处理器模糊测试技术研究综述

摘要 处理器是计算机系统的核心组件，承担着执行指令、处理数据和调度任务的重要职责。为了应对日益剧增的计算需求和安全风险，处理器变得越来越复杂，同时也更容易出错。与可通过及时打补丁修复的软件错误不同，处理器硬件错误是永久性的，无法被轻易修复，会造成持久性危害。因此，在处理器制造前尽早发现设计和实现错误是非常重要的。然而以形式化验证和动态验证为代表的传统处理器验证方法分别存在状态空间爆炸和漏洞挖掘效率低的问题，难以适应复杂的处理器设计。为了弥补传统验证方法的不足，研究人员将软件模糊测试思想应用到处理器验证中，称为处理器模糊测试。模糊测试是当前最为流行的一种自动化软件漏洞挖掘技术，被广泛应用于各种软件测试，其核心思想是通过大量的随机输入来检测程序正确性。由于处理器和软件的固有差异，不能直接将软件模糊测试技术应用到处理器中。为此，研究人员专门设计了针对处理器特性的模糊测试技术。本文深入调研了现有处理器模糊测试研究工作，从挖掘不同漏洞类型的角度分类介绍了处理器模糊测试整体研究情况。然后，分析和总结了处理器模糊测试中使用的关键技术，包括输入构造、测试引导、种子调度和测试结果评估技术。接着，本文探讨了现有研究工作的不足之处，并介绍了针对这些不足的优化工作。最后，本文总结了处理器模糊测试研究现状并讨论了未来可能的研究方向。

关键词 模糊测试；处理器验证；处理器安全
中图法分类号 TP3-05

A Survey on Processor Fuzz Testing Technologies

Abstract Processors are the core components of computer systems, which are responsible for executing instructions, processing data, and scheduling tasks. In response to increasing computing demands and security risks, processors have evolved to become increasingly complex and error-prone. Unlike software bugs that can often be fixed with timely patches, hardware flaws in processors are permanent and hard to repair, leading to lasting damage. Therefore, early detection of design and implementation errors in processors before manufacturing is very important. Conventional processor verification methodologies, including formal verification and dynamic verification, encounter challenges such as state space explosion and inefficiency in bug detection, making them difficult to apply on complex processor designs. To overcome these limitations, researchers have applied software fuzz testing techniques to processor verification, which is known as processor fuzzing. Fuzzing is a widely used automated technique for uncovering software bugs, deployed across diverse software testing scenarios, aiming at validating program correctness through extensive random inputs. However, due to the inherent difference of processors and software, software fuzz testing methods cannot be directly applied to processors. Consequently, researchers have designed fuzz testing techniques specifically targeting processors. This paper conducts a comprehensive review of related research, providing an overview of the current state of processor fuzz testing studies, with a focus on different types of flaws. Then it analyses and summarizes the key techniques in processor fuzz testing, including input generation, test guidance, seed scheduling and test result evaluation. Subsequently, this paper inspects shortcomings of typical studies, introduces optimization research aimed at addressing these shortcomings. Finally, this paper summarizes the current research landscape of processor fuzz testing and discusses on potential directions for future research.

Key words fuzz testing; processor verification; processor security

1 引言

处理器是计算机系统的大脑，负责执行计算机程序中的指令，控制数据流动并协调各个硬件组件的工作。为了满足高性能和功能扩展的需求，处理器设计变得越来越复杂，通常具有数十亿个晶体管和多个内核。与此同时，处理器也变得越来越容易出错，甚至最新的商用处理器也存在硬件错误，严重影响计算机系统的可靠性和稳定性。硬件错误不仅会产生错误结果，还可能会导致隐私数据泄露或损坏，甚至导致计算机系统崩溃或死机，使计算机无法正常工作。例如，Intel Pentium FDIV[1]返回错误的浮点除法结果，Meltdown 漏洞[2]和 Spectre 漏洞[3]会造成敏感信息泄露，AMD Barcelona TLB 错误[4]和 Intel Pentium F00F 错误[1]能冻结处理器等。由于芯片制造后的硬件错误很难修复，且修复成本极高，因此在芯片制造前对处理器进行充分测试以发现潜在错误至关重要。

据统计，高达 70%的处理器开发时间被用于验证设计的正确性[5]。验证是处理器设计与开发生命周期中极为重要的一环，旨在确保处理器实现符合规格要求，发现潜在的硬件错误和漏洞，提高系统的稳定性和安全性。传统的处理器验证方法包括形式化验证和动态验证。形式化验证是一种基于数学模型和形式化语言的验证方法，其最大优势在于能够覆盖完整的状态空间。然而，这种方法在面对大型复杂设计时不可避免地存在状态空间爆炸的缺点。动态验证分为带约束的随机验证方法（Constrained Random Verification, CRV）和覆盖率引导的输入生成验证方法（Coverage Directed Test Generation, CDG）。CRV 虽然改进了完全随机验证方法，但它严重依赖于人工定义的输入约束，是一种部分自动化的验证方法。CDG 使用覆盖率指标引导测试方向，覆盖率的高低与测试充分度直接相关，高覆盖率表明测试更为充分。常见的覆盖率指标包括手动定义的功能覆盖率和代码覆盖率。但由于其输入生成方式仍较为随机，因此 CDG 仅能检测比较简单的处理器漏洞，漏洞挖掘能力低。

模糊测试（Fuzzing）是软件领域当前最有效的自动化测试技术之一[6-8]，它通过输入大量的随机、异常或非预期的数据（即“模糊输入”）来评估目标软件的鲁棒性和安全性，致力于发现应用程序或系统的缺陷和安全漏洞。鉴于模糊测试在软件测试领域的卓越效果，研究人员将其思想和技术引入到处理器模糊测试[9]，以弥补传统处理器验证方法的不足。近年来，处理器模糊测试在学术界受到了越来越多的关注。图 1 统计了从 2018 年开始至今（2024 年 3 月）在计算机领域的所有国际会议和期刊中发表的处理器模糊测试论文数量。可以看出，有关处理器模糊测试的研究论文数量呈逐年增长趋势，凸显了该领域的持续研究热度。和传统处理器验证方法相比，这些处理器模糊测试工作能有效提高漏洞挖掘能力和效率，目前已累计发现了开源处理器中 150 多个功能缺陷和安全漏洞。模糊测试技术同样也在商业处理器验证中发挥了越来越重要的作用，例如，研究人员通过模糊测试技术发现 AMD Zen 2 系列处理器上的 VZEROUPPER 指令可能会泄漏寄存器文件状态[10]。

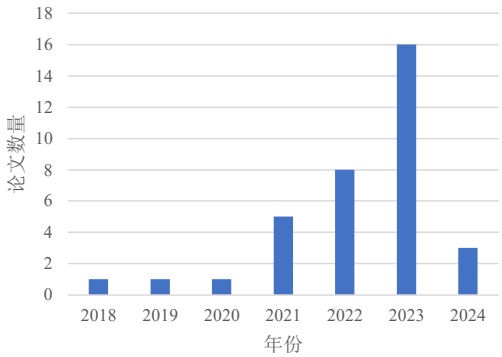


图 1 处理器模糊测试论文数量

Figure 1 Number of papers about processor fuzz testing

处理器模糊测试逐渐成为一个重要的研究方向，已取得可观的研究成果，但目前仍缺乏对这一领域研究现状的系统总结与分析。Fu 等人[11]虽然认为处理器模糊测试是一个有价值的研究方向，但仅调研了两篇处理器模糊测试论文。且由于其发表时间较早，无法反映近年来处理器模糊测试研究情况。现有模糊测试领域的综述论文[6-7, 12]大多集中在软件层，例如操作系统[13]、数据库[14]、网络协议[15]等各类上层软件。硬件层的模糊测试综述主要围绕嵌入式设备和固件[16]，未有工作针对硬件层的处理器模糊测试进行详细调研与分析。据我们所知，截止到投稿前，还未有公开的关于处理器模糊测试研究的综述文章。因此，本文是第一篇专注于处理器模糊测试研究的综述文章。

本文旨在对现有处理器模糊测试研究进行调研、总结与分析，特别关注模糊测试各个环节所使用的核心技术和优化方法，并给出未来可能的研究方向。

本文接下来的文章组织结构如下：第 2 章介绍软件模糊测试和传统处理器验证技术背景，并分析模糊测试技术应用到处理器验证中的难点与挑战。第 3 章根据不同的目标漏洞类型分类介绍典型的处理器模糊测试工作。第 4 章从输入构造、测试引导、

种子调度和测试结果评估四个方面，分析目前处理器模糊测试中的关键技术。第 5 章总结当前处理器模糊测试工作存在的不足，并介绍针对这些不足的优化工作。第 6 章简要概括处理器模糊测试研究现状，讨论和展望了未来可能的研究方向。

2 背景

2.1 软件模糊测试技术

以 AFL[17]为例，典型的软件模糊测试流程如图 2 所示。主要包括待测程序（Program Under Test，PUT）插桩，种子生成、种子选择、种子突变、种子执行和种子保留五个阶段，其中种子即输入。

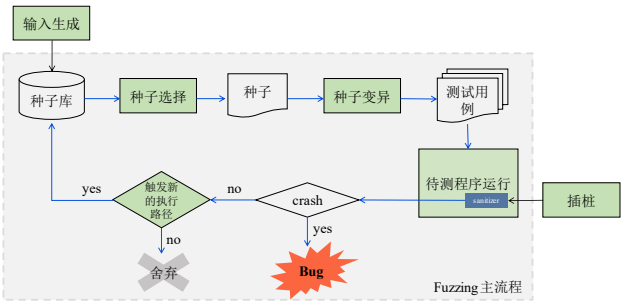


图 2 软件模糊测试流程

Figure 2 Software fuzz testing process

在进入模糊测试主流程之前，AFL 会生成一定数量的随机输入，作为初始种子加入到种子库。此外，还需要对 PUT 进行插桩，以监控和收集 PUT 运行时状态。在每一轮模糊测试中，模糊器（fuzzer）都会从种子库中选择一个种子进行测试。为了产生更加多样的种子，AFL 对已选择的种子进行突变（即修改）。根据突变的比特位数和位置的不同，突变方式可以分为确定性突变和非确定性突变，常见突变操作有比特翻转（bitflip）、删除（delete）、重写（overwrite）和交换（swap）等。突变的新种子作为测试用例被输入到待测程序中开始运行。在运行过程中 AFL 会监控待测程序的行为，一旦识别到程序崩溃，就认为当前输入能检测到待测程序中的漏洞，并分析导致程序崩溃的原因。否则，如果当前输入能触发新的执行路径，则说明该输入是有效的，将其保留在种子库中，以进行后续测试。如果当前输入既不能引起程序崩溃，也没有触发新的执行路径，则舍弃这个无效输入。不断选择、突变和运行种子，循环往复，就可以广泛探索待测程序中的执行路径，快速挖掘潜在漏洞。

目前模糊测试已成为软件测试领域的主流技术之一，被广泛应用于检测操作系统内核、物联网、软件应用、网络协议等中的缺陷与漏洞。相应的模糊测试工具有 SPIKE[18]（网络协议）、SQUIRREL[19]（数据库）、JANUS[20]（文件系统）、Charm[21]（移动设备驱动）、KAFL[22]（内核程序）、HyperCube[23]（虚拟机）。各种模糊测试工具的涌现使得模糊测试技术的测试对象更加多样化，影响也更加广泛。在实际应用中，研究者通过模糊测试技术发现了许多潜伏已久的漏洞，例如著名的 Heartbleed 漏洞[24]、c-ares $100K 漏洞[25]和 libpng 漏洞[26-27]等。

2.2 处理器设计与验证

处理器设计和开发是一个复杂而精细的过程，通常包括多个阶段和任务。首先在需求分析阶段，开发团队确定处理器的指令集架构、性能目标、功耗要求等内容。架构师根据需求分析设计最佳架构，包括寄存器文件、流水线结构、缓存体系结构等整体架构和指令执行单元、数据通路、控制逻辑等微架构。紧接着，开发人员使用硬件描述语言（Hardware Description Language，HDL）如 Verilog、Chisel 等来实现微架构模块。HDL 通常在寄存器传输级（Register Transfer Level，RTL）编写，用于描述复杂的硬件结构，如总线、有限状态机（Finite State Machine，FSM）、队列和加法器等。实现完成后，电子设计自动化（Electronic Design Automation，EDA）工具将 RTL 模型转化为门级（gate-level）电路并进行综合优化，门级电路使用布尔门、多路复用器和触发器等状态元件来实现硬件。然后，EDA 工具将门级设计综合为晶体管级，进行物理布局布线。最后将设计发送到代工厂进行生产和制造，得到芯片实体。

然而，手动编写 HDL 代码是非常耗时且容易出错的[1, 28-29]。在需要重新制造的半导体产品中，超过 40%的是由于在硅片制作后（post-silicon）发现功能性缺陷而需要进行修复[30]。不同于软件可以通过更新补丁的方法来修复漏洞，硬件一旦被“部署”（即生产），想要消除硬件漏洞的影响几乎是不切实际且成本极高的。因此在处理器开发生命周期的早期阶段尽早发现硬件漏洞至关重要，据统计，整个硬件设计周期的 70%时间被用于设计验证（Design Verification，DV）[5]。

对处理器进行充分详细的验证是确保其正确性和可靠性、降低硅后漏洞修复成本的关键手段。目前主要的处理器验证方法分为两大类：形式化验证（Formal Verification）和动态验证（Dynamic Verification）。

形式化验证是一种基于数学和形式化方法的验证技术，通过使用数学符号和形式化语言描述处理器的设计规范，利用数学推理和模型检查等技术来验证设计是否符合规范。这种验证方法不依赖于具体的测试用例，而是通过数学逻辑和严格的推理来

检查设计的正确性。流行的工业形式化验证工具有 Cadence JapserGold[31]和 Synopsys VC Formal[32]。 虽然形式化验证技术在处理器验证中已取得可观的效果[33-34]，但也存在一些不可忽视的缺点。首先，形式化验证依赖于容易出错的人力专业知识，要求验证人员事先了解硬件设计，并具有对设计规范进行数学抽象和建模的专业能力。其次，形式化验证存在状态空间爆炸的问题[35-37]。随着设计规模和复杂程度的增加，处理器设计存在大量的状态组合。为了探索这些状态空间，验证所需的计算资源和时间呈指数级增长。此外，形式化验证通常是离线进行的，不适用于频繁修改设计的验证场景。

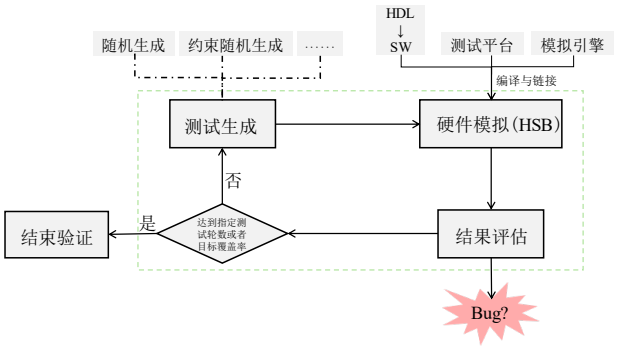


图 3 传统处理器动态验证步骤

Figure 3 Traditional processor dynamic verification steps

动态验证具有良好的灵活性和可扩展性，仍然是如今工业界主流的设计验证方法。动态验证通常包括三个主要步骤：测试生成、硬件模拟和结果评估，如图 3 所示。首先，在测试生成阶段，产生一系列输入作为待测设计（Design Under Test, DUT）的激励。根据输入的生成方式可将动态验证分成两大类：带约束的随机验证（Constraint Random Verification, CRV）和覆盖率指导的测试生成（Coverage-Directed test Generation, CDG）。CRV 是一种部分自动化的测试生成技术，需要手动定义输入格式和约束。为了避免因随机生成导致的验证效率低下问题，CRV 方法还需要一定的人工调优。CRV 原理已被广泛应用于商业 DV 框架 Accellera UVM[38]和开源 DV 框架 cocotb[39]。不同于 CRV 的无向性输入，CDG 旨在通过引导生成能覆盖目标范围的输入。其中覆盖率是引导测试方向和输入有效性的关键指标，常见的覆盖率包括手动定义的功能覆盖率[40]和 DUT 的 HDL 代码覆盖率[41-43]等。但由于可部署性方面的问题，CDG 在实践中并未得到广泛应用[9]。

然后，在硬件仿真期间，使用软件模拟 DUT 行为，接受并响应测试生成阶段产生的输入。具体来讲，要先将硬件 RTL 实现（HDL）转换为软件模型（C/C++），随后将软件模型和测试平台（test bench）进行编译，并将编译结果与仿真引擎（例如 Verilator[44]或者 VCS[45]）链接起来。它们最终组合成硬件模拟二进制文件（Hardware Simulation Binary，HSB），HSB 接收输入，并按照输入中的指令模拟硬件行为。

接着，在硬件模拟结束后，评估 DUT 在给定输入下的行为是否正确。如果 DUT 行为不符合规范和预期，验证人员会定位并修复错误。

动态验证过程不断重复以上三个步骤，直到已探索完所有感兴趣的状态空间或已达到预期覆盖率。当达到指定测试轮数或者目标覆盖率值时，说明已经进行了足够充分的测试，满足测试要求，此时便可选择结束验证过程了。

尽管在实际验证工作中动态验证方法仍然发挥着重要作用，但这类方法检测到的主要是简单的表面错误，较难深入挖掘复杂 DUT 中的深层设计空间。

2.3 处理器模糊测试的挑战

为了弥补以上 DV 方法的不足，研究人员开始将软件测试模糊技术引入到处理器验证[9]，称作覆盖率引导的突变模糊测试（coverage-guided mutational fuzz testing）或处理器模糊测试。处理器模糊测试是 CDG 方法的一种，其与传统 CDG 方法的本质区别在于两者输入生成方式的不同。具体而言，传统 CDG 方法的输入全部来自一定策略（如随机生成）产生的大量生成式输入。而处理器模糊测试方法的输入除了少量的生成式输入（即初始输入）外，主要来自于对有效输入进行变异而得到的突变式输入。其中，覆盖率是评估输入有效性的核心指标。由于处理器模糊测试通过引入突变来最大化利用有效输入，所以相较于传统 CDG 方法，它能更快地达到更高的覆盖率，并具有更强的验证能力和漏洞挖掘效率。

与传统的 DV 验证方法相比，处理器模糊测试具有工作量小、相对简单和可扩展性强等优势。然而，由于软件和硬件执行模型之间存在固有差异，设计针对处理器硬件行为的模糊器面临一些挑战，处理器模糊测试方法目前仍处于初步发展阶段。

挑战 1：无法直接将软件模糊器应用到处理器。首先，处理器和软件输入格式不同。许多软件模糊器采用文件或一组随机变量值作为输入，但处理器的输入大多是连续且没有固定长度的指令序列。其次，软件模糊测试中的漏洞检测方法无法直接应用到处理器。大多数软件模糊器如 AFL[17]使用崩溃（crash）来检测错误，但处理器并不会崩溃，此外，无法使用软件插桩工具对处理器进行插桩。许多软件模糊器

表 1 现有代表性处理器模糊测试研究工作

Table 1 Existing representative studies on processor fuzzing

工作	发表时间	测试方法	使用的仿真器	漏洞类型	检测漏洞数量	是否开源
RFUZZ[9]	2018	灰盒	Verilator	功能缺陷	-	√
DirectFuzz[47]	2021	灰盒	Verilator	功能缺陷	-	×
Logic Fuzzer[48]	2021	白盒	Dromajo	功能缺陷	13	×
DifuzzRTL[49]	2021	灰盒	Verilator	功能缺陷	16	√
Fuzzing HW Like SW[50]	2022	黑盒	Verilator	功能缺陷	5	√
TheHuzz[51]	2022	灰盒	VCS	功能缺陷	11	×
SpecDoctor[52]	2022	白盒	Verilator	安全漏洞	2	√
BugsBunny[53]	2022	灰盒	Verilator	功能缺陷	-	×
Cross-Level Fuzzing[54]	2022	灰盒	Verilator	功能缺陷	7	×
Revizor[55]	2022	黑盒	-	安全漏洞	-	√
SurgeFuzz[56]	2023	白盒	Verilator	功能缺陷	6	√
PSOFuzz[57]	2023	灰盒	VCS	功能缺陷	-	×
GenFuzz[58]	2023	灰盒	Verilator	功能缺陷	-	√
ProcessorFuzz[59]	2023	灰盒	Verilator	功能缺陷	9	√
MorFuzz[60]	2023	白盒	VCS	功能缺陷	19	√
HyPFuzz[61]	2023	灰盒	VCS	功能缺陷	14	×
Xfuzz[62]	2023	白盒	Verilator	功能缺陷	13	×
MMFuzz[63]	2023	灰盒	PyRTL	功能缺陷	-	×
SIGFuzz[64]	2023	白盒	Verilator	安全漏洞	5	√
MABFuzz[65]	2023	灰盒	VCS	功能缺陷	-	×
VGF[66]	2023	白盒	VCS	功能缺陷	-	×
Cascade[67]	2024	灰盒	Verilator	功能缺陷	38	√
INSTILLER[68]	2024	灰盒	Verilator	功能缺陷	12	×
WhisperFuzz[69]	2024	白盒	VCS	安全漏洞	12	×

依赖自定义编译器如 afl-gcc[46]对软件程序在编译过程中完成插桩，并通过插桩代码监控程序运行时状况。但这些编译器无法编译由 HDL 编写的硬件设计，更不能进行自动插桩。最后，处理器模拟速度慢，不适配快速的软件模糊器。通常情况下，对于特定功能，执行其等效软件相比模拟其硬件模型更为迅速。由于处理器设计中存在复杂的相互依赖关系，实现处理器模拟和仿真的并行化非常具有挑战性[70]。

挑战 2：设计处理器模糊器较为复杂。首先，要确定所测试硬件类型的抽象层次和输入格式。硬件的输入可以在各种硬件抽象级别生成，如架构级、RTL 级、门级和晶体管级。每个级别也有各自的输入格式，如事务数据包、数字信号到模拟信号。因此，确定合适的模糊测试抽象级别和输入表示是设计处理器模糊器的首要任务。其次，还要设计硬件友好的覆盖率指标和覆盖率反馈机制。覆盖率指标应该是对硬件行为的抽象，能指示目标状态空间的探索情况。覆盖率插桩代码应该轻量、简单和自动化，能适配多种 DUT，支持在 DUT 运行过程中收集和反馈运行时信息的同时不影响这些 DUT 的正确性和执行效率。此外，设计合适的种子选择和突变策略是模糊器有效性的关键。使用优质种子进行突变能尽快发现 DUT 漏洞，有效的突变算法能快速引导模糊器探索新的状态空间。同时，需要在可靠的硬件设计上评估模糊器的性能。由于常见的 Intel x86 处理器等商业硬件设计并不开源，因此找到认可度高的复杂开源处理器来检测模糊器能力十分重要。最后，任何处理器模糊器都应该与现有硬件设计和验证流程兼容，以更好地融入并集成到工业硬件验证流程中，发挥实用价值。

3 处理器模糊测试分类与代表性工作

自2018年RFUZZ[9]将覆盖率引导的突变模糊测试技术应用到RTL设计验证开始，近几年来涌现出了许多处理器相关的模糊测试研究和工具。表1列出了目前已发表的绝大部分处理器模糊测试研究工作。这些模糊测试研究工作已被实验证实能有效发现处理器功能错误和安全漏洞，并提高处理器的质量、鲁棒性和安全性。

根据测试方法可以将这些研究工作分为白盒测试、灰盒测试和黑盒测试，如表1所示。从表中可以看出约 58%的处理器模糊测试工作属于灰盒测试，灰盒测试需要对处理器内部结构有一定了解，但不需要完全了解，这种方法更适用于大型复杂的处理器设计。约33%的研究工作属于白盒测试，这类工作主要集中在对较小处理器核的测试，因为完全了解和熟悉这类小型设计的内部实现不存在过高的难度。少数研究工作采用黑盒测试方法，这类工作不需要知道处理器的内部实现细节。总的来说，不同的测试方法针对不同复杂度和可访问性的处理器提供了灵活的测试策略，验证人员可以根据具体的处理器特点选择合适的测试方法，以实现高效的模糊测试。

根据要检测的漏洞类型，这些研究工作又可以分为针对处理器功能缺陷（bug）的模糊测试研究和针对处理器安全漏洞（vulnerability）的模糊测试研究。针对功能缺陷的处理器模糊测试往往不针对某一类具体的功能错误，而是随机捕获所有可能出现的设计或实现错误。针对安全漏洞的处理器模糊测试对微架构信息泄漏问题进行检测，例如著名的熔断[2]（meltdown）和幽灵[3]（spectre）漏洞。这些模糊测试研究工作已被实验证实能有效发现处理器功能错误和安全漏洞。

为评估模糊器在不同场景下发现漏洞的效率和能力，研究人员通常选择合适的开源DUT以模拟模糊器在真实环境中的使用情况。从目前的研究情况来看，处理器模糊测试的DUT已涵盖商业IP（Intellectual Property），如TileLink Peripheral IP[71]、DSP（Digital Signal Processing）系统中的FFT（Fast Fourier Transform）块和处理器核。在这些DUT中，开源处理器核占据主要比例，如RISC-V[72]和OpenRISC[73]处理器核。RISC-V是一个开放免费的指令集架构（Instruction Set Architecture，ISA），支持32位、64位和128位寄存器宽度以及各种可选扩展，例如支持原子指令的A扩展、支持压缩指令的C扩展和支持向量操作的V扩展等[74]。RISC-V完全开放使用许可，允许任何人免费使用、设计和实现RISC-V处理器核。目前，RISC-V已被广泛应用于各种领域，成为一个受欢迎的指令集架构。OpenRISC是由OpenCores组织开发的精简指令集架构（Reduced Instruction Set Computing，RISC），同样秉持开放和免费的原则，旨在创建可用于教育和嵌入式系统的开源RISC架构处理器。它支持 32 位和 64 位寄存器宽度，具有可选的浮点算术和向量处理支持。常用DUT以及这些DUT的配置和使用情况如表 2所示，从表中可以观察到大多数的处理器模糊测试工作选择将Rocket[75]、BOOM[76]和CVA6[77]作为DUT，这些处理器核的受欢迎程度与代码质量、社区活跃程度和影响力密切相关。

3.1 针对功能缺陷的处理器模糊测试研究

功能缺陷指在处理器设计或制造过程中出现的各种问题和错误，导致处理器无法按照预期方式或结果运行，影响处理器的正确性、稳定性甚至是性能。常见的功能缺陷类型包括执行结果错误、时序错误、访问控制错误和中断/异常处理错误等。执行结果错误指处理器在执行特定指令时出现错误，导致程序执行异常或结果不正确；时序错误指处理器中的时钟信号、数据传输或控制信号出现问题，导致指令执行顺序混乱或数据传输错误；访问控制错误指处理器未正确识别越界或越权访问，导致读写数据不正确或未正确报告异常；中断/异常处理错误指处理器未能正确处理异常优先级或未正确设置异常程序处理入口，导致在异常处理程序中陷入死循环等情况。

针对功能缺陷的处理器模糊测试研究可根据测试的模块范围再进一步细分为定向模糊测试和非定向模糊测试。定向模糊测试针对处理器中的某些特定区域、目标模块或功能进行检测，而非定向模糊测试则检测整个处理器或RTL设计中的功能缺陷。

3.1.1 非定向模糊测试

非定向模糊测试通过广泛探索硬件行为和状态空间来挖掘处理器各个模块和组件中的功能缺陷。目前，有两种主要的非定向模糊测试技术解决思路：一是模糊输入激励（input-stimuli fuzzing），二是模糊电路逻辑（logic fuzzing）。这两种方法都致力于深入挖掘处理器硬件的行为和状态空间，以提高系统的稳定性和安全性。

模糊输入激励方法：绝大多数的处理器验证工作都是通过向HSB输入激励（stimuli）来进行的，HSB 接收这些输入激励并执行相应的操作。在处理器模糊测试中，输入激励通常可以分为两种格式：比特序列和指令序列，如图 4所示。图4 (a)中的输入格式为比特序列，由0/1字符串组成，每32或64比

表 2 常见 DUT 及使用情况

Table 2 Common DUTs and their applications in fuzzer

指令集类别	DUT	发布时间	代码量	支持的指令集	架构	相关模糊测试工作	
RISC-V	Sodor[78]	2013	47.9k	RV32I	32bit in-order 2/3/5-stage pipeline	RFUZZ DirectFuzz	
	Rocket[75]	2016	69.3k	RV64G	64-bit in-order 5-stage pipeline	RFUZZ MorFuzz PSOFuzz SIGFuzz HyPFuzz INSTILLER	DifuzzRTL MABFuzz PSOFuzz ProcessorFuzz Cascade WhisperFuzz
	BOOM[76]	2016	22.9k	RV64GC	64-bit out-of-order 10-stage pipeline	DifuzzRTL MorFuzz SurgeFuzz SIGFuzz ProcessorFuzz Cascade WhisperFuzz	Logic Fuzzer MABFuzz PSOFuzz SpecDoctor HyPFuzz INSTILLER
	CVA6[77]	2018	269k	RV64GC	64-bit in-order 6-stage pipeline	TheHuzz MorFuzz PSOFuzz Cascade TaintFuzzer	Logic Fuzzer MABFuzz HyPFuzz WhisperFuzz SoCFuzzer
	BlackParrot[79]	2020	279.1k	RV64IMAFD	64-bit in-order 2-stage pipeline	Logic Fuzzer ProcessorFuzz	
	RSD[80]	2019	223.0k	RV32IMF	32-bit out-of-order 6-stage pipeline	SurgeFuzz	
	XiangShan[81]	2022	54.3k	RV64GCBK	64-bit out-of-order 11-stage pipeline	XFUZZ	
	NutShell[82]	2019	23.2k	RV64IMAC	64bit in-order 9-stage pipeline	XFUZZ SpecDoctor	
OpenRISC	mork1x[73]	2013	29.9k	OpenRISC 1000	32-bit in-order 6-stage pipeline	DifuzzRTL TheHuzz HyPFuzz INSTILLER	
	or1200[73]	2000	36.3k	OpenRISC 1000	32-bit in-order 5-stage pipeline	TheHuzz HyPFuzz INSTILLER	

特为可构成一条指令。图 4 (b)中的输入为指令序列，通常指令序列中除了指令外，还包括数据区，即图中的 d_5_0。

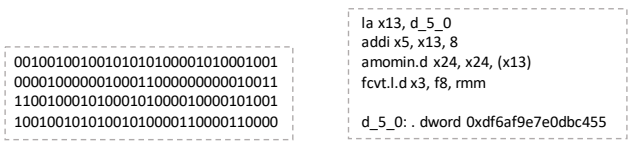


图 4 处理器模糊测试输入格式

Figure 4 Input formats for processor fuzzing

模糊输入激励是对这些不同格式的输入进行突变来生成更多样化的新输入，以探索多种可能的输入组合和场景。在突变比特序列格式的输入时，该方法并不确保突变后比特序列的语法正确性和语义合理性，因为 DUT 是否能正确响应和处理非法指令也是测试内容之一，因此可以直接使用 AFL 中的突变算法。例如，RFUZZ[9]把 RTL 电路中所有顶级模块的输入引脚值映射为比特序列，将其作为 DUT 一个特定测试周期中的输入值，然后便可直接应用 AFL 的确定性和不确定性突变策略来产生新种子，以提升对输入空间的探索和覆盖能力。对于指令序列格式的输入，突变不仅要应用于代码区的指令序列本身，还要应用于数据区的数据。常见的突变方式有增、删、改、指令序列重组等。例如，SIGFuzz[64]、DifuzzRTL[49]、ProcessorFuzz[59]、MorFuzz[60]等，通过突变指令的操作码或其他字段来获得新指令，以构造丰富的指令序列。

模糊电路逻辑方法：对于隐蔽的复杂功能缺陷，很难通过随机指令流/数据流和定向测试发现。因为这些测试方法大都专注于构造丰富的用于 DUT 的外部输入，强调"由外向内"的思路，即通过外部输入来

控制 DUT 的内部行为。然而，DUT 的内部行为对输入来说是透明的，通过随机输入无法精确地探索和覆盖 DUT 的深层行为，尤其是微体系结构行为和状态。因此，有学者提出采用"由内而外"的方法，通过模糊 RTL 电路逻辑来直接控制 DUT 的内部行为。

这类方法的核心思想是将不破坏 DUT 正常功能的电路逻辑插入到 DUT 源码中，以检测 DUT 是否仍能产生正确执行结果。理论上，插入的修改逻辑可以在指令运行时搅动执行路径，使 DUT 脱离常规流程，但对指令的最终执行结果不产生影响。若 DUT 受到这些电路逻辑的影响产生了错误的执行结果，则说明 DUT 的实现存在问题，需要被修复。所插入的不破坏 DUT 正常功能的电路逻辑通常和微体系结构层的处理器状态相关，如 Cache/Buffer 等各种缓存结构，Queue/Table 等各种存储结构。目前这类工作的典型代表是 Logic Fuzzer[48]，Logic Fuzzer 引入了一种称为表突变（Table Mutation）的突变方式，通过在指令执行过程中突变处理器中的各种表结构，如 BTB（Branch Target Buffer）、BHT（Branch History Table）、TLB（Translation Lookaside Buffer）等，来探索输入在某些微体系结构状态下是否能被准确执行。实验结果证实了这类方法可以探索更广泛更隐蔽的处理器微体系结构状态，更快地发现先前未知的功能缺陷。

3.1.2 定向模糊测试

定向模糊测试不专注于整个处理器，而是有明确的目标性，测试目标通常是处理器中的特定模块、组件或功能，旨在通过有针对性地生成测试用例，以发现特定类型的漏洞或验证特定的代码路径。定向模糊测试的输入一般来自随机指令序列生成器，为了更精准快速地接近测试目标，测试人员还可手动定义输入生成约束或模版。为了识别和监测测试目标的覆盖情况，处理器定向模糊测试通常结合静态分析和动态分析两种技术。

静态分析阶段将对 DUT 的源码进行语法和语义分析，以获取电路结构、控制流、数据流等方面的信息，并利用这些静态信息实现插桩，以便在动态分析阶段收集运行时信息。例如，DirectFuzz[47]对 RTL 设计的中间表示 FIRRTL[83]进行静态分析，实现三个关键功能：1）确定目标模块实例中的覆盖点；2）生成实例有向连通图，并计算每个模块实例和目标模块实例的相对距离；3）对 FIRRTL 进行简单插桩，以记录每个测试输入的执行路径。

动态分析阶段会使用静态分析所生成的插桩代码收集并计算输入在 DUT 中的运行时信息，如到目标模块实例（DirectFuzz[47]）或目标区域（BugsBunny[53]）的距离。在以距离为引导信息的处理器定向模糊测试中，为了获取到测试目标的距离，插桩代码不仅要记录输入的执行路径，还需要抽象出电路结构树形或图形表示，以便计算距离。如果一个输入所覆盖的区域到测试目标的距离更短或者能覆盖新的测试目标，那么该输入会被视为有效输入并保存到种子库中用于后续突变。否则，该输入被舍弃。

相比于非定向处理器模糊测试，定向模糊测试的优势在于它将测试资源聚焦到预设的测试目标上，从而更快速地发现特定漏洞或验证特定的执行路径。进而加快漏洞挖掘速度和效率。虽然定向模糊测试以灰盒测试为主，但仍要求验证人员熟悉处理器设计并对 DUT 进行一定的深入分析。因此，定向模糊测试虽然能够带来可观的测试效果，但也具有一定的实施门槛。

3.2 针对安全漏洞的处理器模糊测试研究

处理器安全漏洞指在处理器硬件或微体系结构中存在的，可能导致计算机面临一定的安全风险的缺陷。这类安全漏洞可能允许攻击者通过巧妙构造的输入或恶意代码，绕过正常的安全防护机制，访问敏感数据、执行未经授权的操作，或干扰系统的正常功能。典型的处理器安全漏洞包括缓冲区溢出、侧信道攻击等。这些漏洞可能在硬件设计、源码实现或架构规范层面存在，影响范围广泛。解决处理器安全漏洞通常需要硬件制造商、操作系统开发者和软件开发者的协同努力，通过更新补丁和安全策略来提升系统的整体安全性。

近些年来，由于 Spectre 和 Meltdown 攻击的出现，处理器中的侧信道漏洞越来越引起人们的关注。侧信道漏洞是一类广泛的安全问题，涉及利用时间、功耗、电磁辐射等侧信道来获取敏感信息。其中，瞬态执行攻击是一种典型的侧信道攻击的典型实例。瞬态执行指瞬态指令的执行，瞬态指令会改变微体系结构状态，但不破坏处理器的执行结果的正确性。出于性能优化的目的，现代处理器中广泛存在着瞬态执行机制，如乱序执行和推测执行等。如果攻击者能够通过瞬态指令获取微体系结构状态，那么处理器内部的信息就会被泄露，可能导致严重的安全问题，这攻击被称为瞬态执行攻击，如著名的 Spectre 和 Meltdown 攻击。因此在处理器设计和开发早期阶段检测瞬态执行漏洞是十分重要的，对于保障处理器的安全性非常关键。

目前，检测瞬态执行侧信道漏洞主要还是通过

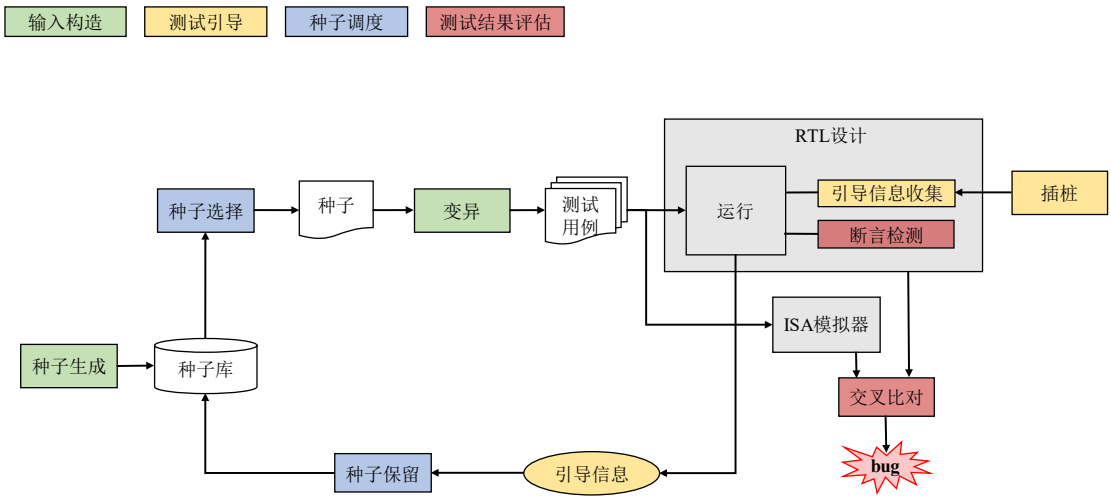


图 5 处理器模糊测试架构

Figure 5 Processor fuzzing architecture

手动检查和实验发现的，这种方法耗时耗力且效率低下。借助模糊测试技术，针对瞬态执行漏洞特点设计相应的自动化模糊检测机制，可以极大提高处理器安全漏洞检测速度和效率。

针对瞬态执行类侧信道漏洞的处理器模糊测试研究的关键之一是创造和识别瞬态执行场景。其中创造瞬态执行场景依赖于特定的随机指令序列，例如必须包含条件跳转指令（如 beq、blt）和访存指令（如 ld、sd），且分支预测错误和正确路径下均包含非空操作。为了识别瞬态执行场景，需要依赖处理器中的微体系结构信息或者运行时的动态信息。例如 SpecDoctor[52]利用 RoB（Reorder Buffer）检测瞬态执行，因为 RoB 总是参与发生瞬时执行时的回滚，且监视 RoB 所需资源较少。SIGFuzz[64] 和 WhisperFuzz[69]分别通过评估指令和输入的执行时间来识别瞬态执行，因为处理器会执行并回滚错误分支预测路径上的指令，从而使整体输入或正确分支路径下某些指令的执行时间增大。

另一方面，基于识别到的瞬态执行场景展开新的瞬态执行攻击也非常重要。首先，模糊器保留能够触发新的瞬态执行场景的输入，定位输入中能利用瞬态执行访问和传输秘密信息的具体指令。模糊器接下来对输入进行突变，在可能的位置插入可以利用侧信道接收秘密的指令。然后模糊器再次执行突变后的输入，通过测量获取秘密所需的周期数判断是否成功实施了一个新的瞬态执行攻击。借助模糊测试方法，可在成熟处理器核中有效发现新的 Meltdown 和 Spectre 攻击的变种，如 Boombard、Birgus 等[52]。

4 处理器模糊测试中的关键技术

现有处理器模糊测试的研究工作，基本都遵循如图 5 所示的架构。首先，生成一些初始种子加入到种子库中。然后，从种子库中挑选种子进行变异（或称为突变）得到新种子，这些新种子被作为测试用例输入到 DUT（RTL 级设计）中。测试用例的运行时信息在 DUT 的运行过程中被收集，并作为测试方向的引导信息，决定了该种子是否被保留。运行时信息收集功能通过对 DUT 插桩代码实现，插桩在测试流程启动前执行，且只需要执行一次。为了判断每个测试用例在 DUT 中的执行结果是否符合预期，需要在 ISA 模拟器中也运行相同的测试用例，通过交叉比对同一个测试用例在 DUT 和 ISA 模拟器中的执行结果来判断是否检测到了漏洞或缺陷。从图 5 中可以看出，处理器模糊测试主要有 4 个关键技术，分别是输入构造技术、测试引导技术、种子调度技术和测试结果评估技术。由于测试目标和实现方式的不同，各个研究工作在这 4 项关键技术上有多种不同的解决方案。

4.1 输入构造技术

输入构造技术是处理器模糊测试的关键，旨在生成丰富有效的测试用例，以覆盖更全面的处理器行为与状态，挖掘处理器中的隐藏漏洞。为了适应不同的测试平台，需要根据各种测试平台的架构特点和输入接收方式设计相应的输入格式。例如，基于 FPGA 硬件仿真平台的模糊测试工具以二进制序列作为输入格式，而基于 Verilator 等软件模拟平台的模糊测试工具则以指令序列作为输入格式。虽然不同测试平台接收的输入格式不尽相同，但输入构造方法都可以分为两大类：基于生成的和基于突变的。

现有代表性处理器模糊测试研究工作所使用的输入构造技术如表 3 所示。

基于生成的输入构造方法：该方法使用开源指令集生成器或自定义的指令序列格式和语法作为约束，随机产生大量测试用例，并将这些测试用例加入到初始种子库。

以 RISC-V 指令集为例，为了支持 RISC-V 产品的验证，出现了许多开源指令集生成器和测试套件可供使用。例如，加州大学伯克利分校开发的 riscv-tests[84]和 riscv-torture[85]、RISC-V 国际协会提供的 riscv-compliance[86]以及 Google 推出的基于 SV/UVM 的随机指令生成器 riscv-dv[87]等。但这些开源工具通常只检查处理器的基本功能，存在测试样例过少或输入过于随机的问题。

为了使输入更加可控且更适应测试平台，许多处理器模糊测试工作开发了自定义的输入格式和生成约束。例如，RFUZZ[9]将电路顶级模块输入引脚的值映射为比特序列，该比特序列被用以表示特定测试周期中的输入值。为了允许模糊测试引擎在每个测试周期中应用不同的值，RFUZZ 将单周期测试输入连接起来，形成了多周期输入。通过随机赋值这些比特序列，即可得到丰富的输入激励。为了寻找软件可利用的处理器漏洞并支持漏洞调试，出现了许多以指令序列作为输入格式的研究工作。这些研究工作提出了不同的指令序列生成器，其中以 DifuzzRTL[49]和 MorFuzz[60]最为代表性。DifuzzRTL 设计了支持中断列表的新型输入格式 SimInput。SimInput 由初始化指令、随机指令、预定义数据区和中断事件组成。指令生成时先随机选择操作码再使用数据区的内容实例化源寄存器、目的寄存器、立即数和访存地址等字段。MorFuzz 改进 DifuzzRTL 中复杂的输入语法和约束，设计了更加轻量的输入模版生成器（Stimulus Template Generator）。MorFuzz 将大量的随机指令分成了两类：magic 指令和 template 指令。magic 指令专门负责从数据区加载数据，如 ld x1, RDM_DATA_ADDR(x0)，template 指令指已确定操作码但未实例化其他字段的指令，如 lh x??, ??(x??)，这些未实例化的指令域将在 DUT 执行过程中被设置并突变。

基于突变的输入构造方法：通过从种子库中选出种子进行突变来产生新输入。突变指对种子进行随机或有目的的修改，以生成新的测试输入，这一思想来自于遗传算法[17]。使用合适的突变粒度、突变时机、突变范围、突变内容和突变算法，可以探索更多非常规的处理器行为，极大地增强了输入的多样性。

从表 3 中可以看出，根据输入格式的不同（如图 4 所示）可以把将突变方法划分为比特粒度的突变和指令粒度的突变。根据突变时机又可以将突变方法分为输入运行后突变和运行时突变。大多数处理器模糊测试工作都是在 DUT 外部对已执行过的种子进行突变的，也有个别模糊测试工作做通过对 DUT 插桩来实现在种子运行过程中完成突变[48, 60]。例如，MorFuzz[60]在 DUT 的取指和译码阶段之间插桩突变引擎，以实现运行时突变指令。

然而，不同的模糊测试工作的突变范围、突变内容和突变策略却不尽相同。

表 3 代表性处理器模糊测试研究采用的输入构造技术

Table 3 Input generation techniques used in representative processor fuzzing studies

代表性研究工作	输入格式	生成式输入	突变式输入				
			突变粒度	突变时机	突变范围	突变内容	突变策略
RFUZZ[9]	比特序列	随机生成	比特	运行后	每个比特	比特值	AFL 中的确定性突变和不确定性突变策略
DirectFuzz[47]	比特序列	随机生成	比特	运行后	每个比特	比特值	距离目标模块越远的输入被突变的次数越多
DifuzzRTL[49]	指令序列	自定义指令序列生成器	指令	运行后	每条指令	整条指令（操作码和其他字段）	随机选择增、删、改，还支持指令序列间合并操作
SpecDoctor[52]	指令序列	自定义指令序列生成器	指令	运行后	瞬态执行部分的指令	整条指令（操作码和其他字段）	随机选择增、删、改策略
TheHuzz[51]	指令序列	自定义指令序列生成器	指令	运行后	每条指令	整条指令（操作码和其他字段）、其他字段（操作码不变）	自定义权重来选择每条指令的突变操作
MorFuzz[60]	指令序列	自定义指令序列生成器	指令	运行时	每条指令	其他字段（操作码不变）	随机字段级突变和语义级突变

以比特序列作为输入的研究工作往往都采用AFL 的确定性突变和非确定性突变算法，通过对比特序列进行精确或随机的比特翻转、随机插入和删除和数值扰动等操作来实现突变，如 RFUZZ[9]和DirectFuzz[47]。两者突变方面的不同点在于，RFUZZ对所有输入都“一视同仁”，均应用相同次数的突变算法，但 DirectFuzz 意识到不同种子的价值是不一样的，到目标模块的距离越远的输入，被突变的次数应该越多。

大多数以指令序列作为输入的研究工作将输入中所有指令作为突变范围，但个别针对特定目的的研究仅突变部分指令，如测试处理器中瞬态执行漏洞的 SpecDoctor[52]仅对瞬态执行部分的指令进行突变。指令突变操作往往有增、删、改，但具体而言指令突变的内容和突变策略方面，不同的研究工作各具特色，如图 6 所示。DifuzzRTL[49]随机选择每条指令是否突变以及所要执行的具体突变操作，然后对整条指令进行突变，包括指令操作码和其他字段。TheHuzz[51]使用自定义权重来选择每条指令所采用的突变操作，除了支持对整条指令进行突变，还可以只对指令的其他字段进行突变（操作码保持不变），这样可以覆盖更多彼此接近的不同数据路径。MorFuzz[60]不突变指令的操作码，而是先通过改变指令的功能字段进行字段级突变，再改变目的寄存器、源寄存器和立即数字段实现语义级突变，以达到探索相近路径的效果。

4.2 测试引导技术

测试引导技术是处理器模糊测试的核心，旨在通过定义合适的引导信息来评估输入的有效性，以引导测试在深度和广度上取得更好的效果。首先，确定适宜的引导信息是测试引导技术的主要挑战。引导信息指明了输入构造的方向，不同的引导信息可能会导致不同的测试结果。常见的引导信息有覆盖率指标（非定向模糊测试）和距离（定向模糊测试）。例如，以覆盖率作为引导信息时，覆盖率指标要求能包罗所有模块和组件，否则可能会遗漏某些潜在的问题和错误。虽然覆盖率达到 100%并不能确保处理器不存在任何缺陷或漏洞，但基于合理的覆盖率指标，高覆盖率通常能够提高发现缺陷或漏洞的概率。其次，为了在 DUT 运行过程中收集这些引导信息，一般需要在 DUT 中插入特定代码（即插桩）。这些代码用来监视、记录和收集 DUT 执行过程中引导信息变化，并反馈给模糊测试平台。但插桩时要注意，一方面过多的插桩代码可能会带来额外的资源开销，导致模糊测试速度变慢。另一方面不正确的插桩代码可能会影响 DUT 的正确性，导致测试结果不准确。所以，测试引导技术不仅要制定适宜的引导信息，还要确保引导信息的插桩代码不会严重减慢或影响 DUT 的运行，以保证测试的有效性。

现有处理器模糊测试代表性研究工作所使用的测试引导技术如表 4 所示。从表中可以看出，大多数非定向模糊测试工作使用覆盖率作为测试引导技术，而定向模糊测试工作使用自定义的引导信息作为测试引导。

非定向模糊测试中最早使用 mux（多路复用器）作为覆盖率，mux 是数字电路中常见的重要元件之一，常用于数据选择、信号路由和控制逻辑等。最简单的 mux 有两个输入端和一个输出端，通过控制信号来选择其中一个输入信号输出。例如，表达式 mux（cond，tval，fval）表示 mux 在 cond 为真的情况下选择 tval 作为输出，cond 为假则选择 fval 作为输出。RFUZZ[9]将 mux 中 cond 的取值作为覆盖率，即每个 mux 中 cond=1 和 cond=0 是两种不同的处理器状态。但 DifuzzRTL[49]认为这种覆盖率是时钟不敏感的，不能精确捕获 FSM 的状态转换和识别跨时钟的多路复用器切换事件之间的相互依赖。因此它将控制 cond

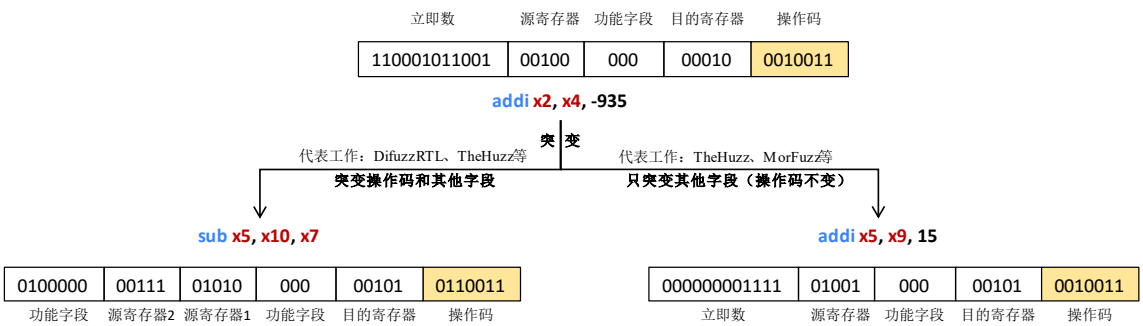


图 6 两类指令突变方式

Figure 6 Two types of instruction mutation methods

表 4 代表性处理器模糊测试研究采用的测试引导技术

Table 4 Test guidance techniques used in representative processor fuzzing studies

处理器模糊测试分类	代表性研究工作	引导技术	引导信息类型
非定向模糊测试	RFUZZ[9]	mux 切换覆盖率	覆盖率
	DifuzzRTL[49]	控制寄存器覆盖率	
	Fuzzing HW Like SW[50]	代码覆盖率	
	TheHuzz[51]	分支/FSM/代码等多种覆盖率	
	ProcessorFuzz[59]	CSR 覆盖率	
	SpecDoctor[52]	是否触发瞬态执行	自定义引导信息
定向模糊测试	DirectFuzz[47]	到目标模块实例的距离	
	SurgeFuzz[56]	标记的信号事件到祖先寄存器的路径长度	

取值的寄存器（称为控制寄存器）作为覆盖率，记录运行时每个时钟周期内所有控制寄存器的值。此外，还有研究工作使用 CSR（Control and Status Register，控制和状态寄存器）覆盖率[59]或代码覆盖率[50]作为测试引导技术。TheHuzz[51]认为仅使用上述单一的覆盖率指标可能会忽略潜在的错误，从而导致测试的全面性不足。因此 TheHuzz 使用了多种由 EDA 工具自带的覆盖率指标，如分支覆盖率、条件覆盖率、FSM 覆盖率、表达式覆盖率和语句覆盖率。

不同于上述非定向的模糊测试工作使用各种覆盖率指标作为引导信息，定向模糊测试通常使用到目标的距离作为引导信息。如 DirectFuzz[47]使用到目标模块实例的距离来引导测试生成方向，渐渐逼近目标模块实例。SurgeFuzz[56]按照从标记的信号事件到祖先寄存器的路径长度来引导测试产生更频繁的异常边界活动。它们都需要对 HDL 代码进行静态分析，基于电路结构来构建有向图，从有向图中抽象距离信息。

4.3 种子调度技术

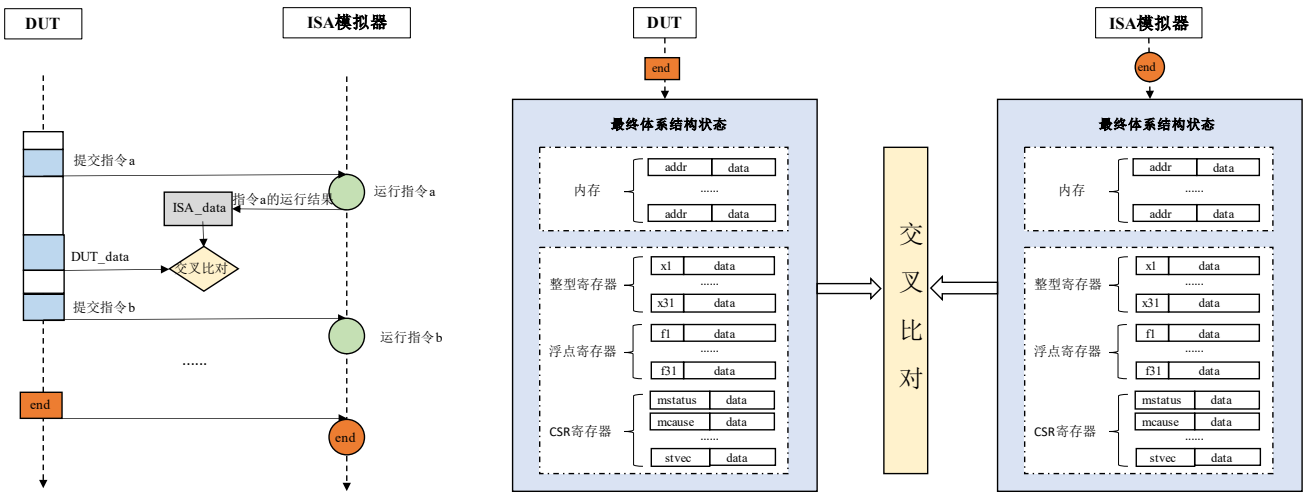
种子调度技术是处理器模糊测试中的重要技术，旨在根据引导信息保留优质种子到种子库，并从种子库中选择合适的种子进行突变。种子调度技术的思想是最大化对覆盖率或反馈信息有贡献的种子的利用度，从这些种子中选择优质种子，基于当前的优质种子突变得到更优种子，因此种子调度技术能直接影响输入的有效性。种子调度技术的难点在于寻找一种合适的种子选择策略，在聚焦于优质种子的同时，避免陷入局部最优解。一个良好种子选择策略能有效引导模糊测试快速探索不同的执行路径，提高漏洞挖掘效率。

对于以覆盖率作为引导信息的处理器模糊测试工作来说，它们都以是否提高覆盖率作为种子保留的依据，只记录那些提高覆盖率的种子，如 RFUZZ、DifuzzRTL 和 TheHuzz，但它们使用的种子选择策略不同。RFUZZ 和 TheHuzz 每次都选择实现了最高覆盖率的种子进行突变，从而保证了新种子的质量。DifuzzRTL 则从种子库中随机选择种子进行突变，不一定是触发最高覆盖率的种子，这样可以尽可能地探索更广阔状态空间。

而对于以距离作为引导信息的处理器模糊测试工作来说，它们只保留更接近测试目标的种子，如 DirectFuzz[47]和 SurgeFuzz[56]。由于每次都选择距离目标模块实例最近的种子很容易使得模糊测试方向陷入局部最优，因此 DirectFuzz 选择将触发新模块实例的种子作为突变对象，并额外引入能量值作为参数来动态调整种子选择策略。SurgeFuzz 中种子选择策略包括随机选择和基于能量分布的加权选择两种，可以根据需要灵活切换两种调度策略以达到最好的测试效果。

4.4 测试结果评估技术

测试结果评估技术是处理器模糊测试发现各种缺陷和漏洞的直接手段，旨在判定给定测试用例是否在 DUT 中产生了正确结果，以识别可能存在的各种问题。不同于软件模糊测试通过崩溃、内存泄漏和退出状态码来指示是否触发了漏洞，处理器本质上并不能提供这样的反馈信息，因为硬件仿真软件稳定地为 DUT 的任何输入产生输出。因此，必须为处理器模糊测试设计专门的测试结果评估方法，来判断给定输入是否触发了 DUT 中的错误。目前，常见的测试结果评估方法主要有断言检测和黄金参考模型（Golden Reference Model，GRM）交叉比对两种。断言检测方法用于在处理器设计过程中对特定条件进行断言或假设，并在仿真过程中检查这些条件是否成立。断言通常是一种声明，用于描述设计的预期行为或性质，如果出现断言违例，则表明设计存在问题。GRM 交叉比对通过将给定输入在 DUT 和 GRM 的输出结果逐一比较，来验证 DUT 是否产生了和 GRM 相同的预期结果，以检查 DUT 设计和实现的



(a) 运行时比对　　(b) 运行后比对

图 7 GRM 交叉比对时机

Figure 7　GRM cross comparison occasion

正确性。GRM 交叉比对结果的任何不匹配都表明 DUT 可能存在错误，需要对错误进行手动分析以确定其原因。GRM 是一个已经验证过的、功能正确的设计模型，通常是指令集模拟器，如 RISC-V 指令集模拟器 Spike[88]和 Dromajo[89]。总之，断言检测方法通常用来验证设计的特定行为，而 GRM 交叉比对方法验证的则是 DUT 整体设计的正确性。

断言检测：目前，只有少量处理器模糊测试工作使用断言检测作为判别处理器行为正确性的方法，如 SurgeFuzz[56]和 Fuzzing HW Like SW[50]。定向模糊测试工作 SurgeFuzz 将检测死锁等问题的断言插桩在 DUT 的 HDL 代码中，以检测 DUT 在运行过程中是否触发了特殊的边界情况。Fuzzing HW Like SW 直接对处理器的 HSB 实施软件模糊测试技术，由于错误的处理器行为时会触发 HSB 崩溃，因此可以直接使用 SVA（SystemVerilog Assertions）对处理器属性和行为进行检查。具体实现上，Fuzzing HW Like SW 在 SVAs 断言违例时向 HSB 进程发送 SIGABRT 信号，终止当前测试并记录错误行为。由于存在依赖人工编写、无法全面覆盖所有测试场景和调试困难等问题，断言检测并未成为处理器模糊测试的主流测试结果评估方法。

GRM 交叉比对：通过对比给定输入在 DUT 和 GRM 的执行结果是否一致，可以直观检查 DUT 行为是否符合预期。若存在不一致，首先需要定位出错的指令，然后再结合波形文件、DUT 源码和指令集手册分析导致不一致的根本原因。交叉比对的内容一般是体系结构状态如寄存器状态和内存内容，其中寄存器状态包括通用寄存器和 CSR 寄存器的值，而内存内容则指特定地址中记录的值，例如数据区等。GRM 交叉比对按照比较时机可以分为两种：运行后比对和运行时比对，如图 7 所示。

运行后比对仅在输入运行结束后才执行 DUT 和 GRM 的交叉比对，这就要求在最后一条指令执行完后要生成双方各自的体系结构状态快照，然后逐一比照这些状态是否都相同，这类方法以 DifuzzRTL 和 TheHuzz 为代表。由于只比较 DUT 和 GRM 的最终状态，这种方法的缺点是输入执行过程中反映在体系结构状态中的错误行为可能被稍后的正确执行覆盖和隐藏。此外，导致状态匹配不一致的根源出错指令可能距离匹配分歧点很远，需要人工追溯根源出错指令，加重调试工作量。

运行时比对方法支持 DUT 和 GRM 协同模拟，解决了运行后比对错误被覆盖和隐藏的问题。运行时比对旨在输入运行过程中，每执行完一条指令，就进行交叉比对。因此它需要构建一个同时运行 DUT 和 GRM 并支持双方之间通信的基础设施，能让它们同时开始执行相同的指令并相互传递消息。为了解决中断等外部事件带来的 DUT 和 GRM 行为不一致的问题，运行时比对方法强制 GRM 的控制流执行和 DUT 一致性的操作，允许它们共同模拟中断陷阱处理程序。这种方法以 MorFuzz 和 Logic Fuzzer[48]为主要代表。

5　处理器模糊测试优化与加速

将软件模糊测试技术应用到处理器验证，在很大程度上提升了处理器设计验证的效率和质量。现有典型的处理器模糊测试研究工作针对上一章提到的输入构造、测试引导、种子调度和测试结果评估技

术做了有效探索，取得了一定的成果。然而，它们的研究工作仍存在一定的不足，还有更多的优化和改进空间。本章将分析并总结现有典型处理器模糊测试研究工作的不足之处，并介绍针对这些不足的优化研究。

5.1 典型处理器模糊测试研究的不足

目前，典型的处理器模糊测试研究工作主要存在以下三点不足：

覆盖率引导作用不足、可扩展性差。覆盖率是处理器模糊测试的关键引导信息，指示目标硬件行为和状态空间的探索情况。良好的覆盖率指标能抽象处理器状态变化并引导模糊测试的广度和深度。然而，现有典型研究工作所使用的覆盖率指标均存在一定的局限性。一方面，覆盖率指标不能充分引导探索处理器状态空间。例如，RFUZZ[69]使用的 mux 覆盖率不能捕捉 FSM 不同周期的变化情况。DifuzzRTL[49] 的控制寄存器覆盖率虽然能反映时钟周期敏感的 FSM 状态，但它仅关注驱动 mux 选择信号的控制寄存器，当 mux 以不同方式实现（例如作为与/或/非门的组合）时，无法检测到 mux，就会忽略相应的控制寄存器，从而导致测试不充分。另一方面，现有覆盖率指标还存在一定的扩展性问题。例如，DifuzzRTL 和 RFUZZ 的覆盖率只适用于 FIRRTL 编译器，而不能直接应用到其他 HDL（如 Verilog 或 System Verilog）。ThuHuzz[51]的覆盖率指标不易用于 FPGA 仿真，从而限制了 TheHuzz 对基于 FPGA 仿真的模糊测试的适用性。此外，部分覆盖率指标统计开销过大。例如，TheHuzz 使用多种 EDA 实现的覆盖率指标，虽然能覆盖相对更全面的处理器状态空间，但在收集覆盖率时会导致 70%的运行时开销。

种子生成和调度策略简单。处理器模糊测试总体继承了软件模糊测试的基本流程，包括初始种子生成、种子突变、种子调度等环节。然而，大多数处理器模糊测试工作对这些环节的实现较为简单，各个环节的实现策略高度依赖随机性，未根据测试目标和达到的覆盖率情况动态调整策略，缺乏配合引导信息的动态调度，从而降低了漏洞检测和设计空间探索的速度。具体体现在种子运行完成度低、模糊测试指令在种子中占比小、种子调度策略单调等方面。据统计[67]，目前最主流处理器模糊测试器 DifuzzRTL 的平均种子执行完成度不到 20%，其中所执行的随机生成的模糊指令占比只有 5.8%（平均值）。此外，典型的处理器模糊测试研究工作使用的种子调度策略都较为简单，未探索种子与触发的状态空间之间的关联。例如，RFUZZ 和 TheHuzz 使用静态的最优覆盖率调度策略、DifuzzRTL 使用随机调度策略以及 DirectFuzz[47]采用新触发点种子优先调度策略。

模糊测试速度慢、吞吐量低。模糊测试的吞吐量与诸多因素有关，如不同输入之间的硬件状态重置、单线程测试用例输入和低效的覆盖率插桩统计等。首先，过多的状态重置是影响测试速度的重要原因。为避免多轮测试过程中的状态积累问题导致测试用例无法复现，需要确保在不同的测试用例之间将处理器重置为指定状态。例如 RFUZZ 使用 MetaReset 和 SparseMem 技术重置 DUT 初始状态，DifuzzRTL、ProcessorFuzz[59]等使用专门的初始化指令序列重置寄存器状态，并在程序中预设数据区来重置内存状态，以支持漏洞的可重现性。由于每轮测试都要执行这些重置操作，使得模糊测试花费了过多时间来执行这些不属于模糊指令的操作，影响测试速度和吞吐量。其次，单线程单输入测试框架也极大地影响了测试吞吐量。现有处理器模糊测试研究工作基本上都基于 CPU 的单线程单输入架构，每轮测试只能接收和执行一个输入，未能充分利用现代计算机的多核和 SIMD（Single Instruction, Multiple Data）等硬件机制，实现多线程并行测试和同时多输入。此外，低效的覆盖率插桩统计代码也会减慢测试速度。例如，相比于小型处理器设计，大型处理器设计的 mux 数量会呈指数型增长，极大加重 RFUZZ 的 mux 覆盖率插桩和统计工作。而 TheHuzz 收集覆盖率所需的运行时开销占总资源消耗的 70%。

5.2 处理器模糊测试优化

针对 5.1 章节中列出的现有典型处理器模糊测试研究的不足之处，涌现了许多针对性的优化研究工作，这些优化工作所聚焦的处理器模糊测试阶段如图 8 所示。表 5 详细列出了这些研究工作的优化目标以及在覆盖率、测试速度和漏洞挖掘能力等方面的优化效果。接下来将从不同的优化目标分类介绍这些处理器模糊测试研究工作。

5.2.1 优化覆盖率指标

优化覆盖率指标着力于探索更高的覆盖率或者设计更能充分体现处理器状态的覆盖率指标，接下来以 HyPFuzz[61]和 ProcessorFuzz[59]为例分别说明优化设计思路。

观察到现有处理器模糊测试存在难以覆盖全部设计空间的问题，HyPFuzz 利用形式化验证来辅助模糊测试探索难以覆盖的空间。当模糊测试方法饱和时（即覆盖率长久未提升），HyPFuzz 从模糊测试切换至形式化验证，并选择一个未被覆盖的状态空间，

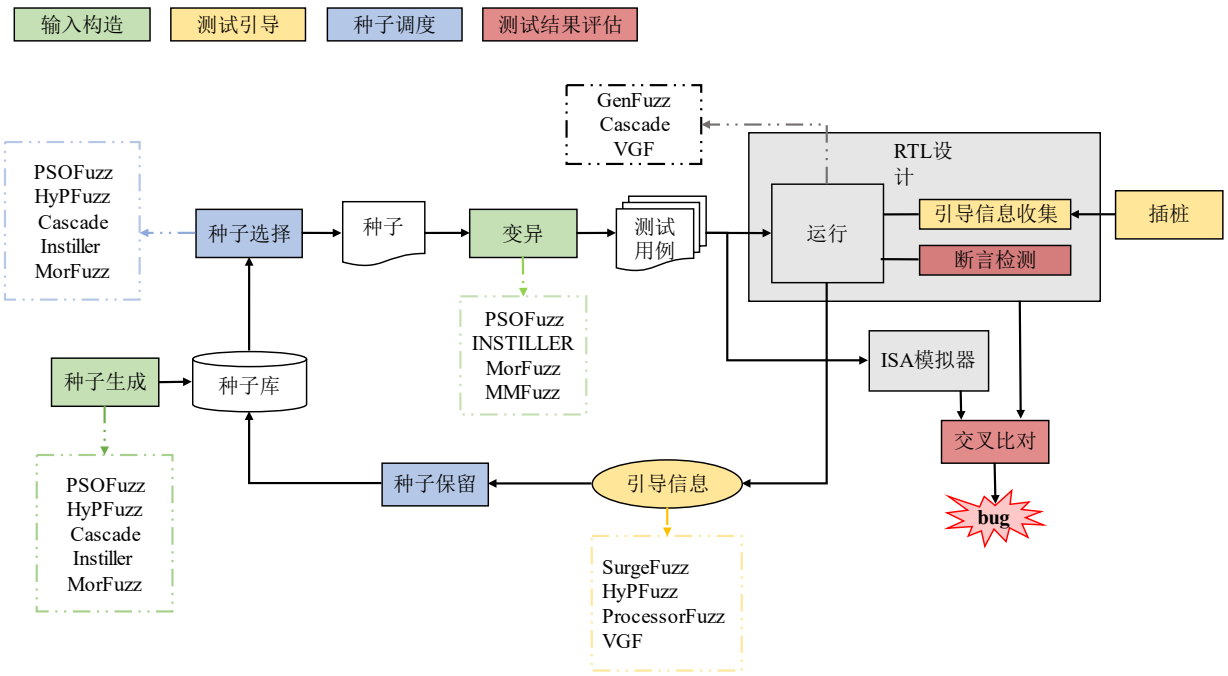


图 8 处理器模糊测试优化工作

Figure 8 Processor fuzzing optimization overview

然后利用形式化验证工具为该状态空间生成能覆盖它的最小实例。接着使用自定义的测试用例转换器将该实例转化为模糊测试能够接受的输入激励，当再次运行模糊测试时，便能覆盖这个之前长久未能被覆盖的状态空间。因此，HyPFuzz 能有效改善传统模糊测试方法探索新状态空间能力不足的问题，在模糊测试方法不能发挥作用时借助形式化验证工具产生能覆盖新状态空间的输入，以追求尽可能覆盖所有的处理器状态，实验结果也验证了 HyPFuzz 在提升覆盖率方面的有效性，经过评估，HyPFuzz 能达到比 TheHuzz 高 1.72%的覆盖率，达到相同覆盖率的速度比 TheHuzz 快 11.68 倍。此外，HyPFuzz 不仅能检测到 TheHuzz 报告的所有漏洞，还发现了 3 个新的漏洞。

ProcessorFuzz[59]认识到 DifuzzRTL 采用的控制寄存器覆盖率在很多情况下并不能反映 FSM 状态的变化。如图 9 所示，处理器从 S0 状态开始，在条件 P1 的控制下转换到状态 S1，然后又在 P1 的控制下由 S1 转到 S2 状态。但 DifuzzRTL 仅能记录控制条件 P1 的状态，无法识别和区分此时进行的是 S0->S1 还是 S1->S2 的状态转换，所以 DifuzzRTL 中的控制寄存器覆盖率并不能作为充分探索处理器状态的指标。因此，ProcessorFuzz 提出了一种基于控制和状态寄存器（Control and Status Register, CSR）变化的覆盖率指标，CSR 寄存器能记录图 9 中 S1 和 S2 的值，即能体现处理器中的状态转换。这种更能充分代表处理器状态的覆盖率指标使得 ProcessorFuzz 在三个开源 RISC-V 处理器核上发现了 6 个新漏洞，并且发现漏洞的速度比 DifuzzRTL 平均快 1.23 倍。

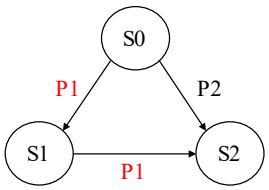


图 9 处理器状态转换图（简单示例）

Figure 9 Processor state transition diagram (simple example)

5.2.2 优化种子生成与调度策略

现有优化研究工作主要采用智能算法（如粒子群优化算法、多臂赌博机算法、蚁群算法）或自定义算法来优化种子生成与调度策略。

PSOFuzz[57]使用粒子群优化（Particle Swarm Optimization，PSO）算法来优化种子生成和变异的过程。具体来说，PSOFuzz 将生成种子最佳指令的概率问题形式化为 PSO 问题，设计了基于 PSO 的种子动态生成算法。借助 PSO 算法，PSOFuzz 可以动态调整每个线程所选择的最佳变异操作，从而更有效地挖掘漏洞。相比于 TheHuzz，PSOFuzz 实现了 15.25 倍的漏洞检测加速和 2.22 倍的覆盖率加速。

INSTILLER[68]提出了一种通过蚁群算法来精简种子的方法，通过在不影响覆盖率的前提下缩短种

表 5 代表性优化研究采用的方法与优化效果

Table 5 Methods and performance in representative processor fuzzing optimization studies

优化目标	代表性研究工作	优化方法	BaseLine	覆盖率大小提升	覆盖率速度提升	新漏洞检测	测试速度提升	其他优化效果
覆盖率指标	HyPFuzz[61]	模糊测试与形式化验证相结合	TheHuzz	1.72%	11.68 倍	3	3.06 倍	-
覆盖率指标	ProcessorFuzz[59]	提出 CSR 覆盖率	DifuzzRTL	-	-	6	1.23 倍	-
种子生成	PSOFuzz[57]	将生成种子最佳指令转换为粒子群优化问题	TheHuzz	2%	6.39 倍	-	15.25 倍	-
种子生成	INSTILLER[68]	使用蚁群算法精简种子	DifuzzRTL	11%	-	-	-	-
种子生成	Cascade[67]	保证种子 100%执行完成度	DifuzzRTL	-	97	37	-	-
调度策略	MABFuzz[65]	将种子选择问题转换为多臂赌博机问题	TheHuzz	0.68%	3.05 倍	-	最高 308.89 倍	种子长度减少了 74%
测试吞吐量	GenFuzz[58]	充分利用 CPU 多核和 GPU，支持多输入	DifuzzRTL	-	2.1 倍	-	-	运行时间加速可达 80 倍

子的长度，得到能触发相同覆盖率的精简种子。INSTILLER 将指令进行聚类，用于将总是同时出现的、共同实现某一功能的多个指令划分到同一个基本块中，并提取它们之间所具有的控制流和数据流关系。当种子的覆盖率与长度的比值连续下降时（即越来越长的指令序列对覆盖率并没有贡献），INSTILLER 对这类种子应用蚁群算法，利用上述关系对种子进行精简，直到覆盖率与长度的比值回升到既往水平。因此，INSTILLER 能避免对覆盖率无贡献的指令占用过多的测试时间，从而允许在相同时间内执行更多的测试用例。INSTILLER 相比 DifuzzRTL 实现了 11%的覆盖率提升，同时触发相同漏洞的种子长度相比 DifuzzRTL 减少了 74%。

Cascade[67]针对种子运行完成度不高的问题，改进了种子生成策略。Cascade 首先生成与控制流无关、仅包含数据流的中间程序。然后基于中间程序运行后的上下文信息构建控制流，将多个中间程序组合成一个完整的输入，该输入具备较复杂的控制流和数据流，同时还能保证高完成度。由于每个种子都能保证 100%的执行完成度，所以 Cascade 可以充分利用生成的所有指令探索更丰富的处理器状态。在实验中，Cascade 发现了 5 个 RISC-V 开源处理器核中的 37 个新漏洞。

MABFuzz[65]基于多臂赌博机（Multi-Armed Bandit，MAB）算法，根据历史收益动态调整变异策略。MABFuzz 将种子选择问题转换为多臂老虎机问题，然后使用算法推测最优的种子选择策略。该方法首先为每个臂创建一个种子集，然后代理根据多臂老虎机算法的初始概率选择一个臂。接下来，执行被选择的臂对应的种子集中的一个种子，并获取相应的覆盖率信息。借助多臂赌博机算法，MABFuzz 可以在种子选择环节综合考虑更有潜力的种子和执行次数较少的种子，以平衡测试资源的分配，从而可以实现更佳的检测效率。相比 TheHuzz，该方法在七个不同漏洞上的检测速度实现了最高 308.89 倍的加速，在覆盖率方面实现了 3.05 倍的提升。

5.2.3 提升测试吞吐量

提升吞吐量指提高相同时间内执行的测试用例数目，从而能执行更多的测试用例，提高模糊测试效率，力求挖掘更多的处理器漏洞。除了上述研究的优化方法间接对测试吞吐量产生了正面影响，一些研究旨在直接优化模糊测试器的吞吐量，以提高测试效率，其中以 GenFuzz[58]最具代表性。

GenFuzz 专门利用 GPU 对 RTL 仿真进行硬件加速，从而提高吞吐量。不同于其他研究工作没有改变模糊测试器串行运行的基本架构，GenFuzz 在模糊测试过程中充分利用 CPU 的多个核心，并使用 GPU 加速 RTL 模拟。有了多核 CPU 和 GPU 的助力，GenFuzz 能支持多输入，提升了模糊测试中时间消耗较大的环节的并行性，从而极大提升了模糊测试效率。实验表明，GenFuzz 的覆盖率提升速度相比 DifuzzRTL 最高提高了 80 倍。

此外，还有研究通过在测试的精度和速度之间进行平衡，以提高模糊测试效率。例如，VGF[66]通过使用修改后的 Verilator 输出模型，在牺牲单次模拟精度的情况下，实现了更高的模糊测试吞吐量，从而实现了总体测试效率的提升。

6 总结与展望

6.1 现有处理器模糊测试研究工作总结

模糊测试是一种主流的自动化软件测试技术，被广泛应用于挖掘各种类型的软件缺陷和漏洞。近些年来，研究学者将软件模糊测试技术应用到处理器验证中，不断改进和优化模糊测试技术以适配处理器验证特性，弥补了传统处理器验证方法速度慢、效率低和可扩展性差等缺点。处理器模糊测试主要聚焦于对输入构造技术、测试引导技术、种子调度技术和测试结果评估技术的研究。现有处理器模糊测试工作按照所针对的漏洞类型可以分为两大类：针对功能缺陷的模糊测试和针对安全漏洞的模糊测试，目前处理器模糊测试研究工作主要集中在挖掘各类开源处理器的功能缺陷，同时对安全漏洞的关注也日益增加。当前处理器模糊测试的测试对象以 RISC-V 开源处理器核为主，热门 DUT 包括 BOOM、Rocket 和 CVA6 等。

近三年来，出现了许多针对典型处理器模糊测试工作的优化和加速研究，它们主要聚焦于：（1）优化覆盖率指标：引入形式化验证或新的覆盖率指标。例如，HyPFuzz 使用形式化验证辅助模糊测试探索难以覆盖的状态空间，ProcessorFuzz 使用 CSR 覆盖率，通过监视 CSR 寄存器的变化有效地提高了模糊测试的效率和覆盖范围。（2）优化种子生成策略：引入智能算法和动态策略来提高测试用例的生成和选择效率。例如，PSOFuzz 和 MABFuzz 利用智能算法优化突变操作符的选择，提高突变策略的效果。Cascade 和 INSTILLER 通过改进种子生成策略和精简测试用例，解决了测试用例完成度低和冗余开销大的问题，进一步提高了漏洞检测的效率和准确性。（3）提高测试吞吐量：引入硬件加速和模拟优化等技术来支持多输入和加快测试速度。例如，GenFuzz 通过利用 GPU 等硬件资源支持并行处理测试用例，VGF 优化 Verilator 等工具的软件模拟速度来提高测试用例执行效率。这些优化技术使得处理器模糊测试更加快速和高效，增强漏洞挖掘能力，提升处理器状态空间覆盖的广度和深度。

据统计，现有处理器模糊测试工具在开源处理器核上累计发现了 150 多个缺陷和漏洞，极大地提高了这些处理器核的安全性和功能正确性。

6.2 未来研究方向展望

未来的处理器模糊测试可能有以下发展方向：

自动化且高效的测试结果评估方案。在处理器模糊测试的测试结果评估环节中，并非所有的交叉比对不一致结果都是由处理器中的缺陷或漏洞导致的。在 ISA 规范手册中有许多未强制规定的约束，称为 platform-specific 或 implementation-specific。也就是说，即使在 DUT 和 GRM 交叉比对中出现了不匹配的情况，也不能断定发现了真正的处理器漏洞。因为这可能是由于 DUT 和 GRM 的某些实现差异导致的误报，而这种实现差异是符合指令集规范的，不能称之为漏洞。由于 GRM 的普适性和 DUT 的特定性，GRM 和 DUT 中不可避免地存在许多实现差异，例如 XFUZZ[62]在实践中遇到的至少 120 个实现差异。

然而，识别漏洞误报是非常耗时的。为了定位不一致信息的原因，验证人员都要先定位出错的源头指令，然后再复现漏洞，得到输入在 DUT 中执行的波形文件。最后根据测试程序、波形文件、DUT 源码和 GRM 源码分析源头指令执行出错的根本原因。然而由于现代处理器设计复杂，包含大量的逻辑和功能模块，定位不一致信息根因是处理器调试中的难点。定位具体原因后，还需要翻阅指令集手册来进一步判定不一致信息是否是由于 DUT 和 GRM 实现差异导致的误报。

此外，排除漏洞误报是非常困难的。同一个漏洞误报原因可能会导致不同的不一致信息表象，分析已知原因的误报是耗时且无意义的。因此，为了减少验证人员的工作量，在测试过程中排除已知漏洞导致的误报是非常必要的。通常人们通过修改 GRM 来避免已知误报重复出现，但这不仅要求验证人员同时熟悉 GRM 和 DUT，也要求对 GRM 的修改不影响其正确性，进一步提高了验证的门槛的难度。

因此，结合模糊测试技术设计自动化且高效的测试结果评估方案以识别和排除漏洞误报显得尤为重要，它可以最大程度减少验证人员的调试工作量，是一个兼具商业价值和学术价值的研究内容。

检测处理器中的隐蔽漏洞。处理器中的隐蔽漏洞指那些不容易被发现或者不容易被利用的漏洞。这些漏洞通常存在于处理器的微架构中，只有在某些特定条件下才能被检测到，例如缓存一致性问题、多核一致性问题、数据依赖性问题等。当前的处理器模糊测试研究工作主要集中在挖掘那些相对容易被检测到的表面漏洞，而对于深层的隐蔽漏洞则尚未进行充分的深入探索。随着表面漏洞被逐渐挖掘完毕，未来的研究工作应当转向挖掘处理器中更深层次的隐蔽漏洞，以进一步提高处理器的安全性和可靠性。

针对性能问题的处理器模糊测试研究。现有处理器模糊测试研究主要集中在挖掘功能缺陷和安全漏洞上，而忽视了检测性能问题。性能问题也是处理

器中的一类重要问题，它专指那些影响处理器性能的缺陷或问题，从而导致处理器在执行任务时出现性能下降、速度变慢或资源利用不当的情况。性能问题可能与处理器内部设计、指令执行、缓存管理和分支预测等各个方面有关，会影响处理器的整体性能表现。因此，基于模糊测试技术探索处理器的性能问题是一个重要的研究方向，以确保处理器在设计规格范围内提供预期的性能水平，优化用户体验，提高系统效率和资源利用率。

将机器学习技术应用到处理器模糊测试。机器学习技术具备自动化学习和泛化能力，能够从数据中学习规律并应用于未知数据，实现智能决策和预测，是当前热门的人工智能技术之一。目前已有大量研究工作将机器学习技术应用到软件模糊测试[90-94]，大大提高了软件测试效率和准确度。然而，处理器模糊测试处于初步发展阶段，现阶段仍以人工分析处理器特性和定制引导策略为主。在不久的将来，相信会有许多研究学者借助机器学习技术的特性，学习输入和特定硬件行为间的联系，更好地优化种子突变和引导策略，实现智能化测试，为处理器验证带来新的可能性。

参考文献

[1] Pentium processor specification update. Intel Corporation. 1999.

[2] Lipp M, Schwarz M, Gruss D, et al. Meltdown: Reading kernel memory from user space [J]. *Communications of the ACM*, 2020, 63(6): 46-56.

[3] Kocher P, Horn J, Fogh A, et al. Spectre attacks: Exploiting speculative execution [J]. *Communications of the ACM*, 2020, 63(7): 93-101.

[4] Revision guide for amd family 10h processors revision 3.92. Amd Corporation.

[5] Fine S, Ziv A. Coverage directed test generation for functional verification using bayesian networks [C]. *Proceedings of the 40th annual Design Automation Conference*, 2003: 286-291.

[6] Li J, Zhao B, Zhang C. Fuzzing: a survey [J]. *Cybersecurity*, 2018, 1(1): 1-13.

[7] Zhu X, Wen S, Camtepe S, et al. Fuzzing: A Survey for Roadmap [J]. *ACM Computing Surveys*, 2022, 54(11s): 1-36.

[8] Liang H, Pei X, Jia X, et al. Fuzzing: State of the art [J]. *IEEE Transactions on Reliability*, 2018, 67(3): 1199-1218.

[9] Laeufer K, Koenig J, Kim D, et al. RFUZZ: Coverage-directed fuzz testing of RTL on FPGAs [C]. *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018: 1-8.

[10] Zenbleed. Ormandy T. https://lock.cmpxchg8b.com/zenbleed.html. Jul. 2023.

[11] Fu W, Arias O, Jin Y, et al. Fuzzing Hardware: Faith or Reality? [C]. *2021 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, 2021: 1-6.

[12] Ren Z, Zheng H, Zhang J, et al. A Review of Fuzzing Techniques [J]. *Journal of Computer Research and Development*, 2021, 58(05): 944-963.
（任泽众, 郑晗, 张嘉元, et al. 模糊测试技术综述 [J]. *计算机研究与发展*, 2021, 58(05): 944-963.）

[13] Li H, Zhang C, Yang X, et al. Survey of OS Kernel Fuzzing [J]. *Journal of Chinese Computer Systems*, 2019, 40(09): 1994-1999.
（李贺, 张超, 杨鑫, et al. 操作系统内核模糊测试技术综述 [J]. *小型微型计算机系统*, 2019, 40(09): 1994-1999.）

[14] Liang J, Wu Z-Y, Fu J-Z, et al. Survey on Database Management System Fuzzing Techniques [J]. *Journal of Software*: 1-25.
（梁杰, 吴志镛, 符景洲, et al. 数据库管理系统模糊测试技术研究综述 [J]. *软件学报*: 1-25.）

[15] Yu B, Su J-S, Yang Q, et al. Survey on Vulnerability Mining Techniques of Network Protocol Software [J]. *Journal of Software*, 2024, 35(02): 872-898.
（喻波, 苏金树, 杨强, et al. 网络协议软件漏洞挖掘技术综述 [J]. *软件学报*, 2024, 35(02): 872-898.）

[16] Yu Y-C, Chen Z-N, Gan S-T, et al. Research on the Technologies of Security Analysis Technologies on the Embedded Device Firmware [J]. *Chinese Journal of Computers*, 2021, 44(05): 859-881.
（于颖超, 陈左宁, 甘水滔, et al. 嵌入式设备固件安全分析技术研究 [J]. *计算机学报*, 2021, 44(05): 859-881.）

[17] American Fuzzy Lop. Zalewski M. http://lcamtuf.coredump.cx/afl/technical_details.txt. 2014.

[18] SPIKE. Aitel D. https://github.com/guilhermeferreira/spikepp.

[19] Zhong R, Chen Y, Hu H, et al. SQUIRREL: Testing Database Management Systems with Language Validity and Coverage Feedback [C]. *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020: 955–970.

[20] Xu W, Moon H, Kashyap S, et al. Fuzzing file systems via two-dimensional input space exploration [C]. *2019 IEEE Symposium on Security and Privacy (SP)*, 2019: 818-834.

[21] Talebi S M S, Tavakoli H, Zhang H, et al. Charm: Facilitating dynamic analysis of device drivers of mobile systems [C]. *27th USENIX Security Symposium (USENIX Security 18)*, 2018: 291-307.

[22] Schumilo S, Aschermann C, Gawlik R, et al. kAFL: Hardware-Assisted feedback fuzzing for OS kernels [C]. *26th USENIX security symposium (USENIX Security 17)*, 2017: 167-182.

[23] Schumilo S, Aschermann C, Abbasi A, et al. HYPER-CUBE: High-Dimensional Hypervisor Fuzzing [C]. *Network and Distributed System Security Symposium*, 2020

[24] Common Vulnerabilities and Exposures. (2014). CVE-2014-0160. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160. Dec. 2013.

[25] c-ares-CVE-2016-5180. Project C-Ares Security Advisory. https://c-ares.org/adv_20160929.html. Sept. 2016.

[26] Common Vulnerabilities and Exposures. (2019). CVE-2019-7317. https://nvd.nist.gov/vuln/detail/CVE-2019-7317. Apr. 2019.

[27] Common Vulnerabilities and Exposures. (2010). CVE-2010-2249. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2249. Jun. 2010.

[28] 12th generation intel core processor specification update revision 008. Intel. 2022.

[29] Cve details: Intel: Vulnerability statistics. Mitre. https://www.cvedetails.com/vendor/238/Intel.html. Aug. 2019.

[30] Part 12: The 2018 Wilson Research Group Functional Verification Study (IC/ASIC Verification Results Trends). Foster H. https://blogs.sw.siemens.com/verificationhorizons/2019/03/13/part-12-the-2018-wilson-research-group-functional-verification-study/. Mar. 2019.

[31] Cadence Webpage. Cadence. https://www.cadence.com.

[32] VC-Formal. Synopsys. https://www.synopsys.com/verification/static-and-formal-verification/vc-formal.html. 2021.

[33] Kaivola R, Ghughal R, Narasimhan N, et al. Replacing Testing with Formal Verification in Intel Core i7 Processor Execution Engine Validation [C]. *International Conference on Computer Aided Verification*, 2009: 414-429.

[34] SymbiYosys (sby) - Front-end for Yosys-based formal verification flows. Wolf C. https://github.com/YosysHQ/SymbiYosys. 2017.

[35] Chen M, Qin X, Koo H-M, et al. System-level validation: high-level modeling and directed test generation techniques [M]. Springer Science & Business Media, 2012.

[36] Huang B-Y, Zhang H, Subramanyan P, et al. Instruction-Level Abstraction (ILA): A Uniform Specification for System-on-Chip (SoC) Verification [J]. *ACM Transactions on Design Automation of Electronic Systems*, 2018, 24(1): 1-24.

[37] Dessouky G, Gens D, Haney P, et al. HardFails: Insights into Software-Exploitable Hardware Bugs [C]. *28th USENIX Security Symposium (USENIX Security 19)*, 2019: 213-230.

[38] Universal Verification Methodology (UVM). Accellera. https://www.accellera.org/downloads/standards/uvm.

[39] cocotb. Cocotb. https://github.com/cocotb/cocotb.

[40] Trippel T, Shin K G, Bush K B, et al. Bomberman: Defining and defeating hardware ticking timebombs at design-time [C]. *2021 IEEE Symposium on Security and Privacy (SP)*, 2021: 970-986.

[41] Piziali A. Functional verification coverage measurement and analysis

[M]. Luxembourg: Springer Science & Business Media, 2007.
[42] Tasiran S, Keutzer K. Coverage metrics for functional validation of hardware designs [J]. *IEEE Design & Test of Computers*, 2001, 18(4): 36-45.
[43] Jou J-Y, Liu C-N J. Coverage analysis techniques for HDL design validation [J]. *Proc Asia Pacific CHip Design Languages*, 1999: 48-55.
[44] Verilator. Snyder W. https://www.veripool.org/wiki/verilator.
[45] VCS. Synopsys. https://www.synopsys.com/verification/simulation/vcs.html.
[46] afl-gcc. Zalewski M. https://github.com/google/AFL/blob/master/afl-gcc.c.
[47] Canakci S, Delshadtehrani L, Eris F, et al. Directfuzz: Automated test generation for rtl designs using directed graybox fuzzing [C]. *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021: 529-534.
[48] Kabylkas N, Thorn T, Srinath S, et al. Effective processor verification with logic fuzzer enhanced co-simulation [C]. *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021: 667-678.
[49] Hur J, Song S, Kwon D, et al. DifuzzRTL: Differential Fuzz Testing to Find CPU Bugs [C]. *2021 IEEE Symposium on Security and Privacy (SP)*, 2021: 1286-1303.
[50] Trippel T, Shin K G, Chernyakhovsky A, et al. Fuzzing hardware like software [C]. *31st USENIX Security Symposium (USENIX Security 22)*, 2022: 3237-3254.
[51] Kande R, Crump A, Persyn G, et al. TheHuzz: Instruction Fuzzing of Processors Using Golden-Reference Models for Finding Software-Exploitable Vulnerabilities [C]. *31st USENIX Security Symposium (USENIX Security 22)*, 2022: 3219-3236.
[52] Hur J, Song S, Kim S, et al. SpecDoctor: Differential Fuzz Testing to Find Transient Execution Vulnerabilities [C]. *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022: 1473-1487.
[53] Ragab H, Koning K, Bos H, et al. BugsBunny: Hopping to RTL Targets with a Directed Hardware-Design Fuzzer [J]. *SILM, June*, 2022.
[54] Bruns N, Herdt V, Große D, et al. Efficient cross-level processor verification using coverage-guided fuzzing [C]. *Proceedings of the Great Lakes Symposium on VLSI 2022*, 2022: 97-103.
[55] Oleksenko O, Fetzer C, Köpf B, et al. Revizor: Testing black-box CPUs against speculation contracts [C]. *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022: 226-239.
[56] Sugiyama Y, Matsuo R, Shioya R. SurgeFuzz: Surge-Aware Directed Fuzzing for CPU Designs [C]. *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 2023: 1-9.
[57] Chen C, Gohil V, Kande R, et al. PSOFuzz: Fuzzing Processors with Particle Swarm Optimization [C]. *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 2023: 1-9.
[58] Lin D-L, Zhang Y, Ren H, et al. Genfuzz: Gpu-accelerated hardware fuzzing using genetic algorithm with multiple inputs [C]. *2023 60th ACM/IEEE Design Automation Conference (DAC)*, 2023: 1-6.
[59] Canakci S, Rajapaksha C, Delshadtehrani L, et al. ProcessorFuzz: Processor Fuzzing with Control and Status Registers Guidance [C]. *2023 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2023: 1-12.
[60] Xu J, Liu Y, He S, et al. MorFuzz: Fuzzing Processor via Runtime Instruction Morphing enhanced Synchronizable Co-simulation [C]. *32nd USENIX Security Symposium (USENIX Security 23)*, 2023: 1307-1324.
[61] Chen C, Kande R, Nguyen N, et al. HyPFuzz: formal-assisted processor fuzzing [C]. *Proceedings of the 32nd USENIX Conference on Security Symposium*, 2023: Article 77.
[62] Xu Y-N, Yu Z-H, Wang K-F, et al. Functional Verification for Agile Processor Development: A Case for Workflow Integration [J]. *Journal of Computer Science and Technology*, 2023, 38(4): 737-753.
[63] Cheng Y, Zou H, He J, et al. MMFuzz: Towards Enhancing RTL Fuzz Testing Using Metric Feedbacks Based on Markov Chain [C]. *2023 IEEE 32nd Asian Test Symposium (ATS)*, 2023: 1-6.
[64] Rajapaksha C, Delshadtehrani L, Egele M, et al. SIGFuzz: A Framework for Discovering Microarchitectural Timing Side Channels [C]. *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2023: 1-6.
[65] Gohil V, Kande R, Chen C, et al. MABFuzz: Multi-Armed Bandit Algorithms for Fuzzing Processors [J]. *arXiv preprint arXiv:231114594*, 2023.
[66] Dai R, Lee M, Hoey P, et al. VGF: Value-Guided Fuzzing--Fuzzing Hardware as Hardware [J]. *arXiv preprint arXiv:231206580*, 2023.
[67] Solt F, Ceesay-Seitz K, Razavi K. Cascade: CPU Fuzzing via Intricate Program Generation [C]. *33rd USENIX Security Symposium (USENIX Security 24)*, 2024
[68] Zhang G, Wang P, Yue T, et al. INSTILLER: Towards Efficient and Realistic RTL Fuzzing [J]. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2024.
[69] Borkar P, Chen C, Rostami M, et al. WhisperFuzz: White-Box Fuzzing for Detecting and Locating Timing Vulnerabilities in Processors [J]. *arXiv preprint arXiv:240203704*, 2024.
[70] Chen W, Ray S, Bhadra J, et al. Challenges and trends in modern SoC design verification [J]. *IEEE Design & Test*, 2017, 34(5): 7-22.
[71] "sifive-blocks". Sifive. https://github.com/sifive/sifive-blocks/tree/master/src/main/scala/devices. 2020.
[72] RISC-V Homepage. Risc-V International. https://riscv.org/.
[73] OpenRISC Homepage. Openrisc. https://openrisc.io/.
[74] The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2. Waterman A, Asanovic K. https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf. May 2017.
[75] Asanovic K, Avizienis R, Bachrach J, et al. The rocket chip generator. Technical Report 2016,EECS Department, University of California, Berkeley, Tech Rep UCB/EECS-2016-17, 2016.
[76] Zhao J, Korpan B, Gonzalez A, et al. Sonicboom: The 3rd generation berkeley out-of-order machine [C]. *Fourth Workshop on Computer Architecture Research with RISC-V*, 2020
[77] Zaruba F, Benini L. The cost of application-class processing: Energy and performance analysis of a Linux-ready 1.7-GHz 64-bit RISC-V core in 22-nm FDSOI technology [J]. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2019, 27(11): 2629-2640.
[78] The Sodor Processor. Celio C. https://github.com/ucb-bar/riscv-sodor.
[79] Petrisko D, Gilani F, Wyse M, et al. BlackParrot: An agile open-source RISC-V multicore for accelerator SoCs [J]. *IEEE Micro*, 2020, 40(4): 93-102.
[80] Mashimo S, Fujita A, Matsuo R, et al. An open source FPGA-optimized out-of-order RISC-V soft processor [C]. *2019 International Conference on Field-Programmable Technology (ICFPT)*, 2019: 63-71.
[81] Xu Y, Yu Z, Tang D, et al. Towards developing high performance RISC-V processors using agile methodology [C]. *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022: 1178-1199.
[82] NutShell. Oscpu. https://github.com/OSCPU/NutShell.
[83] FIRRTL (A Flexible Intermediate Representation for RTL). Uc Berkeley Architecture Research. https://bar.eecs.berkeley.edu/projects/firrtl.html.
[84] riscv-tests. Asanovic K, Johnson D. https://github.com/riscv/riscv-tests. 2013.
[85] Risc-v torture test. Lee Y, Cook H. https://github.com/ucb-bar/riscv-torture.
[86] riscv-compliance. Bennett J, Moore L. https://github.com/riscv/riscv-compliance.
[87] SV/UVM based instruction generator for RISC-V processor verification. Liu T, Ho R, Jonnalagadda U. https://github.com/chipsalliance/riscv-dv. 2019.
[88] Spike RISC-V ISA simulator. Risc-V International. https://github.com/riscv-software-src/riscv-isa-sim.
[89] Dromajo - Esperanto Technology's RISC-V Reference Model. Esperanto Technologies. https://github.com/chipsalliance/dromajo. 2019.
[90] Neural fuzzing: applying DNN to software security testing. Blum W. https://www.microsoft.com/en-us/research/blog/neural-fuzzing/. Nov. 2017.
[91] Godefroid P, Peleg H, Singh R. Learn&fuzz: Machine learning for input fuzzing [C]. *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017: 50-59.
[92] Rajpal M, Blum W, Singh R. Not all bytes are equal: Neural byte sieve for fuzzing [J]. *arXiv preprint arXiv:171104596*, 2017.
[93] She D, Pei K, Epstein D, et al. Neuzz: Efficient fuzzing with neural program smoothing [C]. *2019 IEEE Symposium on Security and Privacy (SP)*, 2019: 803-817.
[94] Nicolae M-I, Eisele M, Zeller A. Revisiting Neural Program Smoothing for Fuzzing [C]. *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023: 133-145.