PDF Download
3721133.pdf
05 February 2026
Total Citations: 0
Total Downloads: 506

Latest updates: https://dl.acm.org/doi/10.1145/3721133

Accepted: 17 February 2025
Revised: 21 January 2025
Received: 07 November 2024

Citation in BibTeX format

RESEARCH-ARTICLE

DeepVerifier: Learning to Update Test Sequences for Coverage-Guided Verification

YUNTAO LU, Chinese University of Hong Kong, Hong Kong, Hong Kong

CHEN BAI, Hong Kong University of Science and Technology, Hong Kong, Hong Kong

YUXUAN ZHAO, Chinese University of Hong Kong, Hong Kong, Hong Kong

ZIYUE ZHENG, The Hong Kong University of Science and Technology (Guangzhou), Guangzhou, Guangdong, China

YANGDI LYU, The Hong Kong University of Science and Technology (Guangzhou), Guangzhou, Guangdong, China

MINGYU LIU, Huawei Technologies Co., Ltd., Shenzhen, Guangdong, China

View all

Open Access Support provided by:

Chinese University of Hong Kong

The Hong Kong University of Science and Technology (Guangzhou)

Huawei Technologies Co., Ltd.

Hong Kong University of Science and Technology

ACM Transactions on Design Automation of Electronic Systems
https://doi.org/10.1145/3721133
EISSN: 1557-7309

DeepVerifier: Learning to Update Test Sequences for Coverage-Guided Verification

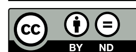
YUNTAO LU, The Chinese University of Hong Kong, Hong Kong, Hong Kong
CHEN BAI, The Hong Kong University of Science and Technology, Hong Kong, Hong Kong
YUXUAN ZHAO, The Chinese University of Hong Kong, Hong Kong, Hong Kong
ZIYUE ZHENG, The Hong Kong University of Science and Technology - Guangzhou Campus, Guangzhou, China
YANGDI LYU, The Hong Kong University of Science and Technology - Guangzhou Campus, Guangzhou, China
MINGYU LIU, Huawei Technologies Co Ltd, Wuhan, China
BEI YU, The Chinese University of Hong Kong, Hong Kong, Hong Kong

Verification is critical in ensuring the reliable operation of modern, complex computing systems. However, as processor designs become increasingly sophisticated, conventional static verification techniques struggle to generate high-quality test sequences that achieve comprehensive coverage. Dynamic simulation-based approaches, which leverage coverage-driven objectives, can increase confidence in correct processor functionality but often suffer from low verification efficiency due to the generation of redundant test sequences and significant computational overhead. To address these challenges, this paper presents DeepVerifier, a novel coverage-guided test generation framework that leverages data-driven learning of existing test sequences and their associated coverage feedback. DeepVerifier uses a language model to learn the semantic representations of test sequences, ensure adherence to syntax constraints, and estimate the relationship between test sequences and coverage scores. By updating test sequences with higher coverage, DeepVerifier can significantly improve the efficiency and effectiveness of the verification process. Experimental results of verifying an out-of-order RISC-V microprocessor demonstrate that the framework accurately estimates the coverage scores of test sequences and updates high-quality sequences that contribute to higher coverage. This coverage-guided test generation technique holds promise for enhancing the reliability of modern processor designs.

1 Introduction

RISC-V represents an open-source instruction set architecture (ISA) [1], [2] that has flourished across various sectors of academia and industry due to its modular and extensible characteristics. In the development of processors utilizing RISC-V ISAs, functional verification plays a pivotal role in ensuring the correctness of functionality; this phase can account for as much as 70% of the design and development lifecycle [3].

Corresponding author: Mingyu Liu.
Authors' Contact Information: Yuntao Lu, The Chinese University of Hong Kong, Hong Kong, Hong Kong; e-mail: ytlu23@cse.cuhk.edu.hk; Chen Bai, The Hong Kong University of Science and Technology, Hong Kong, Hong Kong; e-mail: cbai@cse.cuhk.edu.hk; Yuxuan Zhao, The Chinese University of Hong Kong, Hong Kong, Hong Kong; e-mail: yxzhao21@cse.cuhk.edu.hk; Ziyue Zheng, The Hong Kong University of Science and Technology - Guangzhou Campus, Guangzhou, Guangdong, China; e-mail: zzheng989@connect.hkust-gz.edu.cn; Yangdi Lyu, The Hong Kong University of Science and Technology - Guangzhou Campus, Guangzhou, Guangdong, China; e-mail: yangdilyu@hkust-gz.edu.cn; Mingyu Liu, Huawei Technologies Co Ltd, Wuhan, China; e-mail: liumingyu@huawei.com; Bei Yu, The Chinese University of Hong Kong, Hong Kong, Hong Kong; e-mail: byu@cse.cuhk.edu.hk.



This work is licensed under a Creative Commons Attribution-NoDerivatives International 4.0 License.
© 2025 Copyright held by the owner/author(s).
ACM 1557-7309/2025/3-ART
https://doi.org/10.1145/3721133

Just Accepted

Coverage metrics are quantitative indicators that show the extent to which a design has been verified. The idea behind coverage-guided verification is to implement more advanced methods for identifying untested behaviors in a design. This approach ultimately improves the effectiveness of the verification process, as seen in various constrained functional verification techniques [4], [5], [6].

As processor designs have become increasingly complex, the focus on functional verification has intensified. There is a growing interest in automated verification techniques, aiming to improve coverage metrics and reduce the time required for verification. To address these challenges, advanced approaches have been proposed, including static formal analysis, dynamic simulation, and semi-formal methods.

Static formal techniques use mathematical formulations and equivalence checking [7] to theoretically prove the correctness or generate counterexamples. However, a state explosion issue arises when verifying complex designs with deep hard-to-reach states [8], [9], [10]. This challenge limits the practicality and scalability of formal methods for large-scale designs.

On the other hand, dynamic simulation techniques function by modeling the system under examination within a hardware-level environment. This methodology facilitates the verification of design functionality through the systematic observation of the system's behavior across a range of conditions and scenarios. This approach is widely regarded as the leading solution for the evaluation of large-scale circuits. A simulation verification work [11] achieves coverage objectives through two pioneering techniques: constrained random generation and coverage-guided test sequence generation. The application of constraints effectively narrows the search space of test sequences, while coverage scores strategically direct the exploration of these generated test sequences to maximize the coverage of hardware states.

Instruction stream generators (ISGs), such as FORCE-RISCV [12], RISC-V design verification framework [13], and RISC-V Non-ISA-specific coverage tool [14], have been developed to systematically produce test sequences randomly and repetitively. This approach significantly enhances the coverage of various design frameworks. Semi-formal methods, i.e., a concolic testing approach [15], generate effective test sequences by analyzing the control flow graph (CFG) of register-transfer level (RTL) designs. These sequences are simulated and executed to check coverage scores, which then guide the next round of sequence generation. Successfully applying this method requires a thorough understanding of both the design and the grammatical structure of RTL code to accurately analyze the control and data flows that inform the generation of test sequences. Additionally, inspired by successful software testing practices, fuzzing methods have been applied to hardware verification tasks. Coverage-guided fuzzing techniques have been proposed in [16], [17], [18], [19], [20] to generate test sequences that achieve higher coverage scores, revealing bugs and vulnerabilities in hardware designs. Nevertheless, these methods generate test sequences within a few hours or days. The effectiveness of coverage-guided fuzzing is significantly influenced by the combination of RTL structure analysis and coverage scores. This quality relies on heuristic mutation methods and search algorithms developed by experts, who utilize feedback from coverage scores of test sequences to enhance their strategies.

Recently, the increasing trend of machine learning (ML) techniques has opened up new possibilities for addressing the challenges of generating high-quality test sequences with minimal professional knowledge. ML approaches represent a promising direction to reduce reliance on domain expertise while facilitating greater automation and scalability in test sequence generation. In previous work, a fully connected Bayesian network [21] described the relationships between input test stimuli and coverage metrics. To enhance the representational capabilities of ML models, researchers developed a three-layer artificial neural network [22] that accelerated the test generation process and improved coverage. Additionally, a language model [23] was employed to understand the semantics of RTL (Register Transfer Level) code, allowing it to predict the hit rate of functional coverage, which in turn improved verification efficiency. However, the adoption of ML methods also introduces new challenges. These include the need for substantial amounts of data to learn the relevant representations and implicit relationships, as well as the inherent complexity associated with interpretation.

Just Accepted

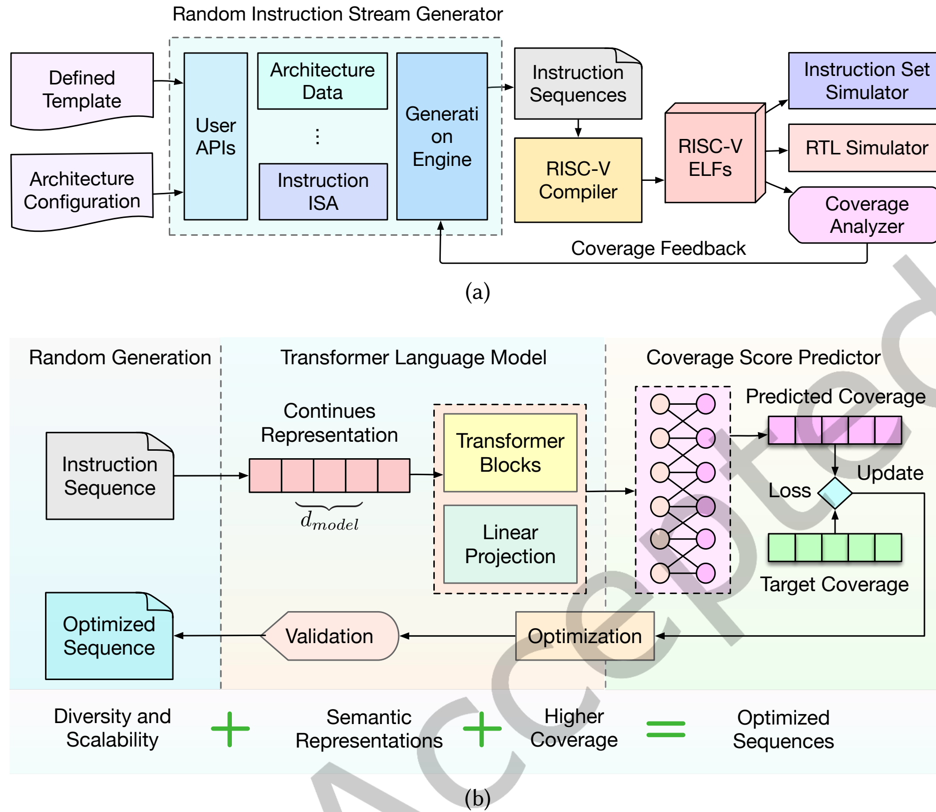


Fig. 1. (a) End-to-end verification workflow for a random test sequence generator used in RISC-V designs. (b) We propose a new strategy for updating test sequences using a transformer language model, which offers feedback on the coverage of a coverage predictor during the early verification stages in the flow.

Just Accepted

In this paper, we introduce a novel framework for generating test sequences called DeepVerifier. This framework is guided by coverage metrics and learns to leverage the relationship between generated test sequences and their associated coverage scores using a language model. Additionally, it employs an objective coverage model to optimize these test sequences for achieving higher coverage. To address the challenges posed by limited data and interpretation issues, we generate data from a large number of test sequences along with their corresponding coverage scores obtained from a simulator.

We highlight the proposed DeepVerifier, focusing on three key aspects. First, we customized a language model to represent RISC-V ISA instruction sequences. This model tokenizes and transforms assembly RISC-V instructions into continuous, low-dimensional embedded vectors, which can then be used as input for a subsequent neural network that estimates coverage scores. Second, we designed a coverage score estimator capable of predicting diverse coverage values for these continuous instruction sequence embeddings. This lightweight predictor allows for fast and accurate predictions across various embeddings of RISC-V instruction snippets. Third, we propose an end-to-end differentiable learning strategy to optimize the generated test sequences. Our sequence update strategy is informed by the predicted coverage values and the deviation between the estimated coverage and the

closure coverage scores. By computing the gradients of the test sequence embeddings for the predicted coverage values, we can update the embeddings using these gradients and adaptive learning rates. This process aims to produce test sequences with higher coverage.

A comparison between conventional coverage-guided test generation methods and DeepVerifier is illustrated in Fig. 1. In traditional methods, test sequences are generated based on feedback scores from the coverage analyzer, following co-simulation with both the instruction set simulator and the RTL simulator, as depicted in Fig. 1(a).

In contrast, DeepVerifier efficiently generates high-quality test sequences using a sequence representation language model, a score predictor, and a sequence optimizer, as shown in Fig. 1(b). The effectiveness of DeepVerifier lies in its ability to accurately represent RISC-V instruction sequences through the language model and utilize a fast and reliable coverage score predictor, which minimizes the waiting time during co-simulation. While we demonstrate our approach on 64-bit RISC-V processors, the methodology is inherently extensible to other ISA families. The transformer-based framework requires only adaptation of the tokenization layer and ISA architectural constraints from specifications for different targets, making it applicable to various processors. The current implementation with a limited set of RISC-V instruction tokens represents a starting point, and the transformer architecture can scale to much larger vocabularies with large-scale tokens to accommodate more complex ISA extensions, additional architectural states, or even mixed-ISA systems.

Additionally, it employs a gradient-based sequence update strategy to enhance test sequences in the optimal direction for achieving higher coverage scores. Our contributions are summarized below:

- A customized novel language model has been developed for learning the semantic and syntactic representations of RISC-V instruction sequences.
- A fast and accurate coverage predictor calculates coverage scores for test instruction sequences.
- A gradient-based sequence update strategy uses the target coverage scores to enhance test sequences, aiming for higher coverage.
- Experimental results demonstrate that our framework effectively integrates representations of instruction sequences, estimates coverage efficiently, and optimizes test sequences, achieving a coverage improvement of approximately 3% to 6%, thereby facilitating coverage closure.

The rest of this paper is organized as follows: Section 2 introduces the fundamentals of verification tasks. Section 3 presents the problem formulation for the test sequence update issue. Section 4 offers an overview of the DeepVerifier framework, including the algorithms employed. Section 5 outlines the experimental setup and evaluates the results obtained from DeepVerifier. Finally, Section 6 summarizes the findings and discusses future research directions.

Just Accepted

2 Preliminaries

2.1 RISC-V ISA

The RISC-V Instruction Set Architecture (ISA) [1] comprises a mandatory base integer instruction set, referred to as `RV32I`, `RV64I`, or `RV128I`, which is characterized by specific register widths. Additionally, it includes various optional extensions identified by single letters; for example, the letters `M`, `A`, `C`, `F`, and `D` correspond to integer multiplication and division, atomic instructions, compressed instructions, and single and double precision floating point operations, respectively. For example, `RV32IMC` specifies a 32-bit core equipped with both `M` and `C` extensions. The designation `G` signifies the `IMAFD` instruction set; therefore, `RV32GC` denotes `RV32IMAFDC`. Each core contains 32 general-purpose registers, labeled from `x0` to `x31`, while the floating point extensions contribute an additional 32 float-point registers. Moreover, there exist specific instructions to access these registers by their designated names, where source registers are referred to as `rs1` and `rs2`, and the destination register is designated as RD. The format and semantics of both the base ISA and its extensions are delineated in the unprivileged ISA specification.

Furthermore, the privileged specification outlined in the RISC-V documentation [2] encompasses several critical functionalities essential for environmental interaction and operating system execution. This specification delineates various execution modes, particularly the mandatory machine mode, along with the extensions for Supervisor and User modes, accompanied by detailed descriptions of the corresponding control and status registers (CSRs). CSRs are specialized registers that are fundamental to the privileged architecture framework. For instance, the machine status register monitors and regulates the current operating state of the hardware thread (hart), the machine ISA register identifies the ISA extensions supported by the processor, and the machine trap-vector base-address register designates the address of the trap/interrupt handler. Additionally, a read-only machine architecture ID register encodes the unique architecture identification of the processor, among other functionalities.

The complexity and modularity of the RISC-V ISA present challenges and opportunities for verification. The diverse instruction categories activate different components of the design, which exercise specific functional units and control paths, making comprehensive coverage crucial yet challenging to achieve through manual or random testing alone. The characteristics of RISC-V motivate our transformer-based approach in several ways. For the instruction representation, our model learns semantic relationships to capture both the syntax of individual instructions. For the coverage correlation, the self-attention mechanism enables modeling relationships between instruction patterns and coverage metrics, which helps identify which instruction combinations effectively exercise different parts of the design. For sequence updates, understanding the semantic structure of RISC-V instructions allows our method to generate valid and effective test sequences.

2.2 Simulation-based Verification

In simulation-based verification, the RTL model of the microprocessor is converted into a high-level object-oriented software class within both open-source and commercial environments [24], [25]. This translated high-level model is subsequently instantiated in the testbench alongside a memory model that is populated with the relevant verification tests. Once the simulation infrastructure is established, it is essential to develop criteria to ascertain whether the simulated tests yield a pass or fail outcome. The most straightforward approach is to conduct correctness checking through directed tests. Upon completion of a test execution, the final result is compared against a pre-calculated reference answer. The determination of success or failure is based on this comparison. While directed tests are extensively utilized within the industry, their primary aim is often to confirm the fundamental functionality of the design or to investigate specific, deliberate corner cases. For over three decades, to rigorously assess the design, both industry professionals and academic researchers have relied on verification binaries that are generated randomly.

A Random Instruction Generator (RIG), also referred to as a test program generator or instruction stream generator, is a software utility designed to produce randomized assembly instruction streams based on predefined configurations. The tests generated by the RIG encompass a wide range of implemented functionalities. This tool is capable of creating intricate test cases that may be challenging for engineers to devise independently [26], [27], [12]. By applying complete random instruction streams to stress the design under test, one can assess the system comprehensively. However, complete randomness lacks control over the generated tests, resulting in a low probability of uncovering deeply concealed bugs arising from complex interactions among various components. To address this limitation, some RIGs offer the capability to exert control over the generated tests via test program templates. These templates provide an abstract representation of the test, delineating a set of constraints that the generator must adhere to. Consequently, this enables the management of both the direction and depth of the generated tests [5].

While simulation-based verification relies heavily on random test generation and manual coverage analysis, our approach enhances this paradigm by introducing learning-based guidance. DeepVerifier leverages the coverage

Just Accepted

feedback from simulation runs to train our transformer model, enabling it to learn the relationship between instruction sequences and coverage outcomes. It creates a more intelligent test generation process that combines the thoroughness of simulation-based verification with the predictive power of deep learning. Instead of purely random exploration or manual guidance, our method automatically learns instruction patterns and makes the verification process more efficient and systematic.

2.3 Language Model in Natural Language Processing

Natural language processing (NLP) constitutes a domain within linguistics and machine learning that concentrates on comprehending all aspects of human language. The primary objective of NLP tasks is not merely to interpret individual words in isolation, but rather to grasp the broader context in which these words are situated.

Moreover, recent advancements in Transformers [28] have resulted in the availability of thousands of pre-trained models capable of performing various tasks across different modalities in text. These tasks include text classification, information extraction, question answering, summarization, machine translation, and text generation, supporting over 100 languages.

The automatic generation of vector representations for assembly instructions at the binary analysis level draws upon methodologies from the natural language processing (NLP) domain [29], treating binary assembly code as analogous to natural language documents. Prominent deep pre-trained language models in NLP include BERT [30], GPT [31], RoBERTa [32], and BART [33]. Reference [29] introduced a pre-trained assembly language model aimed at general-purpose instruction representation learning, which acknowledges the inherent naturalness of assembly instruction sequences. This work underscores the potential for acquiring semantic representations through pre-trained language models, facilitating their application across a variety of downstream tasks.

Advances in NLP, particularly the transformer architecture's success in capturing sequential dependencies and semantic relationships, inspire our approach to hardware verification. We adapt these techniques to treat RISC-V assembly sequences as a domain-specific language, where instructions are tokens and their relationships represent dependencies. Similar to the transformers learning semantic relationships between words in natural language, our model learns relationships between instructions and their impact on verification coverage. Furthermore, we extend beyond direct NLP applications by incorporating instruction types and coverage objectives into the model. This allows DeepVerifier to not only understand instruction sequences but also predict their verification effectiveness and guide test generation.

3 Problem Formulation

The problem is articulated as follows:

Problem 1. Given a RISC-V microprocessor, the task is to devise a method for generating high-quality test cases that effectively assess its functionality. The primary goal is to maximize the scores of established coverage metrics.

In Problem 1, a test case is defined as a sequence of instructions. In this context, the terms "test sequences" and "test cases" are used interchangeably to convey the same concept.

4 DeepVerifier Framework

4.1 Overview

Traditional verification approaches typically rely on random testing to achieve coverage requirements, DeepVerifier, illustrated in Fig. 1(b), serves as a complementary method, specifically targeting hard-to-reach coverage cases through intelligent test sequence generation. While achieving near-100% coverage is crucial in processor verification, DeepVerifier is designed not as a replacement for existing methods, but as a complementary approach

Just Accepted

to enhance coverage efficiency. Traditional random testing remains essential for broad coverage but can be time-consuming when targeting specific coverage cases. DeepVerifier addresses this limitation by learning from existing test sequences to intelligently guide the generation of new tests, potentially reducing the number of instructions and simulation time needed to achieve target coverage levels. Our framework particularly excels at identifying patterns that lead to improved coverage in challenging scenarios, working alongside conventional methods to accelerate the verification process.

Initially, we generate assembly instruction sequences through a random instruction sequence generator [12] in conjunction with a customized fuzzing tool. These sequences are compiled and simulated using RTL simulators like Synopsys VCS [25] or Verilator [24] to capture coverage scores. To optimize simulation time and instruction count efficiency, we preprocess these sequences and utilize them to train a transformer language model that learns the semantic patterns leading to effective coverage.

Our approach comprises three key components designed to reduce the number of instructions needed for achieving target coverage: (1) an encoder that efficiently embeds test sequences into continuous vector representations; (2) a coverage predictor that rapidly forecasts coverage values from these representations; and (3) a decoder that translates optimized continuous representations back into valid instruction sequences. This framework enables quick identification of promising test sequences through iterative parameter adjustment, significantly reducing the exploration time compared to random testing alone.

The instruction stream generator creates test sequences based on predefined templates specifying instruction types, opcode weights, and sequence length. The transformer model's encoder-decoder architecture efficiently converts between text sequences and token vectors, maintaining sequence validity while exploring the coverage space.

A fine-tuned tokenizer works with the transformer model to convert instructions into token vectors matching the vocabulary size. The encoder creates compact word embeddings, while the decoder reconstructs valid sequences. The model training process minimizes cross-entropy loss to align input and output distributions, capturing crucial patterns that influence coverage. The coverage predictor uses the transformer's final hidden state to build a neural network that efficiently maps sequence characteristics to coverage outcomes.

To accelerate coverage improvement, we compute mean square loss between target and predicted coverage scores, using this deviation to update sequence representations. The gradient-based optimization process, guided by scheduled learning rates, efficiently modifies input sequences while maintaining their validity through a correctness verification tool. This approach complements random testing by specifically targeting coverage gaps, potentially reducing the total number of instructions needed to achieve coverage targets.

4.2 Test Sequence Representation

The test sequences comprise a diverse array of assembly programming language snippets of varying lengths, generated by a RISC-V instruction stream generator [12]. Each sequence encompasses a series of instructions, with lengths ranging from 128 to 256. Each instruction comprises opcodes and operands. In alignment with the specifications of the RISC-V ISA [1] and [2], we categorize opcodes into five distinct types and operands into three categories. To effectively represent each instruction, we construct a vocabulary that includes both opcodes and operands, along with a tokenizer designed to convert each token into its corresponding embedded form.

To maintain the representation strategy of the `<s>` and `</s>` tokens utilized in the trained tokenizer, which signify the start and end identifiers of each input sequence, we delineate each instruction sequence by inserting `<s>` and `</s>` at the beginning and concluding points of each sequence. To interpret the instructions within the RISC-V ISA specifications and convert instruction opcodes and operands into continuous vector representations suitable for subsequent utilization by a transformer language model, we refine the tokenizer. An instruction

Just Accepted

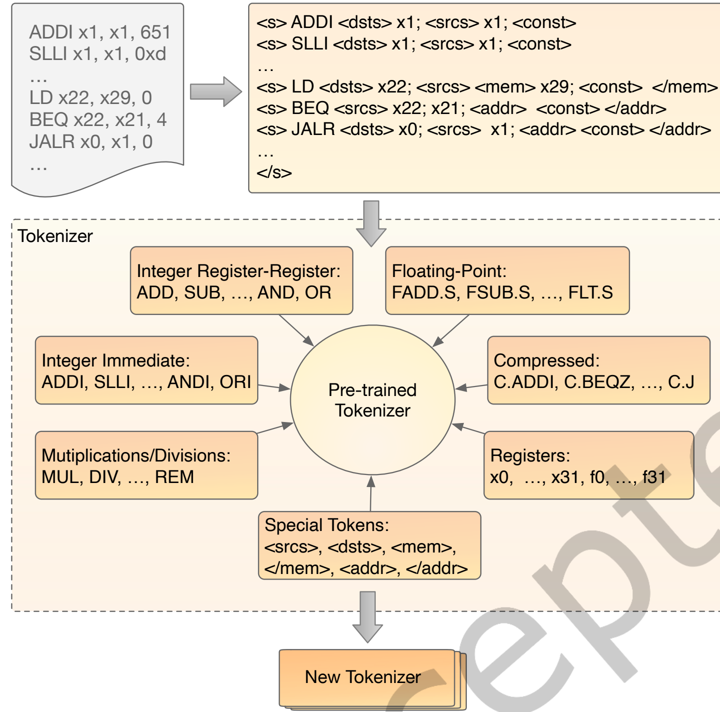


Fig. 2. Creating customized tokens to represent instruction sequences and integrating RISC-V ISA-specific tokens into the vocabulary, along with developing a new tokenizer.

Just Accepted

comprises tokens that include an opcode (e.g., `ADD`, `SUB`), register operands, address operands, and immediate values.

We incorporate opcodes and register symbols as new tokens within the vocabulary. To differentiate between source and destination operands, we employ the tokens `<srcs>` and `<dsts>` to represent source registers and destination registers in an instruction, respectively. For immediate operands, we utilize the `<const>` token to mitigate the presentation of diverse concrete values. Furthermore, for load and store instructions (e.g., `LD`, `SD`) that access memory addresses, we append `<mem>` and `</mem>` labels as tokens at the beginning and end of the instruction to represent the memory access pattern. For conditional and unconditional jump instructions (e.g., `BEQ`, `JAL`) that reference the next instruction address, we propose the use of `<addr>` and `</addr>` tokens to identify the range of address registers. Notably, to facilitate the processing of sequences, we replace commas within the instruction with semicolons and subsequently restore them following optimized modifications to maintain the validity of the instruction format.

This tokenization approach enables our model to capture both instruction-level and sequence-level semantic relationships, which is crucial for generating effective test sequences. Unlike traditional approaches that treat instructions in isolation, our method preserves contextual dependencies that influence coverage outcomes.

4.3 Model Architecture

To learn semantic representations for hardware verifications, the LSTM network in [23] can sequentially learn the dependencies in an instruction sequence, while this approach faces several limitations. The sequential nature

of LSTM processing makes it computationally expensive for long instruction sequences, and its fixed-size hidden state becomes a bottleneck when modeling complex instruction dependencies. The Transformer model [34] is a widely recognized technique for sequence modeling, which offers three key advantages through its architecture. First, its parallel processing and self-attention mechanism excels at capturing long-distance dependencies within input sequences while maintaining computational efficiency. Second, its multi-head attention allows simultaneous modeling of different types of instruction relationships. Third, its explicit position encoding better preserves instruction order constraints critical for verification tasks.

To leverage these advantages of the model to understand both the semantics and syntax of RISC-V assembly sequences, we developed a customized version of BART [33], a fine-tuned transformer-based sequence-to-sequence language model. This customization involved specific adaptations, including a tailored tokenizer and loss functions, aimed at enhancing RISC-V assembly sequence representation and improving coverage score prediction, drawing inspiration from PalmTree [29], a prior assembly language representation model. Unlike [29] that aims for general-purpose instruction embedding across architectures, our model is specifically optimized for verification integration with custom tokenization. Furthermore, we bridge representation learning with coverage-guided verification by introducing components for coverage prediction and sequence updates. This customization includes verification-specific adaptations of a tailored tokenizer and specialized loss functions, enabling both better coverage results and more interpretable verification workflows.

The customized tokenizer has been developed to effectively capture the distinct characteristics of RISC-V assembly sequences, including opcodes, register operands, address operands, and constant immediate values. To address the intricate internal structures of instructions, we represent the instruction sequence by appending special tokens as defined in Section 4.2. Additionally, we generate a specific tokenizer tailored for this purpose, which builds upon the standard tokenizer of BART to incorporate instruction-related tokens into the vocabulary. The processes of preprocessing and tokenizing the instruction sequences, along with the composition of the RISC-V ISA-specific tokenizer utilizing a customized vocabulary, are illustrated in Fig. 2.

The fundamental components of a transformer model encompass an encoder, a decoder, and several linear projection layers designated for downstream tasks. The encoder and decoder are composed of a position-wise input embedding layer, followed by N transformer blocks. Each transformer block includes a multi-head self-attention layer, a fully connected feed-forward network, and multiple additive and normalization layers. The output generated from the final transformer block is subsequently processed through linear projection layers to derive the conclusive output representation of the input sequence.

While our current implementation focuses on RISC-V instruction tokenization, the framework enables expansion to diverse verification scenarios through adaptable token vocabularies and sequence lengths. The transformer model's self-attention mechanism inherently supports sequences well beyond our current implementation, facilitating the verification of sophisticated designs. The extensible nature of our architecture accommodates the integration of advanced instruction sets, architectural states, and control mechanisms. This inherent scalability positions our framework to adapt to emerging processor architectures and their increasing complexity. As modern processor designs introduce more sophisticated features, the transformer's capacity to capture intricate semantic relationships between instructions becomes increasingly valuable for comprehensive verification coverage.

The input embedding and positional encoding layer transforms input tokens $\mathbf{x}$ into a continuous embedded representation denoted as $\mathbf{TE}$, while simultaneously incorporating positional information into these representations, referred to as $\mathbf{PE}$. The input tokens $\mathbf{x} = (x_1, x_2, ..., x_n)$ encompass a sequence of length n, where each $x_i, i \in \{1, 2, ..., n\}$ signifies a token identity index within the vocabulary. An embedding matrix $\mathbf{TE} \in \mathbb{R}^{d_{vocab} \times d_{model}}$ is employed to map each input $\mathbf{x_i}$ into the embedded space with d_{vocab} representing the vocabulary size and d_{model} indicating the dimensionality of the embedding space. The embedding $\mathbf{te}_i$ corresponding to each token x_i

Just Accepted

is articulated by the following Equation (1),

$$\mathbf{te}_i = \text{TokenEmbedding}(x_i), \tag{1}$$

where $\mathbf{TE}_{x_i}$ denote the x_i-th row of the embedding matrix $\mathbf{TE}$ which possesses a dimensionality of d_{model}. For our model, the embedded dimension is defined as 768. To encode positional information for each token within the sequence, sinusoidal positional encoding employs both cosine and sine functions. For a given position $pos, pos \in \{1, 2, ..., n\}$ and dimension $i, i \in \{1, 2, ..., d_{model}\}$, the positional encoding is bifurcated into even indices $\mathbf{pe}_{pos,2i}$ and odd indices $\mathbf{pe}_{pos,2i+1}$. These indices correspond to the sine and cosine functions respectively, as indicated in Equation (2),

$$\begin{aligned} \mathbf{pe}_{pos,2i} &= \text{sine}(\frac{i}{10000^{2i/d_{model}}}), \\ \mathbf{pe}_{pos,2i+1} &= \text{cosine}(\frac{i}{10000^{2i/d_{model}}}). \end{aligned} \tag{2}$$

The final representation $\mathbf{e}_i$ of each input token is derived by concatenating the token embedding $\mathbf{te}_i$ with the positional encoding $\mathbf{pe}_{pos}$. This relationship is expressed as $\mathbf{e}_i = \mathbf{te}_i + \mathbf{pe}_{pos}$.

Through the processes of token embedding and positional encoding, the input sequence $\mathbf{X} \in \mathbb{R}^{n \times d_{vocab}}$ of the length n and a vocabulary size d_{vocab} is converted into a sequence of continuous embedded representations $\mathbf{E} \in \mathbb{R}^{n \times d_{model}}$, which maintains the same length n and a dimension of d_{model}. The attention mechanism employed within each transformer block is designed to capture long-range dependencies present in the embedded sequence and to evaluate the significance of various components through the computation of attention scores. Each transformer block incorporates a multi-head self-attention layer to derive these attention scores. Specifically, each input embedded matrix is linearly projected into three distinct matrices: query $\mathbf{Q}$, key $\mathbf{K}$, and value $\mathbf{V}$ by the projection matrices $\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V \in \mathbb{R}^{d_{model} \times d_k}$, where $d_k = (d_{model}/H), H$ denotes the number of self-attention heads, which is established as 4 in the model. The multi-head attention score is expressed by Equation (3) as follows:

$$\begin{aligned} \text{Attn}_\text{i}(\mathbf{Q}_i, \mathbf{K}_i, \mathbf{V}_i) &= \text{softmax}(\frac{\mathbf{Q}_i \mathbf{K}_i^\top}{\sqrt{d_k}})\mathbf{V}_i, \\ \text{MultiHeadAttention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) &= \text{Concat}(\text{Attn}_1, ..., \text{Attn}_\text{H})\mathbf{W}_O, \end{aligned} \tag{3}$$

where the computation of each self-attention head, the self-attention score is derived by taking the dot product of the query matrix $\mathbf{Q}_i$ and key matrix $\mathbf{K}_i$. This result is subsequently scaled by the square root of the key dimension d_k. The softmax function is then applied to normalize the scores, ensuring that their sum equals one. Following this step, a weighted sum of the value matrix $\mathbf{V}_i$ is calculated to produce the output Attn_i for each self-attention head. The final multi-head attention score is formed by concatenating the outputs of H self-attention heads, after which these concatenated results are projected using the weight matrix $\mathbf{W}_O \in \mathbb{R}^{d_{model} \times d_{model}}$.

The output generated by the multi-head self-attention layer undergoes normalization through the function LayerNorm. This layer is designed to normalize inputs across features by utilizing the mean μ and variance σ of each hidden state $\mathbf{h}$ within a transformer block, as illustrated in Equation (4),

$$\text{LayerNorm}(\mathbf{h}) = \frac{\mathbf{h} - \mu}{\sqrt{\sigma^2 + \varepsilon}}\gamma + \beta. \tag{4}$$

Upon processing through the multi-head attention layer, the resultant outputs, denoted as $\mathbf{X_{attn}}$, are subsequently directed into the feed-forward network present within each transformer block. This feed-forward network comprises two fully connected layers, $\mathbf{fc}_1$ and $\mathbf{fc}_2$, which operate on each hidden state $\mathbf{h}$. A Gaussian

Just Accepted

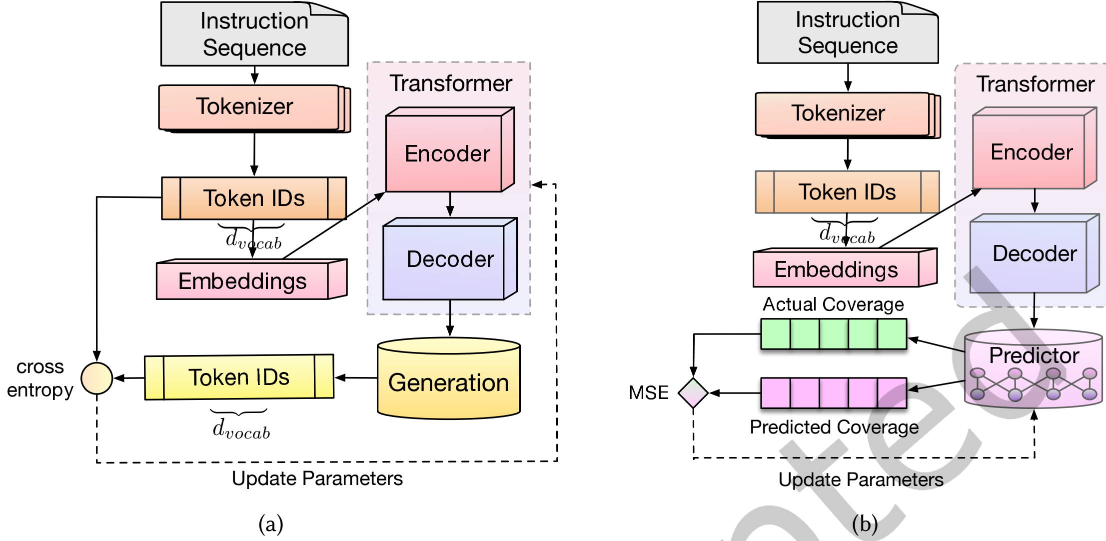


Fig. 3. (a) Training a transformer model for a sequential modeling task focused on sequence generation will help the model learn the semantics and syntax of instruction sequences. (b) Performing a supervised prediction task to assess coverage scores using the fine-tuned transformer model from the generation task will enable us to estimate the relationship between the input sequence and its corresponding output.

Just Accepted

Error Linear Unit (GELU) activation function, as specified in Equation (5),

$$\begin{aligned} \mathrm{GELU}(\mathbf{h}) &= \mathbf{h} \cdot \frac{1}{2}(1 + \Phi(\frac{\mathbf{h}}{\sqrt{2}})) \\ &\approx 0.5\mathbf{h}(1 + \tanh^{\sqrt{\frac{2}{\pi}}(\mathbf{h}+0.044715\mathbf{h}^3)}), \end{aligned} \tag{5}$$

where Φ is the cumulative distribution function for Gaussian distribution.

The following feed-forward network is given by Equation (6),

$$\begin{aligned} \mathbf{fc}_1 &= \mathrm{GELU}(\mathbf{X}_{attn}\mathbf{W}_1 + \mathbf{b}_1), \\ \mathbf{fc}_2 &= \mathrm{GELU}(\mathbf{fc}_1\mathbf{W}_2 + \mathbf{b}_2), \end{aligned} \tag{6}$$

where $\mathbf{W}_1 \in \mathbb{R}^{d_{model} \times d_{ffn}}$, $\mathbf{W}_2 \in \mathbb{R}^{d_{ffn} \times d_{model}}$ represent the projection matrices, while $\mathbf{b}_1 \in \mathbb{R}^{d_{ffn}}$, $\mathbf{b}_2 \in \mathbb{R}^{d_{model}}$ denote the bias vectors corresponding to the first and second fully-connected layers, respectively. The dimension of the feed-forward network, denoted as d_{ffn}, is established as 1024 within the model.

The customized transformer model comprises both encoder and decoder components, consisting of two transformer blocks. The decoder utilizes a preceding output matrix $\mathbf{Y} \in \mathbb{R}^{m, d_{vocab}}$, where m represents the length of the output sequence. To prevent the decoder from attending to future tokens, the self-attention mechanism is masked. This is achieved by assigning negative infinity values $-\infty$ to the attention scores of future positions before the application of the softmax function. The input matrices for the decoder's self-attention are initialized with the outputs from the encoder's self-attention. Specifically, the query matrix $\mathbf{Q}'$ is derived from the masked self-attention of the decoder, while the key matrix $\mathbf{K}'$ and value matrix $\mathbf{V}'$ are obtained from the encoder's outputs.

4.4 Training Tasks

By incorporating coverage feedback into the test generation process, our approach captures the sequence pattern and optimizes coverage objectives. These optimization tasks help discover instruction dependencies in a sequence and obtain the coverage feedback in a short time to guide the successive test generation process. To implement our framework practically, we integrate the existing transformer framework and verification workflows, requiring minimal additional setup while providing substantial benefits in test generation efficiency and coverage improvement. The fine-tuning task illustrated in Fig. 3(a) exposes a transformer model to effectively capture the sequence. Subsequently, we apply the fine-tuned model to conduct the downstream task of predicting coverage scores, as depicted in Fig. 3(b).

Unlike traditional BART implementations that use text infilling as a noising method during training, our sequential modeling task is a fine-tuning task that prioritizes maintaining strict RISC-V ISA compliance throughout the training process. The RISC-V instruction sequences must maintain strict syntax and semantics of ISA compliance, as random masking could disrupt critical instruction dependencies and patterns. The primary objective is to optimize coverage scores through valid and executable test sequences. The transformer model needs to learn the precise relationship between instruction patterns and their corresponding coverage metrics. Adding noise during training could potentially interfere with this coverage-guided learning process. Additionally, unlike natural language tasks where partial text corruption is acceptable, hardware verification demands precise instruction sequences that can be directly compiled and simulated for the coverage analysis.

Training Task 1: Sequential Modeling of Test Generation.

The representation of instruction sequences serves to maintain the architectural information of each instruction within a test sequence, as well as the execution rules governing the overall test sequence. This information encompasses the syntactic rules of RISC-V ISAs and highlights execution dependencies between instructions within the current test sequence tied to a particular design under evaluation. Moreover, this information relates to the association between test inputs and code coverage scores for a given design. By acquiring an understanding of the syntax and execution rules about tokens in a test sequence, and about existing training samples, the proposed model effectively learns the generation of instruction sequencing and the correlation between sequence patterns and objective coverage scores.

$$\mathrm{L}(\mathbf{x}) = \prod_{i=1}^{n} P(x_i|\mathbf{x}_{<i}) = \prod_{i=1}^{n} P(x_i|x_1, x_2, ..., x_{i-1}). \tag{7}$$

Through model training, we can delineate the relationships between each token x_i and acquire the execution rules pertinent to the current test sequence by optimizing the objective function. For example, regarding the existing tokens in a test sequence `{ <s> ADD x1 x2 }`, the transformer model predicts the subsequent token from among the valid register tokens in the vocabulary. Furthermore, the fine-tuned transformer model accurately preserves the syntax of the instruction while also ensuring the proper instruction dependencies. By predicting the subsequent tokens for instruction opcodes and operands, we can ascertain valid register symbols and dependencies, the order of memory access, and the integrity of the control flow. Additionally, a valid test program should achieve correct termination and maintain the integrity of both computational semantics and architectural constraints. On one hand, the fine-tuned transformer model facilitates the appropriate next tokens and execution flow; on the other hand, we have developed a RISC-V instruction syntax interpreter to guarantee the validity of each instruction following successive updates to the sequence.

Training of the transformer model is carried out with a sequence generation task that focuses on recovering the input sequence from the final hidden state, denoted as $\mathbf{h}_n$, of the transformer block through transformations applied in the decoder. The objective of this task is to minimize the difference between the input sequence generated by the encoder and the output sequence produced by the decoder. A cross-entropy (CE) loss function

Just Accepted

is employed to quantify the discrepancy between the predicted token and the corresponding ground-truth token. The objective function is derived from the likelihood of the token sequence, which facilitates the evaluation of the similarity between the source token sequence $\mathbf{x}, i \in \{1, 2, ..., n\}$ of length n and the target token sequence $\mathbf{y}, j \in \{1, 2, ..., m\}$ of length m. Furthermore, the transformer model is updated by minimizing the loss function.

The transformer model operates by applying a multi-head self-attention mechanism to the input tokens $\mathbf{x}$. This process is followed by an embedding and positional encoding phase, which converts distinct token vectors into continuous embedded representations. These representations are then subjected to position-wise feed-forward layers, ultimately generating an output distribution for target tokens through a softmax operation.

The probability $P(y_j = c|\mathbf{x}, \mathbf{y}_{<j})$ of the target token y_j, which corresponds to a token identifier v within the vocabulary, is computed based on the preceding tokens $\mathbf{y}_{<j}$ and the source token sequence $\mathbf{x}$. This calculation involves a linear projection of the last hidden state $\mathbf{h}_n$, resulting in the vector $\mathbf{z}_j$ containing scores for the potential next tokens in a vocabulary of size d_{vocab}. Subsequently, a softmax function is applied to convert these scores into a probability distribution. The cross-entropy loss, which quantifies the divergence between a target sequence $\mathbf{y}$ and input sequence $\mathbf{x}$, is defined in Equation (8),

$$\begin{aligned} \text{Loss}_{\text{CE}} &= -\frac{1}{m} \sum_{j=1}^{m} \sum_{c=1}^{d_{vocab}} \log P(y_j = c|x, y_{<j}) \\ &= -\frac{1}{m} \sum_{j=1}^{m} \sum_{c=1}^{d_{vocab}} \log \frac{\exp(\mathbf{z}_{j,c})}{\sum_{v=1}^{d_{vocab}} \exp(\mathbf{z}_{j,v})}, \end{aligned} \tag{8}$$

where $\mathbf{z}_{j,c}$ represents the score associated with the token ID c within the score vector $\mathbf{z}_j$ corresponding to the target token y_j in the sequence $\mathbf{y}$.

Training Task 2: Coverage Score Prediction of Test Sequences. Following the training of the sequential model aimed at minimizing cross-entropy loss, we utilize the fine-tuned transformer model denoted as $\text{Transformer}_{\text{gen}}$ as the foundational component. In conjunction, we append a multi-layer neural network model referred to as MLP to serve as the prediction head for a coverage prediction task. This task is characterized as a supervised multi-target regression task focused on forecasting coverage scores $\mathbf{s} = (s_1, s_2, ..., s_l)$, where l represents the number of coverage types involved in the measurement, set at $l = 4$ within our framework. The computational process of the transformer body in conjunction with the prediction head model is elucidated in Equation (9),

$$\begin{aligned} \mathbf{h_n} &= \text{Transformer}_{\text{gen}}(\mathbf{x}), \\ \mathbf{s} &= \text{MLP}(\mathbf{h_n}[0]), \end{aligned} \tag{9}$$

where $\mathbf{x}$ represent the input sequence, while $\mathbf{h}_n$ denotes the final hidden state of the transformer architecture. The first token of the hidden state, $\mathbf{h}_n[0]$, is utilized for subsequent predictions. The objective of the training process is to minimize the mean square error (MSE), as defined in Equation (10), between the predicted coverage scores $\hat{s}$ and the corresponding ground truth coverage scores $\bar{s}$.

$$\text{Loss}_{\text{MSE}} = \frac{1}{l} \sum_{i=1}^{l} (\hat{s_i} - \bar{s_i})^2. \tag{10}$$

4.5 Test Sequence Updates and Correction

The sequence update and correction process comprises three principal components: (1) A coverage score predictor, which utilizes a transformer-based neural network consisting of two layers to map instruction sequences to their corresponding predicted coverage scores. (2) A gradient-based optimizer employs an iterative refinement

Just Accepted

approach using gradients derived from the predicted coverage scores. (3) A syntax correction tool, which ensures that the optimized sequences adhere to the ISA compliance and maintain syntactical correctness.

Utilizing the transformer-based model, each instruction within the sequence is tokenized and embedded within a continuous representation. Positional encodings are employed to preserve the sequence order, while special tokens indicating the beginning and end of sequences demarcate the boundaries. To achieve a predetermined coverage score that exceeds the original predicted score, a constrained gradient-based optimizer is implemented to update the continuous representation of the instruction sequence iteratively. The sequence update mechanism is illustrated in Fig. 4, where the dotted arrows in the figure indicate the various levels of continuous representations of sequences. The iterative process commences with a randomly initialized continuous representation, from which it computes the gradients of the predicted coverage scores with ISA constraints and syntactical requirements, guiding the modifications to minimize the difference between the predicted and target coverage scores.

We implement two-level constraints before and following the gradient updating process to ensure that the optimized sequence remains compliant with the ISA and maintains syntactical accuracy. During each gradient update iteration, we enforce ISA constraints by projecting the gradients into the feasible region defined by the ISA. This projection process is essential to convert calculated gradients into constrained gradients, thereby ensuring valid register utilization by setting gradients of reserved registers to zero and adhering to opcode and immediate value ranges pertinent to the specific instruction types. Upon completion of the gradient updates, we validate the optimized sequence through a correction tool that assesses legality and rectifies any incompatible elements. A syntax correctness checker evaluates the syntactic integrity of the instruction sequences by examining instruction dependencies, memory access patterns, and control flow stability. Furthermore, the correction script interprets each instruction within the optimized sequence, reallocates reserved registers by assigning available alternatives, corrects out-of-range immediate values by generating new values within defined maximum or minimum boundaries, and adjusts opcodes to the nearest similar alternative while incorporating random operands.

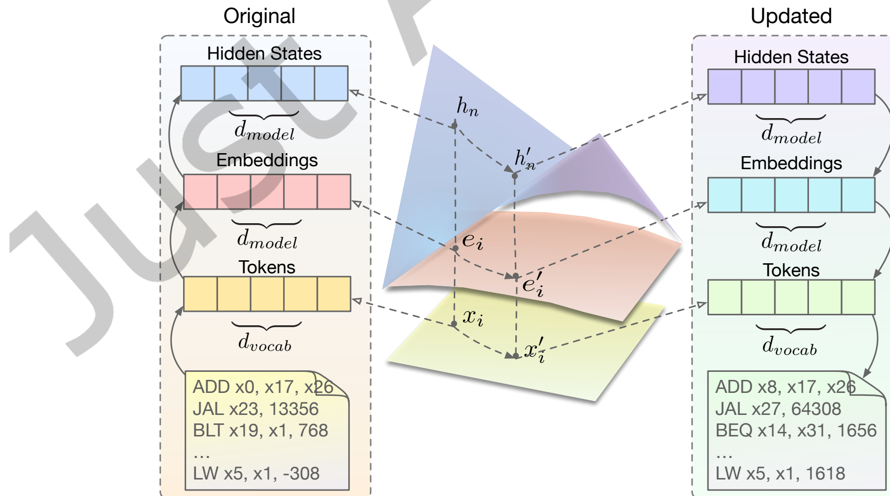


Fig. 4. The gradient-based sequence update flow via transformer language model.

Just Accepted

5 EXPERIMENTS

5.1 Experimental Settings

Dataset. The dataset comprises instruction-level test sequences generated by the open-source ISG known as FORCE-RISCV [12]. This generator adheres to the constraints outlined in the RISC-V ISA specifications. FORCE-RISCV employs a random selection process for instructions, registers, addresses, and immediate values to produce valid instructions that fully support the RISC-V ISAs, encompassing `RV64G`, as well as `RV64 IMAFDC`, `U`, `S`, and `M` privilege levels, traps, exceptions, and virtual memory systems featuring page sizes of 4 KB, 2 MB, 1 GB, and 512 GB.

The instruction stream generator utilizes templates that encompass the instruction set, weights, address constraints, and relative configurations to create executable files in ELF format and disassembled text files of the generated instructions, as depicted in Fig. 5. The test templates, implemented in Python, facilitate control over the instruction generation process by invoking APIs designed for test sequence generation to define constraints and instruction types. These Python templates are designed to be user-friendly, allowing for easy determination of the sequence of generated instructions and the corresponding constraints. Moreover, the generation engine integrates interfaces with the instruction set simulator [35], which is capable of simulating the behavior of the generated instructions. The resultant output includes a standard ELF file and a disassembled text file.

Model Configurations. The transformer language model utilized within our DeepVerifier framework processes input sequences of up to 1024 tokens. It consists of 512 embeddings, 4 blocks of multi-head attention layers with 4 attention heads in each. The model is capable of generating outputs with a maximum length of 1024 tokens, which is subsequently followed by a four-layer feed-forward network. Both the encoder and decoder maintain hidden state dimensions of 512 while the intermediate size of the feed-forward network is set at 1024. The vocabulary encompasses a total of 1216 token types. Key parameters include the maximum input length, embedding dimensions, the number of blocks, the number of attention heads, hidden dimensions, and the number of layers within the feed-forward network.

Training Configurations. We utilize PyTorch version 2.4.0, Transformer version 4.39.2, and Tokenizers version 0.15.2 to implement the model. The training process is conducted on a single Nvidia GeForce RTX 4090 GPU which has 24 GB of memory, alongside the Nvidia CUDA Toolkit version 12.4. The fine-tuned transformer model

Just Accepted

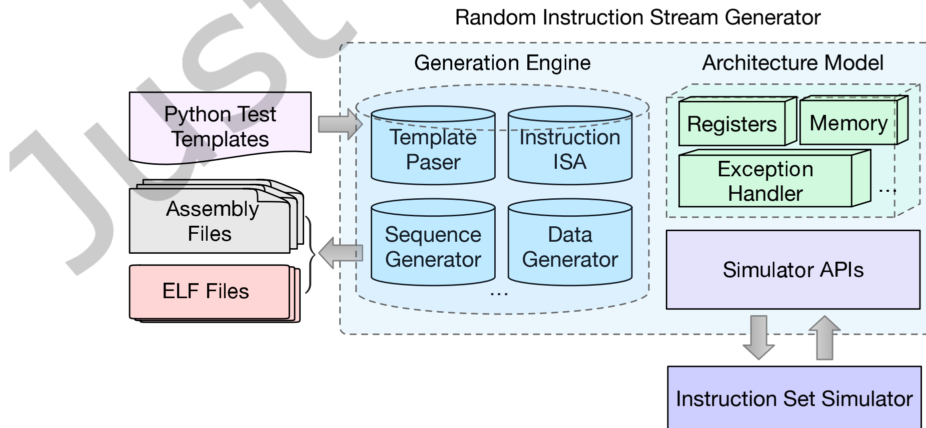


Fig. 5. Dataset generation via random instruction stream generator.

Table 1. Descriptions of coverage metrics.

Coverage	Descriptions
Line	Lines or statements were executed.
Toggle	Signals toggle from 0 to 1 and 1 to 0.
Condition	Both true and false states of conditions were covered.
Branch	If, case statements, the ternary operator (?:) branches of execution.

achieves convergence after training for 100 epochs, utilizing a batch size of 2, an initial learning rate of $5e-5$, and a weight decay of 0.01 as part of the AdamW optimizer. The learning rate is adjusted according to a cosine function, decreasing from the initial value to zero, with several hard restarts after a warm-up period characterized by a linear increase from zero to the initial value. To converge the coverage predictor, we train a neural network with 4 hidden layers, employing an initial learning rate of $1e-3$, a weight decay of 0.01 as part of the AdamW optimizer, a batch size of 32, and 100 epochs. The training and validation datasets are randomly shuffled and split into a ratio of 80% for training and 20% for validation.

The training process involves two main components, the pertaining phase of a language model and a downstream task of the coverage predictor. For the fine-tuning phase using the BART-base model with 139M parameters of open-source Hugging Face [36], each epoch processes approximately 1500 sequences in a total time of 4 hours for the full 100 epochs. The relatively long training time can be attributed to the small batch size of 2, which was necessitated by GPU memory constraints. Despite these limitations, the model successfully converges to achieve optimal performance. The subsequent training of the coverage predictor is considerably faster, completing in approximately 40 minutes due to its simpler architecture and 4 batch size capability.

Coverage Metrics. The coverage score serves as a fundamental evaluation criterion for assessing the efficacy of the proposed sequence update strategy utilizing the transformer model of test sequences. The primary aim is to enhance the coverage score of each test sequence. The definitions of the coverage metrics commonly employed to evaluate the quality of input test sequences are presented in Table 1. By compiling and executing the ELF file corresponding to each test sequence on a simulated processor platform, we can obtain coverage scores for the generated input test sequences. This simulation utilizes a modest configuration of the Berkeley Out-of-Order Machine (BOOM) [37] and RocketChip derived from [38], facilitated by the Synopsys VCS [25] and Verilator [24] simulators.

5.2 Experimental Results

We compare the proposed test sequence update methodology against three different approaches: random generation, mutational fuzzing testing, and methods based on long short-term memory (LSTM). For the random generation, we utilize a template iteratively provided by FORCE-RISCV ISG, employing diverse random seeds to generate 1000 test sequences, each containing between 128 and 256 instructions of the `RV64GC` ISA.

To implement the mutation method, we adopt the same strategy utilized in existing fuzzing test techniques for instructions, as referenced in prior works [20], and [18]. This involves modifying source registers, introducing random immediate values, and generating instructions from a pool of valid instruction candidates for each test sequence within a minute timeframe.

To establish a conventional representation model for a test sequence, we utilize an LSTM-based token representation model [23]. Specifically, we construct a 4-hidden-layer LSTM model that mirrors the embedding dimensions of advanced transformer models, thereby enabling it to share the same coverage score predictor as the transformer model.

Just Accepted

Table 2. Average coverage and runtime of different test sequence generation methods.

Coverage/Runtime	Random [12]	Mutation [20]	LSTM [23]	DeepVerifier
Line	76.753122	79.286422	78.791064	**83.333333**
Branch	71.879106	79.080824	77.777780	**79.286422**
Toggle	62.054130	66.666670	66.032580	**68.977651**
Runtime (s)	955	1066	150	176

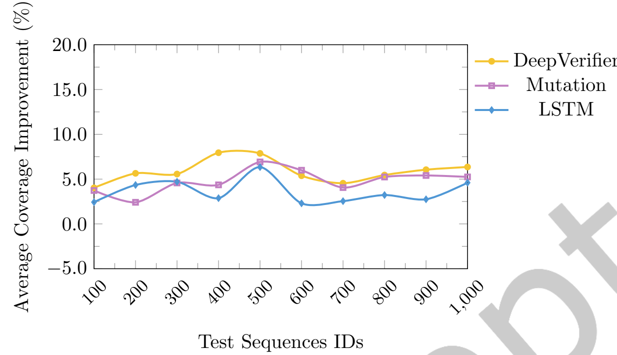


Fig. 6. Average coverage improvement of DeepVerifer, LSTM body method, and mutation method over random generation.

In Table 2, we present the accumulation coverage scores and runtime performance across 1000 test sequences. Branch coverage implicitly includes condition coverage since taking both branches requires conditions to be evaluated to be both true and false. The coverage collector combines them as branch coverage. Our approach demonstrates average coverage improvements of 6.97%, 2.19%, and 3.00% compared to random generation, mutation fuzzing, and LSTM representation, respectively. Notably, while our method requires slightly more runtime than the LSTM model, it achieves this superior coverage in significantly less time than both random generation and mutation fuzzing, indicating better efficiency in reaching coverage targets.

The runtime efficiency of DeepVerifier can be attributed to the ability of a transformer-based model to learn quickly and leverage patterns from existing test sequences and the gradient-based optimization process that efficiently guides sequence generation toward uncovered directions. This is particularly evident when compared to mutation fuzzing, which requires 5.6x more runtime while achieving lower coverage.

The comparative analysis in Fig. 6 demonstrates the superior performance stability of DeepVerifier, maintaining an average improvement of $6-8\%$ over random generation throughout the sequence range. This consistent performance advantage, combined with the favorable runtime characteristics, validates our transformer-based approach's efficiency in achieving higher coverage with reasonable computational overhead.

To provide deeper insights into the performance dynamics, we visualize the coverage improvements for each approach in Figs. 7 to 9. As shown in Fig. 7, our method maintains consistently positive improvements across test cases. This stable performance contrasts with mutation fuzzing (see Fig. 8), where some test sequences show declining coverage, likely due to imbalances in instruction types. The limitation of human-defined mutation strategies, which randomly modify operands and opcodes from a candidate pool, proves insufficient for maintaining diverse test patterns.

The performance of LSTM model in Fig. 9 shows limitations in extracting meaningful patterns from assembly test sequences, with many sequences showing negative improvement compared to random generation. This

Just Accepted

suggests that the simpler sequential modeling of LSTM fails to capture the complex dependencies present in assembly code effectively.

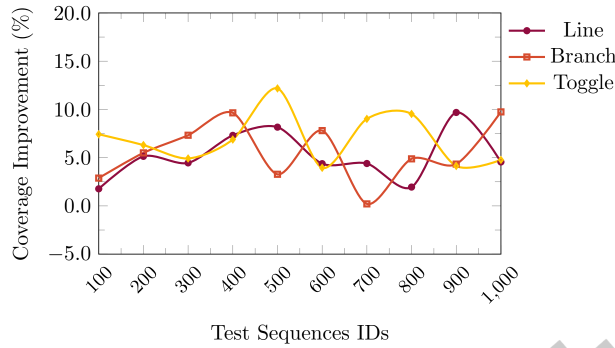


Fig. 7. Coverage improvement of our transformer method over random generation.

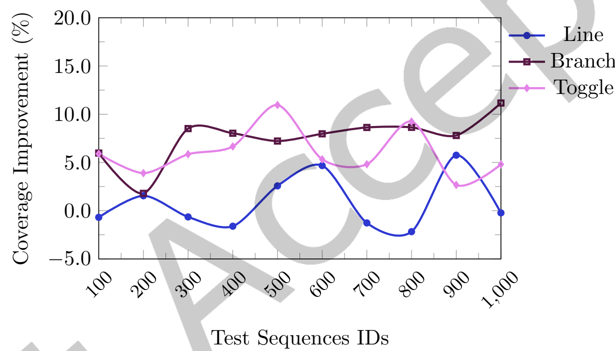


Fig. 8. Coverage improvement of the mutation over random generation.

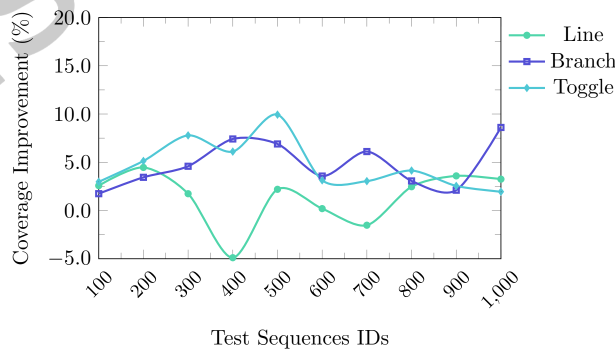


Fig. 9. Coverage improvement of the LSTM body method over random generation.

Just Accepted

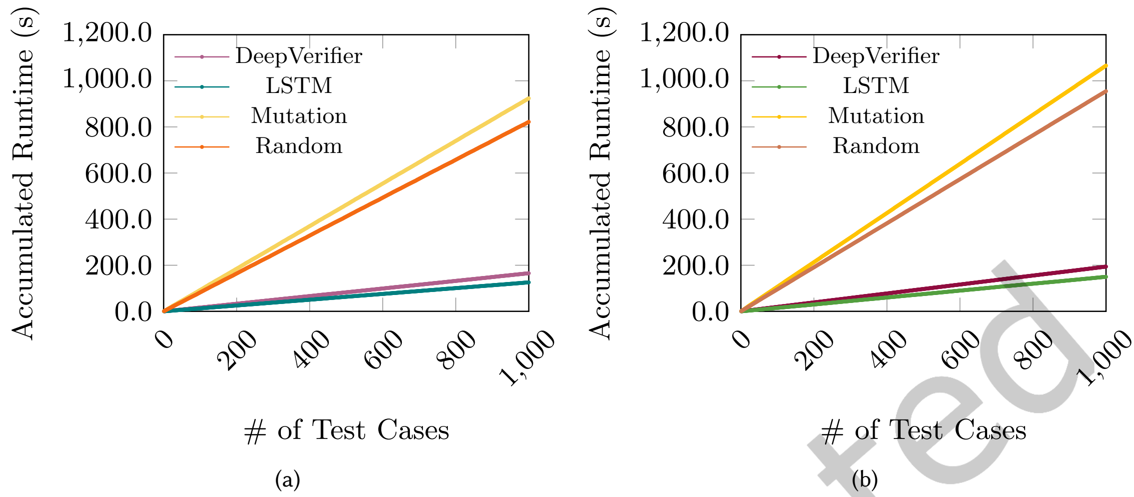


Fig. 10. (a) Accumulated runtime of exercising test cases on RocketChip; (b) Accumulated runtime of exercising test cases on BOOM.

The success of our transformer-based approach stems from the sophisticated representation ability of the transformer language model, which excels in extracting contextual features from instruction-level test sequences, and accurately estimating correlations between input sequences and coverage scores. Furthermore, our approach rapidly evaluates sequence quality during early verification stages and efficiently optimizes test sequence generation through gradient-based updates.

To explore the scalability and generalizability of our approach, we conducted extensive experiments on two different RISC-V processor implementations, the in-order RocketChip and the out-of-order BOOM core. Fig. 10(a) illustrates the accumulated runtime trend for executing 1000 test cases on RocketChip, showing a near-linear growth pattern that suggests increasing complexity as more test cases are processed. In parallel, Fig. 10(b) presents the accumulated runtime of exercising 1000 test cases on the BOOM core, exhibiting a similar trend but with higher overhead due to BOOM's more sophisticated out-of-order execution architecture. The comparative analysis between these two implementations reveals that while both processors show consistent scalability patterns, BOOM requires approximately 1.5x more runtime than RocketChip due to its complex microarchitectural features. This observation validates our approach's generalizability across different processor architectures while highlighting the impact of architectural complexity on verification overhead.

5.3 Ablation Study

The primary objective of ablation analysis is to demonstrate the robustness of our proposed strategy and to confirm the proper functionality of each component. We assess the performance of components that incorporate the semantic representation capabilities of a transformer model, as well as the accuracy of the coverage score predictor. Additionally, we compare the transformer model with LSTM networks to validate its superior capabilities in semantic representations. The evaluation findings will elucidate the novelty and effectiveness of our test sequence generation and updates, which are aimed at enhancing the quality of exercises for processor verification.

In Fig. 11, the variation in loss during the training phase of the transformer model and the downstream coverage predictor is illustrated. Fig. 11(a) depicts the trajectory of the cross-entropy loss for the transformer language

Just Accepted

model, which shows a decreasing trend that approaches a plateau as training epochs progress. This downward trend suggests that the model is effectively learning from the existing data distribution and demonstrating enhanced capacity to minimize the divergence between the learned distribution and the target distribution. The low loss values observed for both training and validation datasets indicate that the transformer model is proficient in representing and extracting semantic information from the input sequences. By utilizing the embedded hidden state vectors from the transformer model, the coverage predictor associates these represented sequences with their corresponding coverage scores. Fig. 11(b) demonstrates that the MSE of the coverage predictor declines over time throughout both the training and validation phases. This reduction signifies an improvement in the accuracy of predicting coverage scores for test sequences as the number of training epochs increases.

To further substantiate the predictive capabilities of the transformer model in the downstream coverage prediction task, Fig. 12 illustrates the deviation of coverage scores generated by the transformer model with the ground truth scores provided by a simulator. On average, the deviation error remains below 1% for each coverage metric assessed. The training metrics detailed in Fig. 11, along with the prediction accuracy demonstrated in Fig. 12, offer compelling evidence that the transformer method effectively represents test sequences. By deriving embedded vectors for these sequences, the downstream coverage predictor accurately characterizes the correlation between sequence representations and their corresponding coverage scores. Additionally, while LSTM models also generate embedded representations of sequences, Fig. 12(d) presents a comparative performance analysis between the transformer model and LSTM.

Through the comprehensive visualizations associated with the training and validation processes of the transformer model, as well as the evaluation of coverage score predictions, we aim to demonstrate the efficacy of the systematic training methodologies employed, as evidenced by precise loss rates. Furthermore, this analysis collectively substantiates the superiority of the transformer model in comparison to traditional LSTM language representation techniques and random test generation approaches. The transformer model exhibits reduced prediction errors and enhanced coverage scores, which are instrumental in optimizing test sequence generation for processor verification.

Just Accepted

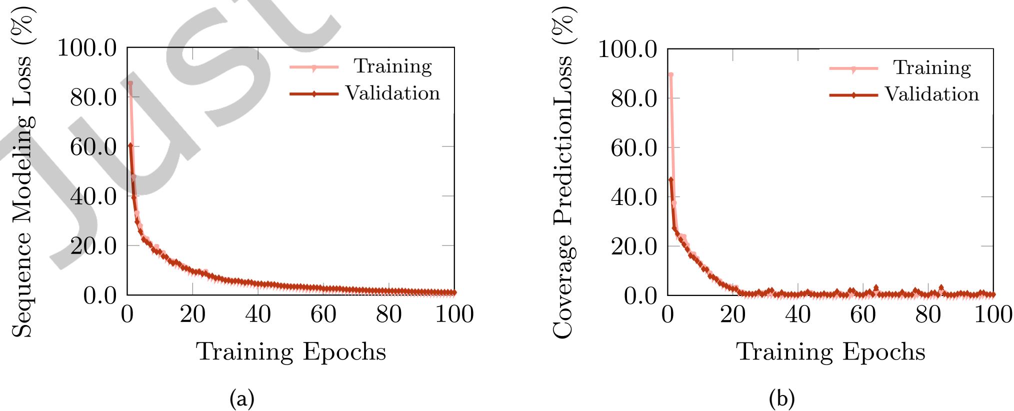


Fig. 11. (a) Training loss of the transformer language model; (b) Training loss of coverage prediction model.

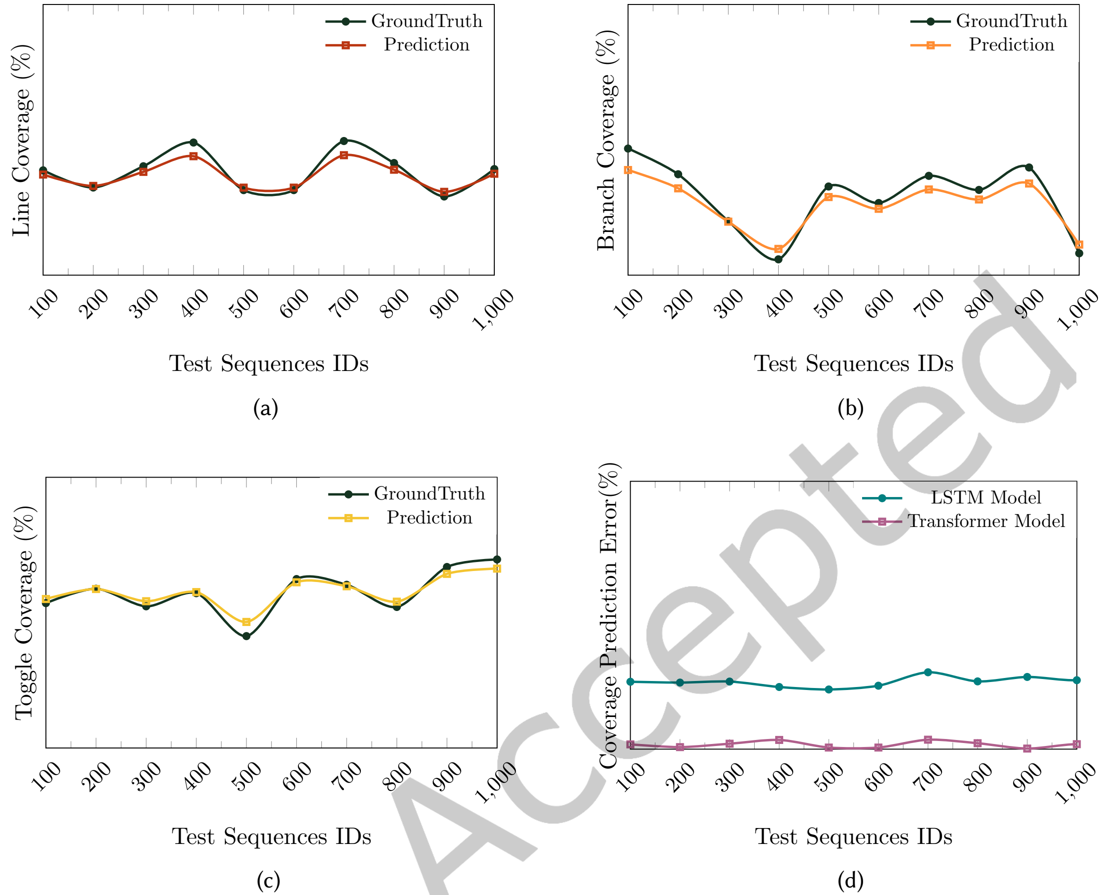


Fig. 12. (a) ~ (c) Deviation of predicted line, branch, and toggle coverage with the ground truth in the test dataset. (d) Comparison of predicted error between transformer and LSTM model body.

Just Accepted

6 Conclusion

Verification efforts are heavily based on the quality and diversity of test sequences, which are essential for effective verification practices. The coverage-directed random generation method produces various test sequences; however, it requires human input to create test templates and configure architectures. In addition, achieving coverage with simulators can be a time-consuming process.

To harness the capabilities of language models for semantic representation and efficient sequence generation, we propose a coverage-directed test generation strategy that employs a transformer model and a coverage predictor based on randomly generated test sequences. This strategy can integrate semantic representations of instructions, quickly align coverage predictions to illustrate the relationship between sequences and coverage, and ultimately enhance sequences for improved coverage.

While our current work demonstrates effectiveness on the RISC-V processor implementation, we acknowledge several important directions for future research, to enhance the scalability to evaluate DeepVerifier on larger

configurations with more complex out-of-order execution features, we need to investigate performance on designs with advanced features and handle increased state space in larger processor designs. In addition to exploring the generalizability of DeepVerifier, we need to extend to support other designs and adapt the instruction embedding and tokenization schemes for different ISAs. Create a more generic coverage modeling approach that can accommodate diverse architectural features, and the current implementation can serve as a proof-of-concept, and extensions would help establish DeepVerifier as a more comprehensive verification solution for modern processor designs.

References

[1] Andrew Waterman, Yunsup Lee, David Patterson, and Krste Asanovic. 2016. The RISC-V Instruction Set Manual, Volume I: User-Level ISA Version 2.1. *ECS Department, UC Berkeley, Technical Report UCB/EECS-2016-118* (May 2016).

[2] Andrew Waterman, Yunsup Lee, Rimas Avižienis, David A Patterson, and Krste Asanovic. 2016. The RISC-V Instruction Set Manual, Volume II: Privileged Architecture Version 1.9. *ECS Department, UC Berkeley, Technical Report UCB/EECS-2016-129* (July 2016).

[3] Harry Foster. 2020. Wilson Research Group Functional Verification Study. https://resources.sw.siemens.com/en-US/white-paper-2020-wilson-research-group-functional-verification-study-ic-asic-fucntional-verification-trend-report. (2020).

[4] Jun Yuan, Carl Pixley, Adnan Aziz, and Ken Albin. 2003. A Framework for Constrained Functional Verification. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 142–145.

[5] Ilya Wagner, Valeria Bertacco, and Todd Austin. 2005. Stresstest: An Automatic Approach to Test Generation via Activity Monitors. In *ACM/IEEE Design Automation Conference (DAC)*. 783–788.

[6] Nathan Kitchen and Andreas Kuehlmann. 2007. Stimulus Generation for Constrained Tandom Simulation. In *ACM/IEEE Design Automation Conference (DAC)*. 258–265.

[7] Rajdeep Mukherjee, Daniel Kroening, and Tom Melham. 2015. Hardware Verification Using Software Analyzers. In *IEEE Annual Symposium on VLSI (ISVLSI)*. 7–12.

[8] Jeyavijayan Rajendran, Vivekananda Vedula, and Ramesh Karri. 2015. Detecting Malicious Modifications of Data in Third-party Intellectual Property Cores. In *ACM/IEEE Design Automation Conference (DAC)*. 1–6.

[9] Yanhong Zhou, Tiancheng Wang, Huawei Li, Tao Lv, and Xiaowei Li. 2015. Functional Test Generation for Hard-to-reach States Using Path Constraint Solving. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 35, 6 (2015), 999–1011.

[10] Rui Zhang, Calvin Deutschbein, Peng Huang, and Cynthia Sturton. 2018. End-to-end Automated Exploit Generation for Validating The Security of Processor Designs. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 815–827.

[11] Nursultan Kabylkas, Tommy Thorn, Shreesha Srinath, Polychronis Xekalakis, and Jose Renau. 2021. Effective Processor Verification with Logic Fuzzer Enhanced Co-simulation. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 667–678.

[12] Inc. Futurewei Technologies. 2024. FORCE-RISCV: An Instruction Sequence Generator (ISG) for The RISC-V Instruction Set Architecture. https://github.com/openhwgroup/force-riscv. (2024).

[13] Google. 2020. RISCV-DV: An SV/UVM-based Open-source Instruction Generator for RISC-V Processor Verification. https://github.com/chipsalliance/riscv-dv. (2020).

[14] Neel Gala and Marc Karasek. 2020. RISC-V Torture Test Generator. https://github.com/ucb-bar/riscv-torture. (2020).

[15] Yangdi Lyu and Prabhat Mishra. 2020. Automated Test Generation for Activation of Assertions in RTL Models. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC)*. 223–228.

[16] Rahul Kande, Addison Crump, Garrett Persyn, Patrick Jauernig, Ahmad-Reza Sadeghi, Aakash Tyagi, and Jeyavijayan Rajendran. 2022. heHuzz: Instruction fuzzing of processors using Golden-Reference models for finding Software-Exploitable vulnerabilities. In *USENIX Security Symposium*. 3219–3236.

[17] Sadullah Canakci, Leila Delshadtehrani, Furkan Eris, Michael Bedford Taylor, Manuel Egele, and Ajay Joshi. 2021. DirectFuzz: Automated Test Generation for RTL Designs Using Directed Graybox Fuzzing. In *ACM/IEEE Design Automation Conference (DAC)*. 529–534.

[18] Vladimir Herdt, Sören Tempel, Daniel Große, and Rolf Drechsler. 2021. Mutation-based Compliance Testing for RISC-V. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC)*. 55–60.

[19] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. 2021. DifuzzRTL: Differential Fuzz Testing to Find CPU Bugs. In *IEEE Symposium on Security and Privacy (SP)*. 1286–1303.

[20] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. 2018. RFUZZ: Coverage-directed Fuzz Testing of RTL on FPGAs. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8.

[21] Shai Fine and Avi Ziv. 2003. Coverage Directed Test Generation for Functional Verification using Bayesian Networks. In *ACM/IEEE Design Automation Conference (DAC)*. 286–291.

[22] Fanchao Wang, Hanbin Zhu, Pranjay Popli, Yao Xiao, Paul Bodgan, and Shahin Nazarian. 2018. Accelerating Coverage Directed Test Generation for Functional Verification: A Neural Network-Based Framework. In *ACM Great Lakes Symposium on VLSI (GLSVLSI)*.

Just Accepted

207–212.
[23] Shobha Vasudevan, Wenjie Joe Jiang, David Bieber, Rishabh Singh, C Richard Ho, Charles Sutton, and oThers. 2021. Learning Semantic Representations to Verify Hardware Designs. *Annual Conference on Neural Information Processing Systems (NIPS)* 34 (2021), 23491–23504.
[24] CHIPS Alliance under The Linux Foundation. 2024. Verilator: The Fastest Verilog/SystemVerilog Simulator. https://github.com/verilator/verilator. (2024).
[25] Inc. Synopsys. 2024. Synopsys VCS® Functional Verification Solution. https://www.synopsys.com/verification/simulation/vcs.html. (2024).
[26] Walker Anderson. 1992. Logical Verification of The NVAX CPU Chip Design. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 306–309.
[27] David A Wood, Garth A Gibson, and Randy H Katz. 1990. Verifying a Multiprocessor Cache Controller Using Random Test Generation. *European Design and Test Conference* 7, 4 (1990), 13–25.
[28] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Empirical Methods in Natural Language Processing (EMNLP)*. 38–45.
[29] Xuezixiang Li, Yu Qu, and Heng Yin. 2021. Palmtree: Learning An Assembly Language Model for Instruction Embedding. In *ACM Conference on Computer and Communications Security (CCS)*. 3236–3251.
[30] Jacob Devlin Ming-Wei Chang Kenton and Lee Kristina Toutanova. 2019. Bert: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*. 4171–4186.
[31] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, and oThers. 2018. Improving Language Understanding by Generative Pre-training. In *OpenAI*.
[32] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A Robustly Optimized Bert Pretraining Approach. In *arXiv preprint*. arXiv:1907.11692.
[33] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. 2020. Bart: Denoising Sequence-to-sequence Ore-training for Natural Language Generation, Translation, and Comprehension. In *Annual Meeting of the Association for Computational Linguistics (ACL)*. 7871–7880.
[34] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. *Annual Conference on Neural Information Processing Systems (NIPS)* 30 (2017).
[35] RISC-V Software. 2019. Spike RISC-V ISA Simulator. https://github.com/riscv-software-src/riscv-isa-sim. (2019).
[36] Facebook. BART(base-size model). https://huggingface.co/facebook/bart-base. ([n. d.]).
[37] Christopher Celio, David A Patterson, and Krste Asanovic. 2015. The Berkeley Out-of-order Machine (BOOM): An Industry-competitive, Synthesizable, Parameterized RISC-V Processor. *ECS Department, UC Berkeley, Technical Report UCB/EECS-2015-167* (June 2015).
[38] Asanovic Krste, Avizienis Rimas, Bachrach Jonathan, Beamer Scott, Biancolin David, Celio Christopher, Cook Henry, Dabbelt Palmer, Hauser John, Izraelevitz Adam, Karandikar Sagar, Keller Benjamin, Kim Donggyu, Koenig John, Lee Yunsup, Love Eric, Maas Martin, Magyar Albert, Mao Howard, Moreto Miquel, Ou Albert, Patterson David, Richards Brian, Schmidt Colin, Twigg Stephen, Vo Huy, and Waterman Andrew. 2016. The Rocket Chip Generator. *ECS Department, UC Berkeley, Technical Report UCB/EECS-2016-17* (April 2016).

Received 7 November 2024; revised 21 January 2025; accepted 17 February 2025

Just Accepted