计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2021,30(11):33−40 [doi: 10.15888/j.cnki.csa.008348]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

面向 RISC-V 的汇编程序语义等价性自动化测试系统①

徐学政, 王　涛, 方　健, 张光达

(军事科学院 国防科技创新研究院, 北京 100097)
通讯作者: 方　健, E-mail: fangjian_alpc@163.com

摘　要: 本文设计并实现了一套面向 RISC-V 的汇编程序语义等价性自动化测试系统. 在面向 RISC-V 开发软件时, 尤其是基于扩展指令 (例如向量指令) 编写高效的程序时, 很难避免以手写汇编的方式编写代码. 例如, 为标准的 C 函数库编写相应的向量版函数. 与编译器自动生成的代码不同, 手写的汇编代码虽然可以最大限度地提高程序的效率, 但因绕过了编译时对程序的约束 (如类型检查、寄存器分配等) 而对开发者提出了更高的要求. 能否对新版本与标准版本的汇编程序进行快速地、自动化的语义等价性测试, 将大大影响代码的正确性和软件开发和调试的效率. 已有面向 RISC-V 的测试框架缺乏对语义等价性测试的支持, 也未考虑程序执行带来的副作用. 本研究基于模拟器的动态测试环境, 设计并实现了一套面向 RISC-V 的汇编程序语义等价性自动化测试系统. 系统通过跟踪机器状态, 捕获程序执行的副作用, 并结合用户定义的测试目标生成测试报告. 实验表明, 本系统相比已有的测试系统, 能够有效地对 RISC-V 汇编程序的语义等价性进行测试.

关键词: RISC-V; 自动化测试; 语义等价性; 汇编程序

引用格式: 徐学政,王涛,方健,张光达.面向 RISC-V 的汇编程序语义等价性自动化测试系统.计算机系统应用,2021,30(11):33–40. http://www.c-s-a.org.cn/1003-3254/8348.html

Automatic Testing System for Semantic Equivalence of RISC-V Assembly Programs

XU Xue-Zheng, WANG Tao, FANG Jian, ZHANG Guang-Da

(Defense Innovation Institute, Academy of Military Sciences, Beijing 100097, China)

Abstract: In this study, we design and implement an automatic testing system for semantic equivalence of RISC-V assembly programs. While developing RISC-V programs, especially developing efficient programs based on extension instructions (such as vector extension), developers often write assembly code manually. For example, for the standard C function library, we often write the corresponding vector version functions for better performance. Without the compiler, the manually developed assembly code can maximize the efficiency of the program, but it skips many important compilation processes (such as type checking and register allocation), thus putting forward higher requirements for the developers. It will greatly affect the correctness of the code and the efficiency of software development and debugging if we can quickly and automatically test whether the rewritten version is semantically equivalent to the standard version of the program. The existing RISC-V testing framework lacks support for semantic equivalence testing and fails to consider the side effects caused by program executions. Based on the dynamic test environment of a simulator, this research designs and implements an automatic testing system for semantic equivalence of RISC-V assembly programs. It can capture side effects caused by program executions through monitoring machine states and generate testing reports with

① 基金项目: 国家自然科学基金 (61802427)
Foundation item: National Natural Science Foundation of China (61802427)
本文由“RISC-V 技术与生态”专题特约编辑武延军研究员、李玲研究员以及邢明杰高级工程师推荐.
收稿时间: 2021-04-28; 修改时间: 2021-05-21, 2021-06-08; 采用时间: 2021-06-11; csa 在线出版时间: 2021-10-22

user-defined testing targets. Experiments show that the system, compared with existing testing systems, can test the semantic equivalence of RISC-V assembly programs.

Key words: RISC-V; automatic testing; semantic equivalence; assembly program

1 引言

1.1 研究背景

RISC-V 是一个基于精简指令集 (RISC) 原则的开源指令集架构, 因其精简、开放、模块化的设计和高可定制的特点在工业界和教育界广受欢迎. 随着面向 RISC-V 的处理器接连问世, 围绕 RISC-V 建设完善的软件生态系统将会大大提高系统和应用软件的设计开发效率, 并降低其维护成本.

在面向 RISC-V 的软件开发过程中, 尤其是基于扩展指令 (例如向量扩展指令或用户自定义指令) 进行程序开发时, 很难避免以手写汇编的方式为程序编写高效的汇编代码. 例如, 为标准的 C 函数库编写相应的向量版本函数. 与编译器自动生成的代码不同, 手工开发的汇编代码虽然可以最大限度地提高程序的效率, 但绕过了编译时对程序的约束 (如类型检查[1]、寄存器分配[2] 等), 因而对开发者提出了更高的要求. 如果开发者经验不足或对指令的理解有误, 将会为代码的正确性和安全性埋下隐患. 以向量扩展指令为例, 为常用函数 (如 strlen、memcpy 等) 的标准版本开发相应的向量版本后, 能否快速地、自动化地测试向量版本与标准版本的函数在语义上是否等价, 将大大影响代码的正确性和软件开发和调试的效率.

1.2 研究现状

国内外学者已提出多种程序测试方法[3–5], 主要分为静态和动态两种测试思路: (1) 静态测试利用形式化的方法[6] (如符号执行[7] 等), 通过对程序语句的语义建模, 模拟程序的执行, 从而达到对程序正确性的验证. 该方法不依赖指令的软硬件运行环境, 能够对程序的分支进行全覆盖从而达到严格的验证结果, 但随着程序规模的扩大, 静态方法的运算量呈现指数级增长. 因别名、分支条件不可解等因素而被迫对程序采取保守的建模方式, 也会为静态分析带来大量误报[8]; (2) 动态测试 (如模糊测试[9] 等) 给定程序的输入值, 运行测试程序以触发程序中的错误. 虽有效避免了误报, 但在程序分支的覆盖率上往往很难匹及静态方法. 此外, 学术界也利用静态和动态相结合的方式[10,11] 对软件测试提供新的思路. CASM-Verify[11] 结合随机测试和符号执行验证密码算法在 X86_64 和 SSE 的汇编实现上的等价性, 但缺乏针对 RISC-V 的支持.

目前, 面向 RISC-V 的指令或系统级软件模拟器 (如 Spike、Qemu[12]、Ovpsim 等) 可以为程序提供动态测试环境. 针对某段待测试的汇编程序, 并给定输入值和期望的输出值, 开发者能够在模拟器上对汇编程序进行功能级测试. 例如, Ovpsim 提供了以测试用例为主导的测试脚本, 开发者可以通过将程序的运行结果 (某些寄存器或内存的值) 与参考值进行比对, 以达到测试的目的. 然而, 这种方式并无法满足语义等价性测试的需求, 因为: (1) 现有框架只针对单一测试程序, 缺乏对多个程序语义等价性测试的框架; (2) 比对程序的实际输出值与参考值, 并不足以支持语义等价性测试, 指令运行过程中产生的副作用 (如是否更改了其他寄存器和内存的值) 并未纳入考虑, 将为程序的正确性和安全性埋下隐患.

本研究基于模拟器的动态测试环境, 聚焦于设计并实现一套面向 RISC-V 的汇编程序语义等价性自动化测试系统, 旨在提高 RISC-V 汇编程序的开发与测试效率. 针对已有框架的不足, 本文设计并实现了针对多个 RISC-V 汇编程序语义等价性的测试框架. 系统可通过跟踪机器状态, 捕获程序执行的副作用, 并结合用户定义的测试目标生成测试报告. 实验表明, 本系统相比已有的测试系统, 能够有效地对 RISC-V 汇编程序的语义等价性进行测试.

本文结构如下: 第 2 节简要介绍了 RISC-V 的应用程序二进制接口; 第 3 节对本文研究的问题进行形式化描述; 第 4 节和第 5 节分别对系统设计和系统实现进行详细介绍; 第 6 节介绍了实验设计及结果并进行了案例分析; 第 7 节进行了总结和展望.

2 RISC-V 的应用程序二进制接口

本节以 RV64G 默认的 LP64D 规范为例, 简要介绍 RISC-V 的应用程序二进制接口 (Application Binary Interface, ABI). ABI 规定了 RISC-V 二进制的寄存器

规范、运行时栈的空间分布、函数调用规范、C 语言类型细节、ELF 文件格式、寻址方式等, 影响了编译工具链中指令生成、寄存器分配、ELF 文件生成等诸多方面. 了解 ABI 规范对于汇编程序的测试至关重要, 例如返回值的传递规范将影响程序的观测值的选定. 本节简要介绍其中寄存器规范、运行时栈分布和函数调用规范 3 个方面.

2.1 寄存器规范

依照 LP64D 规范, RISC-V 的通用寄存器包含 32 个整型寄存器, 32 个浮点寄存器和 32 个向量寄存器.

32 个整型寄存器中, 0 号寄存器 x0 恒为全 0; 16 个为调用者负责保存 (caller-saved) , 寄存器值在函数调用之后的值可能被修改; 16 个为被调用者负责保存 (callee-saved) , 寄存器值在函数调用之后不会更改. 其中 x10–x17 共 8 个寄存器为参数寄存器 (即 a0–a7 寄存器); 32 个浮点寄存器中, 20 个为 caller-saved, 12 个为 caller-saved, 同样具有 8 个参数寄存器; 32 个向量寄存器 (以及 vl、vtype 等) 均为 caller-saved.

2.2 运行时栈的空间分布

运行时栈负责维护函数执行时传递的参数、临时变量、callee-saved 寄存器等. 依照 LP64D 规范, 运行时栈空间从高地址向低地址生长, 以 16 字节对齐. 具体的空间分布如图 1 所示.

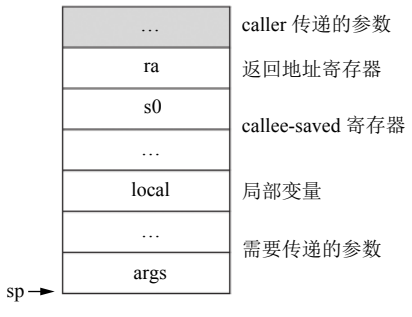

图 1 RISC-V 运行时栈的空间分布示意图

2.3 函数调用规范

函数调用规范 (calling convention) 主要规定函数调用时参数和返回值如何传递, 本节依照 LP64D, 主要分为 5 部分进行介绍: 整型参数、浮点参数、结构体参数、变长参数和返回值.

整型参数传递的主要原则是: 对于大小不超过 64 bit 的参数, 使用单个整型参数寄存器; 对于大小在 64 bit 和 128 bit 之间的参数, 使用连续的两个整型参数寄存器, 低 64 bit 和高 64 bit 分别放在寄存器 $x(n)$ 和 $x(n+1)$ 中, 若仅剩一个整型参数寄存器, 低位使用寄存器, 高位则使用栈传递; 若无可用的整型参数寄存器, 使用栈传递, 参数自右向左依次压栈.

浮点参数传递的主要原则是: 对于大小不超过 64 bit 的参数, 使用单个浮点参数寄存器, 不足 64 bit 的浮点数进行 1 扩展; 若无可用的浮点参数寄存器, 使用整型参数寄存器代替; 若无可用的整型参数寄存器, 将参数自右向左压栈.

结构体参数传递的主要原则是: 若结构体无浮点数成员, 可看作 1 个整型参数, 依照整型参数传递原则处理, 结构体内存分布不变; 若仅包含 1 个 (或 2 个) 成员, 均为浮点数且均为超过 64 bit, 则使用 1 个 (或 2 个) 浮点参数寄存器传递; 若仅包含 1 个整型成员和 1 个浮点成员, 均不超过 64 bit, 则分别使用 1 个整型和 1 个浮点参数寄存器传递; 其余情况均参照整型参数传递原则.

C 语言中的变长参数分为有名和无名参数, 无名参数以省略号代替, 其类型和个数编译时未知, 由调用者通知被调用者本次调用的参数类型和个数 (例如 printf 以格式化字符串的形式), 并默认执行参数的类型提升 (例如将 float 提升为 double). 变长参数传递的不同之处在于: 使用连续的两个寄存器传递 64 bit 至 128 bit 的参数时, 第 1 个寄存器必须为偶数号; 被调用者先将所有使用寄存器传递的无名参数进行压栈, 并计算首个无名参数在栈中的地址.

返回值可看作被调用者向调用者传递的参数, 以前两个参数寄存器传递返回值. 如果返回值超过 128 bit (需要以指针的方式间接传递), 调用者负责为返回值在栈中分配空间, 并隐式地将其在栈中的地址作为第 1 个参数传递, 实际的参数依次后移.

3 问题描述

3.1 机器状态模型

一个简单的机器状态模型可将机器状态 m 用一系列的“键-值”对 (k, v) 来描述. 即:

$$m = \{(k_1, v_1), (k_2, v_2), \cdots\}$$

同时, 值 v 可由下式表示:

$$v = m[k], \text{ if}(k, v) \in m$$

其中, 键 k 可由寄存器名称或内存地址构成, 值 v 可由

十六进制数字表示. 例如, 可将值为 0xffff 的寄存器 x1 表示为 (x1, 0xffff), 值为 0x17 的内存地址 0x7f7e9b18 表示为 (0x7f7e9b18, 0x17).

给定程序 p 和初始 (输入) 机器状态 m^{I}, 机器可通过运行 p 得到新的 (输出) 机器状态 m^{O}, 该过程可用 $p(m^{\mathrm{I}})=m^{\mathrm{O}}$ 表示.

3.2 一般的程序测试模型

我们将一般的基于测试用例的程序测试 (简称一般测试) 模型描述为: 对程序 p 输入一组初始机器状态, 并设置相对应的期望输出的机器状态, 通过判断以下关系是否成立以测试程序 p 的正确性:

$$\forall i \in \{0,1,\cdots,n\},\ p\left(m_i^{\mathrm{I}}\right)=m_i^{\mathrm{O}}$$

实践中, 输入的机器状态可由测试的运行环境 (初始化程序、输入参数等) 给定, 而期望的输出状态却难以完整描述, 一般只通过某个 (或多个) 变量的值描述. 例如, 针对某个输出状态 m^{O}, 期望的函数的返回值为 1, 可记为 m^{O}={(a0, 0x1)}. 显然, 不完整的机器状态描述为测试带来诸多隐患. 例如, 程序满足期望的输出值并通过测试, 但其间可能修改了预期之外的内存值或 callee-saved 寄存器值从而引发安全问题.

3.3 语义等价性测试模型

对于两段程序 p_0 和 p_1 和任意的机器状态 m, 严格的语义等价可描述为:

$$p_0(m)=p_1(m)$$

该定义需对比任意输入状态下的完整输出机器状态以确保语义的等价, 相比一般测试模型具有更为严格的定义以确保捕获程序的副作用. 显然, 我们无法负担对任意机器状态的穷举. 另外, 完整地描述机器状态通常是没有必要的. 例如, 在第 2 节介绍的 ABI 中, RISC-V 的 32 个整型寄存器中有 16 个属于调用者维护, 允许用户程序进行修改而无须恢复现场, 若两个待测程序对某些调用者维护的寄存器进行了不同的修改, 仍可认为其语义等价. 实际上, 在调用者维护的寄存器中, 除用于存放返回值的寄存器外, 大部分寄存器无须在测试中进行描述.

我们将语义等价性测试模型描述为三元组 (P, M, K), 包括一组待测程序$P=\{p_0,p_1,p_2,\cdots\}$, 一组作为输入的机器状态$M=\{m_0,m_1,m_2,\cdots\}$以及一组作为测试标准的键$K=\{k_0,k_1,k_2,\cdots\}$. 当满足以下条件时, 程序通过语义等价性测试:

$$\forall p_x,p_y \in P,\ \forall m \in M,\ \forall k \in K,\ p_x(m)[k]=p_y(m)[k]$$

与一般的测试模型不同的是, 语义等价性测试模型: (1) 规定了一组键作为测试标准, 通常包括返回值寄存器 a0, 被调用者维护的寄存器、某些内存值等, 远远多于一般测试的某几个观测值; (2) 无须给定输入机器状态对应的输出机器状态, 即无须由用户为相应键 k 计算值 v.

4 系统设计

基于第 3.3 节介绍的语义等价性测试模型, 我们设计了面向 RISC-V 的汇编程序语义等价性自动化测试系统. 本节介绍总体框架 (4.1 节) 以及其中的系统配置 (4.2 节)、测试运行 (4.3 节) 和结果分析 (4.4 节) 3 个环节.

4.1 总体框架

测试系统的整体流程如图 2 所示. 其中, 系统配置环节准备了测试所需的测试用例和一组待测程序, 并由用户配置测试目标. 之后, 系统分别运行各个待测程序并同时追踪机器的状态变化, 记录下在运行过程中被更改的机器状态. 最后, 通过比对各个待测程序更改的机器状态, 结合测试目标, 生成语义等价性测试的测试报告.

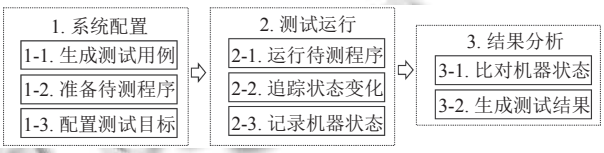

图 2 语义等价性测试系统整体流程图

4.2 系统配置

4.2.1 生成测试用例

测试用例的生成[13,14] 与本系统相对独立, 可以采用手工编写或在给定约束下自动生成两种方法. 与传统的“输入-输出”模式的测试用例不同, 语义等价性测试并不要求用户给定期望的输出值, 只需要比对待测程序的运行结果是否一致即可, 这大大减少了生成测试用例的工作量. 本系统采用测试用例的自动生成, 以函数 int add(int a, int b) 为例, 在给定参数的数据类型 int 的前提下, 系统随机生成一系列测试用例, 如 a=1, b=2; a=99, b=−99 等.

4.2.2 准备待测程序

为保持待测程序运行前机器状态的一致性, 系统

分别对每个待测程序进行“汇编-链接”, 生成可执行文件 (见图 3) . 其中测试用例负责准备输入参数并调用相应的待测程序, 为提示模拟器待测程序运行的开始和结束, 系统在待测程序的开始和结束插入两条指示性的自定义指令 test_start 和 test_end, 并同时在汇编器和模拟器中添加支持. 另一个可行的实现思路是: 通过保存模拟器的状态, 实现在同一状态下分别调用不同的待测程序.

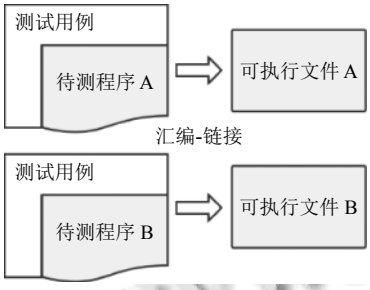

图 3 待测程序生成可执行文件流程图

4.2.3 配置测试目标

用户的测试目标, 即第 3.3 节中的集合 K, 在本系统中是可配置的, 覆盖寄存器及内存的值. 具体地, 用户需要配置: (1) 观测目标, 包括反映程序功能正确性的寄存器或内存的值 (如作为返回值的 a0 寄存器); (2) 约束条件, 包括是否允许待测程序修改被调用者维护的寄存器或其他内存值.

4.3 测试运行

单个待测程序的测试运行流程如图 4 所示. 系统基于 RISC-V 软件模拟器对输入的可执行文件进行译码和执行, 在识别自定义的 test_start 后, 记录初始的机器状态, 运行待测程序, 并同时跟踪机器状态的变化, 直至运行 test_end 后, 记录最终所有被更改的机器状态.

4.4 结果分析

图 5 给出了系统针对一组待测程序的测试运行和结果分析流程图. 系统通过将多个更改的机器状态进行比对, 结合用户配置的测试目标, 形成最终的测试报告. 若所有待测程序在观测目标上产生了相同的结果并满足其他约束条件, 则测试通过.

5 系统实现

本系统基于 Spike 模拟器实现. 支持扩充的指示性指令 (test_start 和 test_end) 所需的汇编器基于 LLVM[15] 后端实现, 链接器使用 RISC-V 的 gcc 工具链 (C 库使用 newlib). 向量扩展指令的指令编码及 Spike 模拟器功能实现遵循 RVV-0.9 规范.

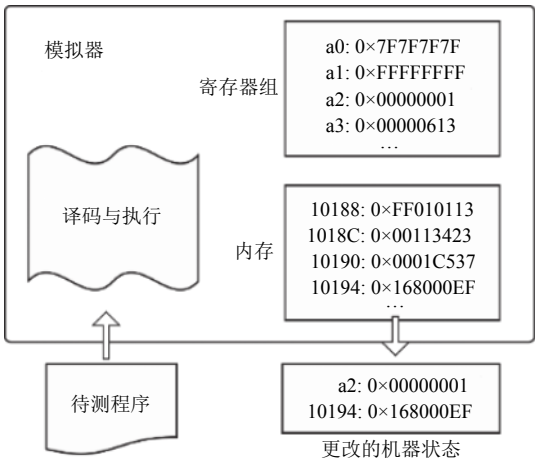

图 4 单个待测程序测试运行流程图

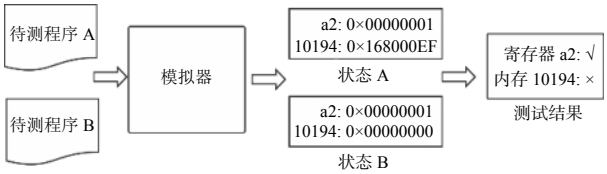

图 5 一组待测程序测试运行及结果分析流程图

系统利用伪随机数实现了支持常用数据类型的测试用例随机自动生成的函数库, 包括整型、浮点型、字符型、字符串型等. 用户通过提供参数类型, 取值范围等信息可自动生成随机的测试用例.

系统支持通过 JSON 文件配置测试目标 (示例见图 6). 其中, 决定程序正确性的观测目标可用户根据 ABI 和程序的具体功能手动设置, 默认观测目标为 a0 寄存器. 另外, 如图 6 所示, 用户可直接配置是否允许修改调用者维护的寄存器和其他内存值.

```
"target": {
    "x10": "", //设置寄存器x10为观测目标
    "0x7f7e9b18": 8
    //设置以0x7f7e9b18为起始地址, 8字节长的内存为观测目标
},
"allow-modification-of-callee-saved-register": false,
 //不允许修改被调用者维护的寄存器
"allow-modification-of-memory": false
//不允许修改除观测目标外的内存地址的值
```

图 6 测试目标配置文件 config.json 示例

系统在运行 test_end 指令时输出寄存器文件以对比寄存器值. 对于内存值, 系统采取在模拟器插桩的方式监测内存变化. 具体地, 在运行 test_start 指令后, 所有 store 类指令会额外记录目标内存地址, 并在执行

test_end 是输出记录的内存地址及内存值.

图 7 展示了针对某个测试用例的测试报告模板, 分为测试结果、观测目标、内存及被调用者维护的寄存器和其他.

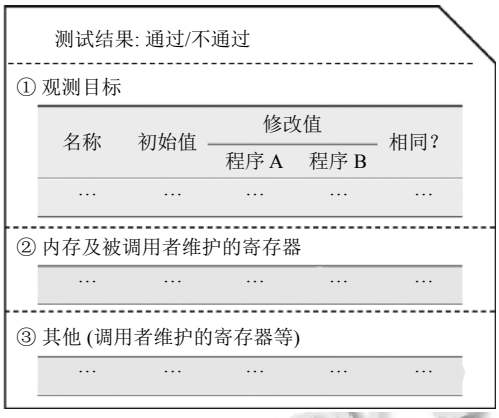

图 7 测试报告模板

6 实验与结果

本实验旨在验证语义等价性测试系统的有效性, 并以一般的测试系统为基准, 评估其测试效果和时间开销. 实验运行于 Intel Core i7-9700 CPU 以及 32 GB 内存的机器, 使用 Ubuntu 20.04 操作系统, 实验结果均为运行 5 次的平均值.

6.1 测试集构造

我们选取 5 个常用 C 函数 (见表 1) 作为系统的测试集, 每个函数具有一个标准版本、一个基于 RVV 的向量版本以及两个变异版本. 为充分验证系统的测试有效性, 我们基于每个向量版本, 为程序引入副作用: (1) 将程序中某个被调用者维护的寄存器更改为调用者维护的寄存器形成变异版 1; (2) 在程序中插入一条 store 指令更改某个指针参数对应的内存中的值形成变异版 2. 表 1 中, “9、11”表示 memcpy 的两个变异版本的代码量分别是 9 和 11.

表 1 常用 C 函数组成的汇编程序测试集

名称	描述	代码量 (行)		
		标准	向量	变异
memcpy	内存拷贝	71	9	9、11
saxpy	向量计算	36	11	11、12
strlen	字符串求长	50	11	11、13
strcpy	字符串拷贝	55	13	13、15
strncpy	字符串拷贝	40	21	21、23

表 2 基于 1000 个测试用例的一般测试和等价测试的有效性和时间对比

名称	标准-向量				标准-变异1、标准-变异2			
	一般测试	时间 (s)	等价测试	时间 (s)	一般测试	时间 (s)	等价测试	时间 (s)
memcpy	通过	0.033	通过	0.148	通过、通过	0.032、0.033	未通过、未通过	0.141、0.142
saxpy	通过	0.134	通过	0.383	通过、通过	0.133、0.133	未通过、未通过	0.377、0.379
strlen	通过	0.031	通过	0.136	通过、通过	0.029、0.028	未通过、未通过	0.131、0.130
strcpy	通过	0.035	通过	0.181	通过、通过	0.037、0.038	未通过、未通过	0.176、0.176
strncpy	通过	0.034	通过	0.175	通过、通过	0.035、0.036	未通过、未通过	0.169、0.169

6.2 系统有效性验证及时间开销评估

为验证本系统的有效性, 实验采取随机生成的 1000 个测试用例分别对 “标准-向量”“标准-变异 1”和“标准-变异 2”共 3 组程序进行语义等价性测试 (简称等价测试), 并同时运行一般测试 (仅比较函数观测目标) 作为对比.

表 2 给出了测试结果和时间开销. 语义等价的 “标准&向量”组通过了一般测试及等价测试. 但面对具有副作用的两个变异版本, 一般测试依然显示通过, 而等价测试通过更加严格的机器状态对比, 有效地捕获到副作用, 并报告测试未通过.

时间开销方面, 等价测试由于追踪、记录、对比机器状态而相比一般测试平均带来约 350% 的额外时间开销. 但由于等价测试通常针对函数级的程序, 时间开销通常是可接受的. 例如向量计算函数 saxpy 在 1000 个测试用例下仅用时 0.14 s 左右.

表 3 函数 strlen 的测试用例及测试目标

类别	具体内容
函数声明	size_t strlen(const char *str)
测试用例	“/FIDtx8('_k1\yzVILE74d$o5hDiw[~”
正确返回值	32, 即a0=0x20
测试目标	观测目标: a0 不允许修改被调用者维护的寄存器 不允许修改运行时栈以外的内存值

6.3 案例分析

本节以函数 strlen 为例, 分析系统的测试有效性.

表 3 给出了 strlen 的函数声明、某测试用例、正确的返回值以及测试目标. 该函数运行结束后, 由标准版函数计算观测得出, 观测目标 a0 应为 0x20, 即整数 32. 另外, 用户规定被调用者维护的寄存器和运行时栈以外的内存值在函数运行后不允许修改.

函数 strlen 的向量版本及相应的两个变异版本的生成方式见图 8. 其中, 变异版 1 修改了被调用者维护的寄存器, 而变异版 2 修改了内存值, 二者均不满足测试目标. 传统的基于返回值对比的测试方法因无 法捕获此类副作用而通过了测试. 基于语义等价性测试系统, 我们分别对“标准-向量”“标准-变异 1”和“标准-变异 2”这 3 组程序进行测试. 三者的测试报告见图 9. 其中, 正确的向量版函数产生了与标准版相同的观测值 (a0) , 并未引入任何违反测试目标的副作用, 成功通过测试. 变异版 1 修改了被调用者维护的寄存器 s1, 变异版 2 修改了内存地址 0x23f00 的值, 均违反了测试目标而未通过测试.

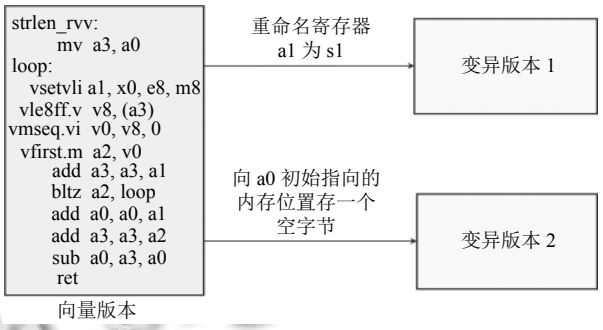

图 8 函数 strlen 向量及变异版本

测试结果: 通过

① 观测目标

名称	初始值	修改值 标准	修改值 向量	相同?
a0	0×00023f00	0×00000020	0×00000020	是

② 内存及被调用者维护的寄存器

s0	……	……	……	是
s1	……	……	……	是

③ 其他 (调用者维护的寄存器等)

v0	……	……	……	否
a1	……	……	……	否

(a) “标准-向量”

测试结果: 不通过

① 观测目标

名称	初始值	修改值 标准	修改值 变异 1	相同?
a0	0×00023f00	0×00000020	0×00000020	是

② 内存及被调用者维护的寄存器

s1	0×0002104d	0×0002104d	0×00023fa0	否
s0	……	……	……	是

③ 其他 (调用者维护的寄存器等)

v0	……	……	……	否
a2	……	……	……	否

(b) “标准-变异 1”

测试结果: 不通过

① 观测目标

名称	初始值	修改值 标准	修改值 变异 2	相同?
a0	0×00023f00	0×00000020	0×00000020	是

② 内存及被调用者维护的寄存器

0×23f00	0×744449462f	0×744449462f	0×7444494600	否
s0	……	……	……	是

③ 其他 (调用者维护的寄存器等)

v0	……	……	……	否
a2	……	……	……	否

(c) “标准-变异 2”

图 9 函数 strlen 的测试报告

此案例表明, 本系统能够通过跟踪机器状态, 结合测试目标, 有效地捕获程序产生的副作用, 提供严格有效的语义等价性测试.

7 总结与展望

本研究基于 Spike 模拟器, 设计并实现一套面向 RISC-V 的汇编程序语义等价性自动化测试系统, 通过比对不同程序运行后的全机器状态 (寄存器、内存等) , 结合用户配置的测试目标, 自动完成测试并生成测试报告. 实验表明, 本系统可成功捕获程序运行的副作用, 为语义等价性有效地提供了更为严格的测试环境.

基于本系统, 未来可通过以下 3 方面继续提高语义等价性测试的有效性和易用性: (1) 结合模糊测试技术, 更为有效地生成测试用例, 提高测试的分支覆盖率和有效性; (2) 结合缺陷定位技术[16,17], 为语句进行可疑性排序, 提高系统的易用性; (3) 采用动静结合的方式[11]对汇编语义等价性进行更为严格的测试.

参考文献

1 Pierce BC. Types and Programming Languages. Cambridge: The MIT Press, 2002.

2 Chaitin GJ. Register allocation & spilling via graph coloring. ACM SIGPLAN Notices, 1982, 17(6): 98–101. [doi: 10.1145/872726.806984]

3 张新华, 何永前. 软件测试方法概述. 科技视界, 2012, (4): 35–37.

4 单锦辉, 姜瑛, 孙萍. 软件测试研究进展. 北京大学学报 (自然科学版), 2005, 41(1): 134–145. [doi: 10.3321/j.issn:0479-8023.2005.01.020]

5 Tse AY. Software testing apparatus and method. US, 5742754. (1998-04-21).
6 王戟, 詹乃军, 冯新宇, 等. 形式化方法概貌. 软件学报, 2019, 30(1): 33–61. [doi: 10.13328/j.cnki.jos.005652]
7 King JC. Symbolic execution and program testing. Communications of the ACM, 1976, 19(7): 385–394. [doi: 10.1145/360248.360252]
8 Yan H, Sui Y, Chen SP, *et al*. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. Proceedings of the 40th International Conference on Software Engineering. New York: ACM, 2018. 327–337. [doi: 10.1145/3180155.3180178]
9 Godefroid P, Levin M, Molnar D. Automated whitebox fuzz testing. Proceedings of Network and Distributed Systems Security. San Diego: Internet Society. 2011.
10 谢肖飞, 李晓红, 陈翔, 等. 基于符号执行与模糊测试的混合测试方法. 软件学报, 2019, 30(10): 3071–3089. [doi: 10.13328/j.cnki.jos.005789]
11 Lim JP, Nagarakatte S. Automatic equivalence checking for assembly implementations of cryptography libraries. 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). Washington DC: IEEE, 2019. 37–49. [doi: 10.1109/CGO.2019.8661180]
12 Bellard F. QEMU, a fast and portable dynamic translator. Proceedings of the annual conference on USENIX Annual Technical Conference. Anaheim: USENIX Association, 2005. 41.
13 宋倩. 基于遗传算法的测试用例生成技术. 计算机系统应用, 2014, 23(11): 264–267. [doi: 10.3969/j.issn.1003-3254.2014.11.050]
14 Godefroid P, Klarlund N, Sen K. DART: Directed automated random testing. Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. New York: ACM, 2005. 213–223. [doi: 10.1145/1065010.1065036]
15 Chris L and Vikram A. LLVM: A compilation framework for lifelong program analysis & transformation. International Symposium on Code Generation and Optimization, 2004. CGO 2004. San Jose: IEEE, 2004. 75–86. [doi: 10.1109/CGO.2004.1281665]
16 陈翔, 鞠小林, 万志, 等. 基于程序频谱的动态缺陷定位方法研究. 软件学报, 2015, 26(2): 390–412. [doi: 10.13328/j.cnki.jos.004708]
17 Wong WE, Gao RZ, Li YH, *et al*. A survey on software fault localization. IEEE Transactions on Software Engineering, 2016, 42(8): 707–740. [doi: 10.1109/TSE.2016.2521368]