SimFuzz：面向 RISC-V处理器模糊测试的相似度引导的块级变异

Hao Lyu, Jingzheng Wu✉, Xiang Ling✉, Yicheng Zhong, Zhiyuan Li, Tianyue Luo
中国科学院大学 中国科学院软件研究所 中国科学院

arXiv:2601.11838v1 [cs.CR] 17 Jan 2026

摘要—**指令集体系结构（ISA）定义了处理器的操作，并充当硬件与软件之间的接口。作为一种开源指令集架构，RISC-V降低了处理器设计的门槛并推动了广泛采用，但也使处理器面临功能缺陷等安全风险。处理器模糊测试是自动检测此类缺陷的有效技术。然而，现有模糊测试方法存在两个主要局限。首先，它们过于强调冗余的测试用例生成，导致忽视跨处理器边界情况。其次，它们过度依赖覆盖率引导。当前的覆盖率指标存在偏差且效率低下，并且一旦覆盖率增长陷入停滞，就会失去效果。**

为克服这些局限，我们提出 SimFuzz，这是一种从历史触发缺陷的输入中构建高质量种子语料库，并采用相似度引导的块级变异来高效探索处理器输入空间的模糊测试框架。通过引入指令相似性，SimFuzz在保留控制流结构的同时扩展种子周围的输入空间，从而无需依赖覆盖率反馈即可实现更深入的探索。我们在三种广泛使用的开源 RISC-V处理器上评估了 SimFuzz：Rocket、BOOM 和香山，共发现17个缺陷，其中包括14个此前未知的问题，且其中7个已分配 CVE 标识符。这些缺陷会影响解码单元和存储单元，导致指令和数据错误，并可能引发内核不稳定或系统崩溃。实验结果表明，SimFuzz在高质量种子语料库上实现了最高73.22%的多路复用器覆盖率。我们的研究揭示了主流 RISC-V处理器中的关键安全缺陷，并为改进功能验证提供了可操作的见解。

索引词—**处理器验证、安全性、模糊测试**

I. 引言

指令集架构是连接硬件与软件的基础桥梁，定义了处理器能够解读并执行的完整操作集合 [4]。作为一种开放且免费提供的指令集架构，RISC-V 降低了处理器设计的门槛，推动其得到广泛采用，目前已有数百万个处理器部署在各处 [25]。与此同时，RISC-V处理器面临日益严峻的安全挑战。GhostWrite [6], 就是一个例子：它是平头哥玄铁 [31] RISC-V处理器中的一项严重缺陷。该缺陷使攻击者能够绕过特权强制执行机制、获得任意内存访问权限并控制外围设备。其根本原因在于玄铁处理器支持一条违反指令集体系结构规范的非标准指令。这类实现特定缺陷通常源于处理器寄存器传输级（RTL）的设计缺陷，并最终表现为功能缺陷。这些缺陷会直接影响软件正确性，即使不存在对抗性输入也可能发生。与软件不同，处理器一经制造便无法打补丁 [5]。由于 RTL 的功能和时序行为在流片后保持不变，因此在制造前进行全面的RTL级验证对于确保正确性和增强安全性至关重要。

现代处理器极其复杂，对其状态空间进行穷尽探索并不可行。因此，处理器模糊测试 [1], [7], [8], [10]–[12], [16], [18], [30], [32]应运而生，成为发现 RTL 级漏洞的一种有效且可扩展的方法。当前的模糊测试器通常依靠覆盖率指标来追踪处理器状态的代理指标。例如，多路复用器（mux）覆盖率 [18]关注多路复用器控制信号。在模糊测试过程中，这些覆盖率指标会引导变异，使变异种子能够探索更广泛的处理器状态。

尽管前景可期，覆盖率引导的处理器模糊测试器仍面临两大主要局限。首先，由于生成了大量冗余测试用例，它们往往无法捕获跨处理器边界情况。例如，PathFuzz [32]使用真实世界的程序（如 SPEC CPU2006）扩充其种子语料库，并对香山 [23]处理器进行模糊测试。即使拥有大量种子，它仍未能发现某些边界情况错误，例如我们研究中发现的错误 B7（表 III）。其次，模糊测试器过度依赖覆盖率引导。ProcessorFuzzer [11] 和 DifuzzRTL [7]设计了新的覆盖率指标来引导模糊测试，但指标的选择往往反映了人为偏见，从而限制了验证有效性 [29]。选择的指标过少会导致验证不足，而追踪过多指标则会增加开销、降低效率。此外，当覆盖率进入平台期 [20], 时，模糊测试器便失去有效引导，严重限制了探索。与此同时，覆盖率提升并不一定能带来有效的漏洞发现 [15]。

由于处理器状态由输入驱动，要探索更广泛的状态，就需要大量能够触发多种状态的高质量测试用例。因此，我们将处理器验证表述为探索处理器输入空间的问题。为此，我们提出了 SimFuzz。首先，SimFuzz 从真实世界处理器中收集触发错误的测试用例，构建高质量种子语料库，确保探索从更可能暴露错误的输入空间区域开始。其次，它在变异过程中保留这些种子的控制流结构，从而维持语义有效性。最后，SimFuzz 引入指令相似性作为引导机制。与覆盖率指标不同，相似度能够捕捉种子之间的细粒度差异，从而更有效地探索输入空间。

我们在三款广泛使用的开源

RISC-V处理器上评估了 SimFuzz：Rocket [2],BOOM [24],和香山 [23]。SimFuzz 在构建的种子语料库上实现了 73.22% 的多路复用器覆盖率。与 Csmith [3] 和 Cascade [30],生成的种子相比，SimFuzz 在覆盖率方面略胜 PathFuzz [32]。更重要的是，SimFuzz 总共发现了 17 个错误，其中包括 14 个此前未知的问题，已有 7 个被分配了 CVE 标识符。这些错误会影响解码和存储单元，导致错误译码和数据错误，并可能造成内核不稳定或系统崩溃。

总而言之，本文做出了三项关键贡献：

- 我们收集了来自三款开源 RISC-V 处理器（Rocket、BOOM 和香山）的触发错误的测试用例，构建了一个真实、高质量的种子语料库。
- 我们提出了 SimFuzz[1]，一种全新的处理器模糊测试框架，以基于相似度的块级变异取代覆盖率引导策略。
- 评估结果表明，SimFuzz 实现了更高的覆盖率，并发现了 14 个此前未知的错误，其中 7 个已获分配 CVE 标识符。

II. 背景

A. RISC-V

RISC-V 是一种开放的指令集架构，旨在实现模块化和可扩展性 [26], [27]。处理器可以仅实现最精简的基础整数指令集，也可以通过扩展增加额外功能。为确保生态系统的一致性，RISC-V 定义了一种标准通用配置，称为 G 扩展集，其中包括整数、乘法/除法、原子操作以及浮点（单精度和双精度）扩展。

G 扩展规定了六种指令格式 [26]：寄存器-寄存器（R 型）、寄存器-立即数（I 型）、存储（S 型）、条件分支（B 型）、高位立即数（U 型）和无条件跳转（J 型）。表 I 总结了这些 32 位指令的格式。

表 I：RISC-V 指令格式

31:25	24:20	19:15	14:12	11:7	6:0	Type
funct7字段	rs2	rs1	funct3字段	rd	操作码	R
imm		rs1	funct3字段	rd	操作码	I
imm	rs2	rs1	funct3字段	imm	操作码	S
imm	rs2	rs1	funct3	imm	操作码	B
imm 字段				rd	操作码	U
imm				rd	操作码	J

指令的语义决定处理器的状态。在 RISC-V 指令中，操作码和 funct 字段共同决定语义。操作码指定指令格式和基本功能，而 funct 字段则进一步细化处理器执行的具体操作。在实际应用中，无法直接从输入空间观察处理器的内部状态空间，因此难以明确推理状态空间覆盖率。然而，处理器状态转换从根本上由指令语义驱动：处理器根据每条指令的语义定义对其进行解释，并更新其架构状态和微架构状态。因此，尽管无法访问确切的内部状态，指令的语义属性可以作为可靠的代理，用于预测这些指令对处理器状态的影响。

B. 处理器模糊测试

开源 RISC-V 处理器的快速发展，使处理器模糊测试成为日益重要的验证技术。模糊测试通过变异种子并观察其与预期行为的偏差，系统地探索处理器状态。RFUZZ [18]是首个硬件模糊测试框架，它利用多路复用器覆盖率验证 RTL 的正确性。在此基础上，后续研究探索了多种输入生成和变异技术。

生成式处理器模糊测试需要创建大量测试用例。Cascade [30]引入了一种非对称的 ISA预仿真机制，在输入生成过程中交织控制流与数据流。然而，其种子往往在输入空间中聚集得非常接近，限制了多样性，并减缓覆盖率的增长。ChatFuzz [9] 和 GenHuzz [19]采用强化学习生成测试用例。尽管这些方法前景可观，但计算和时间成本高昂，且难以探索罕见但关键的状态。总而言之，尽管生成式模糊测试能够生成大量种子并实现较高覆盖率，但它也引入了大量冗余，仍难以有效揭示某些边界情况。

覆盖率引导的变异模糊测试会在指令级别修改现有种子，以扩展输入空间并发现处理器边界状态。DifuzzRTL [7],、TheHuzz [10],和 MorFuzz [8]根据覆盖率指标执行逐指令变异。然而，这些技术往往会破坏原始种子的语义结构，扰乱控制流，从而生成无效或效果较差的测试用例。PathFuzz [32]会预处理种子，将记忆从线性布局转换为足迹布局，然后使用 LibAFL [14] 根据覆盖率指标引导变异。作为一种软件模糊测试库，LibAFL 缺乏硬件感知能力。总体而言，这些覆盖率引导的变异方法严重依赖覆盖率指标，却忽视了种子的语义结构，因而限制了它们充分利用现有测试用例中所蕴含信息的能力。

III. SIMFUZZ

SimFuzz 是一种处理器模糊测试框架，它采用相似性引导的变异策略探索输入空间，而非依赖覆盖率反馈。如图 1 所示，SimFuzz 首先从曾触发真实漏洞的测试用例中构建高质量种子语料库（第 III-A 节）。在变异阶段（第 III-B 节），它会保留每个种子的控制流结构，同时根据指令相似性扩展输入空间。最后，通过差分测试（第 III-C 节）评估变异种子，以揭示被测处理器与 ISA模拟器之间的不一致。

A. 高质量种子语料库构建

种子语料库的质量会显著影响模糊测试的效果 [17]。高质量的种子能引导模糊测试器更快地到达关键执行路径，从而提高早期发现漏洞的可能性。与早期的模糊测试器不同 [11], [12], [18]

[1]https://github.com/has2lab/SimFuzz

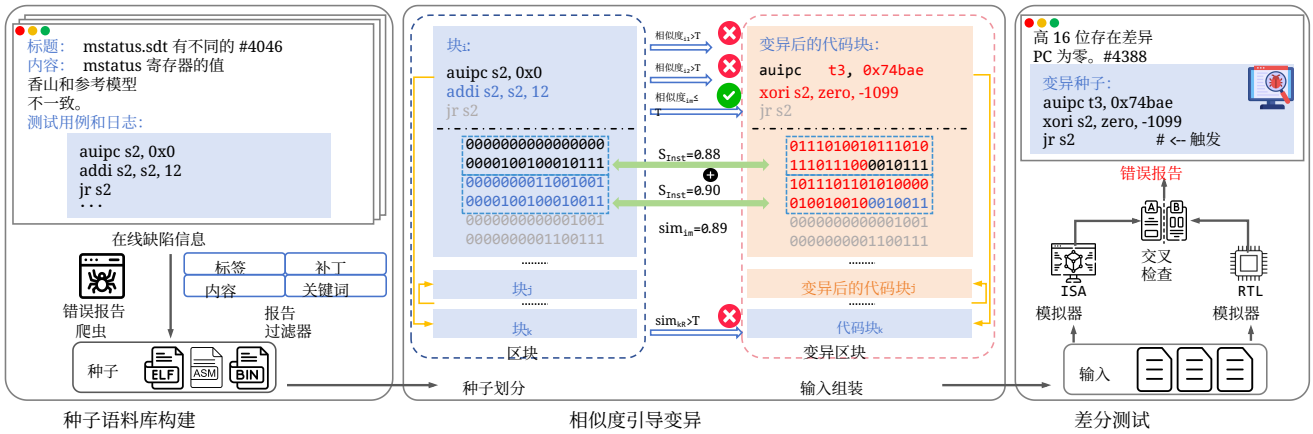


图 1：SimFuzz 工作流概览。

早期模糊测试器使用空语料库或专注于人工生成的种子 [9], [19], [30]。SimFuzz 直接从真实世界中触发错误的测试用例构建语料库。其核心见解是，处理器漏洞往往源于对指令集体系结构规范的误解，或细微的设计偏差。通过重用已触发真实漏洞的测试用例，SimFuzz 确保探索从输入空间中有意义的点开始，而不是从随机点开始。

为构建其种子语料库，SimFuzz 实现了一个在线漏洞报告爬虫和过滤器，目标是三种主要的开源 RISC-V 处理器：Rocket [2]，BOOM [24]，和 Xiang- Shan [13]。漏洞信息来源于 GitHub 问题和 CVE 条目。GitHub 报告会根据标签、补丁、标题和描述自动过滤，然后按照可用的测试资源进行分类：部分报告包含完整的可执行测试用例，部分包含需要补全的不完整代码片段，还有一些仅提供文本描述，需要进行人工分析和测试用例构建。同样，还会分析过去十年的 CVE 条目，以识别相关漏洞，并在可能的情况下重构测试用例。所有最终得到的测试用例，无论是可执行的、补全的还是手动创建的，都会汇总为真实世界种子语料库，构成 SimFuzz 的基础。

这些种子不仅能够实现高质量的模糊测试，还能突出 RISC-V 处理器中容易出错的指令和功能单元，为研究人员和供应商提供有价值的指导。

B. 相似度引导变异

种子语料库中的所有种子，都是处理器输入空间探索中的初始锚点。为了在扩展输入空间的同时保留种子的语义结构，SimFuzz 引入了相似度引导的块级变异。

1）块级变异：块定义为一段连续的指令序列，具有单一出口点，例如会改变控制流的跳转、分支或系统调用。块的出口是一条控制转移指令（CTI），其位置决定了块边界。形式上，对于给定的种子 S，将种子划分为 N 个块：$S = \{B_1, B_2, ..., B_N\}$。每个块 B_i 包含固定数量 M_i 的指令（$Inst$），使得对于每个 $i \in N$，$B_i = \{Inst_1, Inst_2, ..., Inst_{M_i-1}, CTI\}$。CTI 包含目标地址，修改它们会直接影响处理器的控制流。对 CTI 的修改可能导致测试程序崩溃，或偏离预期的执行路径，从而限制对指令流的深度探索。因此，块变异会刻意避免修改 CTI，尽可能保留种子的原始控制流。

进行块变异时，SimFuzz 会选取块中的若干条指令，分别进行指令级变异。变异后的指令会替换原指令，并重新插入块中，从而生成一个新的变异版本。变异后，SimFuzz 会根据指令级相似度计算变异块与原始块之间的相似度，具体定义见算法1。

算法1 块相似度计算函数

```
1: 函数 块相似度(B1, B2)
2:     sim ← 0.0;  size ← min(|B1|, |B2|)
3:     对于 i = 1 到 size 执行
4:         sim ← sim + S(B1[i], B2[i]) × (B1[i] ≠ B2[i])
5:     结束循环
6:     返回 sim/size
7: 结束函数
```

2) 相似度引导：核心思想是在保持控制流的同时对块进行变异。算法2概述了完整的相似度引导块变异过程。如果变异块与其原始块的相似度仍高于阈值 T，则接受该变异，从而扩大输入空间。如果经过多次尝试后，相似度仍未达到 T，则保留该块，不再进行更改。这种方法确保变异在引入多样性的同时仍具有语义意义。块相似度由其包含的指令的相似度决定。

指令级语义相似度通过对执行单元进行建模，反映由指令引发的处理器状态转换的相似程度。基于这一观察，我们设计了一种指令相似度度量，用于近似

算法2 相似度引导的变异算法

要求： 种子语料库 $\mathcal{S} = \{S_1, S_2, \ldots, S_K\}$, T

$\mathcal{S}^* \leftarrow \emptyset$
对于 每个种子 $S \in \mathcal{S}$ **执行**
  $\{B_1, B_2, \ldots, B_N\} \leftarrow$ 种子分段(S)
  对于 $i = 1$ 到 N **执行**
    对于 $j = 1$ 到 R **执行**
      $B_i^* \leftarrow$ **块变异**(B_i)
      $sim \leftarrow$ **块相似度**$(B_i,\ B_i^*)$
      如果 $sim < T$ **则**
        $B_i \leftarrow B_i^*,\ break$
      结束如果
    结束循环
  结束循环
  $S' \leftarrow$ 重新组装种子($\{B_1, B_2, \ldots, B_N\}$)
  $\mathcal{S}^* \leftarrow \mathcal{S}^* \cup \{S'\}$
结束循环
返回 $\mathcal{S}^*$

处理器状态空间相似度，方法是从多个粒度分析指令语义。给定两条 RISC-V 指令——指令 A (I_a) 和指令 B (I_b)，相似度函数将指令相似性分解为四个组成部分：指令类型、操作码、子语义和字段级相似度。

$$S(I_a, I_b) = w_1 S_{\text{tp}}(I_a, I_b) + w_2 S_{\text{op}}(I_a, I_b) + w_3 S_{\text{sm}}(I_a, I_b) + w_4 S_{\text{F}}(I_a, I_b) \quad (1)$$

a) 类型相似度。：指令类型相似度用于衡量两条指令的编码布局之间的相似程度。在 RISC-V 中，指令类型（例如 R型、I 型、S 型）定义了指令的句法结构，包括源寄存器、目标寄存器和立即数字段的数量及位置。类型相同的指令具有完全相同的编码布局，因此其类型相似度最高。对于类型不同的指令，则根据其编码布局的结构重叠程度确定相似度。

b) 操作码相似度：操作码相似度体现了指令之间粗粒度的功能相关性。操作码主要定义指令的高层操作类别，例如算术计算、内存访问、控制流或系统操作。操作码相同的指令具有最高的操作码相似度。属于同一功能类别的操作码具有部分相似度。操作码无关的指令则具有较低的相似度。

c) 子语义相似度：虽然操作码相似度能够捕捉粗粒度的功能行为，但无法区分指令之间细粒度的语义差异。为解决这一问题，我们引入子语义相似度，通过明确考虑处理器执行单元，以更细的粒度对指令语义进行建模。首先，根据指令的主要执行单元对指令进行分组。分派到同一执行单元的指令具有较高的子语义相似度，而面向不同执行单元的指令则具有较低的相似度。在同一执行单元内，子语义相似度还会根据运算特征进一步区分指令，例如算术运算类型、比较语义或数据移动模式。

d) 字段相似度：最后，字段级相似度衡量指令操作数和编码字段（例如寄存器和立即数）的相似度，用于捕捉可能影响数据依赖和微架构行为、但不会改变指令核心语义的差异。

字段相似度通过对应指令字段之间的归一化汉明距离计算：

$$S_{\text{F}}(I_a, I_b) = \sum_{i \in \mathcal{F}} w_i \left(1 - H(f_i^{(a)}, f_i^{(b)})\right) \quad (2)$$

在 (2) 中:

- $\mathcal{F}$ 表示所考虑的指令字段集合，例如 func3、func7、func2 和操作数。• $f_{(a)i}$和 $f_{(b)i}$分别表示指令 I_a和 I_b 中字段 i的值。
- $H(x,\ y)$ 表示两个位向量 x 和 y之间的归一化汉明距离，定义如下:

$$H(x, y) = \frac{\text{HammingDist}(x, y)}{\text{bitlength}(x)}$$

- w_i 是分配给字段 i的权重，满足:

$$\sum_{i \in \mathcal{F}} w_i = 1$$

总体相似度（1）是这些度量的加权组合，能够同时捕捉句法结构和语义意图。这种细粒度度量有助于有效地对指令进行聚类和变异，从而生成有效且多样化的测试用例。

C. 差分测试

变异后，代码块会重新组装成完整的测试用例，并在 RTL 处理器模拟器和 ISA 参考模拟器上执行。它们的行为差异表明可能存在错误。借助这一差分测试框架，Sim-Fuzz 能够揭示 Rocket、BOOM 和香山中的不一致，发现此前未知的错误。

IV. 评估

我们针对 RISC-V GC 指令集实现了 SimFuzz，该指令集受到所有通用处理器的支持。评估分为两个方面：覆盖率（第 IV-A 节）和漏洞发现（第 IV-B 节）。

A. 覆盖率

1) 设置：我们使用 xfuzz 测试框架 [21]，测量覆盖率，该框架支持多种覆盖率指标：行覆盖率、多路复用器覆盖率、翻转覆盖率和分支覆盖率。我们重点关注多路复用器覆盖率和翻转覆盖率，因为这些指标专为硬件设计，并已在其他研究中进行评估。

种子语料库包含我们收集的 42 个可触发历史真实漏洞的测试用例（历史），以及由随机 C 程序生成器 Csmith [3]，（BOOM 项目使用）生成的测试用例，还有 Cascade [30]，（一种最先进的处理器模糊测试测试用例生成框架）生成的测试用例。

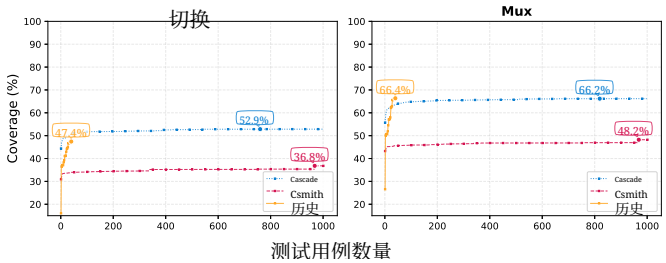
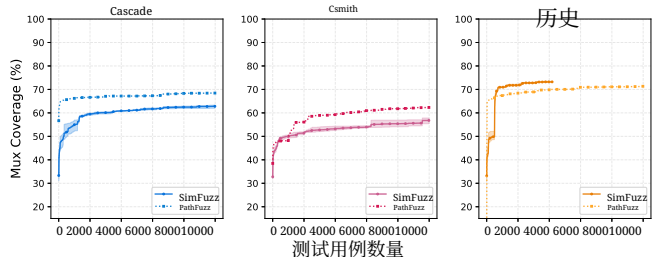
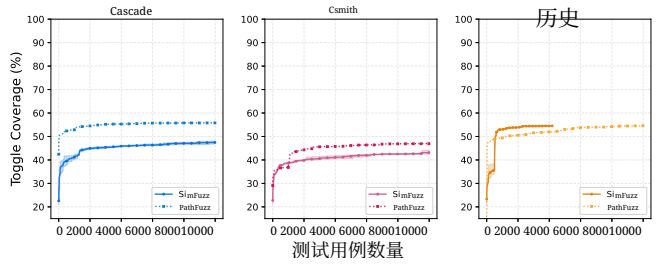


图 2：未进行模糊测试时，不同种子语料库的切换情况和多路复用器覆盖率。历史种子语料库的多路复用器覆盖率最高可达 66.4%。

2) 语料库评估：我们分别使用 Csmith 和 Cascade 生成 1,000 个种子，另加 42 个历史种子。图 2 展示了三个语料库的总覆盖率。历史种子的多路复用器覆盖率最高达到 66.4%，与 Cascade（66.2%）相当，且显著高于 Csmith（48.2%）。尽管规模较小，历史种子仍取得了出色的结果，这凸显了将曾经暴露真实漏洞的测试用例用于种子构建的价值。



(a) 多路复用器覆盖率



(b) 翻转覆盖率

图 3：SimFuzz 与 PathFuzz 的覆盖率。(a) 多路复用器：SimFuzz 的峰值为 73.22%（历史最高）。(b) 切换：SimFuzz 的峰值为 54.51%（历史最高）。

3）设计评估：我们设计了输入空间相似度矩阵，以指导变异过程，并评估直接探索处理器输入空间是否能够有效覆盖处理器状态。在实验中，我们根据初步测试将相似度阈值设为 $T = 0.5$ 。我们分别使用 Csmith 和 Cascade 生成了 1,000 个种子。对于 SimFuzz，我们将 Csmith 和 Cascade 语料库中的每个种子分别变异 10 次，共生成 10,000 个测试用例。历史种子各变异 100 次，共生成 4,200 个测试用例。作为对比，我们将 SimFuzz 与先进的处理器模糊测试器 PathFuzz [32]进行比较。PathFuzz 首先将种子的内存访问足迹展平为线性序列，然后

表 II：不同种子语料库下 SimFuzz 与 PathFuzz 的比较。

种子语料库	覆盖率	SimFuzz(%)	PathFuzz(%)	差异(%)
Cascade	Mux	62.80 ±0.69	68.44	-5.64
	切换	47.46 ±0.65	55.81	-8.35
Csmith	Mux	56.78 ±1.24	62.31	-5.53
	切换	43.12 ±0.81	46.95	-3.83
历史	Mux	73.22 ±0.27	71.33	+1.89
	切换	54.51 ±0.13	54.57	-0.06

*± 数值表示置信区间范围的一半。‡历史种子：SimFuzz（4,200 次测试），其他方法（10,000 次测试）。

应用覆盖率引导变异。我们针对每种种子类型运行 PathFuzz，每种执行 10,000 次。

图 3 展示了 SimFuzz 和覆盖率引导的模糊测试器 PathFuzz 在相同执行次数下实现的多路复用器覆盖率和翻转覆盖率。与 PathFuzz 不同，SimFuzz 不依赖覆盖率反馈，却在历史种子上实现了更高的覆盖率。详细的数值结果见表 II。这些结果表明，SimFuzz 实现了覆盖率
在大多数语料库上与 PathFuzz 相当，甚至在历史种子上略胜一筹，这表明相似度引导的探索无需依赖覆盖率反馈，也能有效到达多样化的处理器状态。

为进一步评估相似度引导的块级变异策略的有效性，我们将 SimFuzz 与其变体 SimFuzz* 进行了比较，后者禁用了相似度引导。如图 4 所示，SimFuzz 在历史种子上的表现始终优于 SimFuzz*。然而，两者的总体覆盖率差距仍然适中。PathFuzz 也呈现出类似的趋势：一旦覆盖率达到一定水平，便会逐渐趋于饱和，几乎没有进一步提升。

这种饱和现象可归因于两个主要因素。首先，某些处理器状态本身就难以触发。其次，鉴于有限的输入空间和变异策略，大多数易于到达的状态可能已经被探索，而剩余状态通常需要更复杂的语义扰动或错综复杂的指令间依赖才能触发。此外，这一观察表明，仅凭覆盖率可能无法充分刻画处理器状态探索的程度。对处理器模糊测试中覆盖率指标的有效性和实用性开展系统性研究，仍是未来工作的重要方向。

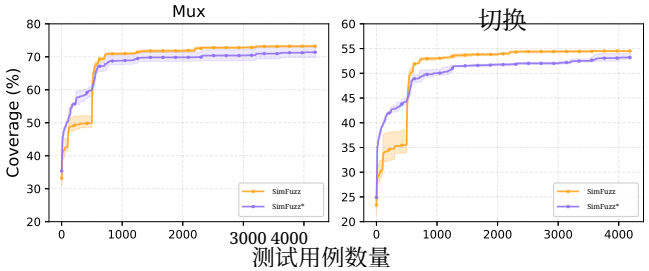


图 4：SimFuzz 与 SimFuzz* 在多路复用器和翻转覆盖率方面的对比。

表 III：SimFuzz 发现的漏洞列表。“From”列表示种子来源：S 表示来自构建的语料库的种子，B 表示 SimFuzz 发现的漏洞。

处理器	受影响单元	From	New	已确认	已修复。
BOOM	**B1:** 浮点单元	S1	✗	✗	✗
香山	**B7:** 解码	S5	✓	✓	✓
	B11: 后端	NA	✓	✓	✓
	B14: 解码	B14	✓	✓	✓
	B16: 加载	NA	✓	✓	✓
NEMU	**B2:** 浮点单元	S4	✗	✗	✓
	B3: 控制	S1	✓	✓	✓
	B4: 控制	S6	✓	✓	✓
	B5: 解码	NA	✗	✗	✓
	B6: 记忆	NA	✓	✓	✓
	B8: 解码	S5	✓	✓	✓
	B9: 控制	S7	✓	✓	✓
	B10: 控制	B10	✓	✓	✓
	B12: 解码	NA	✓	✓	✓
	B13: 解码	B11	✓	✓	✓
	B15: 解码	B14	✓	✓	✓
	B17: 记忆	S8	✓	✓	✓

B. 漏洞发现

SimFuzz 在真实处理器及相关参考模型上共发现了 17 个漏洞，其中包括 3 个已知漏洞和 14 个新发现的漏洞。所有新发现的漏洞均已向社区报告、得到确认并修复。

种子语料库由来自三个开源处理器的 37 份错误报告中的 42 个二进制测试用例组成。Rocket [2] 和 BOOM [24] 是经过十多年发展的早期开源 RISC-V 处理器，而香山 [23] 则是近五年内推出的较新处理器。这三个处理器通常被用作验证研究中的目标 [7], [8], [32]。Rocket 和 BOOM 依赖 Spike [28] 模拟器作为参考模型，而香山则使用 NEMU [22]。参考模型的正确性至关重要，因为模型中的错误可能掩盖真实的处理器漏洞。值得注意的是，一些漏洞同时出现在香山和 NEMU 中，这进一步凸显了参考模型准确性的重要性。表 III 总结了所有已发现的漏洞。由于所有漏洞都是在处理器制造之前发现的，因此目前不存在能够在正常用户权限下于现实场景中执行的漏洞利用程序。下一节将详细介绍若干 RTL 级漏洞。

错误 B7、B8。 这两个错误涉及香山处理器和 NEMU 模拟器中的 AES 加密指令 aes64ks1i [26] 。rnum 字段指定 S 盒 [26]的替换轮，宽度为 5 位（范围为 0-31），但规范仅允许取值 0-0xA。香山和 NEMU 均未强制执行这一限制，导致超出范围的值（例如 rnum = 0x15）仍可正常执行。值得注意的是，该错误此前已在 Rocket 中出现，如今又在香山中重现，凸显了检测 RISC-V 规范中某些边界情况的困难。SimFuzz 利用跨处理器错误数据，有效覆盖了此类边界情况。**错误 B11**。 该错误源于香山的后端控制逻辑。当 jalr 指令执行间接跳转到无效物理地址时，处理器未能正确触发异常处理。具体而言，如果计算出的目标地址超出 48 位物理地址范围，则应发生指令访问故障。然而，后端控制逻辑未将其视为有效的流水线重定向事件。有效信号生成仅考虑分支预测错误，导致无法及时刷新流水线并正确进行异常处理。

漏洞 B13、B14。根据 RISC-V 特权规范， [27], S 模式和 VS 模式下的间接寄存器选择器 siselect 和 vsiselect 应允许访问 0x0-0xFFF 范围的地址。在香山中，硬件将此范围限制为 0x0-0x1FF，而 NEMU 不作任何限制。这种差异导致高级中断架构等关键功能无法在处理器上使用，是一个严重的实现漏洞。

漏洞 B16。香山的加载单元存在漏洞，会导致异常发生时 mtval 寄存器中的值不正确。当加载指令（例如 fld）同时触发地址未对齐和页表转换缺失时，mtval 记录的不是原始故障内存访问的地址，而是一个无关地址。由于操作系统异常处理程序依赖 mtval 来处理页错误或权限违规，这一错误可能导致故障处理不正确，并可能无法为用户级程序提供服务。

C. 其他发现

SimFuzz 揭示，不同开源项目中的处理器存在由相同根本问题引发的漏洞。此外，与已知漏洞相关的缺陷甚至在较新版本中仍然存在。这一发现表明，在相同的指令集架构下，不同处理器可能存在相同或相似的漏洞。因此，在处理器设计和验证过程中，应特别关注已知漏洞，以防止类似错误再次发生。通过补充更多历史漏洞触发种子来扩充此类语料库，有望进一步提高覆盖率并缩短验证时间。我们计划从更多处理器中收集种子，扩大种子语料库，以增强处理器安全性并进一步缩短验证时间。

V. 结论

我们提出了 SimFuzz，一种新颖的处理器模糊测试方法。不同于传统的覆盖率引导策略，SimFuzz 利用来自真实世界处理器的历史漏洞触发测试用例，构建高质量的种子语料库。随后，它采用相似度度量探索输入空间，同时在变异过程中保留这些种子的控制流结构，从而实现对处理器状态更深层次的覆盖。实验结果表明，SimFuzz 的覆盖率高于覆盖率引导方法，发现了 14 个此前未知的漏洞，并获得了 7 个 CVE 标识符，证明了其有效性和实际意义。此外，我们的结果表明，在采用相同架构的处理器之间，可能会出现相同或相似的漏洞，这凸显了将真实世界漏洞样本纳入处理器验证的重要性。

VI. 致谢

作者谨向所有匿名审稿人致以感谢，感谢他们提出的宝贵意见和建议，也感谢 OpenXiangShan 社区的维护者在理解和修复漏洞方面给予的支持。此外，我们还感谢所有与处理器模糊测试相关的开源工作（包括但不限于 [7], [8], [18], [30],[32]）。本文得到国家重点研发计划（2024YFB4506200）和元图重大科研基础设施的支持。

参考文献

[1] Sadullah Canakci、Leila Delshadtehrani、Furkan Eris、Michael Bedford Taylor、Manuel Egele 和 Ajay Joshi。《DirectFuzz：使用定向灰盒模糊测试自动生成 RTL 设计的测试》。载于*2021*年第*58*届529534*ACM/IEEE*设计自动化会议（*DAC*），页码 – 。IEEE，2021年。[2] Chips Alliance。Rocket Chip生成器。https://github.com/chipsalliance/rocket-chip。[3] Csmith。Csmith，C程序的随机生成器。https://github.com/ csmith-project/csmith。[4] Enfang Cui、Tianzheng Li 和 Qian Wei。《RISC-V指令集架构扩展：综述》。*IEEE Access*，11：24696–24711，2023年。[5] Ghada Dessouky、David Gens、Patrick Haney、Garrett Persyn、Arun Kanuparthi、Hareesh Khattri、Jason M. Fung、Ahmad-Reza Sadeghi 和 Jeyavijayan Rajendran。《HardFails：关于可被软件利用的硬件漏洞的见解》。载于第*28*届*USENIX*安全研讨会（*USENIX Security 19*），页码213–230，美国加利福尼亚州圣克拉拉，2019年8月。USENIX协会。[6] Fabian Thomas等。《RISCVuzz：通过差分硬件模糊测试发现处理器架构漏洞》。https://ghostwriteattack.com/。[7] Jaewon Hur等。《DifuzzRTL：通过差分模糊测试查找CPU缺陷》。载于*2021*年*IEEE*安全与隐私研讨会（*SP*），页码1286–1303。IEEE，2021年。[8] Jinyan Xu等。《MorFuzz：通过运行时指令变形增强的可同步协同仿真对处理器进行模糊测试》。载于第*32*届*USENIX*安全研讨会（*USENIX Security 23*），页码1307–1324，美国加利福尼亚州阿纳海姆，2023年8月。USENIX协会。[9] Mohamadreza Rostami等。《超越随机输入：一种新颖的基于机器学习的硬件模糊测试方法》。载于*2024*年欧洲设计、自动化与测试会议暨展览会（*DATE*），页码1–6，2024年。[10] Rahul Kande等。《TheHuzz：使用黄金参考模型对处理器进行指令模糊测试，以发现可被软件利用的漏洞》。载于第*31*届*USENIX*安全研讨会（*USENIX Security 22*），页码3219–3236，美国马萨诸塞州波士顿，2022年8月。USENIX协会。[11] Sadullah Canakci等。《Processorfuzz：利用控制与状态寄存器引导处理器模糊测试》。载于*2023*年*IEEE*硬件安全与可信国际研讨会（*HOST*），页码1–12，2023年。[12] Timothy Trippel等。《像软件一样对硬件进行模糊测试》。载于第*31*届*USENIX*安全研讨会（*USENIX Security 22*），页码3237–3254，美国马萨诸塞州波士顿，2022年8月。USENIX协会。[13] Yinan Xu等。《利用敏捷方法开发高性能RISC-V处理器》。载于*2022*年第*55*届*IEEE/ACM*国际微体系结构研讨会（*MICRO*），页码1178–1199，2022年。[14] Andrea Fioraldi、Dominik Maier、Dongjia Zhang 和 Davide Balzarotti。《LibAFL：用于构建模块化且可复用的模糊测试器的框架》。载于第*29*届*ACM*计算机与通信安全会议（*CCS*）论文集，CCS '22。ACM，2022年11月。[15] Laura Inozemtseva 和 Reid Holmes。《覆盖率与测试套件有效性并无强相关性》。载于第*36*届国际软件工程会议论文集，页码435–445，2014年。[16] Shaoqian Jin、Yulin Li、Liwei Chen 和 Gang Shi。《SSFuzz：为RISC-V处理器生成语法与语义种子》。载于*2024*年大湖区超大规模集成电路研讨会论文集，GLSVLSI '24，页码421–426，美国纽约州纽约市，2024年。ACM。[17] George Klees、Andrew Ruef、Benji Cooper、Shiyi Wei 和 Michael Hicks。《评估模糊测试》。载于*2018*年*ACM SIGSAC*计算机与通信安全会议论文集，CCS '18，页码2123–2138，美国纽约州纽约市，2018年。ACM。

[18] Kevin Laeufer、Jack Koenig、Donggyu Kim、Jonathan Bachrach 和 Koushik Sen。《RFUZZ：面向覆盖率的FPGA RTL模糊测试》。载于*2018*年*IEEE/ACM*国际计算机辅助设计会议（*ICCAD*），页码1–8，2018年。[19] Wu Lichao、Rostami Mohamadreza、Li Huimin、Rajendran Jeyavijayan 和 Sadeghi Ahmad-Reza。《GenHuzz：一种高效的生成式硬件模糊测试器》。载于第*34*届*USENIX*安全研讨会（*USENIX Security 25*）。USENIX协会，2025年8月。[20] Danushka Liyanage、Marcel B¨ohme、Chakkrit Tantithamthavorn 和 Stephan Lipp。《可达覆盖率：估计模糊测试中的饱和度》。载于*2023*年*IEEE/ACM*第*45*届国际软件工程会议（*ICSE*），页码371–383，2023年。[21] OpenXiangShan。使用软件模糊测试器对通用硬件设计进行模糊测试。https://github.com/OpenXiangShan/xfuzz。[22] OpenXiangShan。NEMU。https://github.com/OpenXiangShan/NEMU。[23] OpenXiangShan。香山：开源高性能RISC-V处理器。https://xiangshan.cc/en/。[24] RISC-V BOOM。SonicBOOM：伯克利乱序机器。https: //github.com/riscv-boom/riscv-boom。[25] RISC-V International。RISC-V International在两年内批准40项规范，达成重要里程碑。https://riscv.org/ riscv-news/2024/04/，2024年。[26] RISC-V International。《RISC-V指令集手册第一卷：非特权ISA》。RISC-V International，2024年4月。版本20240411。[27] RISC-V International。《RISC-V指令集手册第二卷：特权ISA》。RISC-V International，2024年4月。版本20240411。[28] RISC-V Software。Spike：RISC-V指令集架构模拟器。https://github.com/ riscv-software-src/riscv-isa-sim。[29] Raghul Saravanan 和 Sai Manoj Pudukotai Dinakarrao。《探索硬件模糊测试中的覆盖率指标：全面分析》。载于*2024*年大湖区超大规模集成电路研讨会论文集，GLSVLSI '24，页码240–245，美国纽约州纽约市，2024年。ACM。[30] Flavien Solt、Katharina Ceesay-Seitz 和 Kaveh Razavi。《Cascade：通过复杂程序生成对CPU进行模糊测试》。载于第*33*届*USENIX*安全研讨会（*USENIX Security 24*），页码5341–5358，美国宾夕法尼亚州费城，2024年8月。USENIX协会。[31] T-Head。OpenC910核心。https://github.com/XUANTIE-RV/openc910。[32] Yinan Xu、Sa Wang、Dan Tang、Ninghui Sun 和 Yungang Bao。《Path-Fuzz：借助足迹记忆拓展CPU模糊测试的边界》。载于*2024*年第*61*届*ACM/IEEE*设计自动化会议（*DAC'24*），美国纽约州纽约市，2024年。ACM。