressing the FP bottlenecks that hinder existing techniques. To achieve this, we introduced **FLAME** (Functional coverage using large language model), offering the following novel and impactful capabilities.

1) *Mapping Semantics Between FPs and Instructions:* For uncovered points, FLAME performs static analysis to extract instruction-related information. Using retrieval-augmented generation (RAG), it retrieves relevant descriptions from design documents, such as processor documentation and FPs definitions. These descriptions establish semantic links between FPs and the corresponding instructions desired by test cases. This method effectively addresses the challenges of complex semantic conditions required for achieving nontrivial functional coverage and the extensive search space associated with input instructions.
2) *Generating Valid and Effective Test Cases:* FLAME leverages RAG to obtain instruction details (e.g., instruction types and usage constraints) from the ISA specification, and then feeds them into LLMs to guide the test generation. In particular, to ensure test validity, FLAME generates tests at the source code level (i.e., generating C programs and then compiling them to instructions for verification) rather than the instruction level, since LLMs have demonstrated superior performance on the former [17], [18], [19] (as investigated in Section V-A1). Also, analyzing patterns of invalid or ineffective test cases is helpful to ensure constraint compliance and maximize validity. Therefore, FLAME enhances its test generation process by incorporating some counterexamples, specific test cases that previously failed to cover certain FPs, into its prompts.
3) *Functional-Coverage-Guided Iterative Testing:* To enhance improvement, FLAME collects functional coverage results and refreshes uncovered FPs as well as counterexamples after executing the generated test cases. This information guides subsequent test generation rounds by updating the target FPs.

To investigate the effectiveness of FLAME, we conducted extensive experiments on four real-world processor designs based on the open-source ISA (RISC-V), including IBEX (with two different versions) [15], CVA6 [16] and CV32E40P [20], which have been widely used in the studies on processor verification [9], [11], [21], [22], [23], [24], [25]. Our experimental results demonstrate the significant superiority of FLAME over the five typical or state-of-the-art baselines (including the widely-studied CRV and CGF techniques, one state-of-the-art processor fuzzing technique, and two state-of-the-art LLM-based techniques). For instance, FLAME covered 34.25%~220% more target FPs than them on average across the four studied processor designs. Particularly, FLAME improves test validity by 28.13% compared to the state-of-the-art LLM-based techniques on average. This demonstrates that FLAME significantly enhanced functional coverage in processor verification by leveraging LLMs' semantic understanding and addressing the mapping and validity challenges as aforementioned. Our experiments also confirmed the contribution of retrieval-based semantic reasoning, source-level test case generation, and iterative coverage refinement to the overall effectiveness of FLAME, and investigated the impact of different LLMs (including GPT-3.5-Turbo, CodeLlama-7B, CodeLlama-13B, and Phind CodeLlama-34B) on FLAME.

The main contributions of our work are as follows.

1) We propose a novel LLM-based technique to enhance functional coverage in processor verification, reducing expert knowledge and manual effort.
2) We build efficient semantic mappings between FPs and instructions, and generate effective test cases from the source code level, in order to address the search space and validity challenges.
3) We conduct extensive experiments on four widely used real-world processors, demonstrating the performance and generalizability of FLAME. We have made the replication package publicly available on our homepage [26].

## II. Background and Related Work

### A. Processor Functional Verification

To ensure the quality and correctness of processor designs, processor functional verification plays a critical role. This process aims to confirm that the design meets specified functional and performance requirements. For this task, functional coverage is a key metric, which measures how thoroughly verification tests explore the design's functionality requirements, rather than just measuring code execution paths like code coverage does [5]. Functional coverage tracks those scenarios, events, and sequences specified in the design documentation, ensuring all intended behaviors are tested. The functional coverage models are structured hierarchically by processor design developers into FPs and groups (a group of FPs with relevant functionalities) [5]. An FP typically represents a fine-grained design feature or instruction behavior, such as the execution of a specific branch instruction or the update of CSRs [5]. For example, in a RISC-V processor, FPs may include "jump and link instruction (JAL) execution" or "CSR write protection check." These FPs are grouped into higher level categories (e.g., control flow and memory access) as functional groups (FGs) to cover the intended functional behavior of the processor systematically.

In the literature, there are a number of techniques targeting this task, which fall into two general categories: static formal verification methods [27], [28], [29], [30], [31], [32] and dynamic simulation-based verification methods [8], [9], [10], [23], [25], [33], [34], [35], [36], [37], [38], [39]. The latter, which includes both directed and random testing, is an effective and widely used approach. In particular, directed testing is an engineer-driven approach in which testers carefully craft specific scenarios to validate known functionalities and verify that features work as intended [36]. However, as designs become more complex, directed testing alone becomes costly and insufficient to cover all scenarios, especially those at the periphery of expected behaviors [36]. Random or constraint-random testing, known as CRV, fills this gap by generating a large number of random inputs guided by specific constraints to test the design. This technique allows a broader exploration of the design's functionality space, making it effective for complex designs [40]. However, CRV still leaves a portion of FPs inefficiently covered.

To alleviate this problem, numerous CGF methods have been proposed [7], [8], [9], [10], [23], [38]. CGF is a well-established verification technique that tests processors by adaptively generating random inputs and mutating seed inputs to simulate unpredictable usage scenarios, guided by various coverage metrics. This approach excels at exposing edge cases, making it particularly effective at uncovering hidden design issues and improving reliability before production. However, because these traditional techniques have only limited interaction with and understanding of the functional coverage model, they often hit bottlenecks in functional coverage growth. These uncovered FPs represent the scenarios that are particularly challenging to test comprehensively in processor design.

To improve functional coverage effectively, several other advanced approaches, especially learning-based approaches, have been proposed [21], [22], [33], [41], [42], [43]. For example, Design2Vec [22] constructed control data flow graphs and learned their syntactic and semantic representations using a graph neural network (GNN), which in turn generates test cases based on this representation. LMT [42] improved the coverage capability at last-mile FPs by learning the relevant variables and constraints using random forest (RF) and generative adversarial network (GAN). However, most of them focus on the submodules or abstract representations of the processor design and are not applicable to the processor architecture itself. That is, these methods do not analyze the impact of instruction inputs on the functional coverage from the processor-level view, making it difficult to migrate these methods to the verification of the entire processor design [10]. Particularly, the growing complexity of modern processors and their diverse instruction sets makes improving functional coverage an increasingly urgent challenge.

### *B. LLM-Based Processor Test Generation*

LLMs have rapidly emerged as transformative tools across diverse domains, offering impressive capabilities in natural language and programming language understanding, code generation, and complex reasoning. Generic LLMs, such as ChatGPT [44] and Llama 3 [45], have exhibited superiority in a range of natural language processing tasks. Meanwhile, code-focused LLMs such as CodeLlama [46] are developed to facilitate programming-related tasks. Various prompting strategies have been developed to invoke LLMs effectively, like zero-shot prompting, few-shot learning [47], chain-of-thought (CoT) [48], and RAG [49]. Zero-shot prompting allows the model to perform a verification or test generation task solely based on the given instruction, without any prior examples. Few-shot learning, a representative form of in-context learning (ICL), provides several representative input–output pairs as demonstrations in the prompt, enabling the model to infer the underlying task pattern and generate more accurate or consistent outputs. CoT refers to a series of intermediate natural language reasoning steps that lead to a final output, which enables LLMs to provide more reliable answers by breaking down their reasoning process into clear, sequential steps. RAG enhances existing models by incorporating knowledge from external databases through a common paradigm containing three main steps: information retrieval, data augmentation, and final generation. Different from simple prompt looping, RAG first retrieves relevant information from external sources, which is then used to augment the original prompt, providing the model with a richer context. Finally, the model generates a response with more accurate, contextually relevant, and factually grounded outputs. By leveraging external information, RAG significantly improves the performance of LLMs, particularly in tasks requiring specialized domain knowledge [49].

Within the software domain, LLMs have shown particular promise in generating test cases [18], [50], [51], [52], [53], [54], [55]. By analyzing code and natural language descriptions, LLMs can understand programming structures and logic to generate test cases targeting specific functionalities and edge cases. This automation helps ensure software meets functional requirements more comprehensively than traditional random or manual testing approaches.

Building on LLMs' advances in software testing, researchers are now exploring their applications for processor verification [11], [12], [13], [14]. In general, they instructed LLMs to understand the semantics of uncovered FPs and then generate test cases to cover them. In particular, Zhang et al. [11] proposed an LLM evaluation framework LLM4DV for processor verification that evaluates the effects of different models and prompts on module-level designs. Xiao et al. [12] proposed an LLM-based test method for processors with different ISAs through zero-shot prompting. They demonstrated the method's effectiveness on both traditional processors and emerging domain-specific architectures. ChatFuzz [13], trained on large machine language datasets, leverages LLMs to understand processor language and generate data/control flow entangled yet random machine code sequences to generate input instructions needed for processor verification. VerilogReader [14] used LLMs to understand Verilog code and coverage, and generate test cases for Verilog modules (module-level) through zero-shot prompting.

These LLM-based approaches demonstrate the promising role of LLMs for processor verification, but simply instructing LLMs makes them face significant limitations in practice. In particular, these approaches struggle to improve functional coverage effectively and efficiently due to the lack of in-depth semantic understanding between the functionalities to be explored and the input instructions. To address this, our approach makes better use of ICL strategies by leveraging RAG to improve the ability to model the instructions—FPs mapping and utilizing the advanced CoT to improve the validity of the test cases generated by LLMs.

## III. Flame Framework

Fig. 1 shows the overview of our novel technique FLAME for processor functional verification, which is dedicated to covering more FPs automatically. FLAME is divided into three parts: knowledge base construction, LLM-assisted test generation, and functional-coverage-guided feedback. FLAME begins by collecting extensive processor-design-related information to build a comprehensive knowledge base that provides essential background information. Then, FLAME uses the RAG technique to retrieve information related to the target FPs from the established knowledge base and generate high-quality test cases based on our devised *documents—instructions—programs* CoT. Finally, a functional-coverage-guided feedback mechanism is utilized, where the previously-generated test cases and coverage result

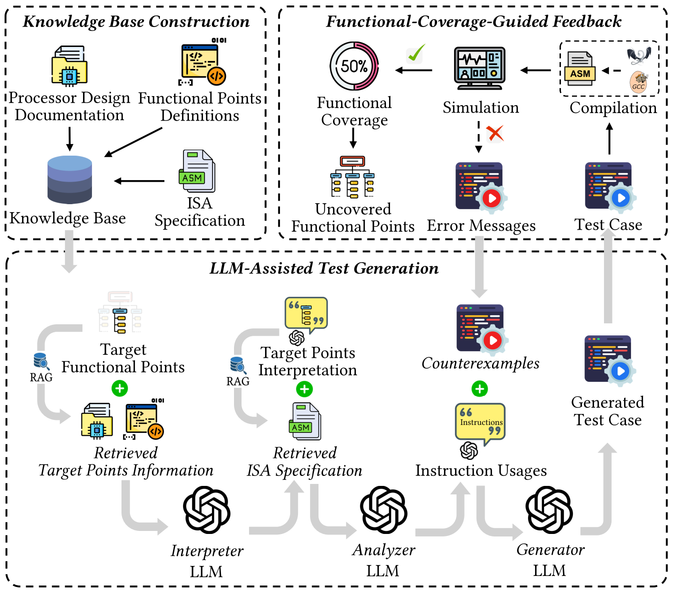
Knowledge Base Construction
Processor Design Documentation
Functional Points Definitions
Knowledge Base
ISA Specification
Functional-Coverage-Guided Feedback
50%
Functional Coverage
Simulation
Compilation
Uncovered Functional Points
Error Messages
Test Case
LLM-Assisted Test Generation
Target Functional Points
RAG
Target Points Interpretation
RAG
Counterexamples
Generated Test Case
Retrieved Target Points Information
Retrieved ISA Specification
Instruction Usages
Interpreter LLM
Analyzer LLM
Generator LLM

Fig. 1. Overview of our proposed FLAME.

information could be provided to LLMs as a reference for the iterative generation process. Note that, due to the cost of LLMs, our technique is not applied to FPs that are easily addressed in practice, specifically those covered efficiently by widely used CRV methods in our work following the existing study [42]. In other words, FLAME focuses on addressing FP bottlenecks in order to achieve cost-effectiveness.

Note that fine-tuning LLMs entails substantial additional costs and may not generalize well across diverse architectures and processor designs. Hence, we focus on employing more efficient prompting techniques through well-designed strategies. While RAG and CoT are common prompting methods in the area of LLMs, their effectiveness is unexplored in the context of processor verification, making their adoption in this challenging domain nontrivial. To address these challenges, we devised a novel approach by carefully leveraging the intrinsic nature of processors rather than relying solely on simple logical reasoning.

## A. Knowledge Base Construction

This phase involves collecting and encoding processor-design-related information to ensure that the following test generation process can leverage comprehensive and structured information. Thoroughly exploiting specification information related to target FPs is crucial for improving complex semantic conditions in uncovered FPs and reducing instruction input space. This information helps reduce the LLM's hallucination threat when invoking RAG, enhancing the relevancy and accuracy of generated responses.

Three types of information related to processor design were collected for knowledge base construction, as shown in Fig. 1: processor design documentation, FPs definitions, and ISA specification. The first two help LLMs understand the processor design under test, while the ISA Specification helps LLMs understand input instructions. Their details are described as follows.

1) *Processor Design Documentation:* These documents provide descriptions of the processor's architecture, functionality, and implementation, serving as the foundational reference for understanding the design and FPs. For example, the IBEX documentation [56] provides information about the coverage plan.
2) *FPs Definitions:* This information contains the definitions for FPs, documenting the design principles and structures behind their implementation. It also outlines the targets for verification, ensuring systematic test coverage across relevant scenarios. For instance, the definitions in [15] primarily record the IBEX micro-architectural events and their behaviors.
3) *ISA Specification:* These specifications define the supported instructions, operational semantics, and encoding formats—essential elements for generating valid and meaningful test cases. For example, this documentation [57] contains the instruction set for RISC-V, detailing the usage of various instructions.

The collected processor-design-related information is systematically processed to facilitate efficient retrieval during LLM-assisted test generation. For design documentation and ISA specification, textual information is divided into chunks using a structured segmentation pipeline that leverages section boundaries, hierarchical headings, and token-length constraints to preserve semantic coherence. A deterministic index is then automatically constructed through a hybrid rule-based and LLM-assisted linking process, where a controlled vocabulary of FP identifiers and instruction names is first generated, and an LLM refines and verifies associations between these identifiers and the most relevant documentation or specification chunks identified by the keywords. Similarly, for FP definitions, source code is segmented into chunks according to the hierarchical structure of FPs and indexed with their related information. This process culminates in the creation of a comprehensive knowledge base, which integrates processor design documentation, FP definitions, and ISA specification.

## B. LLM-Assisted Test Generation

This phase consists of three stages: *interpretation*, *analyzation*, and *generation*, where the LLM serves as interpreter, analyzer, and generator, as shown in Fig. 1. In the first two stages, RAG is employed to retrieve external knowledge from the constructed knowledge base (detailed in Section III-A). This enhances the LLMs' understanding of target FPs and improves the semantic mapping between FPs and instructions, which facilitates addressing the challenges of complex semantic conditions required for achieving nontrivial functional coverage and the extensive search space associated with input instructions.

To improve the validity of generated test cases, we specifically devise a *documents—instructions—programs* CoT for the workflow of three stages: 1) the *interpretation* stage, which will retrieve information about the target FPs and enrich it with LLMs to get detailed information (*documents*) and instruction types; 2) the *analyzation* stage, which will retrieve information about the instructions and analyze it by LLMs based on the functionalities and behaviors of the target FPs to get the specific instruction usages (*instructions*) that cover the FPs; and 3) the *generation* stage, which will generate a C program (*programs*) with the corresponding semantic structure based on the instruction usages from the *analyzation* Stage. This prompting technique exploits the logical reasoning ability of LLMs, thereby improving their performance in test case generation for processor functional verification. Below, we will detail the three stages, respectively.

*1) Interpretation:* At this stage, FLAME first retrieves relevant information from the design documentation and FPs definitions of the target FPs to help the LLM better understand the context. Retrieval is performed through exact keyword lookup using the prebuilt deterministic index constructed during the *knowledge base construction* stage. Since exact matching alone can fail due to naming inconsistencies or incomplete matches in the ISA specification and design documentation, FLAME applies Levenshtein distance [58] to resolve multiple near matches or naming variants and select the most similar documentation chunks. In contrast to cosine similarity, which depends on semantic representations and thus performs poorly on short, lexically similar identifiers [59], Levenshtein distance operates directly on token-level edit operations, making it particularly effective for short FP and instruction names that differ only by a few characters. This retrieval mechanism leverages the offline-constructed index to ensure accurate and consistent mapping between FPs, instruction names, and their corresponding documentation segments, while avoiding the computational cost and stochastic variability of on-the-fly embedding models. These contents, along with the target FPs, are then fed into the LLM for deeper interpretation and improved understanding of the FPs. This stage strengthens the semantic understanding between the target FPs and their associated design documentation, serving as a fundamental component.

*2) Analyzation:* In the second stage, FLAME first segments the LLM's response generated in the first stage, and extracts the keywords related to the instruction types as candidate instructions based on the instruction types defined in the ISA. For each candidate, retrieval from the ISA specification follows the same procedure as in the *interpretation* stage: FLAME first performs exact keyword lookup using the prebuilt deterministic index constructed during the *knowledge base construction* stage, and when multiple near matches or minor naming variants exist, applies Levenshtein distance [59] to select the most similar specification chunks. The retrieved contents, which contain the correct usage of the target instructions, are combined with the detailed FPs information generated in the first stage. The combined information is then fed back into the LLM for deeper analysis and improved understanding of the instructions. By synthesizing the LLM's initial response with the retrieved ISA instruction specifications, the LLM can analyze the target FPs with a focus on identifying key attributes, e.g., the specific instruction types, possible combinations, and applicable operation modes associated with the target points. The outcome of this stage is refined information on instruction types, parameters, and usage scenarios for the target FPs. This enables the LLM to comprehend how the target FPs interact within the design's functional context and establish a precise semantic mapping.

*3) Generation:* In the third stage, FLAME incorporates the specific instruction details obtained in the second stage to generate an effective test case at the source code level (i.e., a C program that can be compiled to instructions for processor verification). Note that FLAME chooses the source code level rather than the instruction level in order to improve test validity, since LLMs have demonstrated superior performance on source code generation [17], [18], [19]. By integrating instructions with the syntactic and operational properties of the C language, the LLM will produce a test case that is explicitly tailored as an input for target FPs. This test case is designed to evaluate target FPs comprehensively by exercising the detailed behaviors, edge cases, and specific conditions identified in previous stages.

## *C. Functional-Coverage-Guided Feedback*

To enhance test efficiency and achieve comprehensive functional coverage, FLAME implements a functional-coverage-guided feedback mechanism. This mechanism involves executing test cases (including compilation and simulation), collecting functional coverage data, selecting target FPs, and utilizing counterexamples to refine the testing process. We now outline the involved key components.

1) *Compilation and Simulation:* To verify the effectiveness of the test cases generated in the prior phase, FLAME simulates them on the processor designs. Initially, the generated C programs are compiled into instruction sequences using an ISA-specific compiler (e.g., RISC-V GCC for RISC-V ISA processors). These instruction sequences are then fed into the processor for simulation using commercial electronic design automation (EDA) tools like Synopsys VCS [60].
2) *Functional Coverage Collection:* To analyze the functional coverage results and guide subsequent test generation, FLAME collects functional coverage information after simulation. During this process, the aforementioned EDA tools generate detailed functional coverage reports, which are automatically parsed to extract functional coverage results. The collected data is analyzed to identify uncovered FPs.
3) *Target FP Selection:* To refine the testing process progressively for achieving comprehensive functional coverage, FLAME establishes a rule for selecting target FPs for the next iteration. In particular, FLAME first calculates the proportion of uncovered FPs within each functional group (defined by design developers based on functionality relevance as presented in Section II-A). The FGs are then sorted based on this proportion, prioritizing the group with the highest number of uncovered FPs. To summarize, after each iteration, the current target functional group is moved to the end of the queue, and the remaining groups are recalculated and resorted based on their uncovered point proportions. This strategy prevents repetitive exploration of a single functional group and ensures that subsequent iterations target new areas, thus improving coverage in a structured and efficient manner.
4) *Counterexample Utilization:* The feedback mechanism plays a crucial role by incorporating coverage information from previously-generated test cases. In particular, to make use of past test cases and provide valuable insights for future iterations, FLAME selects the test cases from the history that can cover FPs within the same functional group as the target uncovered FPs. These selected test cases act as counterexamples, encouraging the LLM to generate more diverse test cases.

In addition, FLAME records information from previously-generated invalid test cases, including error messages and related statements. This information is also utilized as counterexamples during the generation process, helping the LLM avoid generating test cases with similar errors. The counterexamples information is then integrated into the *generation* stage

TABLE I
BASIC INFORMATION ABOUT STUDIED PROCESSOR DESIGNS

| Processor Design | # FG | # FP | # LOC |
|---|---|---|---|
| IBEXv1 | 22 | 2,237 | 151,728 |
| IBEXv2 | 109 | 3,136 | 292,374 |
| CVA6 | 216 | 23,734 | 595,139 |
| CV32E40P | 64 | 775 | 74,068 |

in the form of few-shot learning as shown in Fig. 1. By iteratively incorporating these counterexamples, the LLM is guided to produce more diverse and less error-prone test cases, enhancing the overall testing efficiency and effectiveness.

## IV. Experimental Setup

We formulate three research questions to evaluate FLAME.

1) *RQ1:* How do different LLMs and test case formats perform in FLAME?
2) *RQ2:* How does FLAME perform compared to existing test generation techniques in processor functional verification?
3) *RQ3:* How does each key component of FLAME contribute to the overall effectiveness?

### A. Studied Subjects

*1) Processor Designs Under Test:* To evaluate the performance of FLAME in improving functional coverage, it is essential that the processor designs under test include functional coverage models. These models define FPs and groups, enabling the measurement of functional coverage improvements. Based on this requirement, we collected all the open-source processor designs used in prior work on improving functional coverage [11], [21], [22], including IBEXv1, IBEXv2, CVA6, and CV32E40P.

**IBEX** is a production-quality, open-source 32-bit RISC-V CPU core that supports various extensions [15]. It has undergone extensive verification and multiple tape-outs, making it a reliable benchmark for evaluating verification methods on real-world designs [11], [21]. We used two versions with diverse complexity of functional coverage models (IBEXv1 and IBEXv2). **CVA6** is a 64-bit RISC-V CPU, which implements the RISC-V ISA including integer, multiplication, and division, atomic and compressed extensions, as well as three privilege levels [16], offering insights into verifying high complexity designs [9], [23]. **CV32E40P** is an industrially-supported open-source RISC-V core that implements base integer instructions, featuring both a hardware multiplier and compressed instruction support.

The diversity in design scale, architectural complexity, and feature sets across these processors ensures a comprehensive assessment of FLAME in addressing broad verification challenges. Table I shows the statistical summary of the studied processor designs. The three columns in this table represent the number of FGs, the number of FP, and the number of lines of code (LOC) for each design. For example, CVA6 contains the most FPs (23 734), while CV32E40P contains the least (775).

*2) Target FPs:* Inspired by the existing work [42], we also focused on the FPs that remain uncovered when random test case generation reaches a functional coverage growth bottleneck. As presented in Section III, FLAME focuses on addressing FP bottlenecks for achieving cost-effectiveness.

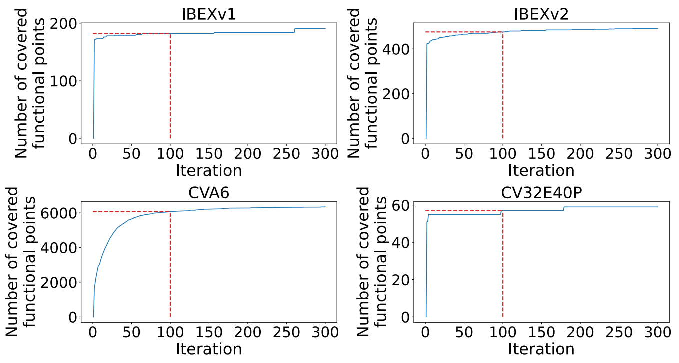

Fig. 2. Functional coverage growth under CRV.

Their research showed that these FPs can be identified by analyzing the growth curve of functional coverage. Following their work, we performed the widely used CRV method [37] (detailed in Section IV-C) on each of the four processor designs to observe the growth of FPs. When the iteration of CRV comes to 100, the growth curve of the four studied processor designs becomes nearly flat, as confirmed in Fig. 2. More specifically, from this figure, the choice of 100 iterations demonstrates a tradeoff between the LLM overhead and overall performance. Using too few iterations may result in unnecessary LLM overhead (which can be addressed by the CRV method), while too many iterations only yield marginal improvements. Accordingly, we ran CRV for 100 iterations per processor (a consistent budget across the four designs for generalizability) and took the uncovered FPs at the end of this run as the target set for FLAME. In other words, after CRV reaches its budgeted saturation, the CRV-covered points (e.g., 6072 FPs on CVA6) are excluded, and the remaining points are defined as the *target FPs* for subsequent evaluation.

### B. Studied LLMs

We selected four popular LLMs, considering source availability (open-source versus closed-source) and their specialization (general-purpose versus code-specific).

1) *GPT-3.5-Turbo* (`GPT3.5`) is one of the influential closed-source LLMs, trained on large amounts of natural language text and code snippets, with reinforcement learning to follow human instructions [44]. We did not choose the latest versions of ChatGPT (i.e., GPT-4 and GPT-4o) due to their high cost, and because FLAME do not require their extended context window. Nevertheless, GPT-3.5-Turbo has shown superiority in various tasks [61], [62], [63].
2) *CodeLlama* is one of the most popular LLMs for code generation and infilling derived from Llama 2 models [64]. In particular, we studied the two sizes: Codellama 7B (`CL7B`) and Codellama 13B (`CL13B`).
3) *Phind CodeLlama* (`PCL34B`) is from Phind models, the fine-tuned versions of CodeLlama-34B, leveraging an internal Phind corpus and instruction-tuning on 1.5B tokens of high-quality programming data [65]. This makes it a state-of-the-art open-source model for code generation.

### C. Baselines

To comprehensively evaluate the effectiveness of FLAME, we compared its performance with five existing test generation techniques, covering a traditional CRV technique, a coverage-guided technique, a latest fuzzing technique, and two latest LLM-based techniques.

1) *CRV* is the kind of constrained random verification method, which is the most widely used in industry and academia [3], [6], [40]. In our verification scenario, we choose the most-studied random instruction generator *RISCV-DV* as an implementation, which is an open-source instruction generator for RISC-V processor verification [37]. *RISCV-DV* requires configuration to use (we used its default configuration in our experiments following the existing work [9], [38]) and generates constrained random test cases based on the constraints of its internal implementation.
2) *DifuzzRTL$_{fc}$* is our adapted version of DifuzzRTL [8], a widely studied CGF method.[1] In our adaptation, based on our target scenario, we used the desired *functional coverage* as the guidance for the mutation process for fair comparison, thus mutating the instruction sequences generated by the CRV and generating instruction sequence mutants as input to the processor design.
3) *Cascade* is the latest processor fuzzing technique proposed by Solt et al. [25]. Due to the CGF's limitation of generating too simple test cases, Cascade constructs test cases with more complex data and control flows by utilizing the instruction set simulator (ISS) to entangle the control flow with the randomized data flow during test case generation. Cascade outputs binary format (ELF) programs as input to the processor design.
4) *LLM-PV$_{ISA}$* and LLM-PV$_{\text{Func}}$, denoted by us, are the two latest LLM-based processor functional verification strategies proposed by Xiao et al. [12]. LLM-PV$_{\text{ISA}}$ takes the functional description of each ISA instruction implemented in the target processor as the prompt to LLMs, while LLM-PV$_{\text{Func}}$ provides LLMs with specific functionality of the target processor. The output of both LLM-PV$_{\text{ISA}}$ and LLM-PV$_{\text{Func}}$ is C programs.

In addition, there are other LLM-based processor verification techniques, including LLM4DV [11], VerilogReader [14], and ChatFuzz [13]. However, LLM4DV and VerilogReader primarily focus on module-level processor verification, whereas our scenario involves full processor design at the architecture level. Furthermore, ChatFuzz requires extensive training datasets, which are unavailable for reproduction. As a result, these methods are not well-suited to our scenario and were not included as baselines in our study.

## D. Evaluation Metrics

We evaluated the performance of FLAME and baselines using these commonly used metrics [8], [9], [10], [11], [21], [42], [66].

1) *Functional Coverage:* The number of target FPs covered during verification, measuring effectiveness.
2) *Pass Rate:* The ratio of successfully executed tests (passing the compilation and simulation) to the total number of generated test cases, reflecting test validity.
3) *Time Reduction:* The reduction in time when achieving the same functional coverage, measuring efficiency.

[1]We did not adapt the state-of-the-art CGF technique, i.e., MorFuzz [9], since it requires significant effort to apply it to our studied processors through communications with its authors.

## E. Implementation and Environment

*1) Studied LLMs:* The open-source LLMs, i.e., Phind CodeLlama (*Phind-CodeLlama-34B-v2* version) [65] and CodeLlama (*CodeLlama-13b-Instruct-hf* and *CodeLlama-7b-Instruct-hf* versions) [46], were downloaded from Huggingface, while the closed-cource GPT-3.5-Turbo (*gpt-3.5-turbo-0125* version) was accessed through its official API [44]. To balance diversity and coherence in the generated test cases, following the existing work [67], [68], [69], [70], and considering the exploratory nature of our task, we set the temperature to 0.7 for these LLMs. The balance is critical for test generation tasks: we expect the LLM to produce correct outputs, yet also explore diverse edge-case scenarios that enhance coverage. The studied LLMs have a maximum context length of 16K tokens. The prompts for LLMs employ universally recognizable delimiters, including symbolic markers and structured field labels, which are naturally interpretable by different model architectures. These delimiters clearly separate prompt segments and maintain a consistent logical structure across models. Since modern LLMs are well trained to recognize such structural keywords, this format ensures reliable parsing and consistent behavior without requiring model-specific adaptations. All the prompt templates can be found on our homepage [26].

*2) Studied Processor Designs:* For the processor-design-related information in knowledge base construction, we collected the documentation of processor designs (including IBEX CPU, CVA6, and CV32E40P), and the FPs definitions in their functional coverage model, as well as their ISA specification (i.e., RISC-V Instruction Set Manual). The targeted RISC-V ISA is RV32IMC by default [15], [16], [20].

*3) Baselines:* We implemented DifuzzRTL$_{\text{fc}}$ based on the DifuzzRTL implementation by replacing its original coverage guidance with our target functional coverage. We adopted the open-source Cascade implementation and extended it to our studied processor designs. We implemented the LLM-PV$_{\text{ISA}}$ and LLM-PV$_{\text{Func}}$ methods based on their paper [12].

More detailed environment specifications are available on our homepage [26] upon request for replication studies.

## V. Results and Analysis

## A. RQ1: Influence of LLMs and Test Formats

*1) Validity Comparison of Test Case Formats:*

*a) Process:* A processor receives input as a sequence of instructions, typically consisting of assembly code. High-level programming languages (e.g., C programs) can be compiled into these instruction sequences. For processor verification, we have two choices: either generating instruction sequences directly or creating C programs and then compiling them into assembly code as part of the verification process. The choice of test case format may influence how LLMs interpret and process information. Therefore, we investigated the effect of the two test case formats on all four studied LLMs (i.e., `GPT3.5`, `CL7B`, `CL13B`, and `PCL34B`) across the four processor designs (i.e., IBEXv1, IBEXv2, CVA6, and CV32E40P).

For each LLM and processor design, we used the fixed prompt template and varied only the command text to steer

TABLE II
PASS RATE COMPARISON OF TEST CASE FORMATS

| Format | PCL34B | CL13B | CL7B | GPT3.5 |
|---|---|---|---|---|
| C Program | 76.60% | 52.51% | 48.95% | 67.40% |
| Instruction | 11.80% | 1.63% | 1.80% | 4.70% |

generation toward either C programs or instruction sequences. For example, for instruction-sequence generation, we used the same template as C-program generation but replaced "C programs" with "instruction sequences." For each setup, we generated an equal number (500) of C programs and instruction sequences as test cases of the processor designs. The pass rate metric was evaluated as the proportion of test cases that successfully passed both compilation and simulation.

*b) Results:* Table II presents the average pass rates for generating C programs and instruction sequences using various LLMs across the four processor designs. As observed, generating C programs consistently achieves higher pass rates compared to generating instruction sequences for all the studied LLMs. For instance, the average pass rates for instruction sequences range from only 1.63%–11.80% across the studied processor designs. In contrast, the average pass rates for C programs range from 67.40% to 76.60%. Our manual investigation of the LLM-generated instructions and execution information revealed that the primary reason for the low pass rates of instruction sequences is their higher likelihood of containing incorrect syntax or unstructured assembly code. This, in turn, results in a higher probability of compilation and simulation failures. The results further confirm the performance of LLMs in generating high-level source code (C programs here) due to training on a considerable amount of source code data. In the future, incorporating assembly code for fine-tuning LLMs may improve the validity of generated instruction sequences. Based on current results, we selected C programs as the test case format for subsequent experiments.

### 2) Effectiveness Comparison of LLMs:

*a) Process:* To examine the impact of LLMs on the effectiveness of FLAME, the same prompt was used for all four LLMs, and the test cases were generated in C programs. All experiments were performed on the same machine under identical conditions, with a fixed duration of 48 h, encompassing both test case generation and simulation. We analyzed the number of target FPs covered by the generated test cases and measured the pass rate to evaluate test validity.

*b) Results:* Fig. 3 presents the comparative results across different LLMs in terms of the number of covered target functional points. Among the four LLMs, the open-source PCL34B demonstrates the best performance (except on IBEXv2, where all four LLMs perform similarly), covering the most target functional points, within the 48-h time budget. It indicates that PCL34B outperforms the other open-source Code Llama-series models (i.e., CL13B and CL7B) and the closed-source GPT3.5 across most processor designs, because PCL34B was fine-tuned on the Phind dataset and achieved 73.8% pass@1 on HumanEval (outperforming GPT-4 and GPT-3.5) [65], highlighting the importance of fine-tuning LLMs specifically on high-quality programming-related data.

In terms of time efficiency, PCL34B achieves the maximum functional coverage obtained by the other models within significantly shorter durations. For instance, on IBEXv1, CVA6, and CV32E40P, PCL34B matches the maximum coverage achieved by GPT3.5 (56 on IBEXv1 and 10 on CV32E40P) and CL7B (967 on CVA6) in just 13.50, 9.29, and 6.98 h, respectively. This represents time reductions of 71.88%, 80.65%, and 85.46%, highlighting the superior efficiency of PCL34B in these scenarios.

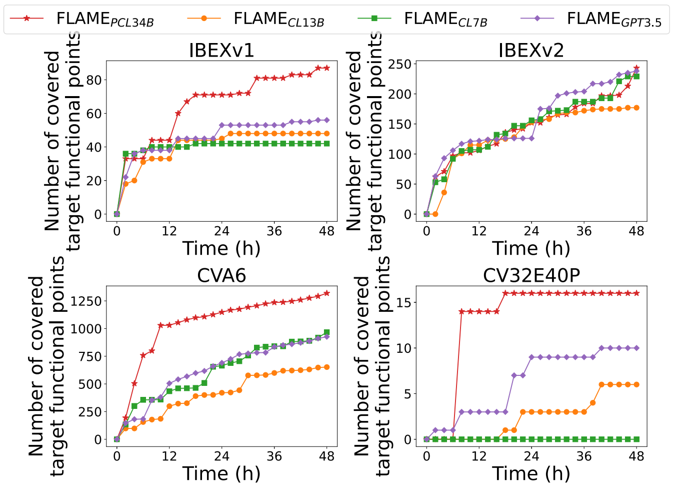

Fig. 3. Functional coverage comparison across LLMs.

TABLE III
NUMBER OF VALID TEST CASES GENERATED BY LLMS

| Processor Design | PCL34B | CL13B | CL7B | GPT3.5 |
|---|---|---|---|---|
| IBEXv1 | 554 | 313 | 397 | 1,087 |
| IBEXv2 | 178 | 88 | 114 | 219 |
| CVA6 | 314 | 192 | 229 | 331 |
| CV32E40P | 477 | 306 | 333 | 995 |

TABLE IV
PASS RATE COMPARISON OF DIFFERENT LLMS

| Processor Design | PCL34B | CL13B | CL7B | GPT3.5 |
|---|---|---|---|---|
| IBEXv1 | 69.42% | 61.49% | 53.43% | 76.01% |
| IBEXv2 | 67.68% | 37.77% | 35.40% | 47.61% |
| CVA6 | 52.86% | 36.52% | 36.46% | 49.48% |
| CV32E40P | 68.53% | 39.79% | 32.62% | 63.74% |
| Average | **64.62%** | 43.89% | 39.48% | 59.21% |

Table III shows the number of valid test cases generated by different LLMs. Among the LLMs, GPT3.5 generates the largest number of valid test cases (e.g., 1087 and 995 for IBEXv1 and CV32E40P). However, the number of valid test cases generated does not directly correlate with improvements in functional coverage. For instance, the PCL34B model does not generate or execute the largest number of valid test cases on any of the processor designs, yet it achieves the most significant improvement in functional coverage effectiveness.

Table IV further presents the performance of LLMs in terms of pass rate. The results show that the test cases generated by PCL34B achieved the highest average pass rates among the four processor designs, with rates of 64.62%, 43.89%, 39.48%, and 59.21%, respectively. Considering the tradeoff between effectiveness (functional coverage and pass rate) and efficiency (number of valid test cases generated), we chose the best-performing PCL34B model for subsequent experiments.

*RQ1 Summary:* Key results show that generating test cases at the source code level (i.e., C programs) using LLMs achieves a higher pass rate. In addition, PCL34B achieves the best tradeoff among the four LLMs, reaching the highest number of FPs within the controlled time and reducing the time to reach the same number by up to 85.46%.

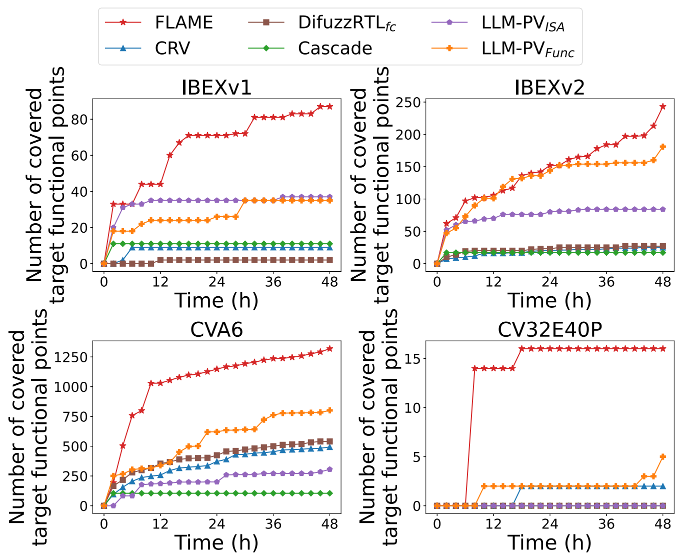

Fig. 4. Functional coverage comparison between techniques.

### B. RQ2: Comparison Between FLAME Versus Existing Techniques

*1) Process:* To compare FLAME with existing processor verification techniques, we selected five typical or state-of-the-art techniques as baselines, including CRV, DifuzzRTL$_{fc}$, Cascade, LLM-PV$_{ISA}$, and LLM-PV$_{Func}$, as detailed in Section IV-C. We used the optimal experiment setting based on the conclusion from Section V-A, i.e., we chose `PCL34B` as the LLM and used C programs as the format of test case generation (for both FLAME and the two LLM-based baselines). Each technique was executed under identical conditions on the same set of target FPs, with a fixed time budget of 48 h/run. We calculated the number of covered target FPs and the pass rate of generated test cases to assess their effectiveness and validity.

*2) Results:* Fig. 4 shows the number of covered target FPs achieved by each technique over time, highlighting the accelerated coverage gains obtained through FLAME. For the studied processor designs, FLAME achieved the best functional coverage (i.e., the largest number of target FPs), outperforming the best baseline (i.e., LLM-PV$_{ISA}$ on IBEXv1, LLM-PV$_{Func}$ on IBEXv2, LLM-PV$_{Func}$ on CVA6, and LLM-PV$_{Func}$ on CV32E40P) by 135.14%, 34.25%, 64.67%, and 220.00%, respectively. Also, FLAME achieved the maximum coverage obtained by these baselines within significantly shorter times: 6.66, 34.27, 8.52, and 6.93 h for the four processor designs, representing time reductions of 86.13%, 28.60%, 82.25%, and 85.56%, respectively. These results indicate that our proposed technique, FLAME, demonstrates superior effectiveness and efficiency in addressing bottleneck FPs.

The existing processor fuzzing techniques, i.e., DifuzzRTL$_{fc}$ and Cascade, have demonstrated limited effectiveness in improving functional coverage. A likely reason is their inability to capture the high-level design intent underlying FPs. As a result, the test cases they generate often fail to exercise the specific instructions and combinations associated with these FPs. Particularly, our study focuses on the FPs that remain uncovered when random test generation reaches a plateau in functional coverage growth, indicating that these uncovered points are much more challenging. The poor performance of these studied baselines on them further underscores the

TABLE V
NUMBER OF VALID TEST CASES GENERATED BY TECHNIQUES

| Processor Design | CRV | D$_{fc}$ [1] | C [1] | LLM-PV$_{ISA}$ | LLM-PV$_{Func}$ | FLAME |
|---|---|---|---|---|---|---|
| IBEXv1 | 1,165 | 899 | 8 | 1,114 | 1,044 | 554 |
| IBEXv2 | 761 | 708 | 93 | 408 | 244 | 178 |
| CVA6 | 753 | 748 | 100 | 22 | 133 | 314 |
| CV32E40P | 239 | 219 | 31 | 1,144 | 973 | 477 |

[1] D$_{fc}$ denotes DifuzzRTL$_{fc}$, and C denotes Cascade.

TABLE VI
PASS RATE COMPARISON OF FLAME AND BASELINES

| Processor Design | DifuzzRTL$_{fc}$ | LLM-PV$_{ISA}$ | LLM-PV$_{Func}$ | FLAME |
|---|---|---|---|---|
| IBEXv1 | 98.81% | 66.77% | 53.54% | 69.42% |
| IBEXv2 | 95.06% | 66.62% | 54.71% | 67.68% |
| CVA6 | 74.02% | 5.42% | 26.13% | 52.86% |
| CV32E40P | 91.83% | 65.48% | 53.17% | 68.53% |

TABLE VII
TOKENS REQUIRED FOR LLM-BASED TECHNIQUES

| Processor Design | LLM-PV$_{ISA}$ | | LLM-PV$_{Func}$ | | FLAME | |
|---|---|---|---|---|---|---|
| | AVG | MAX | AVG | MAX | AVG | MAX |
| IBEXv1 | 396.12 | 786.00 | 451.69 | 1,409.00 | 3,819.45 | 6,734.00 |
| IBEXv2 | 392.49 | 702.00 | 461.13 | 748.00 | 6,606.80 | 9,753.00 |
| CVA6 | 397.24 | 613.00 | 441.63 | 772.00 | 3,776.73 | 8,633.00 |
| CV32E40P | 394.00 | 660.00 | 439.30 | 1,484.00 | 5,220.60 | 10,547.00 |
| Average | 394.96 | 690.25 | 448.44 | 1,103.25 | 4,855.90 | 8,916.75 |

difficulty of our target problem. Similarly, as presented in the existing work [25], the type of functional coverage we target is more sophisticated, requiring significantly greater effort to enhance. The results show that although these existing processor fuzzing techniques can improve other coverage metrics (e.g., multiplexer coverage and control register coverage) [8], [25], they do not necessarily yield similar improvements in functional coverage.

Tables V and VI present the number of valid test cases and the pass rate for each studied technique. As shown in Table V, FLAME generates a relatively smaller number of valid test cases across the four studied processor designs compared to most of the baselines (except Cascade). Notably, for IBEXv1 and IBEXv2, FLAME achieved the largest number of FPs using the smallest number of test cases (except Cascade), with 554 and 178 test cases, respectively. Cascade aims to generate significantly more complex test cases [25], which consequently require substantially longer execution times. As a result, within the fixed time budget of our experiments, all evaluated processors were able to execute only a limited number of valid test cases produced by Cascade.

For the validity of test case generation, the test cases generated by the CRV method (*RISCV-DV*) and Cascade are all correct (100% in pass rate), due to the reliance on manual configuration as well as the rule-based nature. The DifuzzRTL$_{fc}$ is also easier to generate valid test cases, due to the fact that they were mutated from valid seeds of CRV. The pass rate of DifuzzRTL$_{fc}$ is lower than that of CRV because the mutation process leads to the creation of incorrect test cases.

Nonetheless, as shown in Table VI, FLAME achieves the highest pass rate for generating test cases across all processor designs when compared to the other two LLM-based techniques (LLM-PV$_{ISA}$ and LLM-PV$_{Func}$), with pass rates of 69.42%, 67.68%, 52.86%, and 68.53%. While LLM-PV$_{\text{ISA}}$ demonstrates a pass rate slightly lower than FLAME on IBEXv1, IBEXv2, and CV32E40P, its performance drops significantly on CVA6 (the most complicated processor design), with a pass rate of only 5.42%. On the other hand, LLM-PV$_{\text{Func}}$ shows even lower pass rate performance on IBEXv1, IBEXv2, and CV32E40P, and its pass rate on CVA6 is only 26.13%. These findings suggest that FLAME is comparatively capable of generating a smaller number of test cases while maintaining relatively high validity. Particularly, FLAME is affected less by the complexity of processor designs in terms of pass rate than the other two LLM-based techniques.

We further assessed the token usage of all LLM calls to gain an understanding of the computational and monetary costs. Table VII reports the average and maximum number of tokens per iteration for FLAME and the LLM-based baselines. Because FLAME employs a three-stage CoT prompt construction, it naturally incurs higher token consumption than baseline methods that rely on nearly zero-shot prompting. Nevertheless, its average token usage (4.9K per iteration) remains modest relative to modern inference capacities (16K context window for LLMs). The additional token cost primarily reflects the deeper reasoning context encoded in the multistage prompts, which in turn yields higher-quality test generation and improved coverage (up to 220.00%), demonstrating that FLAME achieves a favorable balance between token efficiency and coverage effectiveness.

*RQ2 Summary:* FLAME demonstrates the best functional coverage improvement compared to the baselines and significantly reduces the time required to achieve the same functional coverage, with a reduction of up to 86.13%. Furthermore, FLAME tends to generate a smaller number of test cases while maintaining relatively high validity.

### C. RQ3: Contributions of Key Components in FLAME

*1) Process:* FLAME consists of two main components: 1) the constructed knowledge base, which is used by RAG to retrieve context information of target FPs during the test generation process, and 2) the functional-coverage-guided feedback mechanism, which enhances testing efficiency. The constructed knowledge base is derived from three key sources: 1) processor design documentation; 2) functional points definitions; and 3) ISA specification. The functional-coverage-guided feedback mechanism comprises two primary components: 4) feedback functional coverage information and 5) previously-generated test cases as counterexamples. These five components [1)~5)] collectively form the overall representation and functionality of FLAME, enabling it to improve the effectiveness and efficiency of processor functional verification.

To evaluate the contributions of these key components in FLAME, we constructed five variants by systematically removing one component at a time and measuring the performance, and additionally included the widely used CRV baseline for practical comparison. 1) *FLAME$_{noDOC}$* (which removed the processor design documentation information from FLAME); 2) *FLAME$_{noDEF}$* (which removed the FPs definitions information from FLAME); 3) *FLAME$_{noISA}$* (which removed the ISA specification information from FLAME); 4) *FLAME$_{noFB}$* (which removed the functional-coverage-guided feedback mechanism from FLAME); and 5) *FLAME$_{noCE}$* (which removed the previously-generated test cases as counterexamples from FLAME). The other experiment settings were the same as those in Section V-B. Their performance was assessed using two key metrics: functional coverage and pass rate of generated test cases. This systematic analysis reveals the impact of each component on the overall effectiveness and validity of FLAME and assesses whether the ablated versions still outperform the commonly used industry baseline.

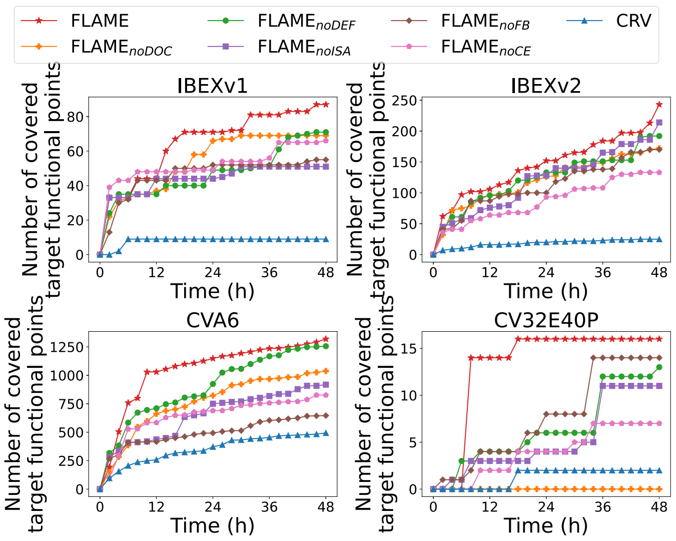

Fig. 5. Functional coverage comparison across variants.

TABLE VIII
NUMBER OF LOC OF FP DEFINITIONS IN PROCESSOR DESIGNS

| Processor Design | Average LOC | Maximum LOC |
|---|---|---|
| IBEXv1 | 78.00 | 80.00 |
| IBEXv2 | 308.58 | 365.00 |
| CVA6 | 44.46 | 145.00 |
| CV32E40P | 14.84 | 36.00 |

*2) Results:* Fig. 5 presents the comparison results between FLAME and its five variants. The results show varying degrees of decline after the removal of individual components; however, these variants still outperform CRV (except FLAME$_{\text{noDOC}}$ on CV32E40P). In particular, the removal of certain components has the most significant negative impact on functional coverage improvement for different processor designs. For instance, removing the ISA specification results in a decrease of 36 FPs on IBEXv1, while the removal of counterexamples leads to a reduction of 110 FPs on IBEXv2. Similarly, removing the feedback mechanism causes a loss of 673 FPs on CVA6, and the removal of processor design documentation decreases FPs by 16 on CV32E40P. As shown in Table VIII, we observe that CV32E40P contains significantly fewer lines of code (LOC) in its FP definitions (14.84 on average and 36.00 on maximum), likely providing limited contextual information for LLMs' interpretation. Consequently, the design documentation plays a more critical role in conveying the semantics and relationships of FPs and their corresponding instructions. When this supplementary documentation is removed, the LLM lacks sufficient context to accurately interpret the target FPs, potentially leading to a more pronounced drop in functional coverage. This indicates different qualities of various kinds of information within different designs and also demonstrates the stable performance of FLAME across diverse process designs.

Tables IX and X present the number of valid test cases generated by FLAME and its variants, along with their cor-

TABLE IX

NUMBER OF VALID TEST CASES GENERATED BY FLAME AND VARIANTS

| Processor Design | $F_{noDOC}$[1] | $F_{noDEF}$ | $F_{noISA}$ | $F_{noFB}$ | $F_{noCE}$ | FLAME |
|---|---|---|---|---|---|---|
| IBEXv1 | 400 | 416 | 529 | 593 | 651 | 554 |
| IBEXv2 | 231 | 178 | 192 | 154 | 191 | 178 |
| CVA6 | 299 | 324 | 332 | 392 | 186 | 314 |
| CV32E40P | 853 | 508 | 577 | 630 | 613 | 477 |

[1] F denotes FLAME.

TABLE X

PASS RATE COMPARISON OF FLAME AND VARIANTS

| Processor Design | $F_{noDOC}$[1] | $F_{noDEF}$ | $F_{noISA}$ | $F_{noFB}$ | $F_{noCE}$ | FLAME |
|---|---|---|---|---|---|---|
| IBEXv1 | 60.88% | 60.03% | 42.90% | 66.26% | 61.88% | 69.42% |
| IBEXv2 | 67.15% | 58.75% | 47.52% | 64.17% | 55.85% | 67.68% |
| CVA6 | 55.58% | 57.65% | 38.69% | 73.96% | 24.73% | 52.86% |
| CV32E40P | 79.13% | 72.16% | 50.88% | 72.83% | 54.34% | 68.53% |

[1] F denotes FLAME.

responding pass rates. As shown in Table IX, FLAME and its variants generate a comparable number of test cases across the four studied processor designs. Table X demonstrates that the variants FLAME$_{\text{noISA}}$ and FLAME$_{\text{noCE}}$ exhibit significant declines in pass rates. For IBEXv1, IBEXv2, and CV32E40P, the removal of the ISA specification has the most substantial negative impact on pass rates, demonstrating the importance of learning correct usages from specifications in general. Conversely, on CVA6, the removal of counterexamples results in the largest reduction in the pass rate, indicating the necessity of learning to generate valid test cases from counterexamples for the most complicated processor design. Although FLAME$_{\text{noDOC}}$, FLAME$_{\text{noDEF}}$, and FLAME$_{\text{noFB}}$ generated relatively more valid test cases than FLAME on CVA6 and CV32E40P, we observe that the test cases produced by these variants are overly simplistic, resulting in a smaller number of achieved target FPs.

*RQ3 Summary:* The experiments show that each component of FLAME has an important role in the overall framework's effect on functional coverage enhancement. In addition, the two modules *ISA specification* and *counterexamples* in FLAME contribute the most to the overall pass rate of test case generation.

## VI. DISCUSSION

To gain deeper insights into the robustness and extensibility of FLAME, we designed two complementary analyses addressing its temporal scalability and environmental generality, which often vary across scenarios in practice. In particular, we 1) examined how FLAME behaves under an extended testing time budget to evaluate its convergence characteristics and long-term scalability, and 2) assessed its sensitivity to compiler variation to understand whether compilation toolchains materially influence functional coverage outcomes.

### A. Scalability Overextended Testing Time

The timing scalability is crucial in industrial verification workflows, where testing budgets often vary dynamically, and methods must retain efficiency across both short and prolonged verification cycles. As shown in Fig. 4, both FLAME and

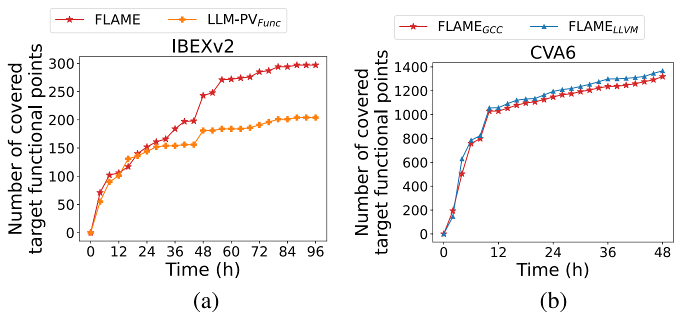

Fig. 6. Extended experimental results. (a) Functional coverage comparison between FLAME and LLM-PV$_{\text{Func}}$ on IBEXv2 in 96 h. Functional coverage comparison between FLAME using GCC and LLVM as compiler on CVA6.

LLM-PV$_{\text{Func}}$ continue to increase the number of covered target FPs within the 48-h window on IBEXv2, suggesting that this budget may be insufficient to assess saturation or relative performance. To investigate whether FLAME maintains its advantage beyond the time limit, we extended the verification process to 96 h [see Fig. 6(a)], where the baseline reached clear saturation. While eventual convergence cannot be ruled out totally, FLAME not only sustained stable growth but also continued to outperform LLM-PV$_{\text{Func}}$, ultimately achieving higher final functional coverage. This demonstrates FLAME's ability to efficiently exploit additional verification time, maintaining consistent growing progress.

### B. Generality Across Compiler Environments

Another factor that may influence FLAME's performance is the choice of compiler, as the LLM-generated test programs are written in C and compiled into executable binaries before simulation. Different compiler toolchains (e.g., GCC and LLVM) may optimize instruction generation differently, potentially affecting instruction distribution and thus impacting functional coverage. To evaluate whether such differences materially influence the FLAME's performance, we conducted an additional experiment using the CVA6 processor as a representative case, comparing results obtained from the LLVM toolchain with those from GCC-based compilation. As shown in Fig. 6(b), both configurations produced highly consistent coverage curves, with FLAME$_{\text{GCC}}$ and FLAME$_{\text{LLVM}}$ achieving 1319 and 1367 covered FPs within 48 h, respectively, which has only a difference of less than 4%, suggesting that the coverage achieved by FLAME is largely insensitive to compiler-specific code generation details. This demonstrates the strong generality of FLAME across compilation environments, reinforcing its robustness in practical verification pipelines that may employ different toolchains.

## VII. THREATS TO VALIDITY

### A. External Threat

This limitation arises from the restricted diversity of processor designs, which may constrain the generalizability of the results to other processor designs with different architectures or custom extensions. While we conducted comprehensive experiments on four widely used open-source processors, these do not fully represent processor designs with diverse ISAs, leaving certain architectural variations untested. Addressing larger and more intricate instruction sets may pose a significant challenge, especially in terms of computational overhead with

LLMs. Drawing inspiration from the research on LLMs in code generation [71], [72], [73], one potential method for relieving LLM's overhead could involve using LLMs to effectively summarize processor information, thereby optimizing prompt constraints. We take it as our future work.

### B. Internal Threat

One of these limitations stems from baseline implementation details and chosen evaluation metrics that may bias performance comparisons. To address this, we strictly adhered to official documentation for baseline reproduction and employed widely accepted metrics to ensure validity and reliability. The configuration settings of LLMs may also introduce potential biases. To mitigate this, we followed established practices [67], [68], [69], [70] by adopting a temperature of 0.7 to encourage diversity in the generated outputs while preserving coherence, which is an appropriate tradeoff for code-related generation tasks that require both correctness and exploratory coverage. To further improve reproducibility, we fixed random seeds and model versions and used a preprocessed, deterministic knowledge base, ensuring consistent retrieval behavior across runs. The third internal threat concerns potentially invalid instructions in LLM-generated sequences. To address this, our simulation harness detects such cases by raising exceptions, terminating execution, recording results, and continuing subsequent iterations. While ensuring evaluation robustness, our study focuses on RV32IMC functional coverage (excluding vector extensions), unlike prior RVV exception-based negative testing [74], and extending our framework to such scenarios remains future work.

### C. Construct Threat

This limitation arises from the reliance on CRV techniques for collecting target FPs, which may not comprehensively capture all critical scenarios. Nevertheless, this approach aligns with the method adopted in the recent study [42]. Another potential threat arises from the hardware configuration differences between LLM-based and non-LLM methods, which could affect execution time measurements. To mitigate this, we deployed only the LLM inference component on GPUs, which is necessary for efficient decoding and meeting memory requirements, while keeping all other pipeline stages on CPUs, consistent with the non-LLM baselines. Our experiments followed each method's standard and practical setup without applying any unfair acceleration. Moreover, since our analysis emphasizes end-to-end verification efficiency rather than raw inference speed, the comparison remains fair and reflects each paradigm's practical computational cost in realistic deployment scenarios.

## VIII. Conclusion

We propose a novel LLM-based test generation framework, FLAME, to cover more FPs in processor verification automatically. By leveraging RAG, CoT, and a functional-coverage-guided feedback mechanism, FLAME establishes semantic mappings between FPs and instructions, enabling the iterative generation of valid and effective test cases. Evaluation of four widely used processor designs demonstrates that FLAME surpasses baselines in functional coverage improvement while significantly reducing the time required to achieve the same functional coverage. In addition, ablation analysis underscores the critical contribution of each component to the overall effectiveness. Future work will focus on expanding the evaluation to encompass a broader range of processor designs and exploring more efficient LLM strategies for further enhancement.

## Acknowledgment

The authors would like to thank the anonymous reviewers for their constructive suggestions to help improve the quality of this article. They confirm that no generative AI tools were used in the preparation of this article.

## References

[1] J. Nurmi, *Processor Design: System-on-Chip Computing for ASICs and FPGAs*. Cham, Switzerland: Springer, 2007.

[2] H. D. Foster, "Trends in functional verification: A 2014 industry study," in *Proc. 52nd ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2015, pp. 1–6.

[3] I. Wagner, *Post-Silicon and Runtime Verification for Modern Processors*. Cham, Switzerland: Springer, 2014.

[4] K. Laeufer, V. Iyer, D. Biancolin, J. Bachrach, B. Nikolić, and K. Sen, "Simulator independent coverage for RTL hardware languages," in *Proc. 28th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst., Volume 3*, Mar. 2023, pp. 606–615.

[5] A. Piziali, *Functional Verification Coverage Measurement and Analysis*. Cham, Switzerland: Springer, 2007.

[6] Y. Naveh et al., "Constraint-based random stimuli generation for hardware verification," in *Proc. AAAI*, 2006, pp. 1720–1727.

[7] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen, "RFUZZ: Coverage-directed fuzz testing of RTL on FPGAs," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2018, pp. 1–8.

[8] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee, "DifuzzRTL: Differential fuzz testing to find CPU bugs," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2021, pp. 1286–1303.

[9] J. Xu, Y. Liu, S. He, H. Lin, Y. Zhou, and C. Wang, "MorFuzz: Fuzzing processor via runtime instruction morphing enhanced synchronizable co-simulation," in *Proc. USENIX Secur.*, 2023, pp. 1307–1324.

[10] S. Canakci et al., "ProcessorFuzz: Processor fuzzing with control and status registers guidance," in *Proc. IEEE Int. Symp. Hardw. Oriented Secur. Trust (HOST)*, May 2023, pp. 1–12.

[11] Z. Zhang et al., "LLM4DV: Using large language models for hardware test stimuli generation," 2023, *arXiv:2310.04535*.

[12] C. Xiao et al., "LLM-based processor verification: A case study for neuromorphic processor," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2024, pp. 1–6.

[13] M. Rostami, M. Chilese, S. Zeitouni, R. Kande, J. Rajendran, and A.-R. Sadeghi, "Beyond random inputs: A novel ML-based hardware fuzzing," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2024, pp. 1–6.

[14] R. Ma et al., "VerilogReader: LLM-aided hardware test generation," 2024, *arXiv:2406.04373*.

[15] (2024). *Ibex RISC-V Core*. [Online]. Available: https://github.com/lowRISC/ibex

[16] F. Zaruba and L. Benini, "The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-GHz 64-bit RISC-V core in 22-nm FDSOI technology," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 27, no. 11, pp. 2629–2640, Nov. 2019.

[17] A. Mastropaolo et al., "On the robustness of code generation techniques: An empirical study on GitHub copilot," in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng. (ICSE)*, May 2023, pp. 2149–2160.

[18] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation," in *Proc. NeurIPS*, 2023, pp. 21558–21572.

[19] X. Du et al., "Evaluating large language models in class-level code generation," in *Proc. IEEE/ACM 46th Int. Conf. Softw. Eng.*, Apr. 2024, pp. 1–13.

[20] M. Gautschi et al., "Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 25, no. 10, pp. 2700–2713, Oct. 2017.

[21] A. Nazi et al., "Adaptive test generation for fast functional coverage closure," in *Proc. DVCON*, 2022. [Online]. Available: https://dvcon-proceedings.org/wp-content/uploads/Adaptive-Test-Generationfor-Fast-Functional-Coverage-Closure-1.pdf

[22] S. Vasudevan et al., "Learning semantic representations to verify hardware designs," in *Proc. NeurIPS*, vol. 34, 2021, pp. 23491–23504.
[23] A. Tyagi et al., "TheHuzz: Instruction fuzzing of processors using golden-reference models for finding software-exploitable vulnerabilities," in *Proc. USENIX Secur.*, 2022, pp. 3219–3236.
[24] C. Chen et al., "Xuantie-910: A commercial multi-core 12-stage pipeline out-of-order 64-bit high performance RISC-V processor with vector extension: Industrial product," in *Proc. ACM/IEEE 47th Annu. Int. Symp. Comput. Archit. (ISCA)*, May 2020, pp. 52–64.
[25] F. Solt, K. Ceesay-Seitz, and K. Razavi, "Cascade: CPU fuzzing via intricate program generation," in *Proc. USENIX Secur.*, 2024, pp. 5341–5358.
[26] (2025). *Homepage*. [Online]. Available: https://superpung.github.io/flame
[27] A. Jayasena and P. Mishra, "HIVE: Scalable hardware-firmware co-verification using scenario-based decomposition and automated hint extraction," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 43, no. 10, pp. 3278–3291, Oct. 2024.
[28] P. Mishra and N. Dutt, "Specification-driven directed test generation for validation of pipelined processors," *ACM Trans. Design Autom. Electron. Syst.*, vol. 13, no. 3, pp. 1–36, Jul. 2008.
[29] H.-M. Koo and P. Mishra, "Functional test generation using property decompositions for validation of pipelined processors," in *Proc. Design Autom. Test Eur. Conf.*, 2006, pp. 1–6.
[30] R. Mukherjee, M. Purandare, R. Polig, and D. Kroening, "Formal techniques for effective co-verification of hardware/software co-designs," in *Proc. 54th ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2017, pp. 1–6.
[31] K. Laeufer, B. Fajardo, A. Ahuja, V. Iyer, B. Nikolić, and K. Sen, "RTL-repair: Fast symbolic repair of hardware design code," in *Proc. 29th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Apr. 2024, pp. 867–881.
[32] A. Ferraiuolo, R. Xu, D. Zhang, A. C. Myers, and G. E. Suh, "Verification of a practical hardware security architecture through static information flow analysis," in *Proc. Twenty-Second Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Apr. 2017, pp. 555–568.
[33] C. Ioannides and K. I. Eder, "Coverage-directed test generation automated by machine learning - a review," *ACM Trans. Design Autom. Electron. Syst.*, vol. 17, no. 1, pp. 1–21, Jan. 2012.
[34] W. Chen, L.-C. Wang, J. Bhadra, and M. Abadir, "Simulation knowledge extraction and reuse in constrained random processor verification," in *Proc. 50th ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, May 2013, pp. 1–6.
[35] K. Campbell, L. He, L. Yang, S. Gurumani, K. Rupnow, and D. Chen, "Debugging and verifying SoC designs through effective cross-layer hardware–software co-simulation," in *Proc. 53rd Annu. Design Autom. Conf.*, Jun. 2016, pp. 1–6.
[36] A. Jayasena and P. Mishra, "Directed test generation for hardware validation: A survey," *ACM Comput. Surv.*, vol. 56, no. 5, pp. 1–36, May 2024.
[37] C. Alliance. (2024). *RISC-V DV: A RISC-V Randomized Instruction Generator*. [Online]. Available: https://github.com/chipsalliance/riscv-dv
[38] Y. Xu, S. Wang, D. Tang, N. Sun, and Y. Bao, "PathFuzz: Broadening fuzzing horizons with footprint memory for CPUs," in *Proc. 61st ACM/IEEE Design Autom. Conf.*, Jun. 2024, pp. 1–6.
[39] G. Zhang, P. Wang, T. Yue, D. Liu, Y. Guo, and K. Lu, "Instiller: Toward efficient and realistic RTL fuzzing," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 43, no. 7, pp. 2177–2190, Jul. 2024.
[40] N. Kitchen and A. Kuehlmann, "Stimulus generation for constrained random simulation," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design*, Nov. 2007, pp. 258–265.
[41] P. Mishra and N. Dutt, "Functional coverage driven test generation for validation of pipelined processors," in *Proc. Design, Autom. Test Eur.*, 2005, pp. 678–683.
[42] M. Yan et al., "Achieving last-mile functional coverage in testing chip design software implementations," in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng., Softw. Eng. Pract. (ICSE-SEIP)*, May 2023, pp. 343–354.
[43] Z. Aref, R. Suvarna, B. Hughes, S. Srinivasan, and N. B. Mandayam, "Advanced reinforcement learning algorithms to optimize design verification," in *Proc. 61st ACM/IEEE Design Autom. Conf.*, Jun. 2024, pp. 1–6.
[44] OpenAI. (2022). *ChatGPT: Optimizing Language Models for Dialogue*. [Online]. Available: https://openai.com/index/chatgpt
[45] A. Dubey et al., "The llama 3 herd of models," 2024, *arXiv:2407.21783*.
[46] B. Roziére et al., "Code llama: Open foundation models for code," 2023, *arXiv:2308.12950*.
[47] P. Sahoo, A. K. Singh, S. Saha, V. Jain, S. Mondal, and A. Chadha et al., "A systematic survey of prompt engineering in large language models: Techniques and applications," 2024, *arXiv:2402.07927*.
[48] J. Wei et al., "Chain-of-thought prompting elicits reasoning in large language models," in *Proc. NeurIPS*, 2022, pp. 24824–24837.
[49] P. Lewis et al., "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Proc. NeurIPS*, vol. 33, 2020, pp. 9459–9474.
[50] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models: Survey, landscape, and vision," *IEEE Trans. Softw. Eng.*, vol. 50, no. 4, pp. 911–936, Apr. 2024.
[51] X. Hou et al., "Large language models for software engineering: A systematic literature review," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 8, pp. 1–79, Nov. 2024.
[52] A. M. Dakhel, A. Nikanjam, V. Majdinasab, F. Khomh, and M. C. Desmarais, "Effective test generation using pre-trained large language models and mutation testing," *Inf. Softw. Technol.*, vol. 171, Jul. 2024, Art. no. 107468.
[53] Y. Chen, Z. Hu, C. Zhi, J. Han, S. Deng, and J. Yin, "ChatUniTest: A framework for LLM-based test generation," in *Proc. 32nd ACM Int. Conf. Found. Softw. Eng.*, Jul. 2024, pp. 572–576.
[54] C. Yang, J. Chen, B. Lin, Z. Wang, and J. Zhou, "Advancing code coverage: Incorporating program analysis with large language models," *ACM Trans. Softw. Eng. Methodol.*, Jul. 2025.
[55] L. Yang et al., "On the evaluation of large language models in unit test generation," in *Proc. 39th IEEE/ACM Int. Conf. Automated Softw. Eng.*, Oct. 2024, pp. 1607–1619.
[56] (2024). *Ibex Documentation*. [Online]. Available: https://ibex-core.readthedocs.io/en/latest
[57] (2024). *The RISC-V Instruction Set Manual*. [Online]. Available: https://github.com/riscv/riscv-isa-manual
[58] Y. Li et al., "A normalized Levenshtein distance metric," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 29, no. 6, pp. 1091–1095, Jun. 2007.
[59] Y. Song and D. Roth, "Unsupervised sparse vector densification for short text similarity," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics, Hum. Lang. Technol.*, 2015, pp. 1275–1280.
[60] (2024). *VCS—The Industry's Highest Performance Simulation Solution*. [Online]. Available: https://www.synopsys.com/verification/simulation/vcs.html
[61] Z. Rasheed, M. Waseem, K. Systä, and P. Abrahamsson, "Large language model evaluation via multi AI agents: Preliminary results,", *arXiv:2404.01023*.
[62] L. Chen, M. Zaharia, and J. Zou, "How is ChatGPT's behavior changing over time?," 2023, *arXiv:2307.09009*.
[63] W. C. Ouédraogo et al., "Test smells in LLM-generated unit tests," 2024, *arXiv:2410.10628*.
[64] H. Touvron et al., "Llama 2: Open foundation and fine-tuned chat models," 2023, *arXiv:2307.09288*.
[65] (2023). *Phind*. [Online]. Available: https://huggingface.co/Phind/Phind-CodeLlama-34B-v2
[66] J. Xu et al., "DejaVuzz: Disclosing transient execution bugs with dynamic swappable memory and differential information flow tracking assisted processor fuzzing," in *Proc. 30th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Aug. 2025, pp. 64–80.
[67] X. Wang et al., "Self-consistency improves chain of thought reasoning in language models," 2022, *arXiv:2203.11171*.
[68] X. Wan, R. Sun, H. Dai, S. O. Arik, and T. Pfister, "Better zero-shot reasoning with self-adaptive prompting," 2023, *arXiv:2305.14106*.
[69] X. Wan et al., "Universal self-adaptive prompting," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2023, pp. 7437–7462.
[70] Z. Tian, J. Chen, and X. Zhang, "Fixing large language models' specification misunderstanding for better code generation," in *Proc. IEEE/ACM 47th Int. Conf. Softw. Eng. (ICSE)*, Apr. 2025, pp. 1514–1526.
[71] X. Jiang et al., "Self-planning code generation with large language models," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 7, pp. 1–30, Sep. 2024.
[72] J. Li, G. Li, Y. Li, and Z. Jin, "Structured chain-of-thought prompting for code generation," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 2, pp. 1–23, Feb. 2025.
[73] J. Li, Y. Zhao, Y. Li, G. Li, and Z. Jin, "AceCoder: An effective prompting technique specialized in code generation," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 8, pp. 1–26, Nov. 2024.
[74] M. Schlaegl and D. Grosse, "Single instruction isolation for RISC-V vector test failures," in *Proc. 43rd IEEE/ACM Int. Conf. Comput.-Aided Design*, Oct. 2024, pp. 1–9.