DOI:10.3876/j.issn.1000-1980.2021.05.014

操作系统形式化验证综述

钱汉伟[1],王承毅[2]

(1.江苏警官学院计算机信息与网络安全系,江苏 南京 210031; 2.江苏科技大学理学院,江苏 镇江 212003)

摘要:介绍操作系统验证理论、语言和工具等技术基础,阐述验证路径、精化关系验证和大规模验证等新的验证方法和理念。比较分析多个操作系统验证项目研究内容、验证方法、主要贡献以及最新进展。分析操作系统验证过程中存在的问题,认为验证成本高、验证工具局限性是制约操作系统形式化验证的关键因素,随着验证工具、框架和定理库的完善,以及深度学习等新技术的应用,操作系统更多的安全性质能够被形式化验证。

关键词:操作系统;形式化验证;定理证明;验证技术;验证方法

中图分类号:TP316 **文献标志码**:A **文章编号**:1000-1980(2021)05-0473-09

Review of formal verification for operating system

QIAN Hanwei[1], **WANG Chengyi**[2]

(1. *Department of Computer Information and Network Security*, *Jiangsu Police Institute*, *Nanjing* 210031, *China*;
2. *School of Science*, *Jiangsu University of Science and Technology*, *Zhenjiang* 212003, *China*)

Abstract: This paper introduces the technical foundations of operating system verification theory, language, and tools, and elaborates new verification methods and concepts such as the verification path, refined relationship verification and large-scale verification. The research content, verification methods, main contributions, and latest developments of multiple operating system verification projects are compared and analyzed. Meanwhile, this study also analyzes the problems in the process of operating system verification and believes that the high cost of verification and the limitations of verification tools are the key factors that restrict the formal verification of operating system. With the improvement of verification tools, frameworks and theorem libraries, new technologies such as deep learning have been improved and applied, thus more security properties of the operating system can be formally verified.

Key words: operating system; formal verification; theorem proving; verification technology; verification method

计算机软件与人们生活密切相关,从网络、手机等日常通信设备到能源、国防等关键基础设施,几乎无处不在。另一方面,由于缺少有效手段控制软件质量,使得软件的漏洞或错误几乎不可避免。每年由于软件的漏洞或者错误导致的损失达上百亿美元。操作系统是所有软件的基础平台,严格保证其可靠性和安全性具有重要价值。目前已知技术中,只有形式化方法能够保证软件不会出错,但是由于操作系统复杂度大、并发度高等问题,其形式化验证一直以来都是一件非常困难的工作。

随着定理证明助手等形式化验证工具功能更加强大,CertiKOS[1]形式化验证理论和框架更加完善,操作系统的形式化设计和验证工作进入一个新阶段,并且取得了一系列重要成果。如 seL4 等操作系统软件的内核功能正确性和无内存泄漏等性质均得到了有效证明。本文对近年来操作系统形式化验证技术、方法和研究进展进行总结,阐述操作系统形式化验证过程中对形式化验证新技术和方法的应用,分析当前操作系统形式化验证存在的问题,展望操作系统形式化验证未来的发展方向。

基金项目:国家自然科学基金(61802155)

作者简介:钱汉伟(1984—),男,高级工程师,博士研究生,主要从事形式化验证研究。E-mail:303560601@qq.com

引用本文:钱汉伟,王承毅.操作系统形式化验证综述[J].河海大学学报(自然科学版),2021,49(5):473-481.
QIAN Hanwei, WANG Chengyi. Review of formal verification for operating system[J]. Journal of Hohai University(Natural Sciences), 2021,49(5):473-481.

1 形式化验证技术

1.1 支撑理论和框架

目前操作系统的形式化验证是用定理证明的方法对操作系统的性质规约进行验证。其中,Hoare 逻辑[2]是性质规约描述的逻辑基础,当前验证工作广泛使用的分离逻辑[3]、并发分离逻辑[4]和并发精化程序逻辑[5](CSL-style relational program logic,CSL-R)等均是以 Hoare 逻辑为基础进行的扩展。霍尔逻辑证明规约用形如"{Pre}P{Post}"的 Hoare 三元组表示,通过前置条件和后置条件,比如某些不变量,放入程序要证明的性质,实现了用一组公理和一组规则描述程序代码应有的性质。分离逻辑对霍尔逻辑进行了扩展,增加了分离合取和分离蕴含谓词及相应的推导规则,能够以自然的方式描述计算过程中内存的属性和相关操作,验证指针程序也更加方便。并发精化程序逻辑将并发分离逻辑的断言语言扩展为关系型断言,更适合证明与多级中断有关的并发程序的上下文精化关系。数据精化的概念最早由 Back[6]提出,Morgan[7]发展了类似的一种精化方法。Roever 等[8]详细论述了数据精化证明方法。实际上,操作系统验证问题也是精化验证问题,操作系统从底层模型到高层规约之间的一致性验证等问题都是以精化理论为基础。携带证明代码(proof carrying code,PCC)[9]将 Hoare 逻辑应用到汇编语言的安全性质检验中,可执行代码和证明捆绑传递的方式为证明连接提供了可能性。基础性携带证明的代码(foundational proof-carrying code,FPCC)[10]将携带证明代码与复杂的推理系统都形式化在一个底层的基础逻辑中,推理系统的可靠性可以用基础逻辑保证,可进一步缩小信任基(trusted base)。在基础性携带证明代码的基础上进一步提出的开放逻辑框架[11](open certified assembly programming,OCAP)则可以将操作系统软件不同模块的验证逻辑结合起来,形成完整的验证系统,极大地保证验证框架的可扩展性,为操作系统模块化验证提供了理论基础。

1.2 形式化验证工具

工具在降低操作系统自动化验证难度、提高验证开发的效率等方面起着重要作用。操作系统的验证问题都将转成定理证明问题,解决定理证明问题最终依赖于定理证明器。目前广泛应用于操作系统验证的定理证明器有 Z3[12]、Isabelle[13]和 Coq[14]等。Z3 是微软著名的自动定理证明器,能够对验证条件进行自动证明。由于大部分证明性质的不可判定性,自动定理证明器只能证明有限的性质,更多的证明则要依赖于人机交互式定理证明器,如 Isabelle 和 Coq 等。Isabelle 是由英国剑桥大学 Paulson 和德国慕尼黑技术大学 Nipkow 于 1986 年共同开发的基于高阶逻辑的证明器。Isabelle 用函数型语言 ML 编写,使用自然演绎规则进行定理证明。证明验证工作在元逻辑(meta-logic)提供的框架下展开,所以它能支持多种逻辑系统。Coq 是法国国家信息研究所(INRIA)开发的一个基于高阶逻辑的证明器,由 OCaml 语言和少量 C 语言编写。最初来源于 1984 年 Coquand 等开发的一个综合依赖类型和多态类型的系统[15],并命名为构造演算(calculus of constructions,CoC)。后来经过扩展,增加了归纳数据类型算法公理化的一些良好性质,形成归纳构造演算(calculus of inductive co nstructions,CiC)。归纳构造演算赋予了 Coq 更强的表达能力。

可信编译器(verified compiler)是另外一类有用的工具,它严格保证了源代码和编译之后机器码之间的语义一致性,使在源码层验证的定理很容易扩展到机器码层。如果验证了用 C 语言编写的操作系统源代码,通过可信编译器生成的相应机器码也可以被认为是通过验证了,这样实现端到端的定理证明变得更加容易。法国国家信息研究所(INRIA)著名的可信编译器 CompCert C[16]可以将 Clight[17]编译成语义一致的 PowerPC、ARM 和 x86 汇编代码。编译器中大约 90% 的编译器算法(包括所有优化和所有代码生成算法)已经在 Coq 中被证明是正确的[18]。CompCertMC[19]和 CASCompCert[20]是对 CompCert 的一个扩展,前者可以保证堆栈的有限性和细粒度的堆栈权限,后者是能够对无竞争的并发 Clight 程序进行认证的单独编译。编程语言 CakeML 自带了一个经过验证的正确编译器[21]。其他的有 C Parser[22]和 AutoCorres[23]工具的组合使用,可以自动将 C 程序转换为语义一致的更高级别的 monadic 表示形式,简化了 C 代码的形式化验证。VST-Floyd 能够提供一套半自动的策略,帮助用户使用 Verifiable C 构建 C 程序的功能正确性证明[24]。

1.3 形式化语言

目前 C 语言是大多数操作系统的程序设计语言,但是 C 语言的操作性语义难以表达不同层次规约,特别是高层的抽象规约,验证时必须要将 C 语言转换成等价的形式化语言。验证 μC/OS-Ⅱ[25]时,先使用 Coq 归纳定义了 C 语言数据结构和程序语句等,再将 μC/OS-Ⅱ的 C 代码手动转换成对应的 Coq 代码进行验证。

DeepSpec 联盟[26]提倡的深度规约(deep specification)具有丰富、双边、形式化和灵活等基本性质。符合深度规约的编程语言与规约语言的界限不再明显。mC2[27]的大多数内核代码都是用 ClightX[28]编写的,ClightX 是对 Clight 的一个扩展,两者能够支持绝大部分 C 语言的语法。

Haskell[29]和 COGENT[30]是纯函数式语言,能够方便地在 Isabelle/HOL 中定义规约进行推理,在操作系统的形式化设计与验证中使用也非常多,如 seL4 的性质验证是基于 Haskell 编写的原型系统[31]。作为函数式程序设计模型的 lambda 演算与自然推理系统这样的证明演算之间具有相同的结构,因此不少人机交互定理证明器内核是用函数式语言编写,这使得函数式语言在用定理证明器时具有更好耦合性。COGENT 保持了 C 语言的操作性和精简性特点,还能够简单地表示高层抽象规约,验证高层抽象定理。COGENT 代码可以经过编译器生成相同语义的 C 代码。BilbyFs 文件[32]系统就是使用 COGENT 语言设计编写完成,并生成 C 代码编译运行。Coq 编写的代码也可以直接抽取出语义相同 Haskell、OCaml 等脚本。FSCQ 文件系统[33]和 DFSCQ 文件系统[34]就是先通过 Coq 设计编写,再自动抽取 Haskell 编译运行。

2　操作系统验证方法

2.1　操作系统精化关系验证

操作系统的验证很大一部分归结为精化关系验证,例如验证操作系统进程调度无饥饿(starvation-freedom)性质时,任何单个语句函数都不能直接推导出结论,需要更高抽象规约描述,具体的代码实现与高层抽象规约对软件行为的定义必须要保持一致(等价)。精化关系则是定义了不同抽象模型或者程序之间的等价关系。操作系统验证要选择合适的精化关系定义。精化关系对运行环境假设太强时,现实运行环境难以满足假设条件,相反假设太弱时,精化关系没有可组合性等良好性质,难以进行有效推理。当前应用较多的上下文精化关系从可观测的角度对等价关系进行定义。它是指在任何上下文环境中 A 替代 B 不产生更多的行为,并且客户程序观察不到 A 与 B 的区别。上下文精化与线性一致性是等价的[35],并发对象的功能正确性通常定义为线性一致性,因此操作系统在并发环境中的功能正确性证明等价于上下文精化关系的证明。实际上,从用户角度来看,操作系统可以看作是为上层的应用提供一组 API 接口服务的抽象虚拟机。同时操作系统是运行在硬件层上,能够提供资源管理等服务的一组中间件的总和。所以说操作系统的正确性就是要求无论应用程序是运行在抽象虚拟机上,还是运行在屏蔽硬件细节的中间件上,应用程序观察不到两者的区别。

操作系统的验证往往包括了从设计到实现等多个不同抽象层的内容,通常在低层抽象模型进行推理证明是困难的,通过层次之间的精化关系证明,低层抽象模型性质证明问题可转化为更加容易证明的高层抽象模型性质问题。

2.2　操作系统验证路径

操作系统形式化验证基本的假设是硬件形式化模型的正确性和验证性质形式化规约正确性。它们本身都是非形式化的,硬件模型正确性取决于真实芯片运行与模型模拟运行是否完全一致,规约正确性取决于人们对操作系统性质的定义是否符合其本意。两者是联系非形式化的现实世界与形式化的计算机世界的一个桥梁。从最上层的规约到最底层的硬件模型之间的形式化部分是操作系统形式化验证的内容,如图 1 所示。由于操作系统验证工作量比较大,有时会选择扩大上层或者底层的信任基,比如假设编译器是正确的,以减少实际验证工作量。

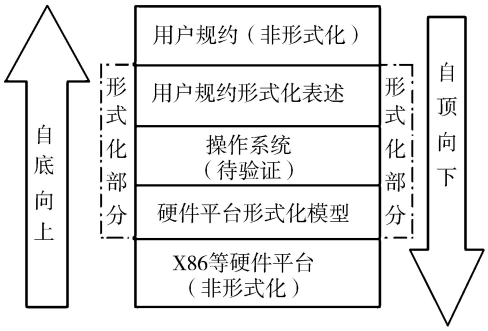

图 1　验证路径示意图

Fig. 1　Schematic map of verification path

从操作系统规约开始自顶向下验证操作系统,是将顶层用户非形式化需求变成系统形式化规约,对系统的功能需求进行定义。对形式化规约建立抽象模型,通过定理证明验证抽象模型的性质。从 CPU 机器码模型开始自底向上验证操作系统,根据 CPU 硬件指令构造机器码形式化模型,再通过建立逐步精化的模型不断向上层抽象,最终证明系统满足某些性质。安全性(safety)是指“不好”的事情(如多个进程进入临界区)从不发生,活性(liveness)是指“好”的事情(如进程申请进入临界区)最终一定会发生,Alpern 等[36]证明了任何性质均可以表示成 2 种性质的交。功能正确性是指代码正

确实现了规约的功能。这几种性质往往是操作系统验证的关键性质。

无论自顶向下还是自底向上的验证途径,最终目标是验证机器硬件的最终实现与用户规约的一致性,并以此为基础证明系统满足某种性质,即证明定理成立。

2.3 大规模软件验证

形式化验证的工作量非常大,整个过程需要人工参与较多,特别是在操作系统这样大规模验证中问题更加突出。seL4 项目中,约 8 600 行 C 代码的证明工作涉及 20 万行 Isabelle 脚本代码,花去了 11 人·a 的工作量。与软件工程解决大规模软件开发问题相类比,产生了证明工程的概念,引入软件工程的思想和经验,可以应用于解决一部分形式化方法中大规模验证的问题。在软件工程的关注点分离等原则的启发下,验证过程中很自然地引入了分层、模块化和复用等方法技术。

操作系统内核函数调用关系错综复杂,局部微小的变动可能影响整个系统。通过认证的方式将操作系统验证分为多个模块,减少单个模块验证的难度,同时将变动带来的影响限制在模块内。复用已经证明过的结果,可以尽量减少重复劳动,将已经验证的模块或者常用证明结论形成定理库,比如 Coq 的 Iris[37] 是一个高阶并发分离逻辑库,支持推理操作系统并发程序的安全性。工具能够提高验证开发的效率,大量减少验证脚本工作量,CSIRO 的 data61 为操作系统的形式化验证开发了一系列相关工具,覆盖验证工作的各个阶段[38]。DeepSpec 项目也在致力于开发适用于形式化验证的一整套工具链。

另外,在操作系统实际验证过程中,选用微内核操作系统作为验证对象是减少验证规模的一个重要方法。以 Linux 为代表的宏内核代码量早已突破 1 000 万行,seL4 等大多数微内核代码都不到 1 万行。从信任基最大限度地缩小和系统安全性等角度,选择微内核进行形式化验证也有着较为明显的优势。目前操作系统验证工作取得一定成果的 seL4、mC2、PikeOS[39]、μC/OS-Ⅱ等也都是微内核架构。

3 操作系统形式化验证研究现状

3.1 Verve

2010 年,微软基于类型化汇编语言(TAL)[40] 和 Hoare 逻辑自动化验证了 Verve 的类型安全和内存安全[41]。完整的 Verve 由硬件抽象层 Nucleus、内核和应用程序组成。Nucleus 由 TAL 编写,提供对硬件和内存的原语访问,实现内存分配、垃圾收集、多堆栈、中断处理和设备访问等功能。内核是在 Nucleus 之上构建的更完整的高级别内核服务,例如抢占线程等,内核是用 C#语言编写并最终编译成 TAL。

验证过程中,所有验证代码最终编译成 TAL,在 TAL 中手工注释(hand-annotating)前置条件、后置条件和循环不变量断言,再由 Boogie[42] 验证器根据注释的安全性和正确性规范进行自动验证。Boogie 验证器底层默认的是一个 Z3 求解器,检查断言是否可满足。

Verve 是第一个证明了类型安全和内存安全的操作系统,证明的自动化程度非常高,但是类型安全和内存安全属于比较弱的性质。如果希望证明结论更有价值,通常需要证明更加强的性质,如功能正确性的证明。

3.2 PikeOS

2009 年,德国 Verisoft XT 团队应用 VCC[43] 验证了 PikeOS 系统内核一个改变线程优先级的系统调用 p4syscall_fast_set_prio 的功能正确性,该系统调用跨越了 PikeOS 内核所有层次,从硬件相关层次到用户接口层次[44]。VCC 是微软等研发的一个用 C 语言开发的工业级验证框架,支持通过添加辅助代码(auxiliary code)和辅助状态(ghost states)验证底层并发的 C 程序。验证过程中,VCC 把 C 代码转换成中间验证语言 Boogie,然后再转换成验证条件,最后由 Z3 实现定理证明或者给出反例。

同年,Verisoft XT 团队应用 VCC 框架形式化验证了 Microsoft 的多处理器虚拟机软件 Hyper-V,Hyper-V 总共包含了 10 万行并发 C 代码和 5 000 行汇编代码。VCC 对 Hyper-V 中 20% 的代码进行了函数合约(function contract)和类型不变量等方面的验证[45]。

Verisoft XT 团队在操作系统验证方面做了小规模尝试,验证了操作系统 PikeOS 和 Hyper-V 的部分代码的部分性质。验证工作 VCC 底层使用 Z3,具有相对较高的自动化程度,但是其表达和证明能力有所不足,存在一定的局限性。验证的代码只占整个操作系统很少的一部分,而且也并未验证 Hyper-V 的功能正确性等重要属性。

3.3　seL4

2009 年,NICTA 的 seL4 团队(现在是 CSIRO 的 Data61 部门的 Trustworthy Systems 团队)验证了高性能操作系统 seL4 微内核的功能正确性和安全属性。seL4 包括 8 600 行 C 代码和 600 行汇编代码。通过形式化验证,证明了 seL4 不会有内存泄漏、空指针访问、算术异常等问题。

验证过程中将 seL4 自顶向下分为 3 个层次。最上层 M_A 是抽象层规约,描述了系统全部行为。这一层包含了系统外部接口足够多的细节,定义了系统能做什么。中间层 M_E 是执行层规约,它是一个可执行模型,也是一个实现,包括了所有数据结构和实现细节的描述。中间层是最上层的细化,进一步描述了系统怎么做。最底层 M_C 是 seL4 的 C 实现,包含了所有具体实现的细节。系统设计和验证中,中间层的 Haskell 原型是一个关键因素,它有效协调了形式化验证和系统性能之间的矛盾。操作系统开发小组使用 Haskell 作为编程语言,形式化验证小组将原型作为中间执行层导入定理证明器进行验证。

系统主要性质通过精化关系来证明。将精化关系记为 $\subseteq$,seL4 的三层模型关系可以表示为 $M_C \subseteq M_E$,$M_E \subseteq M_A$,根据精化的传递性质推出 $M_C \subseteq M_A$。进一步说明了如果一个安全性质在抽象层成立,精化关系保证它在代码实现层仍然成立。

这是首次用形式化方法对一个通用操作系统内核进行功能正确性的验证。不过为了降低验证难度,他们在验证 seL4 的内核过程中,对内核中的 IO、中断和内存管理做了简化处理,比如内核不支持带有细粒度锁的多核并发,回避了中断和抢占导致的内核并发问题。2013 年 Tessin[46] 将单核 seL4 证明结果扩展到基于 BKL 锁的集群多核情况。

3.4　CertiKOS

抽象化、模块化、层次化是操作系统软件设计与实现的重要特征,耶鲁大学的 Flint 团队提出的开放逻辑框架可以将不同计算特征和跨越不同抽象级别的程序模块验证组合,解决了难以设计单一类型的系统或程序逻辑来验证整个操作系统的问题。

2015 年,Flint 团队基于深度规约和认证抽象层(certified abstraction layer)概念,通过分层的方法(将 mCertiKOS 分解成 37 层),完成了对操作系统 mCertiKOS 单核的验证工作。一般接口的实现者很好地为调用者隐藏了自己内部实现的细节,与通常这种“浅规约”(shallow specification)不同,深度规约需要调用者了解所有有关实现的信息,这些信息是调用者的上下文环境重要的组成部分。每个认证抽象层有足够的信息使得性质的证明不再关心已被抽象的 C 语言或者汇编语言实现,可以更好进行逐层精化的证明。每个抽象层都可以证明相应的性质,并且在 mCertiKOS 将各个证明连接起来,复用了相同的证明。

2016 年,以深度规约技术为基础,Flint 团队利用 CertiKOS 构建并且验证了一个细粒度并发多核操作系统 mC2。CertiKOS 是一个构建认证并发操作系统内核的可扩展框架,分为多个认证抽象层,将困难的验证任务分解为多个简单、可自动化的小任务。mC2 包含了 6 100 行 C 代码和 400 行 x86 汇编代码,目前已经应用于地面无人车辆系统的虚拟机(hypervisor)。

相对 seL4 的证明,CertiKOS 框架在于对细粒度锁、真正并发的操作系统内核功能正确性验证取得了进一步的突破,而通常认为并发程序的验证比顺序程序要困难得多。虽然 mC2 是细粒度锁,但是在锁期间,系统是在关中断的环境下运行的,还不是抢占式操作系统。另外 CertiKOS 与 mC2 高耦合,使得 CertiKOS 不适用于已有操作系统内核的验证。

3.5　μC/OS-Ⅱ

为了解决并发环境精化关系可组合性等问题,中科大-耶鲁高可信软件联合研究中心 2012 年提出一种基于依赖-保证的模拟关系(rely-guarantee-based simulation,RGSim)作为通用的并发程序验证手段,并对并发的垃圾收集(GC)等算法进行了验证[47]。依赖-保证模拟关系仅仅保证 2 个外部程序外部可观测行为之间的包含关系,不要求程序执行路径集合之间具有包含关系。

2016 年,他们基于依赖-保证模拟关系提出一个并发分离逻辑风格的并发精化程序逻辑(CSL-R),构造了一个并发的上下文精化验证框架,验证了 μC/OS-Ⅱ的中断处理、任务调度、消息队列、信号量和互斥锁等模块,证明了该系统互斥锁不会发生优先级反转(priority-inversion-freedom,PIF)。一共验证了 1 400 行左右的 C 代码,同时将涉及的汇编代码封装为原语的方式完成底层内核代码的建模,验证的函数覆盖了 μC/OS-Ⅱ中 68% 的常用函数。他们的验证框架同样也通过分层的方法,分别定义了底层机器模型和高层机器模

型,并验证了底层和高层之间的精化关系。验证框架中开发了一系列自动证明策略,大幅度减少了证明脚本量。

这是首次构造了一个可以支持嵌套中断和抢占操作系统内核的验证框架,并验证了商用操作系统部分内核模块。目前验证工作还未实现对 μC/OS-Ⅱ内核代码全覆盖。汇编代码部分封装成原语,因此没有验证任何汇编代码,相对 CertiKOS 对汇编代码的验证工作,缺少了端到端的相关性质定理的验证。几项主要的操作系统验证工作比较见表1。

表1 操作系统验证工作比较

Table 1 Comparison of operating system verification work

验证系统	系统描述	操作系统代码量(已验证)	研究机构	验证框架	主要贡献	定理证明器	验证脚本量/万行
seL4	通用微内核	8 600 行 C 代码,600 行汇编代码	NICTA		功能正确性	Isabelle/HOL	20
Hyper-V	虚拟机	10 万行 C 代码,5 000 行汇编代码	德国航空航天中心 Verisoft	VCC		Z3	
mC2	通用微内核	6 100 行 C 代码,400 行汇编代码	耶鲁大学 FLINT	CertiKOS	多核并行	Coq	90
μC/OS-Ⅱ	可抢占、实时内核、商用	1 400 行 C 代码	中科大	未命名	可抢占、开中断	Coq	22.5
Verve	微内核		微软		类型安全	Z3	

3.6 其他相关工作

2008 年 Myreen[48] 基于 Hoare 逻辑和 VCG(verification condition generation)在机器码层对 LISP 语言解析器(interpreters)进行了验证。Hou 等[49] 基于 LEON3 CPU 指令集建立了一个形式化模型。Zhao 等[50] 定义了一个安全模型,通过验证不变量,发现了 ARINC653 标准存在端口 ID 泄露、进程 ID 泄露等 6 个安全问题。ARINC653 是国际航空电子工程委员会起草的航空应用标准软件接口,定义了隔离微内核的标准规约。Nelson 等[51] 使用 Z3 求解器,实现了 Hyperkernel 内核全部自动化验证,不过所有循环语句都被移出了内核,大大降低了内核的复杂度。2018 年,他们使用 Z3 求解器实现了 Nickel 框架,可以用来设计验证内核接口无隐蔽信道[52]。

作为操作系统最重要的组成部分,文件系统的验证工作也有不少研究。Amani 等[32] 用 COGENT 分别写了 Linux 两个文件系统的实现,并进行了验证。Chen 等[34] 验证了 FSCQ 文件系统,并证明了 FSCQ 在系统任何时候都可以重启,恢复数据后保证不会丢失数据。

4 操作系统形式化验证存在的问题

4.1 软件复杂度高

操作系统是最复杂的软件之一,它大多用 C 语言内嵌汇编语言实现,还包含许多难以分解的相互依赖的组件和程序模块。C 语言中混合汇编语言还需要进行寄存器和栈的操作,导致语义非常复杂。从代码量上看,应用最为广泛的 Linux 操作系统内核代码量早已突破千万行。微内核与宏内核相比,虽然最大程度地减小了内核代码量,但是内核不同部分之间相互依赖性却提高了很多。

内核的并发使得代码执行具有了不确定性。当用户和 I/O 并发执行时,涉及控制权从一个线程转移到另一个线程的控制机制。细粒度锁的多核并发需要复杂的自旋锁实现。这些并发机制的构造都很复杂,执行结果不确定,更难进行推理验证。与此同时,用户关心的常常是整个系统在运行过程中一直保持的全局性质,因此验证过程很可能要涉及操作系统的全部代码。

内核功能复杂性、并发的不确定性和验证性质的全局性的结合使得对操作系统功能正确性等性质的形式化验证变得十分困难。

4.2 验证工作成本高

一方面,虽然通过各种方法降低了操作形式化验证的工作量,但是总体来说仍然居高不下,验证 1 行 C 代码平均需要 25 行左右的证明代码。操作系统软件本身代码量比较大,需要进行验证的工作量更大。以 μC/OS-Ⅱ 系统内核的验证为例,1 400 行 C 代码用了 22 万行 Coq 脚本代码,其中包括验证框架约 6 万行,证

明库约 11 万行,代码证明约 4 万行,证明策略约 1.5 万行,共约 6 人・a 的工作量。而这样的工作量在操作系统形式化验证项目中并不少见。

另外一方面软件工程实践对形式化接受度不高,一般项目很少通过形式化规约定义软件,因此形式化验证工作只有全部依赖于非常专业的人员。专业人员除了需要具有深厚的形式化理论功底,丰富的证明工具使用经验,还要对验证的软件有深刻的了解,过高的门槛使得这样的专业人员非常稀缺。

此外,操作系统需要不断升级以支持适应新的硬件平台和应用程序,版本升级带来部分源代码的改动有可能会导致重新验证系统的巨大工作量。

4.3　验证工具局限性

操作系统形式化验证过程中,大量证明工作需要专业人员通过形式化工具完成。研究人员希望工具能够自动化完成大部分证明工作,也希望工具能有强大的表达能力,能够描述操作系统复杂的性质,但是工具的自动化程度与表达能力强弱往往成反比。如 Z3 等约束求解器可以对验证条件自动求解,具有较高的自动化程度,但是它很难完成操作系统复杂数据结构和软件功能正确性的全部验证。Coq 和 Isabelle/HOL 等人机交互定理证明助手能够使用表达更为丰富的高阶逻辑,但是需要手工输入脚本的工作量比较大。一阶以上的高阶逻辑公式有效性和可证性都是不可判定的,即定理在证明出来之前是无法知道是否可证,更无法找出通用的方法进行自动证明。不可判定性在理论上决定了定理证明工具在自动化方面的局限性。

可信编译器等工具为实现操作系统从抽象规约到执行代码一致性的证明发挥了重要作用。这类工具既要保证源代码和生成代码的语义一致性,还要尽可能考虑优化生成代码执行效率,往往两者难以兼顾。另一方面,编译器优化太复杂,算法很难被认证,其他非形式化的部分也难以验证。认可度较高的可信编译器 CompCert 编译生成的代码运行速度平均比 GCC 编译生成的代码要慢 15% 左右。通过 CompCert 编译产生的目标机器码,因为性能问题就很难被工业界实际广泛使用。函数式语言 COGENT 能够较好地描述操作系统高层抽象性质,但是 COGENT 编译器在生成 C 代码时几乎没有进行优化,其实用性还有待检验。

5　结　　语

操作系统形式化验证从最基本的类型安全等较弱的属性到现在验证功能正确性,高层抽象规约与底层代码精化关系证明,其巨大进步得益于形式化验证技术和工具的发展。同时,软件的形式化验证技术和工具的局限性也在制约着操作系统形式化验证工作走向最终的工业实用和普及。对这些技术和工具的研究创新,将在未来一段时间操作系统形式化验证中占有重要位置。

采用模块化验证,将复杂的操作系统分成多个更加简单的模块分别验证,有助于减轻操作系统验证的复杂度。构建验证框架和验证定理库,增强验证成果的复用,有助于减少验证工作的重复劳动。通过强化学习和规则学习算法构建定理助手智能证明策略,可以加强定理助手在某些代码和逻辑特征场景下的自动证明能力。

随着相关技术和工具的逐步成熟,实现以操作系统形式化验证为代表的证明工程工业普及与应用将成为可能,操作系统软件系统的安全等问题也将从根本上得到解决。

参考文献:

[1] The Flint Group. CertiKOS home page[EB/OL]. [2020-06-02]. http://www.cs.yale.edu/flint/certikos.

[2] HOARE C A R. An axiomatic basis for computer programming[J]. Communications of the ACM, 1969, 12(10): 576-580.

[3] REYNOLDS J C. Separation logic: a logic for shared mutable data structures[C]// Proceedings 17th Annual IEEE Symposium on Logic in Computer Science. Los Alamitos: IEEE Computer Society, 2002: 55-74.

[4] OHEARN P W. Resources, concurrency, and local reasoning[J]. Theoretical Computer Science, 2007, 375(1/2/3): 271-307.

[5] XU Fengwei, FU Ming, FENG Xinyu, et al. A practical verification framework for preemptive OS kernels[C]//International Conference on Computer Aided Verification. Berlin: Springer, 2016: 59-79.

[6] BACK R J. On the correctness of refinement steps in program development[D]. Finland: Abo Akademi University, 1978.

[7] MORGAN C. The specification statement[J]. ACM Transactions on Programming Languages and Systems, 1988, 10(3): 403-419.

[8] ROEVER W, ENGELHARDT K. Data Refinement: model-oriented proof methods and their comparison[M]. Cambridge: Cambridge University Press, 1998.

[9] NECULA G C. Proof-Carrying code[C]//Symposium on Principles of Programming Languages. New York: ACM, 1997:106-119.

[10] APPEL A W, MCALLESTER D. An indexed model of recursive types for foundational proof-carrying code[J]. ACM Transactions on Programming Languages and Systems, 2001, 23(5):657-683.

[11] FENG Xinyu. An open framework for certified system software[D]. New Haven: Yale University, 2007.

[12] Microsoft Research. Z3 Prover code[EB/OL]. [2020-06-02]. https://github.com/Z3Prover/z3.

[13] University of Cambridge, Technische Universität München. The isabelle proof assistant home page[EB/OL]. [2020-06-02]. https://isabelle.in.tum.de.

[14] INRIA. The Coq proof assistant home page[EB/OL]. [2020-06-02]. http://coq.inria.fr.

[15] BERTOT Y, CASTERAN P. Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions[M]. Berlin: Springer, 2013.

[16] INRIA. CompCert home page[EB/OL]. [2020-06-02]. http://compcert.inria.fr.

[17] BLAZY S, LEROY X. Mechanized semantics for the Clight subset of the C language[J]. Journal of Automated Reasoning, 2009, 43(3): 263-288.

[18] LEROY X. The CompCert C verified compiler documentation and user's manual. [EB/OL]. [2020-06-02]. http://compcert.inria.fr/man.

[19] WANG Yuting, WILKE P, SHAO Zhong. An abstract stack based approach to verified compositional compilation to machine code[J]. Proceedings of the ACM on Programming Languages, 2019, 3: 1-30.

[20] JIANG Hanru, LIANG Hongjin, XIAO Siyang, et al. Towards certified separate compilation for concurrent programs[C]//Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. New York: ACM, 2019: 111-125.

[21] KUMAR R, MYREEN M O, NORRISH M, et al. CakeML: a verified implementation of ML[J]. Acm Sigplan Notices, 2014, 49(1): 179-191.

[22] DATA61. C Parser code[EB/OL]. [2020-06-02]. https://github.com/seL4/l4v/tree/master/tools/c-parser.

[23] GREENAWAY D, ANDRONICK J, KLEIN G. Bridging the gap: automatic verified abstraction of C[C]//International Conference on Interactive Theorem Proving. Berlin: Springer, 2012: 99-115.

[24] CAO Qinxiang, BERINGER L, GRUETTER S, et al. VST-Floyd: a separation logic tool to verify correctness of C programs[J]. Journal of Automated Reasoning, 2018, 61(1/2/3/4): 367-422.

[25] MICRIUM. μC/OS-Ⅱ books[EB/OL]. [2020-06-02]. https://www.micrium.com/books/ucosii.

[26] Massachusetts Institute of Technology, University of Pennsylvania, Princeton University, Yale University. Deep specifciation home page[EB/OL]. [2020-06-02]. https://deepspec.org/main.

[27] GU Ronghui, SHAO Zhong, CHEN Hao, et al. CertiKOS: an extensible architecture for building certified concurrent os kernels[C]//KIMBERLY K. 12th USENIX Symposium on Operating Systems Design and Implementation. Savannah: USENIX Association, 2016: 653-669.

[28] GU Ronghui, KOENIG J, RAMANANANDRO T, et al. Deep specifications and certified abstraction layers[C]//SRIRAM K. Proceeding of the 42 nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Mumbai: ACM 2015:595-608.

[29] Haskellorg. Haskell home page[EB/OL]. [2020-06-02]. https://www.haskell.org.

[30] OCONNOR L, RIZKALLAH C, CHEN Z, et al. COGENT: certified compilation for a functional systems language[EB/OL]. [2020-06-22]. https://arxiv.org/abs/1601.05520.

[31] KLEIN G, ELPHINSTONE K, HEISER G, et al. seL4: formal verification of an OS kernel[C]//JEANNA N. Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. New York: ACM, 2009: 207-220.

[32] AMANI S, HIXON A, CHEN Z, et al. Cogent: verifying high-assurance file system implementations[J]. ACM SIGARCH Computer Architecture News, 2016, 44(2): 175-188.

[33] CHEN H, ZIEGLER D, CHAJED T, et al. Using crash Hoare logic for certifying the FSCQ file system[C]//ETHAN L. Proceedings of the 25th Symposium on Operating Systems Principles. Monterey: ACM, 2015: 18-37.

[34] CHEN H, CHAJED T, KONRADI A, et al. Verifying a high-performance crash-safe file system using a tree specification[C]//

Proceedings of the 26th Symposium on Operating Systems Principles. New York:ACM,2017: 270-286.
[35] HERLIHY M P, WING J M. Linearizability: a correctness condition for concurrent objects[J]. ACM Transactions on Programming Languages and Systems, 1990, 12(3): 463-492.
[36] ALPERN B, SCHNEIDER F B. Recognizing safety and liveness[J]. Distributed Computing, 1987, 2(3): 117-126.
[37] Aarhus University. Iris project home page[EB/OL]. [2020-06-02]. https://iris-project. org.
[38] Data61. Proof-engineering tools[EB/OL]. [2020-06-02]. https://ts. data61. csiro. au/projects/TS/proof-engineering.
[39] SYSGO. PikeOS home page[EB/OL]. [2020-06-02]. https://www. sysgo. com/products/pikeos-hypervisor.
[40] MORRISETT G, WALKER D, CRARY K, et al. From system F to typed assembly language[J]. ACM Transactions on Programming Languages and Systems, 1999, 21(3): 527-568.
[41] YANG J, HAWBLITZEL C. Safe to the last instruction: automated verification of a type-safe operating system[C]//Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation. New York:ACM,2010: 99-110.
[42] Microsoft Research. Boogie project code[EB/OL]. [2020-06-02]. https://github. com/boogie-org/boogie.
[43] DAHLWEID M, MOSKAL M, SANTEN T, et al. VCC: contract-based modular verification of concurrent C[C]//2009 31st International Conference on Software Engineering-Companion Volume. New York:IEEE, 2009: 429-430.
[44] BAUMANN C, BORMER T. Verifying the PikeOS microkernel: first results in the verisoft XT avionics project[C]//Doctoral Symposium on Systems Software Verification Real Software, Real Problems, Real Solutions. Berlin:Springer, 2009: 20.
[45] LEINENBACH D, SANTEN T. Verifying the microsoft Hyper-V Hypervisor with VCC[C]//International Symposium on Formal Methods. Berlin:Springer, 2009: 806-809.
[46] TESSIN M V. The clustered multikernel: an approach to formal verification of multiprocessor operating-system kernels[D]. Sydney:University of New South Wales, 2013.
[47] LIANG Hongjin, FENG Xinyu, FU Ming. A rely-guarantee-based simulation for verifying concurrent program transformations[C]//Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages. New York: ACM,2012: 455-468.
[48] MYREEN M O. Formal verification of machine-code programs[D]. Cambridge: University of Cambridge, 2009.
[49] HOU Z, SANAN D, TIU A, et al. An executable formalisation of the SPARCv8 instruction set architecture: a case study for the LEON3 processor[C]//International Symposium on Formal Methods. Berlin:Springer, 2016: 388-405.
[50] ZHAO Yongwang, SANAN D, ZHANG Fuyuan, et al. Refinement-based specification and security analysis of separation kernels[J]. IEEE Transactions on Dependable and Secure Computing, 2017, 16(1): 127-141.
[51] NELSON L, SIGURBJARNARSON H, ZHANG K, et al. Hyperkernel: push-button verification of an OS kernel[C]//Proceedings of the 26th Symposium on Operating Systems Principles. New York:ACM,2017: 252-269.
[52] SIGURBJARNARSON H, NELSON L, CASTROKARNEY B, et al. Nickel: a framework for design and verification of information flow control systems[C]//Operating Systems Design and Implementation. Berkeley:USENIX,2018: 287-305.

(收稿日期:2020-05-20　编辑:刘晓艳)

(上接第 424 页)

[16] 邹玉强. 土石坝沥青混凝土心墙材料蠕变特性研究[D]. 武汉:长江科学院,2017.
[17] 薛晨阳,扎西顿珠,刘斯宏,等. 考虑温度-湿度联合控制的三轴仪研制及初步应用[J]. 水利水电技术,2019,50(3):72-78. (XUE Chenyang,TASHI Dunzhu,LIU Sihong,et al. Development of a temperature-hunmidity joint control considered-triaxial appratus and its preliminary application[J]. Water Resource and Hydropower Engineering,2019,50(3):72-78. (in Chinese))
[18] 李志强,张鸿儒,侯永峰,等. 土石坝沥青混凝土心墙三轴力学特性研究[J]. 岩石力学与工程学报,2006,25(5):997-1002. (LI Zhiqiang,ZHANG Hongru,HOU Yongfeng,et al. Triaxial test study on mechanical characteristics of asphalt concrete in the core wall of earth-rock fill dam[J]. Chinese Journal of Rock Mechanics and Engineering,2006,25(5):997-1002. (in Chinese))
[19] OLDECOP L A,ALONSO E E. Theoretical investigation of the time-dependent behaviour of rockfill[J]. Geotechnique,2015,58(9):765-769.

(收稿日期:2019-12-10　编辑:胡新宇)