分　类　号：TP315　　单位代码：10183
研究生学号：2019544033　　密　　级：公开

吉 林 大 学

硕士学位论文

(专业学位)

融合模糊测试和形式化验证的 RUST 测试工具研究

Research on RUST Testing Tools Integrating Fuzzing and Formal Verification

作 者 姓 名 ：牛聚川
类　　别 ：工程硕士
领域（方向）：软件工程
指 导 教 师 ：张永刚　教授
培 养 单 位 ：软件学院

2022 年 05 月

融合模糊测试和形式化验证的 RUST 测试工具研究

Research on RUST Testing Tools Integrating Fuzzing and Formal Verification

作 者 姓 名：牛聚川

领域（方向）：软件工程

指 导 教 师：张永刚　教授

类　　别：工程硕士

答 辩 日 期：2022 年 5 月 19 日

吉林大学硕士学位论文原创性声明

本人郑重声明：所呈交学位论文，是本人在指导教师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：牛聚川

日期：　2022 年　5　月　24　日

关于学位论文使用授权的声明

本人完全了解吉林大学有关保留、使用学位论文的规定，同意吉林大学保留或向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅；本人授权吉林大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或其他复制手段保存论文和汇编本学位论文。

（保密论文在解密后应遵守此规定）

论文级别：☑硕士 □博士

学科专业： 软件工程

论文题目： 融合模糊测试和形式化验证的 RUST 测试工具研究

作者签名：牛聚川 指导教师签名：

2022 年 5 月 24 日

摘　要

融合模糊测试和形式化验证的RUST测试工具研究

Rust语言是一种兼顾了安全和效率的新型编程语言。近70%的计算机漏洞来源自Memory Management，而Rust语言几乎杜绝了此类问题，这样的特性令Rust语言极适合被主要用于一系列系统软件的开发。随着Rust语言走向成熟，它逐渐得到了工业界的广泛关注。微软公司已经将其应用到关键组件的开发中，Amazon,Google，蚂蚁金服等数百家国内外知名公司也开始利用Rust语言满足用户的安全需求；此外，科研界也对Rust语言产生了浓厚的兴趣：自2017年至今，Rust语言已禅联Stack Overflow论坛“最受欢迎编程语言”的榜首达五届之久。

Rust程序的井喷也带来了新的需求：绝大多数Rust语言的使用者将程序的安全性视为编程的第一准则，他们希望Rust语言未来能取得更大的进步，Rust程序的安全性可以再迈上一个阶梯——然而当前的开发形势下，Rust语言不可能在短期内实现显著的飞跃。若开发者对Rust程序的安全性有更高的要求，最合理的解决方案是用自动化测试工具对Rust程序进行检测。然而，作为一种诞生时间尚短且安全性极佳的语言，Rust语言测试工具的领域几乎是空白的。当前，国内的Rust研究领域并未提出相应的测试工具；国外的Rust测试工具在数量和技术方面都很贫乏：这些工具基本建立于成熟测试工具的技术迁移之上。其中，Seer工具和Seahorn工具的作者就是将它们的成熟测试工具作了修改以支持Rust语言，后续的改进也被搁置。

在软件测试领域中，动态符号执行技术和模糊测试技术的结合在C++与Python程序中取得了很好的表现。模糊测试生成大量测试用例触发目标程序中的错误。而动态符号执行技术通过探索程序路径来发现错误,它利用插桩技术获取路径的约束信息，生成和各条程序路径相对应的测试用例。这两种技术在著名的Driller工具中得以结合，并体现出了强大的潜力。然而，目前Rust领域缺少类似的测试工具，本文将针对这个问题展开工作。

本文作出的贡献如下：

（1）介绍了模糊测试和动态符号执行技术的结合于主流程序语言测试领域的应用现状，提出了改进此类技术以测试Rust语言程序的设计思路。

（2）依据前文提出的方法构建了一个 Rust 测试工具，VeriRust。VeriRust，通过 AFL 模糊测试工具大量生成输入，快速寻找程序中的隔间部分。同时利用 LLVM 技术将 Rust 程序翻译为中间语句 IR，在模糊测试工具失效时，应用针对 Rust 语言进行再集成的 KLEE 动态符号执行工具，在中间语句上寻找可以通过复杂检查的输入。

（3）VeriRust 在 Rust 使用频率最高的两个第三方库组成的数据集，serde_json 和 url 上分别进行了实验。实验结果表明，在 Rust 常用的第三方库之上，本文提出的工具优于一系列单纯应用模糊测试技术和动态符号执行技术的工具。由此体现了相当程度的可行性和有效性。并进一步证明，结合了硬件行业中的形式化验证和软件测试中的模糊测试技术的一类方法，可通过 LLVM 中间语言被迁移到 Rust 语言测试领域。应用此类方法开发的工具在 Rust 实战中展现高覆盖率和准确率，后续的开发改进具有相当的潜力。综上所述，本文的工作对 Rust 语言领域实现了具有可贵意义的初级探索。

关键词：

形式化验证，Rust 语言，模糊测试，动态符号执行，底层虚拟机

Abstract

Research on RUST Testing Tools Integrating Fuzzing and Formal Verification

Rust language is a new kind of programming language that takes both safety and efficiency into account. Nearly 70% of computer vulnerabilities come from memory problems, and Rust language almost eliminates such problems. This feature makes Rust language a very suitable tool for the development of a series of system softwares. As the Rust language matures, it has gradually attracted widespread attention in the field of industry: Microsoft has applied it to the development of key components, while hundreds of other well-known domestic and foreign companies, such as Amazon, Google, and Ant Financial, have also begun to take Rust language into practice. To meet the security requirements of users; in addition, the scientific research community has also developed a strong interest in the Rust language: since 2017, Rust language has been on the top of the "Most Popular Programming Language" list on the Stack Overflow forum for five times.

The soar increase of Rust files has also brought new demands: the vast majority of Rust language users regard safety as the first criterion for programming, and they hope that Rust language will make greater progress in the future, and the safety of Rust programs can be further improved- however, given the current circumstances, it is unlikely that Rust language will make a significant leap in the short term. If developers have higher requirements for the security of Rust programs, the most reasonable solution is to use automated testing tools to test their programs. However, as a young language with excellent security, the field of Rust language testing tools is almost empty. At present, the domestic Rust research field has not proposed corresponding test tools; foreign Rust test tools are poor in quantity and technology: these tools are basically built on the technology migration of mature test tools. Among them, the authors of the Seer tool and the Seahorn tool have modified their mature testing tools to support the Rust language.

In the field of software testing, the combination of dynamic symbolic execution technology and fuzzing technology has achieved good performance in C++ and Python programs. Fuzzing generates a large number of test cases that trigger bugs in the target program. The dynamic symbolic execution technology finds errors by exploring the program path. It uses the instrumentation technology to obtain the constraint information of the path, and generates test cases corresponding to each program path. The two technologies are combined in the well-known Driller tool and show great potential. However, there is currently a lack of similar testing tools in Rust In view of this, thus, this article will conduct some preliminary explorations in this field.

The contributions of this paper are as follows:

First, it introduces the current status of the combination of fuzzing and dynamic symbolic execution technology, and proposes a design scheme to improve such technology to test Rust language programs.

Second, it build a Rust testing tool, VeriRust, based on the method proposed above. VeriRust, which generates large amounts of input through the AFL fuzzing tool, quickly finds compartmentalized parts of the program. At the same time, the LLVM technology is used to translate the Rust program into the intermediate statement IR. When the fuzzing tool fails, the KLEE dynamic symbolic execution tool, which is reintegrated for the Rust language, is used to find the input that can pass the complex inspection on the intermediate statement.

Third, experiments were conducted on the datasets composed of the two most frequently used third-party libraries in Rust, serde_json and url. The experimental results show that, on top of the most commonly used third-party libraries in Rust, the tool proposed in this paper can achieve high code coverage and fully detect program errors. This reflects a considerable degree of feasibility, effectiveness and practicality. It is further proved that the tools that combine formal verification in the hardware industry, fuzzing technology in software testing and LLVM intermediate language conversion can be migrated to the field of Rust language testing. And this kind of tool shows high coverage and accuracy in Rust combat, and the subsequent development

value is guaranteed. In summary, the work in this paper has achieved a valuable preliminary exploration of the Rust language field.

Keywords:

Formal Verification, Rust language, Fuzzing, Dynamic symbolic execution, Low Level Virtual Machine

目　录

第 1 章　绪　论

1.1 研究背景及意义

编程语言设计长期受制于两个目标的冲突：安全与效率。由 Mozilla 基金会推出的新兴编程语言——Rust 语言，就确保了二者的兼顾[1]。Rust 语言的安全性通过编译时检查强制执行的一组严格的安全规则得到保障，这种特性使其被主要用于浏览器、操作系统等系统软件的开发[2]。近年来，Rust 语言得到了工业界和科研界的广泛关注，它的安全优势和性能表现促进了其于系统软件中应用的迅速增长。例如，微软公司就将目光放到了 Rust 语言卓越的内存安全能力上，并以此为基础积极探索用 Rust 取代 C++的可能性。微软公司经过大量实验论证后，最终于 2019 年开始正式应用 Rust 语言完成对核心组件的重写。在随后的几年中，Rust 语言得到了国内外 Amazon,Google，蚂蚁金服数百家知名公司的使用[3, 4]。2020 年 12 月 26 日，国内的第一届 Rust 语言大会——RustChinaConf2020 胜利召开，这标志着国内对于 Rust 语言的应用与研究迈上了新的阶梯。

软件测试，是使用人工或自动的手段来测定某个软件系统的过程，其目的在于检验软件是否满足规定的需求，并进一步计算预期结果与实际结果之间的差别。随着软件行业日新月异的发展，软件测试开始过渡到自动化测试脚本的编写，形式化验证即诞生于软件测试行业这样的发展趋势中。形式化验证旨在用数学方法去证明目标系统不存在错误。在操作中，工具编写人员需要分析问题并建立数学模型，规定系统在不同情境下具备的可能及不可能状态，在此之上建立数学规则，完善测试系统。随着客户对安全性能的需求日益急迫，形式化验证而得以蓬勃发展，区块链交易，交通调度，医疗设备开发等领域就存在形式化验证的身影。

作为一款新兴而充满潜力的编程语言,Rust 语言于系统软件开发领域的潜力提升了软件验证行业相关人员对 rust 语言及相关程序的关注。在事关系统软件等计算机命门部分的开发中，应用形式化验证等价格高昂但结果可靠的验证工

具已经成为工业界的常见做法。以上做法对于 Rust 语言同样适用，但由于诞生时间尚短，很多 Rust 测试工具依然局限于复现单个领域的技术。

在动态程序分析领域中，模糊测试技术已被广泛应用于工业界的实际生产中，模糊测试工具生成大量测试用例输入到目标程序中，以图触发目标程序中包含错误的代码[5]。模糊测试技术的盲目性限制了测试案例有效性，这一过程产生大量冗余的测试用例，浪费了计算机资源。在形式化验证领域中，符号执行技术在最近几年的科研发展相对完备，动态符号执行技术采用白盒技术来生成测试用例，它通过符号值的执行，利用插桩技术获取受试程序中的路径约束信息，以此生成输入，从理论上来讲，生成的输入可以遍历 90%以上的程序路径[6]。

以上的描述揭示了模糊测试和动态符号执行有着优势互补的特性，后续的研究也证明了这一想法的可行性，Nick Stephens 等人提出的 Driller 模型就将两种技术结合起来，利用模糊测试技术和动态符号执行模块二者的互补来寻找更深层次的漏洞。[7]这两种技术的结合实践已经证明了实验的有效性，在此基础上，可以尝试将这种结合应用到其他的领域。比如说，针对 Rust 语言的特性进行领域迁移，也将是一个十分有价值的研究方向。

1.2 国内外研究现状。

主流软件验证工具可以按照技术划分为四大领域：符号执行工具(Symbolic execution tools),自动主动验证工具(Auto-active verification tools),推理验证工具（Deductive verification tools）和自动验证工具（Automatic verification tools）。自 2015 年 Rust 语言正式发布后，从事以上四个领域的科研工作者均提出了对应的验证工具。

符号执行工具旨在通过探索程序路径来发现错误，生成具有高控制覆盖率的测试套件。相对于其他领域的工具，符号执行工具通常不能保证隔绝错误，但它们拥有更好的扩展性，在 Rust 新代码库的实验会有更好的表现。David Renshaw 等于 2018 年提出了 Seer 工具，Seer 基于 miri 工具，使用 z3 作为求解器后端，并增加了对符号执行技术的支持。Seer 以符号形式表示 Rust 程序的输入，并对其保持一组约束。当 Seer 到达程序中的一个分支点时，它会调用其求解器后端来计算在给定当前约束的情况下哪些延续是可能的。Marcus Lindner 等于 2020

年提出了 Cargo-KLEE 模型[8]，它使用著名的 KLEE 验证工具来验证 Rust 程序的方法[9]。为了解决困扰 KLEE 工具的路径爆炸问题，Cargo-KLEE 采用了契约驱动的开发方法来实现模块化验证。这种改进让路径爆炸问题仅涉及每个函数的路径数，而不是整个程序的路径数。Cargo-KLEE 在相当程度上缓和了路径爆炸，但该问题依然限制着模型的效率。

自动主动验证工具可帮助测试人员额外验证代码的一些关键属性，例如数据结构非变量（Data structure invariants）、函数结果等。为了实现以上的额外功能，测试人员需要向代码中添加函数协定（函数的前置/后置条件）、循环非变量、类型非变量等信息来辅助该工具。2020 年，Vytautas Astrauskas 等提出了 Prusti 插件，这也是当前这一领域唯一完备的 Rust 验证工具[10]。Prusti 被嵌入到 Rustc 编译器上，利用 Rust 的类型系统与基于权限的推理方法来完成检验。具体来说，该工具将 Rust 转换为中间验证语言 Viper 的代码，并将任何错误从 Viper 转换回 Rust。最终得到遵循 Rust 类型系统的代码。

推理验证工具可显示诸如完整功能正确性（Full functional correctness）之类的内容：输出正是它们应有的样子。推理验证工具通常会生成一组验证条件，然后使用交互式定理证明器对其进行证明。2018 年，Derek Dreyer 等提出 RustBelt 工具来验证不安全的 Rust 代码，RustBelt 实际上并不验证 Rust 代码：测试人员需要手动将 Rust 代码转录为 λ-Rust，然后使用 RustBelt 使用 IRIS 和 Coq 定理证明器来验证该代码。

自动验证工具常被应用于应对运行异常不存在（Absence of Run-Time Exception，AoRTE）这一情况。运行错误通常包括以下情况：未除以零（No division by zero），无整数溢出（No integer overflow），内存安全问题等等。自动验证工具的优势在于，工具开发人员不必编写专有的测试规范。开发人员的工作量缩减到构建验证工具，以及向代码中注入额外的断言。这种特性使得测试人员不需要大量培训即可使用这些工具，这大大增强了这些工具的适用性，这为他们作为组件被整合到程序的整体开发中带来了更大的可能性。

应用于 Rust 语言的自动验证工具可按照技术类别继续划分为以下四种：

1. 指示器与静态分析（Linters and static analyses）

在日常代码的编写中，数量庞大的初级语法错误都会被编译器捕获并报错，这些可以于编辑器中提供即时反馈的内嵌工具——指示器经常会被忽视，但它们确实展现了不可或缺的实用性。当构建代码时，指示器可以内置到 CI 流程中，任何检查代码的人都可以很便捷地验证该段代码是否发生了错误。Rust-analyzer 一类的常用工具就属于这个范畴。

2. 工具化解释器（Instrumented interpreters），工具化解释器通过执行程序来进行检测，这种执行代码的方式与正常运行的方式基本相同，只是耗时普遍会更长。此类工具的主要代表是等提出的 Miri 工具，它整合了 Stacked-Borrows aliasing model，在运行时会主要检查未定义行为 (Undefined Behaviour)。

3.动态符号执行工具(Dynamic symbolic execution，DSE)，动态符号执行是一种易使用的工具，它们可以被认为是常规测试和模糊测试强大的变体——这些技术已经被很多测试工具开发人员充分掌握。与工具化解释器的工作原理类似，DSE 工具通过探索程序中的路径来执行错误识别。不同之处在于，用户无需自行创建测试用例，DSE 会替用户代劳，自动寻找通过程序各级检查的可行路径，以及可能遵循同一路径的所有可能值的原因，相对于工具化解释器，动态符号执行工具的优势在于所需的时间和计算资源更少。

DSE 工具的性能主要受两方面限制：路径爆炸和循环问题。路径爆炸带来了这样的问题：通过程序的路径数量会以指数级扩张，测试人员在现有技术制约下不可能检查通过程序的所有可能的路径。循环问题指，DSE 只会检查循环的有限次的可能情况，通常会忽略多次乃至无限循环的程序路径可能。

截至 2022 年 2 月，该领域唯二拥有可行版本的工具是 RVT 工具和前文提及的 cargo-Rust 工具。作为符号执行的衍生技术的产物，cargo-Rust 工具也可被划分到动态符号执行工具中。GitHub 用户 alastairreid 等提出的 RVT 工具可以将 Rust 代码转换为 LLVM-IR 并使用 KLEE 等 DSE 工具进行检查，其优秀的扩展性允许程序和模糊工作结合。

4.边界化模型检查器(Bounded model checkers，BMC)

DSE 工具倾向于以深度优先的方式探索可行路径，而 BMC 工具则采用宽度优先的方式探索可行路径。以代码的循环部分为例，在考虑第二次迭代的执行路径

前，BMC 工具会花费更多资源继续探索循环的第一次迭代中可能出现的所有可执行路径。

BMC 工具为代码验证结果提供了更好的保证：循环部分通过了前 k 次迭代的验证，不意味着代码就不会出错，但这个参数 k 越大，测试人员对代码的信心就越大。而对于一个有界循环来说，探索了足够的迭代就等同于证明循环的正确性。

BMC 工具受到问题路径的困扰。若一个条件语句的一个分支具有该 BMC 工具不支持的一些特性(如外部调用或内联汇编代码等)，BMC 工具通常就会根本无法推理出条件。相反，DSE 可以通过跳过问题分支转而探索另一个分支。

BMC 领域的 Rust 测试工具包括以下内容：Arie Gurfinkel 等人于 2015 年提出的 SeaHorn 工具在后续的改进中可以支持 Rust 语言的编译，它是基于 LLVM 中间表示的验证工具[11]。它的缺点在于很少或根本不支持动态链接数据结构的测试。Montgomery Carter 等人于 2016 年提出的 SMACK 工具将 MIR 转换为 Microsoft Research 的 Boogie 中间验证语言，提供对一系列不同验证工具的访问。上述的两种工具无法支持 SDLB 库一类的 Rust 基础库。也就是说，它们只能处理无法涉及到 Rust 基础类库一类的核心内容。

最后需要强调的是，目前没有任何验证工具能对 Rust 语言作完整验证，绝大多数的验证工具都无法处理操作系统设备驱动一类的复杂程序。此外，不安全的代码、闭包、标准库等问题也需要后续的 Rust 语言验证工具一一克服。若想开发出与主流 C 语言验证工具相媲美的成果，Rust 语言验证工具的开发者们仍有很长的路要走。

1.3 论文组织结构

本文对动态符号执行技术与模糊测试技术的结合领域进行了研究，在研究了大量文献和工业界新工具的基础上，提出了将二者结合的技术利用 LLVM 中间语言作为桥梁，迁移到 Rust 语言测试领域的想法，并根据这种设想开发了一种新的 Rust 语言验证工具——VeriRust。

本文主要分为四个章节，具体结构如下：

第一章，绪论。该部分主要介绍本文提出的的课题项目的背景，介绍国外科研人员对 Rust 测试的研究进展，在此之上对该课题的意义进行简单的分析与阐述，并于最后介绍该论文的主要内容和整体结构。

第二章，背景知识。该部分主要介绍 Rust 语言的特性及语法，模糊测试的定义，模糊工具的分类和使用步骤，LLVM 的基本介绍以及符号执行及动态符号执行技术的流程和原理。

第三章，VeriRust 工具的模型及算法实现，该章节按照 VeriRust 内部各组件来介绍本文提出的模型。依次介绍了模糊测试引擎，LLVM 预处理，和动态符号执行引擎的原理和具体实现。

第四章，针对 VeriRust 工具，本文利用 Rust 语言中最常用的两种板条箱组成的数据集进行了基本测试，并在此之上进行验证模糊测试模块和动态符号执行模块功效的消融实验。以证明工具的可行性和有效性

第五章，总结与展望。该部分内容总结本文工作，阐述 VeriRust 工具实现的功能，并在此基础上展望对 VeriRust 工具未来的可行研究方向。

第 2 章 背景知识

2.1 Rust 语言

Rust 语言，国内译为“锈语言”，它是一种与 C++类似的编程语言。Mozilla 基金会员工 Graydon Hoare 于 2006 年创建了 Rust 项目，2009 年后 Mozilla 基金会开始正式赞助该项目的开发。Mozilla 基金会支持该语言开发的目的是创造一种 C 和 C++语言的“后继语言”，可以在语言级别上充分利用硬件的多核计算功能，实现最大化的性能提升，并在此之上克服 C/C++中长久存在的一些安全问题，比如内存泄漏风险和未自动化的内存管理功能等。

第一个有版本号的 Rust 编译器于 2012 年 1 月发布。这之的 2015 年 5 月，Rust 语言第一个稳定版本，Rust 1.0，由 Mozilla 正式公布。截止到 2022 年 2 月，Rust 语言已经更新到 1.58.0 版本。这些发展历史证明 Rust 语言已经足够成熟[19]。

根据 Rust 官网的介绍，Rust 语言具有三大吸引开发人员的优势：高性能、强可靠性、高生产力。而作为一款对标 C/C++语言的新语言，Rust 语言的优势体现在以下技术细节中：

1. Rust 消除数据竞争，能实现广泛的线程安全。当前的 Rust 原生库中已经包含大量的组件，辅助实现并发/并行化 Rust 程序。包括但不限于数据并行、线程池等。针对容易忽视安全性的第三方库组件，Rust 也会以要求原生库的标准，一致地确保所有代码和数据的线程安全。在 Rust 语言中，线程安全要求统治着程序的方方面面。

2. Rust 语言支持异步高并发编程。Rust 语言支持 async/await 异步编程模型。该模型维护一个叫 Future 的概念，它允许开发者设定一个尚未得出的值，并在彻底解决真实值前对其进行各种操作。Rust 语言用执行器组件去轮询 Future，用反应器控制所有的 I/O，这赋予了用户更强的控制力。

3. Rust 支持安全的编译期计算。Rust 可以支持类似于 C++那样的编译期常量求值，但使用中的表现略逊于 C++，因为 Rust 语言对于安全性的要求限制了该功能的自由使用。

2.1.1 Rust 语言的示例

程序 2.1 展示了一段简单的 Rust 代码，它显示了 Rust 语言的编程风格

程序 2.1：Rust 语言示例

```
1  fn main()
2     let v = vec![1,2,3,4,5]; //vec 定义了一个动态增长的数组
3     let mut s = &v[..]    //let 关键字声明新变量
4  loop{                  //loop 语句是 Rust 提供的无限循环语句。
5      match s{
6          [a,rest @ .,b]=>{
7            println!("{},{}",a,b);
8            s = rest;
9           }
10          [a]=> {
11            println!("{",a);
12            break;
13          }
14         []=> break,
15       }
16     }
17 }
```

此段代码用到了模式匹配功能，实现从一个数组首尾同时开始遍历的功能。

2.2 模糊测试

模糊测试（Fuzzing）是一种应用广泛的软件测试技术。应用此类技术的工具以自动或半自动生成的大量随机数据为测试依据，输入到待检测的程序中，监视程序异常，如程序崩溃，断言(Assertion)错误等。模糊测试常常用于检测软件或计算机系统的安全漏洞，而模糊工具（Fuzzer）自动化了这一过程的大部分，这类工具能吸收初始程序知识，并报告任何特殊的程序状态。

不犯错的软件是不存在的。对于一个待测试程序来说，发现漏洞远比未发现错误更有意义。模糊测试等软件测试技术的最终目的就是尽力发现更多的潜在程序漏洞。

2.2.1 模糊工具的分类

朴素模糊工具生成完全随机的输入，提供给程序进行测试。朴素模糊工具相当容易实现，但它们很难在有效时间内到达感兴趣的程序状态。

现代模糊工具建立在朴素模糊工具之上，它可以分成三种主要类型：基于突变、基于生成和进化模糊工具。

基于变异的模糊工具盲目地突变或操作提供的输入。模糊工具不知道预期的输入格式或规范，他们也不能明智地选择突变。模糊工具以符合规范的测试用例为种子，应用一系列变异策略来微调种子，这些变异策略包括但不限于字节删除，随机比特反转、字节复制。Michael Eddington 等开发的 Peach 工具属于这个范畴，它也同时整合了基于变异和基于生成的模糊工具。

基于生成的模糊工具通过规范获取关于预期的输入格式或协议的信息。这类模糊工具会根据这些规范生成或制作输入。除了上文提到的 Peach，GitHub 创作者 Ryan 等贡献的 Sulley 工具也属于这个领域，它是一个针对 Python 开发的模糊工具框架，被用于文件传输协议、网络协议的测试。

进化模糊工具是当前最先进的模糊工具，此类工具建立在基于突变的模糊工具上，它会选择部分输入，而不是所有输入进行突变。一般来说，进化模糊工具要么突变一个输入，要么选择两个或多个输入并执行交叉，结合所选输入的组成部分来产生一个新的输入。应用其他随机化技术也是可行的。具体来说，进化模糊工具会观察每个各阶段输入对程序行为的影响，评估输入的性能，选择要进行的随机化输入，改变测试行为。例如，一些模糊工具会依据测试用例执行时的代码覆盖率进行排序，选择得分最高的输入进行突变。最具代表性的进化模糊工具是 AFL，honggfuzz 和 libFuzzer[13,14,16]。它们皆是由谷歌开发的。其中，AFL 工具是模糊测试领域最具开创性的模型——它改变了模糊测试领域的发展道路[15]。在 AFL 工具得以广泛应用之前，模糊测试技术的应用十分有限，作为一种依靠大量生成测试用例试错的技术，模糊测试技术的低效显而易见。AFL 工具的开创性

在于它引进了覆盖反馈的技术：评测生成测试用例的质量不能完全依靠重复执行来完成，覆盖反馈技术简化了这一流程,在测试用例进行路径遍历时，AFL 工具会通过插桩代码获知当前测试用例是否实现了更多的分支覆盖。当覆盖率不增长，那么该测试用例就会被抛弃。由此，有用的测试用例得到遴选。

根据运行时从被测程序中获取信息的多少，模糊测试工具又可以被分为黑盒、白盒和灰盒三类。获取的信息可以是路径覆盖信息、程序内存使用情况、CPU 利用率或其他对指导测试用例生成有帮助的信息。

传统的黑盒模糊测试也称为“黑盒随机测试”，黑盒模糊测试通常采用基于变异的方法生成测试用例，但不需要从被测软件中获取任何运行时信息，只根据预设的变异策略对给定的种子测试用例进行变异来产生新的测试用例。随着技术的发展，黑盒模糊测试也逐渐开始使用基于生成的策略，如利用预设的语法规则或输入格式相关的知识来产生有效的输入数据。

白盒模糊测试技术是黑盒模糊测试的改进方法。为了克服黑盒模糊测试显著存在的盲目性，白盒模糊测试会探究被测软件内部细节，获取代码逻辑知识，以达到更好的代码覆盖率。白盒模糊测试常用的技术包括动态符号执行和启发式搜索算法，通过这些技术生成的测试用例可以覆盖绝大多数程序路径。然而，路径生成的复杂计算制约了白盒模糊测试的测试规模。

灰盒模糊测试结合了黑盒和白盒模糊测试的设计理念，它借助从被测软件中获取的少量信息来指导测试用例生成，以高效地发现程序中存在的错误。灰盒模糊测试常使用代码插桩技术获取有关程序内部的信息，插桩指向被测程序插入一些代码，在运行时，这些代码在不干扰程序正常运作的前提下，额外获取当前路径一类的程序信息，这些信息将为测试用例的生成作出指引[17]。

2.2.2 模糊测试工具的工作阶段

预先工作。这里的工作主要指代程序知识。在开始模糊测试过程之前，用户会将程序知识整合到模糊工具中。这些知识包括但不限于感兴趣的程序状态的构成，需要探索的输入接口等。基于突变的模糊工具需要额外程序知识整合成的输入语料库，或是以一组程序输入的方式指定输入接口。基于生成的模糊工具需要

以输入规范的形式获取额外的程序知识，例如文件格式或协议描述。进化模糊工具，要求与基于突变的模糊工具基本一致，需要语料库以获取额外的程序知识。

生成输入。这个阶段的目标就是创建输入，以探索新的程序状态。在该阶段，模糊工具利用已知的程序知识生成输入，并通过接口提供给程序[18]。模糊工具旨在生成最相关的输入，减轻下一阶段中进行筛选的工作量。由此可见，生成的输入实际上只有一部分会被真正送入程序中。

选择输入。该阶段中，模糊工具对上一阶段的输入作出筛选，并将筛选结果发送到程序。一个模糊工具的目标，理论上说，是高效地测试新的程序路径，但在实践中，生成的大多数输入最终都在执行少数相同的路径。为了解决这一问题，模糊工具必须有效地使用其输入语料库，并最小化发现新的程序路径所花费的计算量。输入测试调度（Input test scheduling），又称种子选择（seed selection），被用作对抗前文提及的输入倾向，以探索有限的程序路径。输入测试调度对输入和输入顺序作排序及选择，预测哪些新的输入最有可能导致新的程序状态。对于漏洞评估，测试调度通常会筛选输入，以最大化发现错误的数量。输入测试调度是有效地探索大的或无限的输入搜索空间的关键，但为特定的程序和模糊工具寻找正确的调度策略仍然是一个研究挑战。FuzzSim 等工具在过程的许多迭代中使用输入性能信息快速比较输入选择策略[19]。

监控程序。在该阶段中，模糊工具提供程序选择的输入，并监控程序以识别感兴趣的程序状态。“感兴趣的程序状态”代表着程序在该状态下表现出一个特定的行为。在大多数情况下，这意味着程序出现崩溃。

评估输入。在该阶段中，模糊工具评估输入的表现。许多模糊工具使用代码覆盖率（code coverage）来估算输入的效果。如果生成的输入导致代码的新部分得以执行，那么该输入就增加了代码覆盖率。libFuzzer 工具使用数据覆盖率（data coverage）作为度量，这种测量方式聚焦于和过往探索作比较，若出现新的数据值，则证明了工具的有效性。其他的一些模糊工具使用错误发现（bug discovery）作为评价指标，在这种评价标准下，工具的评分与程序错误的发现呈正比关系[20]。

图 2.1 总结了上文中模糊工具的工作流程。

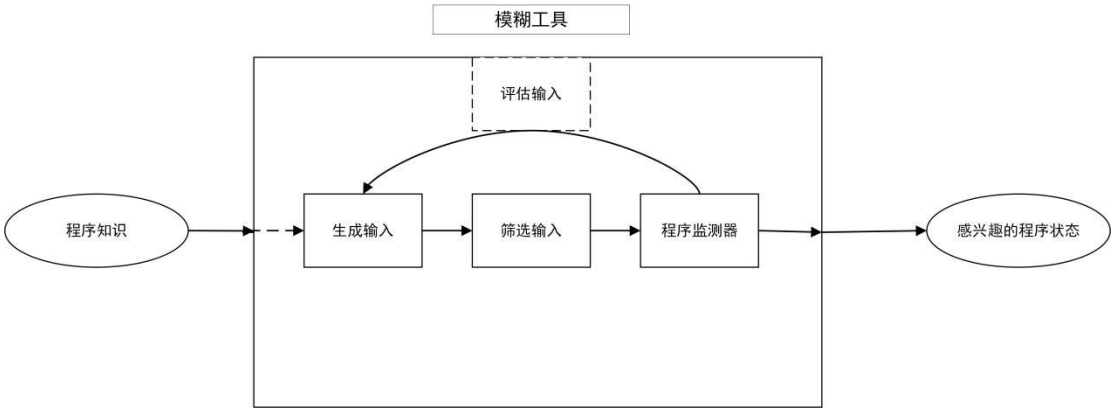

图 2.1　模糊工具的工作流程

2.3 LLVM

LLVM(Low Level Virtual Machine),中文译作底层虚拟机[21]。从当前的发展上来看，LLVM 项目早已脱离了虚拟机的范畴，现在的 LLVM 项目是模块化和可重用的编译器和工具链技术的集合。在实际应用中，LLVM 是很多编译器的基石。C 语言常用的 Clang 编译器就隶属 LLVM 的子项目，Rust 语言的专用编译器 Rustc 同样建立在 LLVM 的强大功能上[22]。

整体上看，LLVM 的框架与传统的编译器极为相似，它们都建立在三阶段架构上：前端——优化器——后端。传统编译器中，前端分析源代码的词法、句法、语义,排除错误,建立起描述该段程序语言的抽象语法树（Abstract Syntax Tree, AST），经过一系列转换获取新的表示。在优化器中，抽象语法树可以进一步被转换优化，并于后端部分转化为字节码的表示方式。

传统优化器非常易用，但它有一个显著的缺陷：重用能力较差。若想得到不同的字节码，需要针对该种语言单独设计前端，优化器和后端。随着高级编程语言种类的扩增，这种低效的方式被 LLVM 提出的方案取代：在 LLVM 的前端中，不同类型的高级编程语言的抽象语法树都将被统一转化为中间语言 IR,优化器对统一的 IR 进行优化操作,后端则将优化完成的 IR 根据需求转化为不同类型的机器码[23]。

图 2.2 展示了基于前文描述的设计思路下，LLVM 工具呈现的框架。

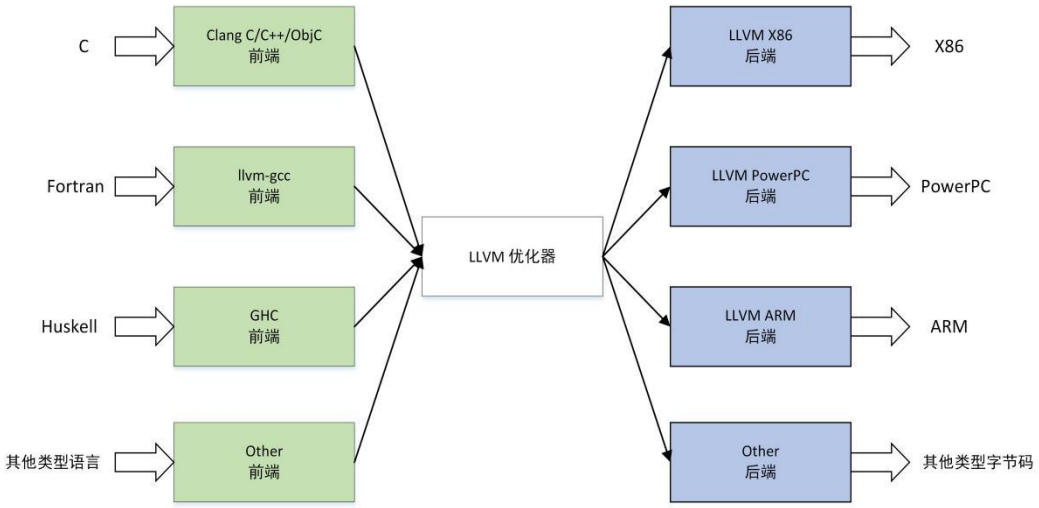

图 2.2　LLVM 改进技术下的三段框架

图 2.2 展示了 LLVM 工具对源代码进行编译的流程：。

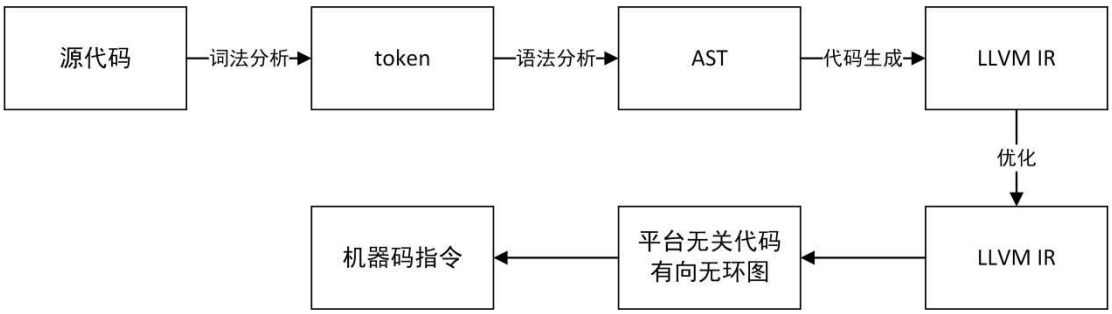

图 2.3　LLVM 编译流程

2.4 动态符号执行

符号执行概念的提出已有四十年的历史。近年来，约束求解等辅助技术的迅速发展使得这一理论得以被广泛应用到实际项目中[24]。

符号执行依靠生成测试用例执行测试，存在的每个用例需要和程序中的各条路径能一一对应。生成具体的测试用例的能力是评判符号执行技术优劣的重要依据，一个优秀的符号执行工具能做到在最短时间内生成大量输入，以此发现最多的对应的程序路径，并检查各路径上出现的程序错误，包括断言违背、安全漏洞、内存崩溃等[25,26]。生成的测试用例将被程序保存，也可——重现程序错误时的状态，并用于测试人员的后续工作在此之上分析与改进程序。

2.4.1 符号执行原理

符号执行，也称静态符号执行。在正常的程序使用中，软件以程序中的变量的具体值作为输入，完成一次具体执行。而在符号执行一类的测试流程中，软件以符号作为输入，将程序变量的值表示为以符号输入值为基底的符号表达式，并将程序输出值表示为以符号输入值为参数的函数[27,28]。

下述是符号执行的典型过程：沿着每个执行路径，符号执行会维护一个程序计数器，用于标识要执行的下一个语句；一个符号内存状态 σ，它指代从程序变量到符号表达式的映射；以及一个符号路径约束PC，它通常是一个无量词的一阶布尔公式。以上这些信息的集合就是一个符号程序状态。

在符号执行开始时，σ 被初始化为一个空映射，PC初始值被重置为$true$。随着符号执行过程的深入，σ 和PC都将被更新：经过赋值语句时，σ 通过将变量映射到符号表达式来更新，经过条件语句时，PC得以累积信息。

这里可以以实例的方式来说明：假设当前路径经过这样一个条件语句$if(c)\ \ S_1\ else\ S_2$。并且，在到达这个语句前，传入的执行路径已经经行了$n-1$个条件语句。那么当前的PC已经包含了执行到这个语句时的所有符号路径约束，PC可以表示如公式 2-1：

$$PC = (pc_1 \wedge pc_2 \wedge \ldots \wedge pc_n) \qquad \cdots\cdots\cdots\cdots\cdots\cdots (2\text{-}1)$$

其中PC是沿此路径的条件语句的真或假分支的符号约束。要执行条件语句，另一个符号执行会被衍生，以并行地经行两个分支。到此阶段时，条件c在当前符号内存状态下被计算为$\sigma(c)$，PC被更新到$PC \wedge \sigma(c)$，以跟踪真分支的执行。同时，一个新的路径约束$PC' = PC \wedge \neg\sigma(c)$将被创建，以跟踪假分支的执行。之后，一个约束求解器被调用来进行PC和PC'的可满足性检验，如可满足模理论(Satisfiability Modulo Theory, SMT)求解器[29]。如果PC是可满足的，那么符号执行将在当前实例的上下文中继续执行。同样，如果PC'是可满足的，则生成一个具有符号状态 σ 和路径约束PC'的新的符号执行实例。但如果以上两个条件PC和PC'都无法满足，符号执行就将在相应的执行路径终止。

最终，当程序的执行路径的符号执行达到结束或出现错误时，约束求解器将被调用，以求解PC并生成具体的输入值[30,31]。在理想条件下，约束编码和求解都

是完美的，发散不会发生，最终，在正常执行过程中，测试过程生成的输入向量会令原始版本的受测程序遍历与符号执行流程相同的路径[32]。

2.4.2 动态符号执行原理

当遇到包含公式的PC时，通常的符号执行技术就显得力不从心，这推动了动态符号执行技术的诞生。动态符号执行是符号执行技术的一种变体，通过使用来自具体执行的动态信息来辅助应对更复杂的约束。动态符号执行有两个主要变体，即并行测试（Concolic testing）和执行生成测试(Execution-Generated Testing，EGT)[33]。在并行测试中，受试软件同时执行具体值和符号值。它维护一个具体的内存状态，除了PC和符号内存状态之外，还将所有变量映射到它们的具体值。并行测试从在随机或任意输入向量上具体执行程序开始，并沿着执行路径执行符号执行以生成PC。假设程序执行发生在执行路径$p = p_1, p_2 \ldots, p_{n,}$上，其中每个 p_i 对应于在第i个条件语句处所取的真分支或假分支与执行与符号约束。公式 2-1 的定义中，$PC = pc_1 \wedge pc_2 \wedge \ldots pc_n$。为了向不同的路径进行探索，程序下一步要生成新的输入数据，所以，并行测试部分会选择一个语句处的约束pc_j，其中 $1 \le j \le n$。新的路径约束PC可以用公式 2-2 表示。

$$PC = \left(pc_1 \wedge pc_2 \wedge \ldots \wedge pc_{j-1}\right) \wedge \neg . pc_j \qquad \cdots\cdots\cdots\cdots (2\text{-}2)$$

接下来，测试工具用可满足性模理论约束求解器提取PC的具体输入向量。在使用生成的输入向量运行程序后，本次的执行会沿着测试时的具体路径$p' = p_1, \ldots, p_{j-1}\overline{p_j} \ldots,$。程序当前所有执行路径将形成该程序的执行树，程序执行树的内部节点代表着程序的分支点，也就是条件语句[34]。执行树的一个分支则代表着迄今为止探索的执行子树，其中包括过往实验中所有的执行路径和覆盖的分支。在接下来的迭代中，并行测试会从执行树中选择一个分支，以生成下一个输入向量，引导测试的探索方向[35,36]。图 2.4 展示了一个执行树的实例。

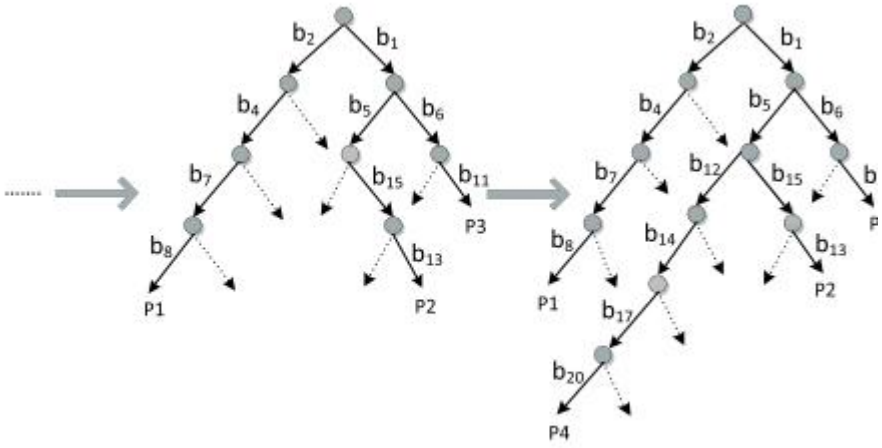

图 2.4　执行树的实例

如果由所选分支的否定所生成的符号路径约束无法被满足，DSE 将选择另一个分支进行探索。由以上描述可以得知，并行测试中的每次迭代都包括分支选择、约束求解和受试程序的并行运行[37]。上述并行测试模式的通常流程如图 2.5 所示。

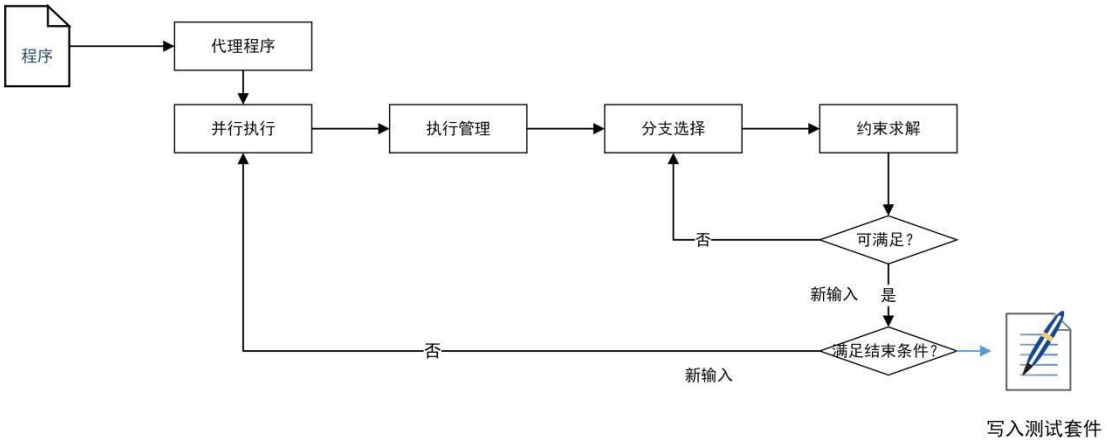

图 2.5　并行测试模式的通常流程

约束求解器很难有效并精确地处理不透明的库调用或高资源消耗的算法，在这种境况中，利用并行测试中的具体值进行替换可以简化PC。这种替换和简化很可能会引入不完整性，但它能为并行测试生成输入值。

执行生成测试方法在 EXE 和 KLEE 模型中被采用[38, 39, 40]。它的具体执行方式与传统的符号执行类似，但它不维护符号变量的具体值。在这种方法中，如果一些操作数是符号的，它们的一个可能的具体值可以通过求解当前的PC来获得一个令人满意的赋值。为了记录程序中所探索部分的信息，执行生成测试维护了一个

类似的二进制符号执行树，内部节点同样代表程序中的条件语句，而叶子结点代表着当前状态。

2.5 本章小结

本章主要介绍在研究过程中应用的技术知识。其中，在 2.1 部分，本章介绍了 Rust 语言的基本情况，它的特性，和一些基本的语法；在 2.2 部分，本章介绍了模糊测试技术的定义，当前模糊测试工具的分类，和测试工具执行的流程；2.3 部分对 LLVM 技术做了简单介绍，重点强调了它的流程；在 2.4 部分，本章介绍了动态符号执行，它是一种用来描述组合约束问题的约束表示语言，其中，2.4.1 部分详细介绍了符号执行技术的流程及公式，2.4.2 部分详细介绍了动态符号执行技术的流程及公式;最后,在 2.5 部分对本章的主要内容进行总结描述。

第 3 章　VeriRust:混合形式的 Rust 形式化验证工具

现有的 Rust 测试工具基本来自于对其他语言的成熟工具的技术迁移，在符号执行领域，基于 KLEE 改进的 Rust 测试工具取得了成功，然而将符号执行工具与其他技术结合的后续工具尚未得到提出。VeriRust 工具的贡献就在于它填补了 Rust 语言领域的空白，实现了可贵的初步探索：它成功地将以 Driller 代表的结合工具的技术迁移到 Rust 领域，并在覆盖率和准确率取得了可观的效果。VeriRust 工具克服了 Cargo—KLEE 等符号执行工具无法处理大规模程序的问题，可以在处理多个类库的同时不受状态空间爆炸的影响。VeriRust 工具的成功进一步证明，以 LLVM 中间语言作为桥梁，形式化验证和模糊测试技术的结合工具可被迁移到 Rust 语言测试领域，

3.1 VeriRust 工具的整体架构

VeriRust 的设计理念来自对输入的分析：程序的用户输入可以被人工划分为两种不同类型，即一般输入和特殊输入。

（1）**一般输入**表示普遍而通用的一种输入，这些输入基本不会影响程序的执行流程。

（2）**特殊输入**表示少数几个可能的输入值。这些少数派输入能改变程序执行流程。

VeriRust 工具对后一种输入类型进行检查，将应用程序分成了多个部分。执行流通过对特定输入的检查在各个部分之间移动，在单独的一部分中，应用程序处理一般输入。

VeriRust 工具通过结合若干部件得以实现其功能。下文将介绍各部分组件的功能概述，并在后续部分详细阐述。

Rust 解析模块。在执行模糊测试的同时，程序会通过 LLVM 将 Rust 的 bitcode 文件，也称 IR（中间语句）进一步处理，执行比较 Rust 版本，提出 Rust 库等操作，以在后续过程中喂给动态符号执行模块。

模糊测试模块。用户调用 VeriRust 时，模糊测试引擎随即被调用。模糊测试引擎利用测试用例加以调试，加速初始模糊步骤。调试完成后，模糊测试引擎

从应用程序的第一个部分块展开探索，当它检查到了一个具有复杂判断逻辑的部分，模糊测试引擎就会进入“卡死”状态，模糊测试部分无法继续搜索程序中的新路径，工具无法对剩余的代码进行覆盖处理。

动态符号执行模块。当模糊测试引擎卡死时，VeriRust 调用它的选择性动态符号执行组件。该组件将分析应用程序，并沿用先前的模糊测试步骤所发现的唯一输入来预先约束输入，以防止路径爆炸。在跟踪模糊工具发现的输入后，动态符号执行组件将调用约束求解（constraint-solving）引擎，识别将强制执行到以前未探索的路径下的输入。如果模糊引擎在卡住之前覆盖了之前的隔间，这些路径表示执行流进入新的隔间。

协调管理模块。一旦动态符号执行组件识别出了新的输入，它们就会被传递回模糊测试组件，模糊测试组件会以输入为种子，产生新的变异，从而继续模糊化测试新的部分。工具将在模糊测试部分和并行执行部分之间循环，直到发现感兴趣的输入——这通常意味着发生应用程序崩溃。

VeriRust 的整体架构如图 3.1 所示：

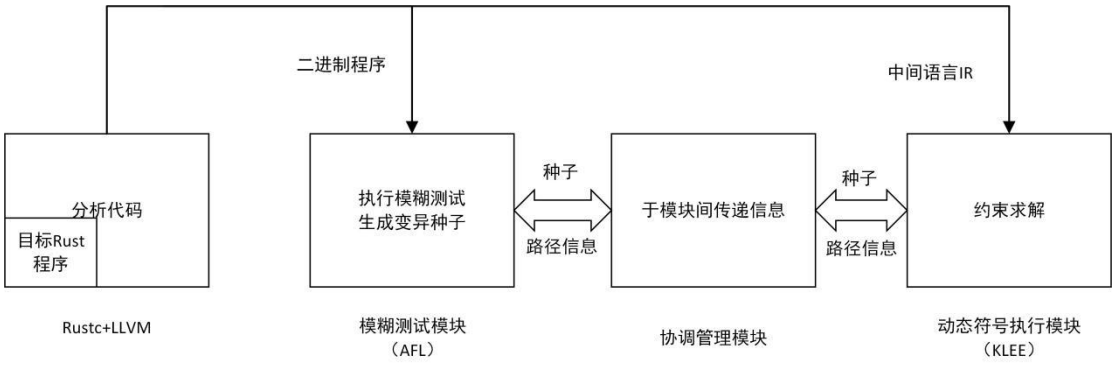

图 3.1 VeriRust 顶层架构图

VeriRust 的整体流程如调度图 3.2 所示：BC 格式的 Rust 二进制文件被输入到工具中，并交由模糊测试模块和 LLVM 框架并行处理；模糊测试模块将执行经典的模糊测试，若超时仍未发现路径错误，可视为异常情况。当前的路径状况将作为语料库保存，动态符号执行模块将于此语料库的状态之上开始执行，实现探索路径的工作。若发现错误，模糊测试模块将更新路径状态，并于此路径状态上继续执行测试工作。

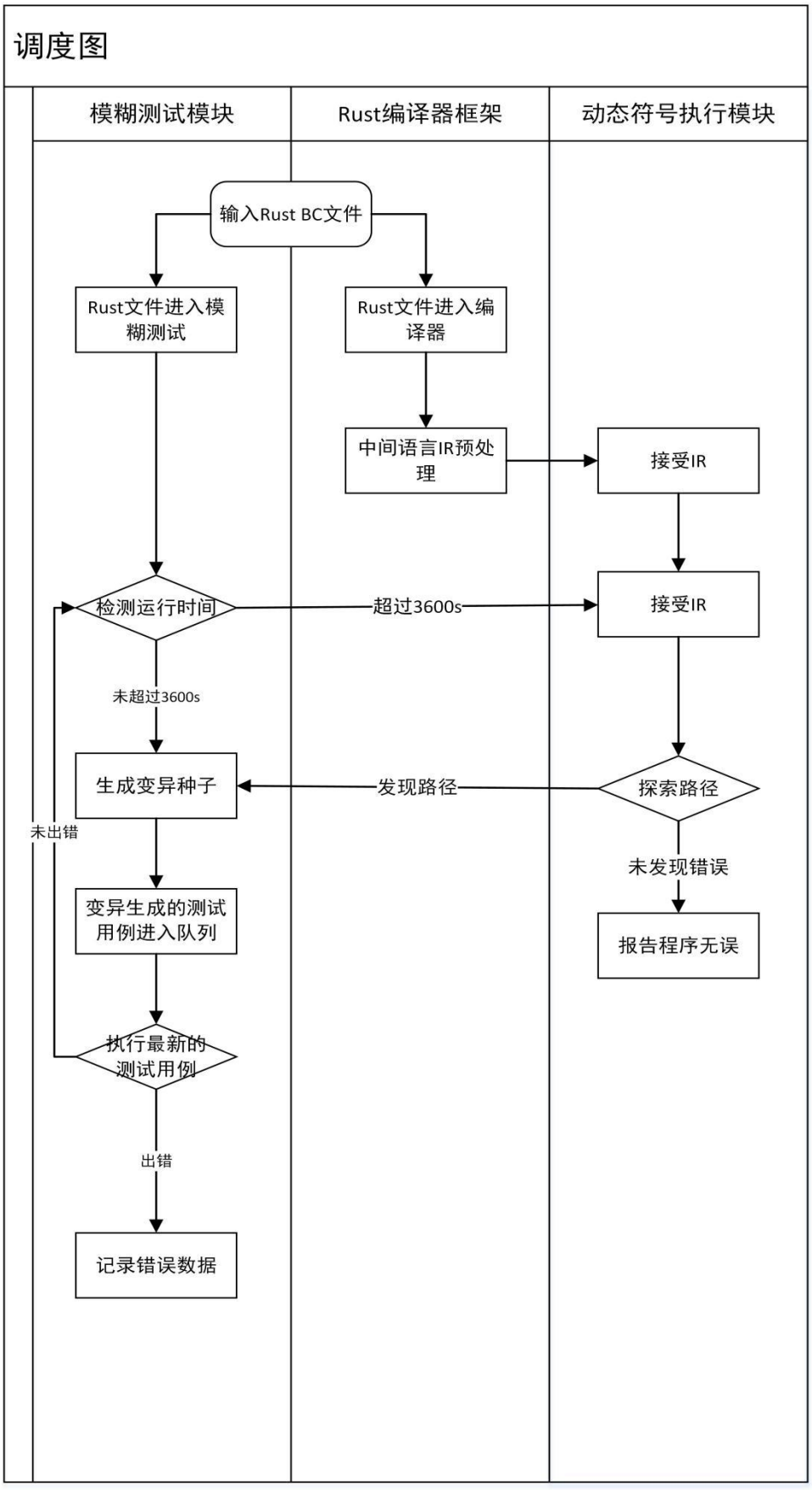

图 3.2 VeriRust 的调度图

3.2 Rust 代码转化部分

3.2.1 Rust 语言转化的实现

在 C/C++语言的测试中，测试人员利用 Clang 编译器将 C 原生代码转化为 IR 语言，将生成的机器码喂给模糊工具。类似地，VeriRust 的测试将基于 Rustc 编译器进行机器码的生成。使用者提供二进制程序代码，LLVM 将其翻译为独立的中间语言（IR），它可以进一步转换为二进制机器码，并依靠后端程序获得目标语言的汇编语言。从细节上看，编译器先将 Rust 源代码编译为 HIR（高级 IR）；HIR 转换为 MIR（中级 IR）；MIR 转换为 LLVM IR7（低级虚拟机 IR）；最后，它将 LLVM IR 编译为机器代码并链接结果[22]。

在 VeriRust 的 LLVM 转化部分中，实际需要完成的功能包括以下部分

1. 由于 Rust 语言正处于旺盛的更新迭代周期中，Rust std 库的函数也会发生变化，因此，需要将 BC 文件的 Rust 版本与当前测试工具的兼容版本对应。

2. 提取目标程序中的 std 库，并于重编译后再次整合。

此外，当使用像 KLEE 这样的基于 LLVM 的工具时，为了使 KLEE 能在 Rust 语言上平稳执行，程序中的一些具体设定需要对应地作出修改，包括：

1. 一些配置标志将被传递给 cargo，以确保 Rustc 正确配置 verification-annotations 板条箱。

2. 使用 kcachegrind 来查找 Rust IR 中的验证瓶颈

3. 为了支持命令行参数，预处理器“rvt-patch-llvm”会在 main 开始时调用初始化程序。

3.3 模糊测试部分

模糊测试技术以白盒的形式测试程序：它从外部监视程序的状态，并使用大量输入来测试程序，利用一系列的反复过程来检验应用程序崩溃的发生。

3.3.1 AFL 工具简介

本文提出的工具的模糊测试模块的设计，承接于工业界流行的 American Fuzzy Lop 工具，也称 AFL 工具。AFL 工具是一个专注于安全的暴力模糊工具，

可用于编译时检测模式或传统的盲模糊器模式。AFL 工具依据代码覆盖率进行排序和评估，覆盖率通过计算适应度函数得到结果。

AFL 工具的检测可以在编译时完成，也可以通过 QEMU 管理程序完成。执行一次检测时，VeriRust 搭载的 AFL 工具的算法流程如下：

1. 加载用户提供的初始测试用例，将其添加到队列中。

2. 从队列中获取下一个输入文件，修剪测试用例尺寸，使其达到最小尺寸。

3. 使用各种传统的模糊测试策略反复测试、变异文件。

4. 当 3 中的变异文件导致新的状态转换时，记录上述数据。同时将变异输出的新条目添加到队列中。

从队列中获取下一个输入文件，重复上述过程。

基于遗传算法的模糊工具。作为一种基于变异的模糊测试工具，通过遗传算法进行输入生成，根据遗传的规则突变输入。并根据适应度函数对它们进行排序。适应度函数是代码覆盖率的评价方式之一，若输入触发的执行路径与其他输入触发的路径不同，适应度函数会给予更高的评分。

状态转换跟踪。模糊测试工具从其输入中观测到控制流转换的并集，加以追踪，并将其作为源和目标基本块的元组。在遗传算法中，当他们发现新的控制流转换时，该输入将被制成预备的种子，且享有更高的优先级。这种处理的结果是，一个导致应用程序沿不同路径执行的输入将被更优先地视作种子处理。

公开化的循环。对于模糊测试引擎和动态符号执行引擎二者来说，处理循环是都一个复杂的问题。为了减少循环路径空间的大小，按以下流程执行启发式操作：

3.3.2 模糊测试部分的流程

1. 当模糊测试引擎检测到一个路径包含循环迭代时，一个二次计算将被调用，以确定是否应从该路径取种育种。

2. 模糊测试引擎确定该路径中所执行的循环迭代的总数，并调出导致路径进入相同循环的输入的历史数据，将二者进行比较。

3. 上述路径都按照它们的循环迭代计数的对数被放置到“桶”中。遗传算法从每个桶中取出一条路径作为种子。

上述的处理流程最终能将每个循环涉及的路径由N降至$\log N$。

3.3.3 模糊测试部分的限制分析

模糊工具会随机变异输入，而遗传算法指导的模糊工具反过来会突变过去通过二进制文件生成唯一路径的输入，所以它们能够快速发现处理通常输入的不同路径。然而，生成能通过复杂检查的特定输入成为了模糊工具要面对的挑战。

以一条简单的判断语句为例：

$$if\ (x == 123456)\ vulnerable( )$$

若要通过这一路径，最坏情况下，模糊工具需要在种子上做 10^6 次变异。在一个程序中，这样简单语句不在少数，若一直使用模糊工具测试，积累的时间消耗和计算机资源消耗是无法接受的。

3.3.4 过渡到动态符号执行部分

遇到模糊部分无法通过的复杂检查时，工具将过渡到动态符号执行部分，在现有输入之上进一步构筑特殊输入。这一过渡期间的算法原理如下。

当模糊部分度过了预设数量的变异，却依然没有识别到新的状态转换时，工具会认为引擎已经进入卡死状态。模糊工具会检测当前代码部分，寻找感兴趣的输入。

通过以下两个条件，工具的该部分获取到“感兴趣的输入”：

1. 应用程序在当前输入指引下所采取的路径，是第一个触发某些状态转换的路径。

2. 应用程序在当前输入指引下所采取的路径，是第一个进入唯一的循环桶空间（Loop bucketization）的路径。

这些条件限制了传递给动态符号执行组件的输入数量，同时确保了传递给动态符号执行部分的测试有更大的概率寻找到可行的路径。

3.4 动态符号执行部分

并行执行的优势在于，它可以搜索约束求解器可以满足的任何路径的输入。这使得它可以用于解决散列函数一类较为复杂的解决方案，这种情况下，模糊测试的暴力解法往往会失效。当 VeriRust 工具确定模糊测试部分无法找到其他状态转换时，将调用并行执行引擎。这时，模糊测试无法针对性地生成特定输入以满足代码中的复杂检查，需要求助于动态符号执行引擎。用于利用符号求解器，将能够达到但不能满足复杂检查的现有输入转换为能够达到并满足这种检查的新输入。

当 VeriRust 调用动态符号执行引擎时，它会传递由模糊引擎标识的所有“感兴趣的”输入。每个输入都被象征性地跟踪，以识别模糊引擎无法满足的状态转换。当确定这样的转换时，动态符号执行引擎生成输入，将通过这个状态转换驱动执行。在动态符号执行引擎完成处理所提供的输入后，其结果被送回模糊测试引擎的队列中，并将控制被传递回模糊引擎，以便它可以快速探索应用程序的新发现的部分。

3.4.1.KLEE 工具简介

在 Driller 工具中，动态符号执行引擎采用了表现更好的 Angr 引擎，Angr 引擎需要吞入 Valgrins VEX IR 中间语言进行测试，而 Rust 编译器建立在 LLVM 项目之上，Rust 二进制程序只能被转化 LLVM IR 中间语言。最终的设计方案中，VeriRust 工具退而求其次采用 KLEE 引擎作为动态执行引擎。

KLEE 引擎是一款经典的动态符号执行工具，该引擎的改进基于 EXE 引擎之上，依靠中间语言进行操作。中间语言被解释为程序代码对符号状态的影响。符号状态使用符号变量来表示可能来自用户的输入或其他非常数的数据

符号变量是一个可以产生许多可能的具体解决方案的变量，常用 X 或 Y 来表示。其他值，如程序中硬编码常量，被建模为具体值。随着执行过程的进行，符号约束将被添加到这些变量中。

约束，可以理解为对符号值的潜在解的限制声明，$X < 50$ 一类的公式即是最常见朴素的约束。具体值则是满足上述约束的任何 X 值。

在执行过程中，KLEE 分析引擎跟踪内存和寄存器（前文的符号状态）中的所有具体值和符号值。在到达的程序中的任何一点上，引擎都可以执行约束解析，以确定在满足该状态的情况下，所有符号变量约束的可能输入。当这样的输入传递到应用程序的正常执行流程时，应用程序也将转移到该输出处。

KLEE 的符号存储模型可以同时存储具体值和符号值。它使用一个基于索引的内存模型，其中读地址可以是符号化的，但写地址总是具体化的。这种方法维持分析功能的可行性：如果读写地址都是符号，重复地读写相同的符号索引会增加符号约束，或者让存储的符号表达式更加复杂。因此，符号机制下，写地址操作总是具体化为一个单一有效的解决方案。在一定条件下，符号值会被具体化为一个单一的潜在解。

3.4.2.动态符号执行部分的执行流程

在大多数测试工作中，模糊工具利用随机位翻转等常见的变异策略就可以胜任大部分路径的探索工作。不仅如此，在模糊工具和并行执行均可以探索到路径的情况下，模糊工具本地执行的表现将远优于并行执行，综合考虑这两点，在 VeriRust 中，大部分的工作都将由模糊工具完成，只有当存在更难解决的约束问题时，并行引擎才会被调用，这种设计方式可以大幅度提升路径探索的效率。

当模糊工具无法利用变异输入来发现新的执行路径时，并行执行引擎将被调用。它跟踪模糊工具发现的路径，识别进入到并行执行组件的输入，利用以上信息执行有限的符号探索。此外，当模糊工具组件发现崩溃的输入时，并行执行引擎会将该输入重新随机化，以恢复依赖于随机性和其他环境因素的崩溃输入部分。并行执行引擎的工作流程如下：

1. 跟踪预约束：VeriRust 使用并行执行从模糊工具中跟踪感兴趣的路径，并生成新的输入。由于只分析表示应用程序处理该输入的路径，这种操作允许 VeriRust 一定程度上规避并行探索中固有的路径爆炸，

A. 当跟踪从模糊工具传递到符号执行部分时，符号执行引擎将针对模糊工具未发现的新转换进行搜索。VeriRust 的并行执行引擎按照与模糊工具相同的路径跟踪输入。

B.　当 VeriRust 遇到一个条件控制流传输时，它会检查反转该条件是否会导致发现一个新的状态转换。如果可以，VeriRust 将生成一个示例输入，它将通过新的状态转换而不是原始的控制流来驱动执行。

C.　VeriRust 的并行执行引擎将引导模糊工具引擎到应用程序的新隔间部分。生成输入后，VeriRust 继续按照匹配路径查找额外的新状态转换。

2. 输入预约束：为了执行中生成相同的轨迹，并使有限的并行探索可行，符号执行引擎需要预约束处理。VeriRust 同样要使用预约束来确保并行执行引擎的结果与本地执行中的结果相同，这样还能确保工具依然能发现新的状态转换。在预约束执行中，输入的每个字节被约束至可以与模糊工具输出的每个实际字节相匹配。当发现新的可能的基本块转换时，预约束会被短暂地删除。这样，VeriRust 可以求解一个偏离该状态转换的输入。

程序 3.1 展示输入预约束如何在本文的 VeriRust 中工作：

在该 Rust 程序中，为了到达 vulnerable()函数 ，测试工具必须在第 18 行提供输入(0x12d615h8)这个值。由前文中对模糊测试引擎部分的实现介绍可知，模糊工具几乎不可能利用枚举猜到这一数字，VeriRust 无法通过模糊器部分在该程序上发现任何新的状态转换，动态符号执行引擎将被调用。

当调用并行执行来跟踪输入时，动态符号执行引擎首先约束符号输入中的所有字节，以匹配跟踪输入中的那些字节。由于程序是符号执行的，每个分支只会推及一种可能性，程序因此会遵循一条路径，这样，路径爆炸问题不会发生。当程序执行到达第 18 行时，VeriRust 认识到此处的替代状态转换未在模糊测试过程中得以进行。为此 VeriRust 删除在执行开始时添加的预约束，其中不包括通过符号执行程序而放置的谓词。字符数组 x 中的字节部分受路径的约束，magic 的值受$if(magic == 0x12d615h8)$的约束。就这样，并行执行引擎创建了一个包含 24 个 B 字符，并包含 magic 参数所需的值 0x12d615h8 的输入。在符合条件的输入的指导下，工具得以通过第 18 行的检查

程序 3.1：预约束示例

```
1.  fn check(x : u8, depth : i32){
2.     if depth >= 100  {
3.       return  0;
4.        }
5.      else   {
6.        let count = (x =='B')? 1: 0;
7.        count += check(x+1, depth+1);
8.        return  count;
9.      }
10.   }
11.   fn main(void) {
12.    let x:[i32,100];
13.    let  magic;
14.    read (0, x, 100);
15.    read(0, &magic, 4);
16.    if (check(x, 0) == 24)
17.     if magic == 0x12d615h8
18.      vulnerable();
19. }
```

3.有限的符号探索：并行执行的成本高昂，为了减少对它的调用，VeriRust 引入了一个符号探索存根（stub），以在新发现的状态转换基础上进一步预先探索更多的状态转换。这样，VeriRust 在发送后续请求之前，模糊测试引擎就可以预先找到进一步的状态转换，而无需重新追溯之前的路径。该部分的流程如下：

A.为 VeriRust 定义一个限制参数 N，它代表着一次探索中可遍历的基本块的上限，符号探索存根在此限制下探索状态转换节点的周围区域。

B.遍历了数量为 N 的基本块后，VeriRust 就会计算遍历时发现的所有路径的输入。这种做法起到缓冲的作用，避免模糊器部分在提供给 VeriRust 生成的输入后，就很快陷入卡死状态。

C.通过以上步骤生成一个新的输入后，该输入需要通过 VeriRust 多部分的复杂检查，随即返回模糊器部分，以更深入二进制计算。

4.再随机化：在程序运行期间引入的随机值可能会干扰模糊化的效果。下面以 Rust 程序 3.2 为例来进行说明：

程序 3.2：随机值影响

```
1   fn main(){
2    let mut challenge - rand::thread_rng().gen_range(0, 10000);;
3    let mut response;
4    let test = std::fs::File::create("test.txt")
5     test.write_all(&mut challenge);
6     test.read_to_string(&mut response).unwrap();
7     if challenge == response
8       println!("Abort");
9    }
```

用户执行该程序的结果是给出一个随机输入。显而易见，如果没有监控程序输出，我们就永远无法知道 challenge 的具体值。这构成了模糊化的不稳定性因素。

针对此问题，VeriRust 的处理步骤如下：

A. 一旦发现漏洞，VeriRust 将使用符号执行追踪引发崩溃的输入。

B. 恢复需要满足目标二进制文件构成的动态检查的输入字节。

C. 通过检查程序崩溃时的符号状态,查找应用程序输出和引发崩溃的输入之间的关系，VeriRust 可以确定应用程序的 challenge-repsonse 协议。

在程序 3.2 中，我们可以看到，read_to_string()函数的调用所引入的符号字节被限制为等于 write_all()函数调用所写出的字节。通过将这些关系确定化，工具可以生成相应的使用规范，用来应对真实测试情况下的随机性。

3.4.3.动态符号执行部分所受的限制

在并行执行测试的传统流程中，操作人员从程序的开头开始执行测试，使用符号执行引擎探索路径状态，以尽可能多地寻找错误。这种方法符合测试人员的

逻辑模式，但它也受到两个问题的限制：

第一，并行执行极为耗费时间。模糊工具以本地执行的方式进行测试，以一种“漫灌”的方式输入测试用例到具体程序中，这已经是一种耗时甚高的测试方式。但相比较来说，并行执行流程要涉及到搜寻路径，解释程序的代码，耗时更甚。此外，约束解决步骤会涉及到 NP 完全问题(Non-deterministic Polynomial Complete)的解决[15]，这亦需要消耗大量时间进行计算。

第二，制约着动态符号执行技术的状态爆炸问题同样存在于本文应用的 KLEE 工具中,随着并行执行引擎对程序的探索，路径的数量会呈指数级增长，在这种情况下，受到硬件功能的限制，对程序更深更广的探索是不可行的。以程序 3.3 中代码为例。

程序 3.3：限制示例

```
1 int check(char *x, int depth){
2   if depth >= 100 {
3      return 0;
4    } else{
4      fn count =(*x =='B') ? 1 : 0;
5      count += check(x+1,depth+1);
        return count;
6    }
7  }
8 fn main() {
      let x : [u8,100];
      read(0,x,100);
9     if check(x,0)== 24
10       vulnerable();、
11  }
```

在符号执行引擎中进行测试时，简单程序的测试也可能需要天文数字级别的计算。示例代码展示了一个简单的功能：主函数调用 check()函数统计，当用户恰好输入 24 个‘B’字符时，vulnerable()函数会被触发。但在符号执行体系下

测试时，CPU 对 check()函数的每次调用都涉及到变量 x 和字符‘B’进行值的比较，每次执行都将每个模拟状态一分为二，可能的状态达到 2^{100} 个，这个处理量超过了 CPU 的极限，这种情形下，状态爆炸不可避免地发生于此次符号执行流程中。相比之下，一个基于状态转换的模糊测试工具选择输入时，并不推理受测程序的整个状态空间，而只关注输入触发的状态转换。即，它更加关注次数，在程序 3.3 中，模糊工具只关注第 5 行的 check()是否成功。也就是说，无论 B 字符出现在输入中的哪个位置，模糊工具也只会根据它于输入中的数量来判断状态。这种做法有效地规避了路径爆炸。

3.5 本章小结

本章介绍了 VeriRust 工具的设计概况和各部分的工作原理。3.1 小节整体介绍了 VeriRust 的设计思路来源和各组成部分。3.2 小节介绍了建立于 LLVM 的编译器框架对于 Rust 代码转化为 IR 的流程。3.3 小节详细介绍了模糊测试部分的情况和细节，包括 AFL 工具的相关知识，模糊测试引擎的流程，缺陷，以及向动态符号执行部分转化时的操作。3.4 小节介绍了动态符号执行引擎部分的工作情况和细节，包括 KLEE 工具的相关知识，动态符号执行部分的流程和缺陷。

第 4 章 实验与结果分析

4.1 实验工具

serde_json 数据集:Rust 语言中,常见的 JSON 格式数据表示包括文本数据,无类型或松散类型的表示，以及强类型的 Rust 数据结构。

serde_json 是 Rust 的一个常用板条箱，即对基本库起到补充作用的第三方库。serde_json 提供了快速制作 JSON 格式数据的方法，以及在 JSON 数据的常用表示之间转换的的高效方法。

serde_json 数据集来源自一个调用了 serde_json 库的 Rust 目标程序,通过编译器转化为 BC 文件。它的大小为 15.56Mb。在现实程序中，大量调用各类基础库及三方库的程序是绝对的主流，因此，VeriRust 工具也需要这一类的程序进行测试。

url 数据集：url_crate 与 serde_json 常年位居 Rust 语言的第三方库中使用量的前三名。url_crate 板条箱提供了强大的网址解码功能，它可以将代码中的 str 类型的 URL 字符串解析为 URL 结构体,为用户利用 URL 结构体的方法提供便捷的通路。此外，它还能实现将给定的 URL 结构体还原为基础 URL 结构体等方法。

url 数据集与 serde_json 数据集类似，来源自一个调用了 url 库的 Rust 程序经过转化组成，大小为 17.12Mb。

执行实验的机器环境配置如下：

（1）操作系统：64 位 ubuntu 20.04

（2）编译器版本：Rustc 1.17

（3）实验设备：32 核的 Intel Core i7-6700 CPU

（4）内存：64GB。

4.2 评估指标与测试目标

在实验中应用的评估指标如下：

（1）**覆盖率**：在实际的验证中，不排除这样的情况：该程序的质量很高，程序在很长一段时间的测试中未出现问题，这时很难对验证工具的效果下定论，需要引入额外的判断标准加以判断。代码覆盖率在这时就起到了作用。

（2）**可测量性**：科研者们提出了一系列优化技术来解决可测性问题。例如，状态合并将有助于指数级地减少路径的数量，当符号表达式的复杂性失去控制时，就会调用具体化。指数搜索空间的存在令形式化工具经常遇到性能问题。对于有经验的用户，VeriRust 提供了一组丰富的命令行工具来配置、识别和消除性能瓶颈。

（3）**可用性**：在理想的场景中，用户只将 VeriRust 作为常规的模糊工具运行，甚至不会注意到后台运行的形式化工具。更具体的说法是，并行执行引擎将在需要时被自动调用，而无需手工确认。例如，当用户需要测试分支的可达性，或检查特定内存访问指令的绑定可能性时，并行执行引擎就会启动。

通过阅读工业界大量论文，可以得出这样一个结论：可用性差是阻碍工业界正式应用新技术的一个主要障碍。通过将形式化验证技术隐藏在模糊测试功能后，并尽可能地重复使用模糊测试工具，VeriRust 不会给终端用户带来额外的负担。

由于当前该领域尚缺乏可以比较的模型，实验的测试目标将如下安排：

（1）Rust 的一系列基础类库，如 SDLB，背景部分中介绍的工具都是无法支持的，它们指出的 feature 不够丰富，无法覆盖涉及到的一些库，因而无法处理不同的指令格式。虽然 VeriRust 更多的是在第三方库上做测试，但由于前文提及的 serde_json 库和 url 库理所应当地需要依赖一些 Rust 基础类库才能得以运作，VeriRust 工具已经可以在一定层面上解决基础库的问题。这一复杂的设计需要通过实战得以证明可用性。

此外，符号执行工具对于代码的规模十分敏感，这一类工具很容易因为代码过大而宕机。作为一种结合工具，它的实用性需要进行测试，若其可以达到百万

乃至千万的级别，那么工具就到达了工业界一系列实用工具的水准，体现了开发的价值

（2）模糊测试与动态符号执行技术在其他语言领域的应用非常有效，在 Rust 领域，此类结合的效果尚未得以证明，通过消融实验，可以很大程度上解决这个疑问。

4.3 实验及结果

4.3.1 VeriRust 可行性实验及分析

由于 Rust 领域缺少类似的结合工具，本文的实验并不会聚焦于与其他工具对比，而是单纯地展现结合技术在迁移至 Rust 领域后的表现效果,并证明 VeriRust 工具的可行性。

可行性实验将于前文介绍的 serde_json 数据集和 url 数据集上进行，它们来源自 Rust 语言最常用的板条箱，即第三方库。表格 4.1 展示了实验的结果：

表 4.1 VeriRust 于常见第三方库数据集上的实验数据

项目	最长路径指令	路径	运行时间(min)	初始种子数	生成种子数	程序大小(Mb)
url	5297283	243	725	158	2215	17.12
serde_json	2952632	97	136	219	1027	15.56

通过阅读工业界的论文可知，符号执行一类的形式化验证的一大问题在于它们缺乏处理大规模项目的能力，通常情况下，在一次路径探索中，路径指令不会超过十万级别。而 serde_json 数据集和 url 数据集上的实验都表明 VeriRust 工具处理的最长路径指令可以达到百万级别。

尤其要强调的是，VeriRust 工具可处理的最长路径指令达到了 5297283，这个数字意味着 VeriRust 有能力处理包含 8000 到 10000 行代码的程序——这已经

达到了中等规模的嵌入式代码的级别。这意味着，VeriRust 具备了向商用级别的测试工具发展的可能性，因为工具已经具备了测试一个领域的主流规模程序的能力。

VeriRust 的可行性由此得到了有效的证明。

4.3.2 有效性实验

本小节将通过实验验证模糊测试与动态符号执行技术结合的有效性：作为新领域的实验性工具，前文对 VeriRust 的实验更多聚焦于可行性而非改进的有效性。由于符号执行工具和模糊测试工具固有的性能差异，无法通过简单的可行性实验验证结合是否实现了“1+1>2”的效果。验证此类技术结合在 Rust 领域是否有效，需要回归到准确率实验中。

实验将不再局限于前文提及的 serde_json 与 url 数据集上进行。为了确保可以对引用库的程序进行检查，选取的程序或者引用了第三方库，或者大规模调用了一些 Rust 基础库.结合原本的数据集，一共选取了 20 个程序进行测试,其中 3 个可以确定发生了明显的错误，主要集中于循环问题和内存控制问题上。

实验设置的对比工具如下

基本模糊测试工具。此部分在现有的 VeriRust 上简单修改即可完成，修改不会涉及到对 AFL 的核心逻辑的改变。在该测试中，VeriRust 的并行执行节点被停用，每个二进制文件分到 4 个 AFL 模糊工具核。当 AFL 引擎无法发现新的路径时，它不会得到任何辅助工具的协助。

基本的符号执行工具。为了最大化地减少不同工具对于结果的干扰，在此处的实验中，沿用了 VeriRust 采用的改进集成的 KLEE 符号执行引擎来进行动态符号执行测试。KLEE 会探索状态空间，从入口点开始，检查内存泄露等故障，进而分析每个二进制文件。当状态爆炸确实发生时，KLEE 引擎使用启发式方法对深入探索应用程序的路径进行优先排序，以最大化代码覆盖率。

本文提出的 VeriRust 工具。当测试 VeriRust 时，每个 Rust 文件被模糊引擎分配到 4 个核，此外，并行执行引擎分配给文件 64 个核。并行执行池处理队

列中的符号执行作业，当 VeriRust 的模糊测试部分陷入卡死状态时，符号执行引擎请求跟踪模糊节点的跟踪时间被限制在一个小时内，占用内存被限制在 1G 以内，以避免进行大规模跟踪时耗尽计算资源。

表 4.2 展示了应用了 VeriRust 工具相关技术的工具于 93 个程序上的实验结果。

表 4.2 多种技术工具的准确率对比

方法	错误发现数量	卡死状态次数
VeriRust	4	4
DSL 工具（改进 KLEE）	2	-
DSL 工具(Cargo_KLEE)	3	-
模糊测试工具（AFL）	1	6

在有效性实验中，相比较 AFL 工具发现的 1 个错误，和 Cargo_KLEE 发现的 3 个错误，VeriRust 工具明显可以发现更多的错误，并且陷入卡死状态的次数更少。此外，VeriRust 采用的改进 KLEE 工具相比较 Cargo_KLEE 效果略微逊色，这可能是因为 Cargo_KLEE 工具作为单一工具，优化集成不需要考虑到一系列结合带来的微调。

实验结果表明，在 C++领域中展现出卓越性能的，通过动态符号执行增强模糊工具的技术，在 Rust 语言领域同样能体现出结合的优越性：VeriRust 能够比单独的动态符号执行工具和单独的模糊测试工具，在 Rust 程序上发现更多的错误。

4.4 本章小结

本章介绍了测试 VeriRust 的实验。4.1 节介绍了实验应用的数据集和机器环境；4.2 节介绍了实验的评估指标与测试目标；4.3 节介绍了 VeriRust 的可用性实验，通过对常用第三方库数据集的实验证明了提出的 VeriRust 工具于 Rust

领域的有效性与开创性；4.3.2 节介绍了 VeriRust 的消融实验，这部分实验证明了模糊测试引擎和动态符号引擎结合的有效性，也证明了此类技术于 Rust 领域结合的有效性。

第 5 章　总结与展望

5.1 总结

近年来，随着科研界和工业界的投入，Rust 语言程序的安全性使其在系统软件领域愈发受到重视，要求进一步提高 Rust 程序安全性能的呼声也愈发强烈。在这种情境下，开发 Rust 语言领域的自动测试工具成为了最具性价比的选择。然而，Rust 自动测试工具的发展现状尚不理想，许多有效的技术亟待开发人员的迁移。比如，在 C++与 Python 语言测试工具中，动态符号执行技术和模糊测试技术的结合已经成为一种高效而流行的方法，自诞生以来就发挥着巨大的作用。然而，Rust 语言的测试工具领域缺乏这样的结合。针对这样的现状，本文为 Rust 语言设计并实现了基于模糊测试技术和动态符号执行技术的测试工具，它的主要工作和贡献如下：

1. 本文总结了当前 Rust 语言测试的技术发展，阐述了 Rust 测试领域发展的不足，介绍了模糊测试技术与动态符号执行技术结合工具在其他语言测试领域取得的效果，在此之上提出结合模糊测试技术与动态符号执行技术的 Rust 测试工具的设计方案。

2. 针对上述设计方案，本文提出了 VeriRust 工具，它是一种针对 Rust 语言开发的混合形式验证和自动测试生成工具。VeriRust 利用模糊测试引擎快速寻找隔间，同时利用 LLVM 技术将 Rust Bytecode 进行预处理。当模糊测试引擎无法通过复杂检查时，经过集成改进的动态符号执行引擎会被启动以探索路径，结果会被传回模糊测试引擎以继续快速测试。通过这样的机制，工具可以详尽地遍历控制流图中的路径，在自动扩大覆盖率的同时寻找程序漏洞。

3. VeriRust 工具在 Rust 程序最常用的两个第三方库的数据集上进行了测试，实验结果证明了 VeriRust 在 Rust 程序上体现了相当的可用性。此外，本文针对模糊测试模块和并行执行模块的效果进行了消融实验，进一步证明，模糊测试技术与动态符号执行技术的结合在 Rust 领域同样具备“1+1>2”的效果。通过上述实验，VeriRust 工具实现了有意义的探索，它证明了模糊测试技术与动态

符号执行技术结合工具在 Rust 测试领域的有效性，亦在一定程度上填补了 Rust 测试领域的空白。

5.2 展望

VeriRust 工具正处于完整开发的初步阶段，一个可用的工具已经被提出。首要的改进是优化代码，以另 VeriRust 能进一步适配 Rust 语言，目标是最终提出一个稳定的工具版本。此外，由于 Rust 语言已经比较成熟，当前公布的新版本已经展现了足够的可靠性，可以尝试在最新版本的 Rust 语言上进行优化改进。

稳定的版本开发完成后，后续的工作将转向继续提升 VeriRust 的性能，并加强其可用性。后续的改进路线包括：

1. 静态分析工具。静态分析工具的优势在于，它可以操作程序中所有可能的执行分支，当前，Rust 语言已经得到 Rust-analyzer 一类的静态分析工具的支持。通过整合静态分析工具，VeriRust 可以在对受试代码进行集成之前，就预先发现代码存在的一些问题，节约一定的算力。

2. 云计算 API。整合云计算 API 是工业界的常见作法，VeriRust 的计算能力可以通过这一改进获得极大的提升，并能于云端获取更多的测试用例检验效果。当前工业界已经有一系列成熟的产品可以与本文的工具作结合。

3. 制作板条箱。通过将模型进一步精简，VeriRust 可以被制成一个简单的板条箱，以便其他研究人员测试与使用，这将有助于发现 VeriRust 本身的缺陷。

4. 改进动态符号执行引擎。VeriRust 针对 LLVM 的位代码开发，而不是专门来自 Rust 程序的位代码。这样的限制使 VeriRust 工具只能被禁锢到 KLEE 一类基础的动态符号执行工具中——尽管针对 Rust 改进的 KLEE 引擎已经被提出，但可用的实例还是数量极为有限。从根本上解决这一问题需要大量的文献阅读与代码工作。

参考文献

[1] Balasubramanian A, Baranowski M S, Burtsev A, et al. System programming in rust: Beyond safety[C]//Proceedings of the 16th Workshop on Hot Topics in Operating Systems. 2017: 156-161.

[2] Jung R. Understanding and evolving the Rust programming language[J]. 2020.

[3] Evans A N, Campbell B, Soffa M L. Is Rust used safely by software developers?[C]//2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE). IEEE, 2020: 246-257.

[4] Evans A N, Campbell B, Soffa M L. Is Rust used safely by software developers?[C]//2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE). IEEE, 2020: 246-257.

[5] Godefroid P, Levin M Y, Molnar D A. Automated whitebox fuzz testing[C]//NDSS. 2008, 8: 151-166.

[6] Kuznetsov V, Kinder J, Bucur S, et al. Efficient state merging in symbolic execution[J]. Acm Sigplan Notices, 2012, 47(6): 193-204.

[7] Stephens N, Grosen J, Salls C, et al. Driller: Augmenting fuzzing through selective symbolic execution[C]//NDSS. 2016, 16(2016): 1-16.

[8] Lindner M, Aparicius J, Lindgren P. No panic! Verification of Rust programs by symbolic execution[C]//2018 IEEE 16th International Conference on Industrial Informatics (INDIN). IEEE, 2018: 108-114.

[9] Cadar C, Dunbar D, Engler D R. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs[C]//OSDI. 2008, 8: 209-224

[10] Astrauskas V, Müller P, Poli F, et al. Leveraging Rust types for modular specification and verification[J]. Proceedings of the ACM on Programming Languages, 2019, 3(OOPSLA): 1-30.

[11] Gurfinkel A , Kahsai T , Komuravelli A , et al. The SeaHorn Verification Framework[C]// International Conference on Computer Aided Verification. Springer International Publishing, 2015.

[12] Sabbaghi A, Keyvanpour M R. A Systematic Review of Search Strategies in Dynamic Symbolic Execution[J]. Computer Standards & Interfaces, 2020, 72: 103444.

[13] Zalewski M. american fuzzy lop (2.52 b)[J]. Retrieved April, 2019, 10: 2020.

[14] Shoshitaishvili Y, Wang R, Hauser C, et al. Firmalice-Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware[C]//NDSS. 2015, 1: 1.1-8.1.

[15] Paz A, Moran S. Non deterministic polynomial optimization problems and their approximations[J]. Theoretical Computer Science, 1981, 15(3): 251-277.

[16] Boehme M, Cadar C, Roychoudhury A. Fuzzing: Challenges and Reflections[J]. IEEE Softw., 2021, 38(3): 79-86.

[17] Bucur S. Improving scalability of symbolic execution for software with complex environment interfaces[R]. EPFL, 2015.

[18] Baranowski M, He S, Rakamarić Z. Verifying Rust programs with SMACK[C]//International Symposium on Automated Technology for Verification and Analysis. Springer, Cham, 2018: 528-535.

[19] Bhattacharjee J. Basics of Rust[M]//Practical Machine Learning with Rust. Apress, Berkeley, CA, 2020: 1-30.

[20] Takashima Y, Martins R, Jia L, et al. Syrust: automatic testing of rust libraries with semantic-aware program synthesis[C]//Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. 2021: 899-913

[21] Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation[C]//International Symposium on Code Generation and Optimization, 2004. CGO 2004. IEEE, 2004: 75-86.

[22] Lattner C. LLVM and Clang: Next generation compiler technology[C]//The BSD conference. 2008, 5.

[23] Milanesi C. Printing on the Terminal[M]//Beginning Rust. Apress, Berkeley, CA, 2018: 1-8.

[24] Yun I, Lee S, Xu M, et al. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing[C]//27th USENIX Security Symposium (USENIX Security 18). 2018: 745-761.

[25] Jung R, Jourdan J H, Krebbers R, et al. Safe systems programming in Rust[J]. Communications of the ACM, 2021, 64(4): 144-152.

[26] Chen P, Chen H. Angora: Efficient fuzzing by principled search[C]//2018 IEEE Symposium on Security and Privacy (SP). IEEE, 2018: 711-725.

[27] Rawat S, Jain V, Kumar A, et al. VUzzer: Application-aware Evolutionary Fuzzing[C]//NDSS. 2017, 17: 1-14.

[28] Aschermann C, Schumilo S, Blazytko T, et al. REDQUEEN: Fuzzing with Input-to-State Correspondence[C]//NDSS. 2019, 19: 1-15.

[29] Manès V J M, Han H S, Han C, et al. The art, science, and engineering of fuzzing: A survey[J]. IEEE Transactions on Software Engineering, 2019, 47(11): 2312-2331.

[30] Permenev A, Dimitrov D, Tsankov P, et al. Verx: Safety verification of smart contracts[C]//2020 IEEE symposium on security and privacy (SP). IEEE, 2020: 1661-1677.

[31] She D, Pei K, Epstein D, et al. Neuzz: Efficient fuzzing with neural program smoothing[C]//2019 IEEE Symposium on Security and Privacy (SP). IEEE, 2019: 803-817.

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

[33] Zhang G, Zhou X, Luo Y, et al. Ptfuzz: Guided fuzzing with processor trace feedback[J]. IEEE Access, 2018, 6: 37302-37313.

[34] Ispoglou K, Austin D, Mohan V, et al. {FuzzGen}: Automatic Fuzzer Generation[C]//29th USENIX Security Symposium (USENIX Security 20). 2020: 2271-2287.

[35] Kim K, Jeong D R, Kim C H, et al. HFL: Hybrid Fuzzing on the Linux Kernel[C]//NDSS. 2020.

[36] Chen B, Havlicek C, Yang Z, et al. CRETE: A versatile binary-level concolic testing framework[C]//International Conference on Fundamental Approaches to Software Engineering. Springer, Cham, 2018: 281-298.

[37] Kim M, Kim D, Kim E, et al. Firmae: Towards large-scale emulation of iot firmware for dynamic analysis[C]//Annual Computer Security Applications Conference. 2020: 733-745.

[38] Pham L H, Le Q L, Phan Q S, et al. Enhancing symbolic execution of heap-based programs with separation logic for test input generation[C]//International Symposium on Automated Technology for Verification and Analysis. Springer, Cham, 2019: 209-227.

[39] Blair W, Mambretti A, Arshad S, et al. HotFuzz: Discovering algorithmic denial-of-service vulnerabilities through guided micro-fuzzing[J]. arXiv preprint arXiv:2002.03416, 2020.

[40] Boehme M, Cadar C, Roychoudhury A. Fuzzing: Challenges and Reflections[J]. IEEE Softw., 2021, 38(3): 79-86.

作者简介及在学期间所获得的科研成果

作者简介

牛聚川（1996.12.30），男，汉族，黑龙江省哈尔滨人，吉林大学软件学院软件工程专业专业型研究生。2019 年 9 月至 2022 年 5 月于吉林大学符号计算与知识工程实验室进行学习和研究。

致　谢

光阴似箭，日月如梭，执笔至此，感慨颇多。此时的我，已经是一名即将毕业的研三学生，但闭上双眼，我仍能回忆起第一次来到吉林大学的欣喜与期待。

首先，我要将最大的感谢献给指导我三年的恩师，张永刚教授，张老师为人平易近人，性格和蔼包容，博学而明智。在学术研究的探索中，张老师尽可能地为我们提供学术交流的机会，在百忙之中也一定要抽时间加入我们的学术讨论，老师的指导、师兄的建议和同侪的交流，让我获益颇多。在这个过程中，我得以增长了学术思维，更锻炼了自己的沟通能力和心理素质。多谢了张老师给予的平台，能让我在这三年中迅速成长。在此，我衷心祝愿张永刚老师的身体健康，家庭美满，事业能更进一步！

感谢张一民老师，您为我的科研带来了雪中送炭般的帮助。没有您的指导，我很难找到科研的前行方向。在此，祝您的科研工作取得更好的成果，希望您的名字能更多地出现在顶会的目录中！

感谢李耀麟和张天杭师兄，在科研方面，我总是显得笨手笨脚，感谢你们两年来对我的指导与包容，没有你们，本就愚钝的我在科研道路上想必会遇到更多的磕绊。愿你们在科研和工作中实现自己的梦想，我跟不上你们的脚步，但我会以你们为同龄人的标杆，不懈地努力下去。

感谢 A506 实验室的同届战友，也要将祝福献给仍在积极探索奋斗的师弟师妹们，感谢你们在三年科研学习过程中给予过我的关心与陪伴，更要感谢你们对我的包容与肯定，你们为我树立了自信，让我在硕士生活中得以确立自我认知和努力的方向。感谢三年的陪伴，希望以后的人生中有缘再见！

感谢我的父母，他们是我永远的防波堤，每当我陷入迷茫之中，父母都会为我指引道路。希望在不久的未来，我能成长得足够成熟与坚强，强大到能为他们遮风避雨。

最后，感谢吉林大学，于此三年的学习生活重新塑造了我的性格，也改变了我的人生——在此表示衷心的感谢！祝愿吉大未来的发展道路一片坦途！