软件学报 ISSN 1000-9825, CODEN RUXUEW
2026,37(9):3491−3507 [doi: 10.13328/j.cnki.jos.007609] [CSTR: 32375.14.jos.007609]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

# 基于 Auto-active 与交互式集成的 L4 线程管理形式化验证*

章乐平[1], 赵永望[2,3], 王布阳[4], 李建欣[1]

[1](北京航空航天大学 计算机学院, 北京 100191)

[2](浙江大学 计算机科学与技术学院/网络空间安全学院, 浙江 杭州 310007)

[3](区块链与数据安全全国重点实验室 (浙江大学), 浙江 杭州 310007)

[4](浙江望安科技有限公司, 浙江 杭州 311100)

通信作者: 赵永望, E-mail: zhaoyw@zju.edu.cn

**摘　要**: 相较于初代微内核, 第 2 代微内核 L4 在性能和灵活性方面显著提升, 并在众多领域获得广泛应用. 操作系统内核的正确性与可靠性对系统稳定运行起着决定性作用. 聚焦于 L4 微内核的关键机制——线程管理, 对其展开形式规约与验证. 首先构建安全规约以描述安全性质, 复用标准的 L4 API 功能规约明确功能正确性, 同时自动生成基于 C++源代码的实现规约. 为缓和功能规约与实现规约间的巨大差异, 引入中间规约. 其中, 前两种规约采用 Isabelle/HOL 形式化语言编写, 后两种则以 Python 语言表达. 通过解释 (interpretation)、建立正向模拟 (forward simulation) 等方法, 精化证明各规约间的一致性. 在证明过程中, 将交互式验证与 Auto-active 验证方法相结合, 提升验证自动化能力的同时, 减少人工证明工作量. 最终发现源代码中存在 3 个违反正确性和安全性的问题, 并针对这些问题提出解决方案.

**关键词**: 形式规约; 自动生成; 精化验证; 交互式; Auto-active; L4 线程管理

**中图法分类号**: TP311

中文引用格式: 章乐平, 赵永望, 王布阳, 李建欣. 基于Auto-active与交互式集成的L4线程管理形式化验证. 软件学报, 2026, 37(9): 3491–3507. http://www.jos.org.cn/1000-9825/7609.htm

英文引用格式: Zhang LP, Zhao YW, Wang BY, Li JX. Formal Verification of L4 Thread Management Based on Auto-active and Interactive Integration. Ruan Jian Xue Bao/Journal of Software, 2026, 37(9): 3491–3507 (in Chinese). http://www.jos.org.cn/1000-9825/7609.htm

## Formal Verification of L4 Thread Management Based on Auto-active and Interactive Integration

ZHANG Le-Ping[1], ZHAO Yong-Wang[2,3], WANG Bu-Yang[4], LI Jian-Xin[1]

[1](School of Computer Science and Engineering, Beihang University, Beijing 100191, China)

[2](College of Computer Science and Technology/School of Cyber Science and Technology, Zhejiang University, Hangzhou 310007, China)

[3](State Key Laboratory of Blockchain and Data Security (Zhejiang University), Hangzhou 310007, China)

[4](Zhejiang Wonsec Technology Co. Ltd., Hangzhou 311100, China)

**Abstract**: Compared with original microkernels, L4 greatly improves performance and flexibility and is widely used in various fields. The correctness and reliability of the operating system kernel play a decisive role in the stable operation of the system. This study presents formal specifications and verification for thread management, a key mechanism of L4. First, a safety specification is developed to describe safety properties, a standard L4 API specification is reused to define functional correctness, and an implementation specification based on the C++ source code is automatically generated. To alleviate the significant differences between the requirement specification and the

* 基金项目: 国家自然科学基金“叶企孙”科学基金 (U2341212); 国家自然科学基金重点项目 (62132014); 浙江省自然科学基金重点项目 (LD24F020006)
本文由“形式化方法与应用”专题特约编辑安杰副研究员、顾斌研究员、李建文教授推荐.
收稿时间: 2025-09-08; 修改时间: 2025-10-28; 采用时间: 2025-12-08; jos 在线出版时间: 2025-12-24
CNKI 网络首发时间: 2026-03-26

implementation specification, an intermediate specification is introduced. The first two specifications are formalized in Isabelle/HOL, and the latter two are expressed in Python. The consistency among these specifications is guaranteed through refinement proofs, such as interpretation and the establishment of forward simulation. During the proofs, interactive verification and Auto-active verification are integrated, which improves the level of automation while reducing manual proof obligations. Eventually, three issues that violate correctness and safety properties are identified in the source code, and solutions are proposed for these issues.

**Key words**: formal specification; automatic generation; refinement verification; interactive; Auto-active; L4 thread management

微内核[1]运行于操作系统 (OS) 的特权模式, 遵循内核代码最小化原则. 它将线程管理、地址空间管理等操作系统核心服务封装于内核层, 而把文件系统、设备驱动等服务置于用户层. 这种设计架构的优势在于, 当某个模块出现错误时, 仅会影响相关应用程序, 而不会导致整个操作系统崩溃, 从而保障了系统的稳定性和可靠性. L4[2,3]作为第 2 代微内核, 秉承极简设计理念, 仅提供线程、地址空间和时间这 3 个基本抽象概念, 以及映射和进程间通信两种机制. 其设计旨在实现灵活性、可靠性与高性能, 整个内核实现代码仅约 1 万行 (LOC). 凭借简洁高效的架构, L4 有效解决了初代微内核[4]存在的性能问题, 在众多领域得到广泛应用[5−8], 涵盖嵌入式系统、云计算等多个重要场景.

内核作为操作系统的核心基础组件, 其正确性和稳定性直接决定了操作系统能否正常运行. 即使是极其微小的错误, 也可能导致系统执行错误, 甚至引发系统瘫痪, 严重影响系统的可用性和用户体验. 因此, 提升内核的正确性和可靠性成为操作系统领域亟待解决的关键问题. 本文聚焦于 L4 微内核的核心模块之一, 即线程管理, 并对其展开验证工作. 以先前基于 Pistachio 的 L4 API 验证[9]工作为基础, 进一步向源代码层深入探索, 旨在提出一种有效且高度自动化的 C++代码验证方法. 同时, 将线程管理机制作为验证案例, 检验该方法的有效性和实用性.

回顾以往相关研究工作[10−22], 本文对 seL4[10]、CertiKOS[13]和 Hyperkernel[14]等典型验证项目进行了分析, 主要存在 3 个方面的问题. 首先, 在 seL4 项目中, Greenaway 等人[23]使用 C-parser 工具将内核代码从 C 语言转换为 Isabelle/HOL[24]形式, 但由于本文所要验证的 L4 微内核采用 C++编写, C++的语义相较于 C 语言更为复杂, 因此如何对 C++源代码进行有效建模, 成为亟待解决的难题. 其次, 使用定理证明器手动验证内核代码的效率极低且成本高昂. 以 seL4[10]项目为例, 验证 9 000 行代码投入了 10 人年的工作量; 对于 CertiKOS[13]项目, 验证 6 000 行代码也耗费了 3 人年, 如此高的人力和时间成本, 使得探索更为高效的验证方法成为必然趋势. 第三, Nelson 等人[14]提出了相对自动化的 Push-button 验证框架, 用于 Hyperkernel 功能正确性证明, 但该框架的验证能力较弱, 尤其在面对复杂性质时尤为明显, 例如验证信息安全性质[25−27] (尽管该团队后续提出 Nickel 框架[15]以自动推理信息流安全属性, 但前提是必须手工找出足够的不变式).

针对上述问题, 本文提出了一种将交互式与 Auto-active 验证[28]相集成的方法. Auto-active 验证[28]是指在验证时要求开发者在实现层编写前后置条件 (也通常被称为函数契约 (function contract)) 或不变式等证明注释[29], 然后利用求解器自动证明. 典型工具包括 Dafny[30]、F*、Frama-C[31]等. 以下通过一系列问题及详细解答, 阐述本文集成交互式与 Auto-active 验证方法的核心思路.

● 为何选择集成交互式验证, 而不是仅采用 Auto-active 验证? 一方面, 采用 Auto-active 验证能够利用其自动化能力, 而结合交互式验证方式是为了防止验证能力出现断层, 例如, Auto-active 方式在验证信息流安全属性时表现效果较差, 验证该属性时需找到必要的不变式, 而 Auto-active 通常采用 SMT 求解器作为验证引擎, 很难引导并找到这些不变式, 交互式定理证明器在该方面有天然优势; 另一方面, 已有诸多研究在交互式定理证明器中完成了高层次 (功能) 标准规约和验证工作, 但此类工作往往耗时较长. 例如, L4 API 的验证[9]工作包含约 30 000 行代码, 耗费了 15 人月的时间. 为避免工作冗余, 这些已有的规约和证明成果可进行复用, 实现高效验证.

● 为何选择 Python 作为规约语言、Z3 求解器作为 Auto-active 验证的证明引擎? 原因在于, Python 语言应用广泛、语法简洁, 便于程序员使用, 并且常用的 SMT 求解器例如 Z3、CVC5、Yices2 为 Python 语言提供了良好的接口, 很容易通过 Python 进行调用完成自动化证明; 选择 Z3 求解器的原因在于其核心算法高效且针对软件验证进行了优化、能处理程序验证中出现的各种复杂约束, 从而提升验证能力和效率.

● 如何对 C++源代码进行建模? 本文首先将源代码编译为 LLVM 中间表示 (intermediate representation, IR) 模型, 然后进一步自动生成 Python 格式的实现规约. LLVM IR 作为中间表示形式, 具备良好的通用性和扩展性, 能

够将不同编程语言编写的代码转换为统一的中间表示形式, 为后续处理提供便利, 即通过引入 LLVM IR, 不仅能够验证 C++代码, 还能验证诸如 C、Rust 等其他语言的目标代码, 只要保证该语言能够被 LLVM 编译, 这极大地拓展了验证方法的适用性.

● 为何在功能规约基础上扩展安全规约? 一方面, 从通用性角度出发, 安全规约从功能规约中抽象出一系列属性, 为系统的安全性奠定基础, 同时安全规约能够应用于其他 L4 系列的微内核线程管理模块; 另一方面, 计划在未来将研究工作拓展到信息流安全验证领域, 并期望通过通用标准 (CC)[32]的最高级别 EAL7 认证. 要达到这一目标, 需要提供多层次的规约, 包括安全规约、功能规约以及更细粒度的规约, 而安全规约正是其不可或缺的组成部分.

● 如何保证不同语言编写的规约之间的一致性? 首先, 为缩小因引入 LLVM IR 模型 (汇编级别) 而导致的功能规约与实现规约之间的较大差距, 本文引入 Python 实现的中间规约, 该规约基于功能规约 (Isabelle/HOL 描述) 自动生成, 本文假设自动生成过程的正确性, 即中间规约和功能规约的一致性; 其次, 通过精化方法证明中间规约和实现规约的一致性, 从而保证整个规约体系的可靠性.

基于上述方法, 本研究对基于 Pistachio 0.4[33]的微内核 (L4 的一个开源实现) 的线程管理机制进行形式规约和验证. 验证过程中, 使用 Isabelle/HOL 进行交互式验证, 使用自研基于 Z3 求解器的验证框架 (LLVM IR symbolic verification engine, LISVE) 进行 Auto-active 验证. 复用以往工作中线程管理 API 的标准形式规约[9] (依据内核参考手册建立) 作为本文功能规约, 同时涵盖了该规约在安全属性方面的证明.

本研究的具体贡献主要体现在以下 3 方面.

1) 手动构建安全规约, 基于已有功能规约自动生成中间规约, 并从源代码自动生成实现规约, 这些规约全面覆盖了 L4 线程管理的各项功能, 构建了完整的规约体系.

2) 通过解释[24]、建立正向模拟[34]等验证方法确保了这些规约之间的一致性, 证明了 L4 线程管理的功能正确性和安全属性, 从而为 L4 微内核的可靠性提供了有力保障.

3) 在验证过程中, 发现了源代码中存在的 3 个违反正确性和安全性的问题, 并在本文中进行了修复, 有效提升了 L4 线程管理机制的代码质量.

本文第 1 节介绍技术背景和相关工作. 第 2 节阐述本文验证框架. 第 3 节详细描述针对 L4 线程管理机制的形式规约, 包括安全规约、功能规约、中间规约和实现规约. 第 4 节展示精化验证过程. 第 5 节分析和评估验证结果、指出发现的问题并给出建议. 最后总结全文.

## 1 背景知识

本节介绍与本文研究相关的背景知识, 首先介绍了 L4 线程管理的技术实现, 其次阐述了相关工作, 通过 Auto-active 验证、交互式验证和 Push-button 这 3 种方式分别对操作系统相关工作展开了描述.

### 1.1 L4 线程管理

线程是 L4 系统的基本执行单元, 一个或多个线程构成一个进程. 操作系统启动时, L4 微内核预先规划线程数量, 并创建 σ0 和 rootserver 等特权线程. 这些特权线程具备更高权限, 可执行更多系统服务 (系统调用).

● 标识符. 在 L4 中, 每个线程均拥有唯一的线程标识符 threadid, 分为全局标识符和局部标识符. 全局标识符在整个系统中具有唯一性, 用于标识线程; 局部标识符仅在用户模式下有效, 用于在线程地址空间内标识自身. 此外, L4 还提供 anythread 和 nilthread 两个特殊 ID, 用于特定场景.

● TCB. 线程控制块 (TCB) 是线程模块的核心数据结构, 存储线程的绝大部分信息. TCB 由内核 TCB (KTCB) 和用户 TCB (UTCB) 两部分组成. KTCB 记录线程在内核模式下可访问的信息, 包括线程标识符、线程空间、线程状态, 以及与调度、进程间通信相关的字段; UTCB 记录线程在用户模式下可用的信息, 如线程消息、错误标志、线程处理程序等.

● 调度器与分页器. L4 为每个线程配置调度器和分页器, 分别对应 KTCB 和 UTCB 中的调度器字段和分页器字段, 这两个字段本质上均为全局线程标识符. 调度器负责设置和修改指定线程的调度信息, 如优先级; 分页器在

线程执行出现缺页异常时发挥作用. 由于每个线程均拥有自身调度器 (调度器本身也是线程), 为避免出现死循环, L4 规定 rootserver 作为调度器的终止线程 (其调度器为自身), σ0 作为分页器的终止线程 (其分页器为 nilthread), 且除 rootserver 外, 其他线程不会将自身设为调度器.

● 中断线程. L4 将每个中断抽象为线程. 当中断触发时, 内核通过进程间通信 (IPC) 通知对应线程, 由该线程将中断传递给处理线程, 从而实现微内核中中断处理在用户模式下完成. 中断线程在系统启动时创建, 创建后不可删除, 仅存在启用和禁用两种状态. 启用时, 其分页器设置为处理线程; 禁用时, 分页器设为自身, 因此中断线程的分页器字段与其他线程含义存在差异.

● 线程管理操作. 线程管理操作通过系统调用 ThreadControl 实现, 该系统调用功能丰富, 涵盖线程创建、修改、删除等操作. 图 1 展示了该系统调用的部分代码片段, 通过为不同参数赋值, 可选择执行不同函数. 对于不同函数, 同一参数可能具有不同含义, 如分页器 ID, 既可用于激活线程, 也可用于更新线程分页器.

```
SYS_THREAD_CONTROL (threadid_t dest_tid, threadid_t space_tid, threadid_t scheduler_tid,
        threadid_t pager_tid, word_t utcb_location)
if (EXPECT_FALSE (!is_privileged_space(get_current_space()))) {...} // check permissions of
        current thread
if (EXPECT_FALSE (!dest_tid.is_global())) {...} // check the validity of the destination
        thread id
if (dest_tid.get_threadno()<get_kip()->thread_info.get_system_base()) {...} // handle
        interrupt threads
if (space_tid.is_nilthread()){...} //deletion
else{ // creation or modification
    ... // compute and get the thread's tcb location
    if (dest_tcb->exists()) // modification
    {
        ... // check the validity of parameters in modification mode
        if (dest_tcb->get_space() != space) {...} // modify the thread's space
        if (!pager_tid.is_nilthread()) // activate the thread or modify the thread's pager
        {
            if (!dest_tcb->is_activated()) // the thread is inactive
            {
                ... // check the validity of parameters
                if (!dest_tcb->activate(&thread_startup, pager_tid)) {...} // activation
                else {...} // modify the thread's pager
            }
        }
        if (!scheduler_tid.is_nilthread()) {...} // modify the thread's scheduler
    }
    else // Creation
    {
        dest_tcb->create_inactive(dest_tid, scheduler_tid); // allocate tcb for the thread
        ... // set the thread's attributes
        if (!pager_tid.is_nilthread()) // give the thread a pager to activate the thread
        {
            if (!dest_tcb->activate(thread_startup, pager_tid)){...} // activation
        }
    }
}
```

图 1　线程控制 (ThreadControl) 系统调用代码片段

### 1.2　操作系统验证相关工作

● Auto-active 验证. 依据 Auto-active 的验证方式, 本文总结如下: Hawblitzel 等人[17]完成了从 Ironclad 应用到内核的全栈验证, 他们使用 Dafny 定理证明器[30]为实现代码编写约 3 万行证明注释, 并证明这些应用满足其规约. Ferraiuolo 等人[18]验证了 ARM TrustZone 上安全监视器的功能正确性、完整性和保密性, 他们使用 Vale[35]语言实现软件, 并为汇编指令添加前后置条件和循环不变式以描述指令行为, 最终在 Dafny 定理证明器中完成证明. 还有一些研究致力于在软件中编写注释以辅助证明搜索, 例如在 ExpressOS[36]中验证安全不变式, 在 Verve OS[37]中验证类型安全性.

● 交互式验证. 使用交互式定理证明器进行验证的案例众多. 例如, Tuch 等人[21]和 Klein 等人[22]对 L4 微内核的虚拟内存子系统进行建模, 并验证了 3 个不变式, 随后使用 B 方法将应用程序编程接口 (API) 建模为功能规约,

但未进行验证[38]. 之后, 他们通过精化方法对 seL4 微内核的功能正确性进行了较完整的验证[21], 并进一步验证了内核的信息流安全属性[22], 建模和证明工作在 Isabelle/HOL 完成. Costanzo 等人[12]和 Gu 等人[13]对 CertiKOS 系统验证, 通过定义一系列逻辑抽象层和上下文精化关系, 在 Coq 中经过 37 层抽象, 证明了 mC2 内核的功能正确性. 在先前的工作中, 提出了一个符合 ARINC653 标准的分离内核验证框架[20], 通过逐步精化框架, 为分离内核建立了两层功能规约, 并在 Isabelle/HOL 中证明了其安全性. 在验证和代码规约审查过程中, 发现 ARINC653 标准中存在 6 个可能导致信息泄露的安全漏洞.

● Push-button 验证. 与 Auto-active 验证不同, Push-button[14−16]试图提供更高程度的证明自动化, 但放弃了通用性, 通过一系列极其严格的设计约束来换取全自动化. 例如, Nelson 等人[14]使用 Push-button 方法验证 Hyperkernel 时, 首先对验证对象 (即 Hyperkernel 本身) 做出了精心限制, 使得 SMT 求解器能够无需人工帮助即可理解并验证它, 其限制简单总结为: 1) 禁止动态内存分配; 2) 功能保持最小化, 代码量相对较小; 3) 代码结构规则化; 4) 手工建立高层清晰的规范. 经过编写规范、执行验证文件 (Python verify.py, 一键式)、自动建模和证明, 最后验证高层规范与实现规范之间的一致性, 该方式很难应用于现有系统的验证. 随后, 他们将验证属性扩展到信息流安全领域, 提出了用于验证非干扰性的 Nickel 框架, 并使用该框架验证了 NiStar、NiKOS 和 ARINC653 标准[15], 该框架仍然存在找出必要不变式难的缺点. 此外, 他们还提出了用于开发自动验证器的 Serval 框架[16].

● 分析与比较. Auto-active 验证通过在实现层编写证明注释的方式辅助验证过程, 在某些特定场景下取得了一定成果, 但对于复杂系统而言, 编写大量证明注释会显著增加开发成本和工作量, 限制了其在大规模复杂系统验证中的应用. 完全采用交互式方式能够对系统进行深入、全面的验证, 但往往需要耗费大量时间和人力, 完全通过手工书写规约和证明, 验证效率较低, 难以满足快速验证的需求. Push-button 虽然提供了较高的自动化程度, 但对验证对象有较强的限制, 无法适应复杂系统多样化的验证需求. 与上述相关工作相比, 本文提出的 Auto-active 验证与交互式验证集成的方法, 充分整合了两种验证方式的优势, 既能够防止验证能力出现断层, 又能借助自动化工具提高验证效率, 在 L4 线程管理的验证实践中取得了良好效果, 为操作系统内核验证提供了一种全新的思路和有效方法, 具有重要的理论意义和实践价值.

## 2 方法概述

本节介绍验证 L4 线程管理的整体方法. 重点阐述形式规约和核心框架 LISVE.

### 2.1 形式规约

如图 2 所示, 本文提出的 L4 线程管理验证架构总体涉及 4 个层次, 分别是安全模型层、功能规约层、中间表示层和实现规约层, 每个层次定义了对应的形式规约.

● 安全规约. 安全规约采用 Isabelle/HOL 编写, 包含抽象状态机 (abstract state machine, ASM)、变量及事件标签, 以及安全属性. 标签和安全属性采用参数化方式定义, 它们在功能规约层被实例化, 此举目的是让安全规约能适应更多 L4 微内核版本的验证, 从而提高安全规约通用性. 变量标签包含 σ0、rootserver 线程等元素, 事件标签描述 L4 线程管理机制的所有操作. 安全属性的定义依赖上述参数化变量, 具体性质参考了功能规约层已证明了的安全属性.

● 功能规约. 功能规约基于 Pistachio 内核参考手册构建, 覆盖 L4 微内核的完整 API. 其中线程管理规约沿用既有定义, 在此作为功能正确性的判定依据. 该规约不仅明确了线程创建、激活、删除等操作的接口契约 (contract), 还定义了与线程相关的状态转换的合法约束等. 为了保证与上述规约的一致性, 借助 Isabelle/HOL 证明功能规约是安全规约的一个实例, 具体采用解释方法完成实例化证明.

● 中间规约. 中间规约用于缩小功能与实现规约的差距, 该差距由两方面导致: 1) 描述语言不同; 2) 引入 LLVM IR. 因此, 借助经典控制变量的思路, 保持中间规约和实现规约的描述语言一致, 即均采用 Python 表示; 然后, 确保中间规约是功能规约在不同语言下的另一实现, 即代码结构上类似. 中间规约借助于 LISVE 自动生成, 覆盖上述功能规约中所有子句的转换能力.

● 实现规约. 实现规约基于 C++源代码自动生成, 该过程包含两个关键步骤. 第 1 步, 利用 clang 编译器将 L4 线程管理的 C++源代码编译为 LLVM IR 模型. 在编译过程中, clang 编译器对 C++代码进行全面的语法分析和语义分析等操作, 将其准确转化为 LLVM IR 指令序列, 这些指令序列详细描述了程序的执行逻辑和数据操作过程.

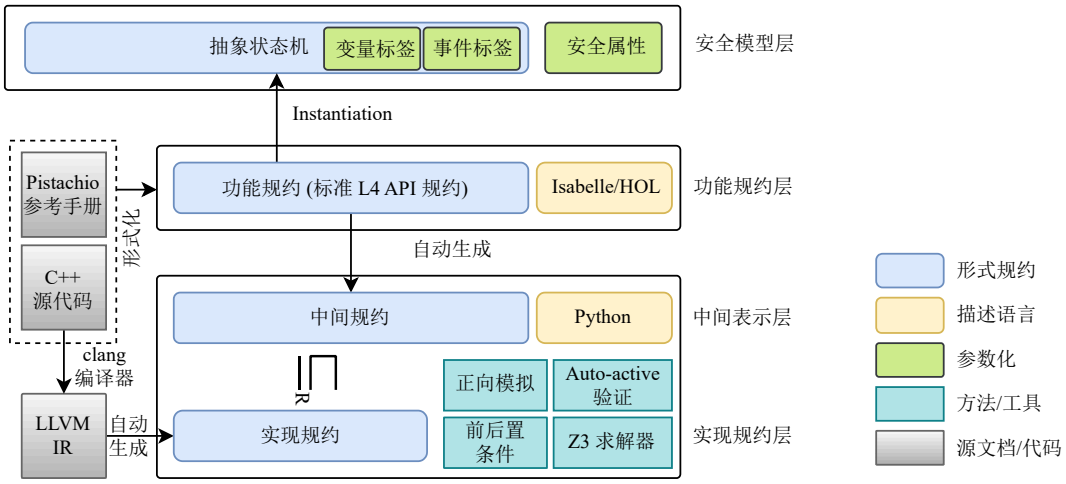


图 2 验证框架

以表 1 左侧程序作为示例, 经过 LLVM 编译指令 clang++编译后, 生成的关键 LLVM IR 程序如表 1 右侧所示. 具体编译指令为:

clang++ -g -nostdlib -O1 -S -emit-llvm example.cpp -o example.ll,

其中, -g 表示生成调试信息, -nostdlib 表示不链接标准库例如 libstdc++, -O1 表示在编译过程中会进行简单优化, 比如删除未使用的代码, -S 和-emit-llvm 两参数结合生成 LLVM IR 代码而非机器汇编. 执行指令后最后生成 LLVM IR 程序文件 example.ll. 第 2 步, 将生成的 example.ll 利用 LISVE 进一步转换为 IR 符号模型, 即实现规约, 详细过程在第 2.2 节中描述.

表 1 C++编译为 LLVM IR 程序示例

| C++代码 | LLVM IR代码 |
|---|---|
| 1. int add(int a, int b){<br>2. int sum = a + b;<br>3. return sum;<br>4. } | 1. ModuleID = ‘example’<br>2. source_filename = “example.cpp”<br>3. define i32 @add(i32 %a, i32 %b) {<br>4. entry:<br>5. %sum = add i32 %a, %b<br>6. ret i32 %sum<br>7. } |

### 2.2 LLVM IR 符号验证器 LISVE

LLVM IR 符号验证器 LISVE 工作流如图 3 所示, 首先输入 Isabelle 脚本和 LLVM IR 程序, 即本文中的需求规约以及从源代码编译后的 LLVM IR 中间表示, 然后将其转化为中间规约和实现规约, 并给出两者规约精化关系, 之后三者编码到 SMT-LIB 格式[39]的表达式并送入 SMT 求解器 Z3 中进行证明, 最后返回验证成果或给出反例.

敏捷规约语言: 考虑到形式化建模和验证门槛较高, 选择 Python 作为微内核自动化方式的规约语言, 该语言有如下优势: 1) 流行、普遍并且发展较快; 2) Python 语法简单, 使用门槛较低, 适合业界人士使用; 3) 易于结合 SMT 求解器验证; 4) 在可行性方面, 目前已存在采用 Python 语言建模和验证的案例. 在表达能力方面, Python 语言支持自定义类型、自定义函数包括递归函数、定义集合类型、函数式编程以及面向对象编程, 这些机制足够支持微内核系统的建模. 例如, 在类型定义方法上, 可以采用类来封装各类型, 将全局变量组合到一个类中从而定义状态变

量; 采用布尔表达式表示真假命题, 谓词实质为返回值为布尔类型的函数, 值的映射采用函数实现.

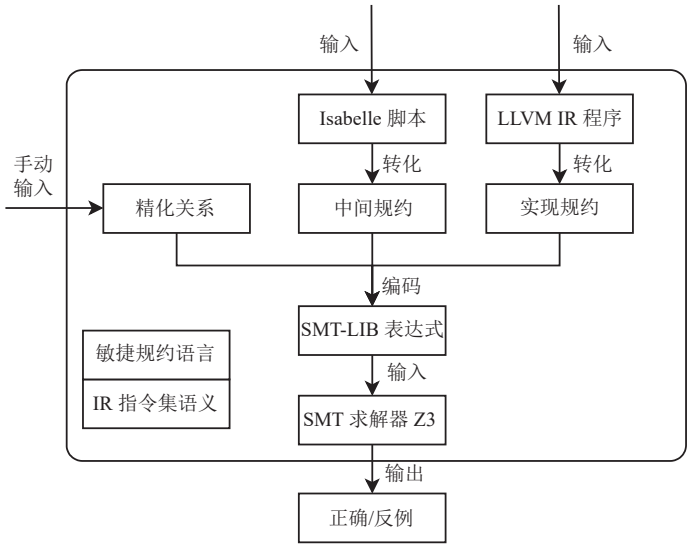


图 3 LLVM IR 符号验证器 LISVE 工作流

验证目标 LLVM IR: 本文在源代码的建模和验证方面, 选择 LLVM IR 作为验证目标, 原因主要包括 4 点: 1) LLVM IR 是平台无关的, 与高级语言例如 C/C++相比, 其语义更简单, 未定义行为更少, 与低级汇编相比, 抽象了底层硬件的细节如栈指针等机器特定细节, 仅保留高层信息; 2) LLVM IR 呈静态单赋值形式 (static single assignment, SSA), SSA 意味着程序中每个变量只能被赋值一次, 可简化数据流分析和优化; 3) LLVM IR 支持多语言编译, 将不同语言的源代码统一编译为 LLVM IR 后验证, 提高通用性; 4) LLVM 在编译过程中会将全局变量、函数等进行分块, 很容易按照指定规则将其转化为 Python 语言实现的模型. 分块具体如下.

- 模块 (module): 模块是 LLVM IR 的顶层结构, 包含一个或多个函数和全局变量. 每个模块代表一个翻译单元或一个独立的程序.
- 函数 (function): 函数是模块的基本构建块, 包含函数头和函数体. 函数体由基本块 (basic block) 组成.
- 基本块 (basic block): 基本块是一个指令序列, 包含没有分支的代码. 基本块以一个标签开始, 并以一个控制流指令 (如跳转或返回) 结束.
- 指令 (instruction): LLVM IR 的指令类似于低级汇编指令, 每条指令执行一个操作, 如算术运算、内存访问或控制流操作.

规约自动生成: 一方面, 对于中间规约的自动生成, LISVE 考虑了功能规约中线程管理机制所涉及的 Isabelle/HOL 语法和语义, 对关键字 type_synonym、record、datatype 和 definition 引导的定义进行解析, 每个定义会被解析为抽象语法树 (abstract syntax tree, AST), 然后进一步将语法树生成为 Python 代码, 在这些定义中, 处理了操作符、表达式、函数调用、自定义类型等, 其中, 运算符、表达式对应关系如表 2 所示. 另一方面, 对于实现规约的自动生成, LISVE 对 LLVM IR 程序进行解析并自动生成 Python 规约. 如图 4 所示, 在将目标程序编译为 LLVM IR 程序后, LISVE 对其分块, 得到数据结构的声明、全局变量的定义以及函数的定义这 3 个部分; 然后, 将数据结构部分定义为 IR 符号模型的类型、全局变量定义翻译为 IR 符号模型的状态字段、函数定义部分翻译为 IR 符号模型的事件; 进一步组合后形成基于状态机的 IR 符号模型, 即实现规约.

得到实现规约后, 需要提供 Python 语言实现的 LLVM IR 指令集语义以形成完整的实现规约. 本文对 LLVM IR 中 49 个指令进行了语义定义并集成到 LISVE 中, 包括控制流指令、二进制操作指令、位操作指令、内存操作指令、转换操作指令和其他操作指令. 如表 3 所示, 绿色部分表示 LLVM IR 指令集中已定义语义的指令.

以 LLVM IR 指令集中加法指令为例, 其原型为:

<result> = add nuw nsw <ty> <op1>, <op2> ; yields ty:result.

表 2　对应转换规则 (部分)

| No. | Isabelle/HOL | Python |
|---|---|---|
| 1 | **let** g **in** f | **lambda** x: f(g(x)) |
| 2 | f(g(x)) | f(g(x)) |
| 3 | **if** exp1 **then** exp2 **else** exp3 | exp2 **if** exp1 **else** exp3 |
| 4 | exp1 → exp2 | exp2 **if** exp1 **else** True |
| 5 | exp1 ≠ exp2 | exp1 != exp2 |
| 6 | exp1 = exp2 | exp1 == exp2 |
| 7 | exp1 ∧ exp2 | exp1 **and** exp2 |
| 8 | exp1 ∨ exp2 | exp1 **or** exp2 |
| 9 | exp1 := exp2 | exp1 = exp2 |
| 10 | s(\|exp1 := exp2\|) | s.exp1 = exp2 |
| 11 | set1 ∪ set2 | set1 \| set2 |
| 12 | set1 - set2 | set1 - set2 |
| 13 | x ∈ set | x **in** set |
| 14 | x ∉ set | x **not in** set |

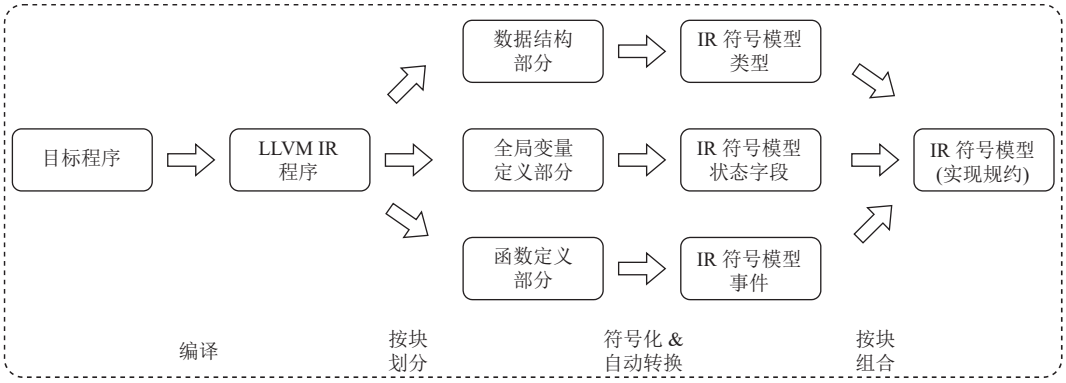


图 4　实现规约自动生成过程

表 3　LLVM IR 指令集

| 类别 | 指令名称 |
|---|---|
| 控制流指令 | ret, br, switch, indirectbr, invoke, call, resume, unreachable, catchswitch, catchret, cleanupret |
| 二进制操作指令 | add, fadd, sub, fsub, mul, fmul, udiv, sdiv, fdiv, urem, srem, frem |
| 位操作指令 | shl, lshr, ashr, and, or, xor |
| 内存操作指令 | alloca, load, store, fence, cmpxchg, atomicrmw, getelementptr |
| 转换操作指令 | trunc, zext, sext, fptrunc, fpext, fptoui, fptosi, uitofp, sitofp, ptrtoint, inttoptr, bitcast, addrspacecast |
| 其他操作指令 | icmp, fcmp, phi, select, callbr, va_arg, landingpad, catchpad, cleanuppad |

对应 Python 语言实现的语义分别如下.

```
1. def add(self, ctx, return_type, a, atype, b, btype, nuw=False, nsw=False):
2.     assert atype == return_type
3.     assert atype == btype
4.     if nuw:
5.         assert util.path_condition_implies(
6.             ctx, util.bvadd_no_overflow(a, b, signed=False), print_model=True)
```

```
7.  if nsw:
8.      assert util.path_condition_implies(
9.          ctx, util.bvadd_no_overflow(a, b, signed=True), print_model=True)
10. return a + b
```

这段代码严格校验操作数和返回值类型一致性, 并可选地通过 nuw (无符号不溢出) 和 nsw (有符号不溢出) 标志进行溢出检查: 若启用标志, 则利用路径条件分析确保运算不会溢出 (无符号或有符号), 验证通过后才执行位向量加法 $a + b$ 并返回结果.

LLVM IR 实现规约推理: 在推理基于 LLVM IR 的实现规约时, 需要考虑两个核心步骤: (1) 处理面向 LLVM IR 实现规约中的任何未定义行为; (2) 将实现规约映射到 Z3 求解器. LLVM IR 中有 3 种类型的未定义行为, 分别是立即未定义行为 (immediate undefined behavior)、未定义值 (undefined values) 和中毒值 (poison values)[40], LISVE 参考文献 [14] 对三者的处理方式.

● 立即未定义行为表示执行程序时立刻出现异常, 如除零. 在验证过程中, LISVE 会附带检查程序中可能造成异常的相关条件, 对于有除零的情况时, 必须保证除数非零, 才能执行到该程序, 从而防止立即未定义行为的执行.

● 未定义值是一种延迟的未定义行为, 表示任意值 (如从未初始化内存读取的值). LISVE 用可能取任何具体值的新符号变量表示未定义值.

● 中毒值类似于未定义值, 但如果执行有副作用的操作, 会触发立即未定义行为. LISVE 采取简单的方法: 用避免中毒的条件保护可能产生中毒值的 LLVM IR 指令, 实际上将它们视为立即未定义行为.

在排除未定义行为后, 可将 LLVM IR 类型和指令映射到 SMT. 例如, 一个 32 位 LLVM IR 整数映射到 32 位 SMT 位向量; LLVM IR 的 add 指令映射到 SMT 的位向量加法; 常规内存访问映射到未解释 (Uninterpreted) 函数操作[41].

Auto-active 验证: 为保证中间规约和实现规约一致性, 本文基于精化方法, 建立两者规约的正向模拟过程中, 证明在行为上实现规约每个行为的可达状态是中间规约的一个子集. 由于精化、等价性以及信息流安全验证基于关系型霍尔逻辑 (relational Hoare logic, RHL) 理论, 验证过程更难以自动化, 这是因为其核心是推理两个程序状态间的复杂关系, 需要同时跟踪和关联两个庞大的状态空间, 搜索证明的策略更具挑战性. 而传统霍尔逻辑是针对单个程序验证, SMT 在该方面验证能力较强, 并且考虑到验证代码为顺序程序. 因此很自然将关系型霍尔逻辑表达式描述为霍尔三元组[42] {P}c{Q}形式, 其中, P 代表前置条件, 详细描述操作执行前系统应满足的状态; c 为具体操作对应的代码; Q 是后置条件. 转换后含义为: P 表示中间规约和实现规约中的每个功能执行前状态相等关系; 两者的执行过程分别为 c1 和 c2, 即霍尔逻辑中顺序执行规则, 本文保证 c1 和 c2 的执行过程不相交, 即 c1 和 c2 操作不同的变量; Q 表示执行后的两个新状态满足的相等关系. 定义好三元组后, 选用 Z3 求解器作为证明引擎并将三元组送入其中, 然后 Z3 求解器会对其进行约束求解, 判断在前置条件 P 成立的前提下, 执行代码 c 后是否能够满足后置条件 Q, 一旦失败则给出错误提示或反例, 最终完成整个 Auto-active 验证过程. 针对约束求解过程中出现的问题, 本文采取了一系列优化措施. 面对复杂完整的函数, 通过将其分解为更多子函数、添加辅助定义等方式, 有效减少符号路径数量, 避免因路径过多导致求解超时问题; 面对线程管理与其他模块耦合度高的情况, 在验证模块组合性时, 基于函数契约验证方法, 将一些关键子函数从线程管理机制中解耦, 消除大量不必要的求解过程, 降低时间开销, 确保 Z3 求解器能够高效完成一致性证明工作, 提高验证效率和准确性.

## 3 L4 线程管理机制的形式规约

### 3.1 安全规约

本研究首先构建抽象状态机以模拟系统执行过程, 随后为该状态机添加一系列安全属性. 安全规约是一个参

数化系统, L4 线程管理的相关元素作为系统输入, 这种设计便于添加或替换元素, 能够为 L4 家族其他内核构建新的规约体系. 以下将详细阐述 L4 线程管理的抽象状态机与安全属性.

L4 线程管理的抽象状态机: 构建 L4 线程管理抽象状态机分为两个步骤. 第 1 步, 运用 locale 定义一个与上下文无关的抽象状态机 $M$, 其为一个四元组 $M=<S,E,s_0,step>$. 其中, $S$ 代表状态集合; $E$ 为事件集合; $s_0$ 是初始状态; $step$: $S\times E\to S$ 作为状态转换函数, 依据不同事件对状态集合 $S$ 进行相应更新. 第 2 步, 定义标签集 $V$. 本文将 L4 线程管理相关的核心元素定义为一系列抽象标签, 包括变量标签和事件标签, 变量标签用来记录系统运行过程中线程的状态信息, 事件标签对应线程管理的各类操作, 变量标签和事件标签具体内容如下.

- 变量标签: σ0、rootserver、intthreads、anythread、nilthread、threads、active、space、scheduler、pager.
- 事件标签: Create、CreateActive、Activate、Delete、SetPager、SetScheduler、SetStatus、ActivateInterrupt、DeactivateInterrupt.

然后将上述标签集 $V$ 添加到抽象状态机 $M$ 中, 也即构建了新的参数化系统, 在后续实例化该参数化系统时, 需输入对应于上述标签的信息, 从而形成关于 L4 线程管理的抽象状态机 $M_V$. 在 $M_V$ 中, $S$ 用来记录系统中所有线程的详细信息, 包括线程标识符、线程状态、调度器及分页器等关键数据; 事件 $E$ 涵盖 L4 线程管理机制所支持的全部操作, 例如线程创建、删除、修改等; 初始状态 $s_0$ 准确反映系统启动时特权线程的初始配置, 包含系统启动时创建的特权线程信息, 如 σ0 和 rootserver; 状态转换函数 $step$ 则严格按照 L4 线程管理规则, 实现状态的正确迁移, 例如当发生线程创建事件时, $step$ 函数会在状态集合 $S$ 中添加新线程的相关信息. 为便于后续描述安全属性, 定义了可达函数 $r$: $S\to bool$, 并建立 $step$ 与 $r$ 之间的逻辑关系, $\forall s\ e.\ r\ s\longrightarrow r(step\ s\ e)$, 即对于任意一个可达状态, 当通过事件操作使系统从该状态转换到另一个状态, 则新状态也是可达的.

安全属性: 本文共建模和证明了 18 个安全属性. 安全意味着系统在任何状态下都不会出现指定的错误, 本文将安全属性表示为可达函数引入的不变式, 以此描述系统状态在执行过程中保持不变的关系. 不变式 $inv$: $state\Rightarrow bool$ 是状态上的谓词, 以已创建的线程一定存在调度器为例, 该属性定义如下.

属性 1: 已创建的线程一定存在调度器.

$$\forall s\ t.r\ s\longrightarrow t\in \text{threads}\ s\longrightarrow(\exists sche.\ sche\in \text{threads}\ s\wedge \text{scheduler}\ s\ t=\text{Some}\ sche),$$

其中, 采用 $r\ s$ 作为前提, 很容易表示系统状态满足安全属性的公式, 蕴含式 $r\ s\longrightarrow inv\ s$ 是通用形式, 也是最终要验证的目标定理, 当每个状态都满足该属性时, 该蕴含式成立. 其他部分安全属性定义如下.

属性 2: 活跃线程一定是已创建线程的子集, 并且活跃线程一定存在分页器.

$$\forall s\ t.\ r\ s\longrightarrow t\in \text{active}\ s\longrightarrow t\in \text{threads}\ s\wedge \text{pager}\ s\ t\neq \text{None}.$$

属性 3: 特权线程 σ0、rootserver 和中断线程 intthreads 都是活跃线程.

$$\forall s\ t.\ r\ s\longrightarrow t\in\{\sigma 0,\text{rootserver}\}\cup \text{intthreads}\longrightarrow t\in \text{active}\ s.$$

属性 4: 线程的调度器和分页器一定是活跃线程.

$$\forall s\ t\ sche.\ r\ s\longrightarrow \text{scheduler}\ s\ t=\text{Some}\ sche\longrightarrow sche\in \text{active}\ s.$$

属性 5: 线程是否能被创建取决于系统配置表里是否存在该线程.

$$\forall s\ t.\ r\ s\longrightarrow t\in \text{threads}\ s\longrightarrow t\in \text{THREADS CONFIG}.$$

### 3.2 功能规约

功能规约依据 Pistachio 的内核参考手册编制, 是 L4 微内核 API 的标准定义, 它们明确了线程管理 API 的功能特性与行为准则, 也是判断线程管理功能实现正确性的重要依据. 在功能规约中, 详细描述线程管理 API 的输入参数、输出结果及对系统状态的影响. 以线程创建为例, 规定创建线程所需参数, 如线程标识符、线程的调度器等; 说明线程创建成功后系统状态的更新操作, 包括在 TCB 中添加新线程信息、分配系统资源等; 定义线程创建失败的情形及返回值, 便于开发者判断异常并处理, 确保线程创建功能正确稳定运行. 以下给出创建线程的功能规约描述.

```
1. Create_req:: SysConfig ⇒ State ⇒ globalid ⇒ spaceName ⇒ globalid ⇒ State
2. Create_req config s gno sp sche =
3. if gno ∈ GLOBAL_GNO config ∧ sche ∈ GLOBAL_GNO config
4. then
5.       (let s = … # 创建地址空间
6.          s = s⦇threads:=threads s ∪ {gno},
7.          space:=(space s)(gno:=Some sp),
8.          sched:=(sched s)(gno:=Some sche),
9.          status:=(status s)(gno:=Some Aborted)⦈
10.      in …)
11. else s
```

在第 5 行中, …表示创建地址空间的相关内容. 本文仅建模和验证线程管理模块, 关于线程模块中涵盖其他模块的情况, 本文处理方式是: 通过前后置条件 (函数契约) 模拟其他模块行为的所有可能性, 而不实际对这些行为建模.

### 3.3 中间规约

如前文所述, 由于功能规约与实现规约在语言类型和抽象层次方面存在显著差异, 中间规约应运而生, 其核心作用是在二者之间搭建一座沟通桥梁, 降低验证工作的难度与复杂性. 中间规约采用 Python 语言表示, 由功能规约自动生成得到.

以上述创建线程函数为例, 从功能规约自动生成如下中间规约 (注释除外).

```
1. def Create_req_imm(config: SysConfig, s: State, gno: globalid, sp: spaceName, sche: globalid) → State:
2.     """
3.     对应 Isabelle 的 Create_req 函数实现
4.     Args:
5.         config: 系统配置
6.         s: 当前状态
7.         gno: 要创建的全局 ID
8.         sp: 空间名称
9.         sche: 调度器 ID
10.    Returns:
11.        更新后的状态
12.    """
13.    # 检查 gno 和 sche 是否在配置的全局 ID 集合中
14.    if gno in config.GLOBAL_GNO and sche in config.GLOBAL_GNO:
15.        # 创建状态的深拷贝
16.        new_s = copy.deepcopy(s)
17.        # 更新线程集合
18.        new_s.threads.add(gno)
19.        # 更新空间映射
20.        new_state.space[gno] = sp
21.        # 更新调度映射
```

```
        new_state.sched[gno] = sche
        # 更新状态映射
        new_state.status[gno] = Aborted
        return new_state
    else:
        # 条件不满足时返回原始状态
        return s
```

容易看出, 在生成规约时, 对每个函数名自动添加后缀*_imm*, 针对 record 类型的变量更新操作时, 会自动生成带前缀 new_的新变量, 从而记录新的状态.

### 3.4 实现规约

利用 LLVM 提供的 clang 编译器将 C++源代码编译为 LLVM IR 后, 通过工具将 LLVM IR 转换为实现规约. 仍以创建线程为例, 如下给出了其实现规约的关键代码. 在该定义中, *func_create_inactive* 表示创建线程函数 (前文创建线程函数实质是创建非活跃的线程), *func*_为自动生成实现规约时自动添加的前缀, 参数 *ctx* 表示记录全局上下文, 即类似于状态, 参数 arg1、arg2、arg3 分别表示调用该函数的对象、目标线程标识符、调度器标识符, arg2 和 arg3 分别对应于功能规约中的 *gno* 和 *sche*, 而 arg1 对应于状态字段中的当前线程字段.

```
def func_create_inactive(ctx, arg1, arg2, arg3):
    …
    # 检查类型有效性
    assert itypes.has_type(ctx, arg1, itypes.parse_type(ctx, '%tcb_t*')), 'Incorrect size'
    # 保存调用创建线程函数的对象
    ctx.stack["%0"] = arg1
    # 检查类型有效性
    assert itypes.has_type(ctx, arg2, itypes.parse_type(ctx, 'i64')), 'Incorrect size'
    # 记录线程标识符
    ctx.stack["%1"] = arg2
    # 检查类型有效性
    assert itypes.has_type(ctx, arg3, itypes.parse_type(ctx, 'i64')), 'Incorrect size'
    # 保存调度线程标识符
    ctx.stack["%2"] = arg3
    …
```

需要说明的是, 在功能规约、中间规约和源代码中, 创建线程函数的参数还涉及地址空间 *sp*、分页器 pager 等, 而在实现规约中不涉及这些参数, 这是由于: 考虑在后续验证过程出现的模块间高耦合以及验证超时等问题, 将部分功能分解为若干子功能, 其中, 创建线程功能实际为线程管理系统调用的子功能之一, 生成的实现规约与 *sp* 和 pager 等参数无关, 因此实现规约中并未体现. 在精化验证阶段, 也无需建立这些无关变量的精化关系.

## 4 精化验证

本节将介绍 L4 线程管理的验证工作. 在从源代码自动生成实现规约后, 核心任务包含两个: 其一, 证明实现规约的行为与中间规约在功能正确性方面保持一致; 其二, 证明功能规约是安全规约层在安全属性方面的实例. 本文分别借助建立正向模拟和解释这两种方法来达成上述目标.

### 4.1 基于解释的实例化证明

解释方法被用于证明安全规约与功能规约之间的一致性. 回顾解释方法的总体思路, 将功能规约定义为安全规约 ($M_V$ locale) 的一种解释 (Isabelle 中采用 interpretation 关键字进行定义), 具体表示为:

$$M'_V = \textbf{interpretation}\ M_V\{\text{s0/s0\_req}, \text{step/step\_req}, \sigma 0/\sigma 0\_\text{req}, \text{rootserver/rootserver\_req}, \ldots/\ldots\},$$

其中, 后缀为 req 的变量在功能规约中定义, .../... 省略了安全规约中其他标签的解释. $M_V$ 中的每个变量都被解释为具体的变量. 此时, 需要确保这种解释是可行的, 即要证明两个关键性质: 1) 不存在类型冲突; 2) $M_V$ 中的所有假设必须在实例 $M'_V$ 中得到证明. 第 1 个证明可由 Isabelle/HOL 自动完成, 它会强制确保类型匹配, 若不满足则报告错误. 在此, 解释方法在一定程度上减轻了证明负担, 因为与正向模拟相比, 仅需保证变量类型匹配无冲突, 无需额外定义精化关系并手动证明. 对于第 2 个证明, 其重点在于验证功能规约中具体事件是否满足安全规约. 由于功能规约中功能安全属性已得到证明, 而安全规约是功能安全属性的抽象, 因此采用已有的功能安全属性证明引理并结合一些自动化策略, 如 auto、simp 等即可证明. 进一步地, 得到如下定理.

**定理 1**. 功能规约是安全规约的一个实例.

### 4.2 基于 Auto-active 验证的正向模拟证明

下面先给出正向模拟的定义并构建精化关系, 然后阐述符号路径分解和基于函数契约的验证方法.

正向模拟的过程如图 5 所示, 可形式化为 $FSim\ R\ A\ C \longrightarrow A \sqsubseteq C$, 其中 $A$ 为中间规约, $C$ 为实现规约, $R$ 为二者的精化关系, 这意味着一旦建立正向模拟 $FSim\ R\ A\ C$, $C$ 则是 $A$ 的一个精化. 推理正向模拟需要确保: 1) 初始状态满足 $R$; 2) 事件驱动的状态转换过程保持 $R$, 形式化表述如下:

$$FSim\ R\ A\ C \equiv R\ (Init\ A)(Init\ C) \wedge (\forall S_A\ S_C\ E_A\ E_C.\ R\ S_A\ S_C \longrightarrow R\ (step\ A\ S_A\ E_A)(step\ C\ S_C\ E_C)).$$

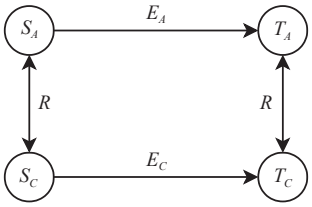


图 5　正向模拟

对于创建线程事件而言需满足:

$R\ s\_imm\ s\_imp \longrightarrow$

$$R\ (Create\_req\_imm\ \text{Config}\ s\_imm\ gno\ sp\ \text{sched})\ (func\_create\_inactive\ s\_imp\ current\ gno\ \text{sched}),$$

其中, $s\_imm$, $s\_imp$ 是中间规约和实现规约中的全局状态; Config 为静态配置表; $sp$ 为地址空间标识符, 此处可为任意值, 仅作参数, 与该函数无关; $current$ 变量获取了实现规约中当前线程 TCB.

精化关系: 建立正向模拟需要提供表明两规约数据一致的关系, 在所有状态字段上建立对应等式, 字段包括 globalid、status、scheduler、pager、error、handler 等. 例如, 中间规约的调度器字段 $s\_imm.scheduler$ 与实现规约的调度器字段值相等.

$$s\_imm.scheduler[dest\_tid] = \text{dest\_tcb.getelementptr}(ctx, offset\_sched, \ldots).read(ctx),$$

其中, getelementptr 是 LLVM IR 内存操作指令之一, 根据偏移值来找到指定的位置, 这里试图通过 $offset\_sched$ 找到 dest_tcb 中的 scheduler 字段. 在证明两规约的所有函数都建立该关系后, 可得出如下定理.

**定理 2**. 实现规约是中间规约的精化.

进一步地, 根据规约在正确性与安全属性上的传递性, 满足如下定理成立.

**定理 3**. 实现规约满足功能正确性与安全属性.

符号路径分解: 对于一些复杂函数, 约束求解可能会生成大量符号路径, 导致超时或失败. 本文提供的方法主

要是符号路径分解, 具体包括 3 个措施: (1) 根据系统调用参数, 将功能点划分为更细的子功能; (2) 进一步划分子功能点, 对条件表达式的各个分支进行拆分, 若顺序语句中存在逻辑表达式, 对所有的可能情况进行拆分; (3) 循环表达式采用有界模型检查 (bounded model checking, BMC) 方式将非确定次数的循环表达式转为确定性次数的循环表达式, 进一步将其转为深度为该次数的条件表达式, 最后继续按照第 2 步中对条件表达式进行拆分. 在少数情况下, 将构建的约束表达式放入 Isabelle/HOL 中引导证明.

基于函数契约的验证: 针对模块组合验证问题, 本文主要采用了基于函数契约的验证方法, 为线程管理机制中涉及的其他模块定义契约, 即模块的假设和承诺. 假设 (assumptions) 描述模块对外部环境的期望, 承诺 (guarantees) 则描述模块在满足假设前提下的行为. 通过为每个模块定义清晰的契约, 可以在模块的接口上进行组合验证, 确保模块之间的交互不会违反各自的契约. 组合验证时, 首先验证各模块分别满足各自的契约, 然后在整个系统层面上验证模块契约是否能够相互兼容. 以创建操作为例 (见第 4.2 节 *Create_req*), 地址空间创建被嵌入定义中, 在证明时, 实际上移除该操作, 替换为某些结果状态. 可分析地址空间的创建过程可能出现以下情况.

1) 前提是地址空间已存在, 承诺创建操作失败.

2) 前提是地址空间不存在, 承诺创建成功.

3) 前提是地址空间不存在, 承诺创建操作异常.

对应 3 种契约, 形式化描述为: 前置条件为已创建空间的集合 spaces 是否存在 sp, 后置条件可总结为地址空间标识符是否应加入记录所有已创建空间的集合 spaces, 若是, 则记录该新空间中线程的集合 space_threads 应设为非空集合{*gno*}, 即创建 *gno* 时, 地址空间 sp 也会被创建, 且 *gno* 的地址空间为 sp, 否则不执行任何操作.

## 5 验证结果与评估

本节分析和评估本文建模和验证工作. 首先统计了代码情况, 然后展示验证过程中发现的问题, 并给出解决办法或建议, 最后讨论了本文工作优势.

代码统计: 本文采用 Isabelle/HOL 构建安全规约, 复用 L4 API 的标准规约作为基于 Pistachio 的 L4 线程管理的功能规约, 并通过解释方法证明了这两个规约之间的一致性. 随后, 从功能规约自动生成了中间规约, 从 C++源代码自动生成了实现规约, 建立了中间规约和实现规约两者之间的正向模拟过程, 中间规约与实现规约一致性. 经过一系列证明, 断言实现规约满足功能正确性和功能安全属性要求. 同时, 由于交互式验证与 Auto-active 验证的集成应用, 证明效率得到显著提升. 规约与验证工作的详细投入情况如表 4 所示. 除功能规约外, 我们花费约 2.5 人月开发形式规约, 规约代码量约 6 300 行. 在证明过程中, 几乎无需手动编写大量证明脚本. 需注意的是, 功能规约的证明统计数据仅包含实例化证明细节, 实例化证明中主要任务是对安全规约的推理, 而这些安全规约是功能规约层功能安全属性 (已证明) 的抽象, 因此仅需 30 行代码完成实例化证明. 此外, 尽管中间规约由功能规约自动生成, 但为了与实现规约的一致性, 不改变行为语义前提下, 后续进行了部分调整或优化.

表 4　规约和证明代码统计

| 规约类型 | 规约 | | | 证明 | | |
|---|---|---|---|---|---|---|
| | 定义数 | 代码行数 (k) | 耗时 (人月) | 定理数 | 代码行数 | 耗时 (人月) |
| 安全规约 | 18 | 0.1 | 1 | — | — | — |
| 功能规约 | 10 | 1.2 | 1 | 1 | 30 | — |
| 中间规约 | 10 | 1 | 1.5 | — | — | — |
| 实现规约 | 10 | 4 | — | 10 | — | 3 |
| 总计 | 48 | 6.3 | 3.5 | 11 | 30 | 3 |

已发现的问题: 在对 L4 线程管理机制进行建模与验证时, 发现了以下 3 个问题. 以下对其进行详细介绍, 同时对每个问题提出了解决办法或建议.

1) 越界访问风险: 当前源代码未设置任何策略对目标线程 ID 的范围进行限制, 这可能导致线程能够访问线

程控制块 (TCB) 区域之外的内存段. 为解决该问题, 建议在源代码中添加条件表达式, 明确规定仅当给定线程 ID 的值不超过 TCB 区域内线程 ID 的最大值时, 相关操作才能执行, 以此限制线程访问范围, 避免越界访问带来的安全隐患.

2) 死锁隐患: 回顾第 1.1 节中 L4 线程的背景知识可知, 调度器最终应终止于 rootserver 线程. 然而, 当前源代码中存在允许调度环出现的情况, 例如线程 a 的调度器设置为线程 b, 而线程 b 的调度器又可被设置为线程 a. 这种调度环的存在会导致死锁, 阻碍系统正常运行. 一个简单有效的解决方案是将每个线程的调度器统一设置为 rootserver 线程, 从而打破潜在的调度环, 确保调度机制的正常运转.

3) 非法删除: 特权线程能够删除任意普通地址空间中的线程, 这一操作可能引发系统异常. 例如, 若待删除的非特权线程 A 是另一个线程 B 的调度器, 当线程 A 被成功删除后, 线程 B 将无法找到其调度器, 进而引发异常. 此外, 这种删除操作还可能导致与调度机制相关的 TCB 字段无法正常修改, 违反正确性规约. 为此, 建议在删除线程之前, 先检查该线程与其他线程之间的依赖关系, 避免因不当删除引发系统异常和违反规约的情况发生.

讨论分析: 以下从多个方面对本研究工作进行讨论: 首先, 通过自动生成源代码模型, 并借助 Z3 求解器进行证明, 极大地减轻了证明负担, 提高了验证效率. 其次, 采用将目标代码转换为 LLVM IR 的方法, 这一技术手段能够处理多种语言实现的源代码建模问题, 包括 C、C++、Java 等. 而目前大多数现有研究工作仅能处理单一语言的代码, 例如 seL4 项目使用 C-parser 工具将 C 代码转换为 Isabelle/HOL 中的模型, 且仅涵盖部分 C 语言语义. 相比之下, 本研究方法在处理源代码的能力上具有明显优势, 为更广泛的操作系统内核验证工作提供了可能. 第三, 本研究开发的大部分脚本具有良好的可复用性, 主要体现在两个方面: 1) 安全规约中定义的属性独立于功能规约, 可作为其他 L4 微内核线程管理的规约进行复用, 增强了规约的通用性和扩展性; 2) 安全规约中的标签具有抽象性, 使其更易于实例化为具体规约, 方便在不同场景下快速应用, 减少重复开发工作, 提高研究效率和成果的可移植性.

方法限制: 本文方法建立在以下假设基础上: 1) 中间规约和实现规约自动生成过程正确; 2) 编译后 LLVM IR 程序语义与 C++源代码语义一致, 即 LLVM IR 编译器可靠; 3) 不考虑并发程序. 此外, 关于 L4 线程管理程序中, 不涉及循环表达式, 在本文集成框架中针对循环表达式的处理是采用有界模型检查 (BMC) 方式, 即仅能保证在有限次数内程序的正确性.

## 6 总 结

在本文中, 针对 L4 微内核的线程管理机制, 提出一种将 Auto-active 验证与交互式验证相集成的方法, 并对其进行全面的形式规约和验证工作. 研究构建了安全规约、功能规约、中间规约和实现规约这 4 个层次的规约体系, 通过精化验证方法证明了各规约之间的一致性, 成功发现并修复了源代码中存在的 3 个违反正确性和安全性的问题, 为 L4 微内核线程管理的可靠性提供了有力保障. 本文通过将交互式验证与 Auto-active 验证相结合, 充分发挥了两种验证方式的优势. 一方面, 能够利用交互式定理证明器的优点, 防止验证能力断层, 并有效复用已有的规约和证明成果, 避免重复劳动, 提高验证工作效率; 另一方面, 借助 Z3 求解器强大的自动求解能力, 显著提升了验证的自动化程度, 大幅减少了人工证明工作量, 降低了验证成本. 同时, 采用自动生成实现规约的方法, 通过引入 LLVM IR 模型, 为后续验证其他编程语言编写的代码奠定了基础, 使该验证方法具有良好的扩展性和通用性.

然而, 本研究仍存在一些有待改进之处, 也为后续研究指明了方向. 一方面, 尽管在一定程度上缓解了模块耦合对验证效率的影响, 但对于更为复杂的系统, 模块之间的交互和依赖关系将更加错综复杂, 如何进一步优化验证方法, 提升对复杂系统的验证效率, 仍是需要深入研究的重要课题. 另一方面, 目前研究主要集中于功能正确性和部分安全属性的验证, 未来可将研究范围拓展至更广泛的安全领域, 如信息流安全验证等, 进一步完善 L4 微内核的安全性体系. 此外, 期望将该验证方法应用于 L4 系列的其他内核以及更多不同类型的操作系统内核, 为提高操作系统整体的正确性和可靠性提供更强大的技术支持, 推动操作系统验证领域的发展与进步.

## References

[1] Liedtke J. On micro-kernel construction. ACM SIGOPS Operating Systems Review, 1995, 29(5): 237–250. [doi: 10.1145/224057.

224075]
[2] Liedtke J. Toward real microkernels. Communications of the ACM, 1996, 39(9): 70–77. [doi: 10.1145/234215.234473]
[3] L4Ka Team. L4 experimental kernel reference manual. Version X.2. 2011. https://www.l4ka.org/l4ka/l4-x2-r7.pdf
[4] Accetta MJ, Baron RV, Bolosky WJ, Golub DB, Rashid RF, Tevanian A, Young M. Mach: A new kernel foundation for UNIX development. In: Proc. of the 1986 USENIX Summer Conf. USENIX Association, 1986. 93–113.
[5] Heiser G, Elphinstone K. L4 microkernels: The lessons from 20 years of research and deployment. ACM Trans. on Computer Systems (TOCS), 2016, 34(1): 1. [doi: 10.1145/2893177]
[6] Elphinstone K, Heiser G. From L3 to seL4 what have we learnt in 20 years of L4 microkernels? In: Proc. of the 24th ACM Symp. on Operating Systems Principles. Farminton: ACM, 2013. 133–150. [doi: 10.1145/2517349.2522720]
[7] Härtig H, Hohmuth M, Liedtke J, Schönberg S, Wolter J. The performance of μ-kernel-based systems. In: Proc. of the 16th ACM Symp. on Operating System Principles. Saint Malo: ACM, 1997. 66–77. [doi: 10.1145/268998.266660]
[8] Lackorzynski A, Warg A. Taming subsystems: Capabilities as universal resource access control in L4. In: Proc. of the 2nd Workshop on Isolation and Integration in Embedded Systems. Nuremburg: ACM, 2009. 25–30. [doi: 10.1145/1519130.1519135]
[9] Zhang LP, Zhao YW, Li JX. A comprehensive specification and verification of the L4 microkernel API. In: Proc. of the 30th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Luxembourg City: Springer, 2024. 217–234. [doi: 10.1007/978-3-031-57249-4_11]
[10] Klein G, Elphinstone K, Heiser G, Andronick J, Cock D, Derrin P, Elkaduwe D, Engelhardt K, Kolanski R, Norrish M, Sewell T, Tuch H, Winwood S. seL4: Formal verification of an OS kernel. In: Proc. of the 22nd ACM SIGOPS Symp. on Operating Systems Principles. Big Sky: ACM, 2009. 207–220. [doi: 10.1145/1629575.1629596]
[11] Murray T, Matichuk D, Brassil M, Gammie P, Bourke T, Seefried S, Lewis C, Xin G, Klein G. seL4: From general purpose to a proof of information flow enforcement. In: Proc. of the 2013 IEEE Symp. on Security and Privacy. Berkeley: IEEE, 2013. 415–429. [doi: 10.1109/SP.2013.35]
[12] Costanzo D, Shao Z, Gu RH. End-to-end verification of information-flow security for C and assembly programs. In: Proc. of the 37th ACM SIGPLAN Conf. on Programming Language Design and Implementation. Santa Barbara: ACM, 2016. 648–664. [doi: 10.1145/2908080.2908100]
[13] Gu RH, Shao Z, Chen H, Wu XN, Kim J, Sjöberg V, Costanzo D. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In: Proc. of the 12th USENIX Symp. on Operating Systems Design and Implementation (OSDI 2016). Savannah: USENIX Association, 2016. 653–669.
[14] Nelson L, Sigurbjarnarson H, Zhang KY, Johnson D, Bornholt J, Torlak E, Wang X. Hyperkernel: Push-button verification of an OS kernel. In: Proc. of the 26th Symp. on Operating Systems Principles. Shanghai: ACM, 2017. 252–269. [doi: 10.1145/3132747.3132748]
[15] Sigurbjarnarson H, Nelson L, Castro-Karney B, Bornholt J, Torlak E, Wang X. Nickel: A framework for design and verification of information flow control systems. In: Proc. of the 13th USENIX Symp. on Operating Systems Design and Implementation (OSDI 2018). Carlsbad: USENIX Association, 2018. 287–305.
[16] Nelson L, Bornholt J, Gu RH, Baumann A, Torlak E, Wang X. Scaling symbolic evaluation for automated verification of systems code with Serval. In: Proc. of the 27th ACM Symp. on Operating Systems Principles. Huntsville: ACM, 2019. 225–242. [doi: 10.1145/3341301.3359641]
[17] Hawblitzel C, Howell J, Lorch JR, Narayan A, Parno B, Zhang DF, Zill B. Ironclad APPs: End-to-end security via automated full-system verification. In: Proc. of the 11th USENIX Symp. on Operating Systems Design and Implementation. Broomfield: USENIX Association, 2014. 165–181.
[18] Ferraiuolo A, Baumann A, Hawblitzel C, Parno B. Komodo: Using verification to disentangle secure-enclave hardware from software. In: Proc. of the 26th Symp. on Operating Systems Principles. Shanghai: ACM, 2017. 287–305. [doi: 10.1145/3132747.3132782]
[19] Zhao YW, Sanán D, Zhang FY, Liu Y. Reasoning about information flow security of separation kernels with channel-based communication. In: Proc. of the 22nd Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Eindhoven: Springer, 2016. 791–810. [doi: 10.1007/978-3-662-49674-9_50]
[20] Zhao YW, Sanán D, Zhang FY, Liu Y. Refinement-based specification and security analysis of separation kernels. IEEE Trans. on Dependable and Secure Computing, 2019, 16(1): 127–141. [doi: 10.1109/TDSC.2017.2672983]
[21] Tuch H, Klein G. Verifying the L4 virtual memory subsystem. 2004. https://cgi.cse.unsw.edu.au/~kleing/papers/verifying-l4-vm.pdf
[22] Klein G, Tuch H. Towards verified virtual memory in L4. 2004. https://isabelle.in.tum.de/~kleing/papers/towards-verified-vm.pdf
[23] Greenaway D, Lim J, Andronick J, Klein G. Don't sweat the small stuff: Formal verification of C code without the pain. ACM SIGPLAN Notices, 2014, 49(6): 429–439. [doi: 10.1145/2666356.2594296]

[24] Nipkow T, Paulson LC, Wenzel M. Isabelle/HOL: A Proof Assistant for Higher-order Logic. Berlin: Springer, 2002. [doi: 10.1007/3-540-45949-9]

[25] Rushby J. Noninterference, Transitivity, and Channel-control Security Policies. Menlo Park: SRI Int'l, 1992.

[26] Sabelfeld A, Myers AC. Language-based information-flow security. IEEE Journal on Selected Areas in Communications, 2003, 21(1): 5–19. [doi: 10.1109/JSAC.2002.806121]

[27] Von Oheimb D. Information flow control revisited: Noninfluence = noninterference + nonleakage. In: Proc. of the 9th European Symp. on Research in Computer Security. Sophia Antipolis: Springer, 2004. 225–243. [doi: 10.1007/978-3-540-30108-0_14]

[28] Leino KRM, Moskal M. Usable auto-active verification. 2010. https://fm.csl sri.com/UV10/submissions/uv2010_submission_20.pdf

[29] Leino KRM. Extended static checking: A ten-year perspective. In: Wilhelm R, ed. Informatics. Berlin, Heidelberg: Springer, 2000. [doi: 10.1007/3-540-44577-3_11]

[30] Leino KRM. Dafny: An automatic program verifier for functional correctness. In: Proc. of the 16th Int'l Conf. on Logic for Programming, Artificial Intelligence, and Reasoning. Dakar: Springer, 2010. 348–370. [doi: 10.1007/978-3-642-17511-4_20]

[31] Cuoq P, Kirchner F, Kosmatov N, Prevosto V, Signoles J, Yakobowski B. Frama-C: A software analysis perspective. In: Proc. of the 10th Int'l Conf. on Software Engineering and Formal Methods. Thessaloniki: Springer, 2012. 233–247. [doi: 10.1007/978-3-642-33826-7_16]

[32] Herrman DS. Using the Common Criteria for IT Security Evaluation. New York: Auerbach Publications, 2002. [doi: 10.1201/9781420031423]

[33] L4Ka Team. L4 pistachio source code, release version 0.4. 2022. https://www.l4ka.org/154.php

[34] de Roever WP, Engelhardt K. Data Refinement: Model-oriented Proof Methods and Their Comparison. Cambridge: Cambridge University Press, 1998.

[35] Bond B, Hawblitzel C, Kapritsos M, Leino KRM, Lorch JR, Parno B, Rane A, Setty S, Thompson L. Vale: Verifying high-performance cryptographic assembly code. In: Proc. of the 26th USENIX Security Symp. Vancouver: USENIX Association, 2017. 917–934.

[36] Mai HH, Pek E, Xue H, King ST, Madhusudan P. Verifying security invariants in ExpressOS. In: Proc. of the 18th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Houston: ACM, 2013. 293–304. [doi: 10.1145/2451116.2451148]

[37] Yang J, Hawblitzel C. Safe to the last instruction: Automated verification of a type-safe operating system. In: Proc. of the 31st ACM SIGPLAN Conf. on Programming Language Design and Implementation. Toronto: ACM, 2010. 99–110. [doi: 10.1145/1806596.1806610]

[38] Kolanski R, Klein G. Formalising the L4 microkernel API. In: Proc. of the 12th Computing: The Australasian Theory Symp. Hobart: Australian Computer Society, Inc., 2006. 53–68.

[39] Barrett C, Stump A, Tinelli C. The SMT-LIB standard: Version 2.0. 2010. https://smt-lib.org/papers/smt-lib-reference-v2.0-r10.12.21.pdf

[40] Lopes NP, Menendez D, Nagarakatte S, Regehr J. Provably correct peephole optimizations with alive. In: Proc. of the 36th ACM SIGPLAN Conf. on Programming Language Design and Implementation. Portland: ACM, 2015. 22–32. [doi: 10.1145/2737924.2737965]

[41] de Moura L, Bjørner N. Z3—A tutorial. Technical Report, Albuquerque: Microsoft, 2012.

[42] Hoare CAR. An axiomatic basis for computer programming. Communications of the ACM, 1969, 12(10): 576–580. [doi: 10.1145/363235.363259]

## 作者简介

**章乐平**, 博士生, 主要研究领域为操作系统形式化方法, 自动化验证, 信息流安全.

**赵永望**, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域形式化方法, 操作系统内核及安全, 编程语言, 安全攸关系统与软件, 模型学习与程序合成.

**王布阳**, 硕士生, 主要研究领域为形式化验证.

**李建欣**, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为社交网络, 机器学习, 大数据, 可信计算.