软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2023,34(8):3527−3548 [doi: 10.13328/j.cnki.jos.006869]

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

L4 虚拟内存子系统的形式化验证*

章乐平[1], 赵永望[2,3], 王布阳[1], 李悦欣[1], 冯潇潇[1]

[1](北京航空航天大学 计算机学院, 北京 100191)

[2](浙江大学 网络空间安全学院, 浙江 杭州 310007)

[3](浙江大学 移动终端安全技术浙江省工程研究中心, 浙江 杭州 310007)

通信作者: 赵永望, E-mail: zhaoyw@zju.edu.cn

摘 要: 第二代微内核 L4 在灵活度和性能方面极大地弥补了第一代微内核的不足, 这引起学术界和工业界的关注. 内核是实现操作系统的基础组件, 一旦出现错误, 可能导致整个系统瘫痪, 进一步对用户造成损失. 因此, 提高内核的正确性和可靠性至关重要. 虚拟内存子系统是实现 L4 内核的关键机制, 聚焦于对该机制进行形式建模和验证. 构建了 L4 虚拟内存子系统的形式模型, 该模型涉及映射机制所有操作、地址空间所有管理操作以及带 TLB 的 MMU 行为等; 形式化了功能正确性、功能安全和信息安全三方面的属性; 通过部分正确性、不变式以及展开条件的推理, 在定理证明器 Isabelle/HOL 中证明了提出的形式模型满足这些属性. 在建模和验证过程中, 发现源代码在功能正确性和信息安全方面共存在 3 点问题, 给出了相应的解决方案或建议.

关键词: L4; 形式化验证; 内存管理; 映射; 信息流安全; Isabelle/HOL

中图法分类号: TP311

中文引用格式: 章乐平, 赵永望, 王布阳, 李悦欣, 冯潇潇. L4 虚拟内存子系统的形式化验证. 软件学报, 2023, 34(8): 3527–3548. http://www.jos.org.cn/1000-9825/6869.htm

英文引用格式: Zhang LP, Zhao YW, Wang BY, Li YX, Feng XX. Formal Verification of Virtual Memory Subsystem in L4. Ruan Jian Xue Bao/Journal of Software, 2023, 34(8): 3527−3548 (in Chinese). http://www.jos.org.cn/1000-9825/6869.htm

Formal Verification of Virtual Memory Subsystem in L4

ZHANG Le-Ping[1], ZHAO Yong-Wang[2,3], WANG Bu-Yang[1], LI Yue-Xin[1], FENG Xiao-Xiao[1]

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

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

[3](Zhejiang Engineering Research Center of Mobile Terminal Security Technology, Zhejiang University, Hangzhou 310007, China)

Abstract: L4, the second-generation of microkernel, greatly compensates for the shortcomings of the first-generation of microkernel in flexibility and performance, which has attracted the attention of academia and industry. The kernel is the basic component for implementing the operating system. Once it has errors, it may cause the breakdown of whole system, further causing losses to users. Therefore, it is very important to improve the correctness and reliability of the kernel. Virtual memory subsystem is a key mechanism to implement L4 kernel. This study focuses on the formal modeling and verification of this mechanism. A formal model is presented, which involves all operations of mapping mechanism, all management operations of address space, and MMU behavior with TLB. The properties of functional correctness, safety, and security are formalized. Through reasoning about partial correctness, invariants and unwinding conditions, it is shown that the proposed model satisfies these properties in the theorem prover Isabelle/HOL. During modeling and verification, three problems are found related to functional correctness and security in source code. The corresponding solutions or suggestions are given in this study as well.

Key words: L4; formal verification; memory management; mapping; information flow security; Isabelle/HOL

* 基金项目: 国家自然科学基金(62132014); 浙江省尖兵计划(2022C01045)
本文由“约束求解与定理证明”专题特约编辑蔡少伟研究员、陈振邦教授、王戟研究员、詹博华副研究员、赵永望教授推荐.
收稿时间: 2022-09-05; 修改时间: 2022-10-13; 采用时间: 2022-12-14; jos 在线出版时间: 2022-12-30

微内核[1]是运行于操作系统特权模式下并遵循内核代码最小化原则的一类内核, 它将操作系统必要服务例如线程管理和地址空间管理等置于内核层, 其他服务如文件系统和设备驱动等置于用户层. 这种设计使得系统中一个模块出现错误只会引起相关的应用程序发生故障, 而不会导致整个系统崩溃. L4[2,3]是第二代微内核, 它遵循微内核的最小化设计原则, 仅提供了 3 个基本抽象(线程、地址空间和时间)和两个机制(映射和 IPC), 其设计目标是追求灵活性、可靠性以及高性能, 整个内核的实现约一万行 C++代码. 该内核在极大程度上解决了初代微内核[4]性能低下的问题, 在学术界和工业界得到广泛应用[5−8].

L4 内核作为操作系统的核心组件, 运行于特权模式下, 能够绕过硬件保护机制. 恶意软件可能利用该特点, 通过内核代码潜在的漏洞例如隐蔽通道(covert channel, 隐通道)来窃取机密数据, 从而导致用户财产损失, 甚至可能威胁到生命. 为提高其正确性和可靠性, 本文聚焦于 L4 虚拟内存子系统这一关键机制, 对实现该系统的地址空间和映射机制形式建模与验证. 到目前为止, 与本文相关的验证工作[9−21]存在 3 个问题: 首先, 经过形式化验证的虚拟内存模型中, 虚拟页之间的关系大多数都是呈线性的, 很少具备与 L4 类似的树状结构, 因此在验证该类结构需要重新探索相关理论知识; 其次, 已有的关于 L4 虚拟内存子系统模型是不完整的, 未考虑灵活页面数据结构、对页面的访问权限字段、对地址空间的操作以及带 TLB 的 MMU 行为等; 此外, 现有内存模型中验证的属性不够完善, 通常都是单方面的, 仅验证内存隔离或仅验证功能安全, 尚未同时考虑功能正确性、功能安全和信息安全这 3 个方面性质的验证.

形式化建模和验证 L4 虚拟内存子系统时, 主要面临 3 个方面的挑战.

第一, 实现 L4 地址空间的数据结构较复杂, 采用灵活的页面类型 Fpage 来指定页面大小, 操作大小不一致的页面会增加验证的复杂度;

第二, L4 采用多级页表来维护虚拟地址和物理地址的关系, 在此基础上, L4 映射机制采用映射数据库(mapping database, MDB)来维护地址空间之间页面和页面的关系. 对于任意一个页面, 它与其他地址空间的页面关系呈树形结构, 在验证该模块时, 由于路径的不确定性, 导致验证工作量呈指数增长;

第三, L4 为了防止多级页表带来深层次递归从而导致栈数据溢出等问题, 使用非递归方式来实现页面映射操作, 同时调用了维护页面之间关系的处理函数, 这使得页面映射算法相当复杂. 以映射页面 *Map_Fpage* 函数为例, 该函数包括 3 层循环, 约 560 行代码, 占实现 L4 虚拟内存子系统总代码量的四分之一. 内核开发人员注释该算法是整个内核中最复杂的算法.

针对上述问题和挑战, 本文提出了针对 L4 虚拟内存子系统的形式建模和验证方法. 整个过程划分为该系统的建模和属性的验证两个阶段.

- 在建模阶段, 本文基于 pistachio0.4 版本的源代码[22]构建, 所构建的模型以状态机为基础, 定义了包括灵活页面类型 Fpage 的所有数据结构, 建模了映射机制所有操作、地址空间管理操作和带 TLB 的 MMU 行为;
- 关于待验证的属性, 本文形式化了 3 类属性: 功能正确性、功能安全(safety)和信息安全(security)相关属性. 功能正确性保证所构建的形式模型正确定义了功能需求, 同时能够检验功能需求是否合理或正确; 功能安全指系统不会出现页面映射关系形成环、TLB 数据不一致等类似的错误; 信息安全方面的性质基于信息流安全属性定义, 这些属性能够有效防止恶意程序利用隐通道窃取内存中有价值的数据. 在验证阶段, 本文通过推理部分正确性、不变式和展开条件(unwinding conditions)的方法, 证明了形式模型满足功能正确性、功能安全性以及部分信息流安全属性.

所有工作在定理证明器 Isabelle/HOL[23]中完成, 对建模和验证过程发现的问题进行分析, 并给出相应的解决办法或建议.

本文具体贡献如下:

1) 提出了针对 L4 虚拟内存子系统的形式模型. 该模型完整建模了系统中所有功能,包括对页面的操作、带有 TLB 的硬件机制 MMU 的行为以及地址空间的管理, 其中, 在对页面的操作中拓展了复杂数据结构以及访问权限字段的建模, 并解决了已有模型[9]不完整问题;

2) 形式化了L4虚拟内存子系统的功能正确性、功能安全性和信息流安全属性: 功能正确性依据L4内核开发手册[3]和部分源代码构建; 功能安全性覆盖了已提出的所有不变式[9], 并拓展了更多不变式, 提高了功能安全性质完整度; 信息流安全属性考虑无干扰(noninterference)和无泄漏(nonleakage), 两者基于本文提出的传递性类型的信息流策略定义;
3) 使用Isabelle/HOL完成了3个方面属性的证明. 其中: 功能正确性被表示为霍尔三元组形式的引理, 利用各功能的自身性质以及Isabelle/HOL理论库和策略证明这些引理成立; 功能安全相关属性被表示为一系列不变式, 通过归纳方法完成不变式的证明; 信息流安全属性的证明依赖于展开条件, 本文完成了对无泄漏的全部证明和对无干扰的部分证明;
4) 发现了L4虚拟内存子系统的设计和源代码中共存在3个问题, 这些问题导致该子系统不满足功能正确性和无干扰属性. 针对这些问题, 本文提出了相应的解决方案或建议.

本文第1节介绍技术背景和相关工作. 第2节详细阐述L4虚拟内存子系统形式模型的构建过程: 首先, 定义相关数据结构和状态; 其次, 建模该子系统中的所有行为, 包括映射机制中页面操作、带TLB的MMU行为、地址空间管理操作. 第3–5节分别描述关于L4虚拟内存子系统的功能正确性、功能安全和信息安全相关性质的定义和证明. 第6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 技术背景与相关工作

本节介绍与本文研究相关的技术背景和相关工作: 首先介绍了L4虚拟内存子系统的技术实现和已存在的验证情况, 其次阐述了内存管理机制的相关工作, 最后介绍了信息流安全的背景知识和相关验证案例.

1.1 L4虚拟内存子系统

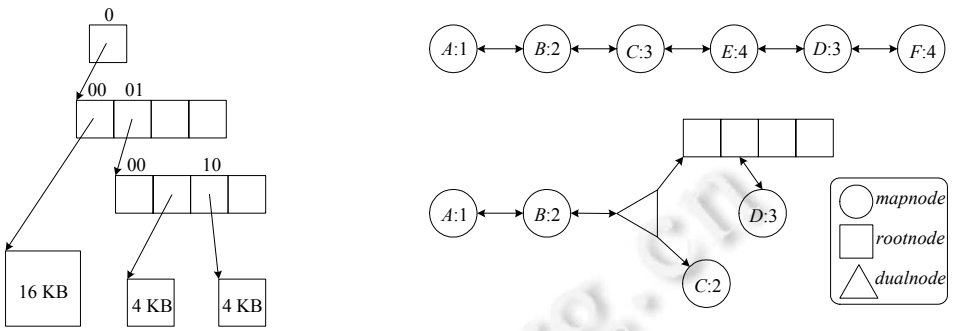
L4虚拟内存子系统包括地址空间和映射等机制.

- 地址空间是一个逻辑的概念, 它表示一个线程能够访问的虚拟地址范围. 对32位机器来说, 它的覆盖范围是0–4 G. 每个地址空间中包含一个页表, 实现了虚拟地址到物理地址的转换, 同时是实现内存隔离的一种方式. L4对地址空间的管理操作包括创建地址空间、初始化地址空间和删除地址空间;
- 映射机制是L4内核的特色之一, 该机制除了维护页表数据, 还维护不同地址空间中页面的相互关系.

系统刚开机时, 会创建一个特权空间 *Sigma*0*Space*, 同时将物理内存一一映射到该空间, 因此它占据所有的物理内存并拥有处理物理内存的最高权限. 之后, *Sigma*0*Space* 中的唯一线程 *sigma*0 作为一个分页器(pager), 会管理物理内存的分配. 具体操作是: 将自身空间的页面映射到其他地址空间, 获得页面的地址空间, 能够进一步地将该页面映射到其他地址空间. 在此过程中, 采用多级页表记录地址空间中被映射页面与物理页的转换关系以及访问权限. 如图1(a)所示: 通过多级页表获取物理页面, 由于L4支持灵活的页面映射, 所获得的页面大小是可变的. 页面成功被映射到其他地址空间后, 需要使用MDB来记录每个物理页与地址空间的关系, 任何一个物理页与地址空间的关系在逻辑上呈树形结构, MDB 采用双向链表实现该结构, 并且构建双向链表的方式是前序遍历. 如图1(b)所示: MDB中包括 *mapnode*, *rootnode* 和 *dualnode* 这3类节点. 其中, *mapnode* 代表了页面实际的映射. 例如, *A*:1 表示指定页面映射到地址空间 *A* 中, 所在深度处于第1层, 即最开始被映射到 *A* 中. 在图1(b)简单情况中(上图), *A*:1 的相邻节点 *B*:2 表示页面被 *A* 映射至 *B*, 进一步被映射至 *C* 和 *D* 中. 对于较复杂的情况(下图), MDB 结构中包括 *rootnode* 和 *dualnode* 节点: 前者是一个数组, 在映射一个大页面分割成的小页面时构建, *rootnode* 中每个元素与 *mapnode* 连接; 后者包含一个 *rootnode* 和一个 *mapnode*, 用于处理页面既被分割为小页面被映射又被直接映射的情况.

由于L4使用数据结构Fpage来指定不同大小的页面, 在允许的条件下, 只需执行一次映射操作即可将大页面映射到指定空间, 从而提高页面映射的效率, 进而提高了性能. 然而, 该方式大幅度增加了页面映射函数的算法复杂度. 在页面映射函数中, 约95%的代码用于实现对页面大小的调整, 调整原因包括两个方面: 映射方和被映射方所提供的页面大小不一致; 在一次操作中, L4能够处理的页面大小与映射方或被映射方的大小不一致. L4采用多级页表来实现对虚拟页和物理页关系的维护, 并使用非递归算法替代递归算

法实现从大页面到小页面的深层次遍历, 这导致在该函数中调整页面大小的算法十分复杂, 实际维护页面关系和页表数据的代码仅 30 行左右.



(a) 多级页表 (b) MDB

图 1 虚拟内存子系统数据结构

目前, 关于 L4 虚拟内存子系统的相关建模和验证工作较少, 仅包括 Klein 等人[9]提出的关于 L4 虚拟内存子系统的抽象模型, 该抽象模型定义了页面操作、页表查询函数以及地址空间管理中的创建地址空间操作, 但是存在的不足是未考虑页表包含的访问权限、TLB、特权空间以及地址空间删除操作. 本文对这些未考虑的组件或功能进行了补充. 在 Klein 等人的抽象模型中, 每次仅对一个页面进行操作, 然而 L4 定义了灵活页面类型 Fpage, 用于提供大小不同的页面, 在页面映射时将其拆分为小页面然后循环处理. 对此, 本文补充了 Fpage 类型的定义, 并实现了处理不同大小页面的函数. 由于该部分内容的拓展大幅度增加了验证复杂度, 所以本文大部分工作是在完成此函数关于不变式的验证. 关于属性, Klein 等人的抽象模型提供了 3 个不变式, 本文依据新构建的模型对三者进行重新定义, 同时拓展了更多的不变式. 后续 Klein 等人在上述模型基础上拓展了精化模型[10], 精化模型的建模粒度更接近于代码实现, 但仍未处理上述存在的问题.

1.2 内存管理机制

内存管理机制是指操作系统对内存进行合理地划分和有效地动态分配, 目前在形式化领域对该机制完成的工作包括内存模型的构建[11−16]、内存管理机制的验证[17−21,24,25]以及内存分配算法的形式化开发[26−31]等. 例如: Saraswat 等人形式化了将程序和内存读写操作分离的弱内存模型[11], 并保证顺序执行的程序不会产生竞争; Leroy 等人使用定理证明器 Coq 构建了低级命令式语言实现的内存模型[12], 该模型支持带指针的程序语义以及对此类程序的转换进行推理. 部分模型[13−15]仅涉及内存分配和和释放的操作. 内存管理机制的验证工作中, 较典型的案例有 seL4[24], CertiKOS[17,25], Zephyr[21], Verisoft[18]以及管理程序(hypervisor)[19,20]等. 其中, 微内核 seL4 是 L4 后一代内核, 两者内存管理机制实现方式不同: seL4 它不采用映射机制以及数据结构 MDB, 依据应用程序所拥有的权能(capability)对内存进行分配,通过分配算法来检查所分配的内存是否在空闲区域内, 同时保证该内存不会与其他对象所占据的内存重叠,严格实现物理内存隔离, 在 Isabelle/HOL 中实现了该机制的建模和验证.

在文献[21]中, 我们采用 Rely-Guarantee 方法完成了实时操作系统 Zephyr 中并发式伙伴(buddy)内存管理机制的建模和验证, 证明了该机制满足功能正确性和不变式成立, 并发现源代码中存在 3 处错误.

1.3 信息流安全

信息流安全属性包括经典无干扰[32]、无泄漏[33]和无影响[34]以及它们的变体.

- 无干扰指清除影响指定域 d 的所有事件后, 从初始状态 s_0 开始执行所得到的事件序列的结果与执行原始事件序列结果相同, 该定义强调动作的保密性;
- 无泄漏是指对于指定序列 es 和指定域 d, 如果保证 s 和 t 状态对于能够影响 d 的所有域都等价, 那么从 s 和 t 状态分别执行 es 后所得到的新状态是等价的. 该定义保证了 d 中的数据不会被恶意篡改, 强

调了数据的保密性;

- 无影响是二者的结合, 既保证系统满足无干扰, 又满足无泄漏.

目前, 关于信息流安全属性的验证已存在大量的工作[35−41], 主要将这些属性应用到实际操作系统或安全机制中, 较经典的案例包括:

- Murray 等人完成了微内核系统 seL4 关于非传递性(intransitive)无干扰的变体验证[35,36], 该工作将 seL4 作为分离内核, 将进程作为安全域, 提出了主体关于客体的访问控制策略, 并将其转换为进程之间的信息流策略. 该策略满足信息流 $a\rightsquigarrow b$ 和 $b\rightsquigarrow c$, 但不满足 $a\rightsquigarrow c$,以保证信息流策略的非传递性. 通过强制执行信息流策略, 在 Isabelle/HOL 中验证了约 8 800 行的内核代码满足信息流安全属性;
- Costanzo 等人在定理证明器 Coq 中实现了并发操作系统内核 mCertiKOS 关于保密性(confidentiality)的验证[37], 将无干扰应用于保密性的定义, 通过观察函数定义进程的可见区域, 利用展开条件证明状态等价性, 为系统提供了高级安全性保证;
- 文献[38]提出了设计和验证信息流控制系统的框架 Nickel, 使用 Nickel 验证操作系统内核 NiStar 满足无干扰属性, 其信息流策略定义为所有普通进程之间不存在信息流, 只能通过内核间接传递信息;
- 在文献[39]中, 我们以分区作为安全域, 完成了 ARINC 653 标准关于无干扰、无泄漏、无影响以及它们变体的形式化验证, 验证过程中发现基于该标准的设计存在隐通道, 对这些通道进行了修复. 后续在文献[40]中提出了信息流安全属性的推理框架, 构建了 ARINC 653 标准的顶层规范, 通过精化方法将顶层规范拓展到更细粒度的规范, 然后将推理框架应用于上述规范的验证.

本文关于信息流安全属性的验证借鉴了上述部分工作的思路, 但同时存在差异. 例如: 安全域的定义与上述均不同; 构建无泄漏的关键定义中, 信息流策略为传递性类型. 更多细节在后续章节中详细介绍.

2 L4 虚拟内存子系统的形式模型

本节介绍 L4 虚拟内存子系统的形式模型. 所构建的模型以状态机模型为基础, 将该系统的运行视为状态机的执行. 状态机的组成要素主要包括状态和事件: 状态记录了实际系统的变量值; 事件描述实际系统的功能, 每个事件对应一个迁移规则, 并驱动状态发生迁移. 针对 L4 虚拟内存子系统, 本文构建的状态覆盖了该系统的所有组件, 包括地址空间相关字段、页表、TLB 和堆等, 定义的事件涵盖映射机制中对页面的所有操作、带 TLB 的 MMU 行为以及对地址空间管理操作. 以下首先介绍状态以及定义状态的类型, 然后分别介绍各类操作.

2.1 类型和状态

依据实现 L4 虚拟内存子系统的数据结构, 可抽象如图 2 所示的地址空间结构图. 在该图中, 最下方圆角矩形表示物理内存, $r1$ 和 $r2$ 表示物理页. 直角矩形表示地址空间($sp1$–$sp5$), 其中, $sp1$ 实际指 *Sigma0Space*. 每个地址空间中的页面由页号($v1$–$v7$)和访问权限(r,w,e)组成, 箭头表示页面之间的映射关系. 例如: $sp1$ 将页号为 $v2$ 的页映射给了 $sp3$ 中页号为 $v4$ 的页, 并赋予了所有权限. 在映射过程中, 需保证页面之间不能形成环状结构, 并且被映射的页面所具有的访问权限不能超过映射者.

本文用 *spaceName_t* 来表示地址空间名, *vaddr_t* 和 *paddr_t* 分别表示虚拟地址和物理地址, *vpage_t* 和 *rpage_t* 分别表示虚拟页和物理页, 均为自然数类型. 访问权限定义为 *perms_t* 类型, 其值包括可读(*pfRead*)、可写(*pfWrite*)和可执行(*pfExecute*). 在图 2 中, 一个页面或者是属于某个地址空间中的虚拟页, 或者是物理页, 可定义统一的页面类型 *page_t* 如下:

datatype page_t=Virtual spaceName_t vpage_t|Real rpage_t.

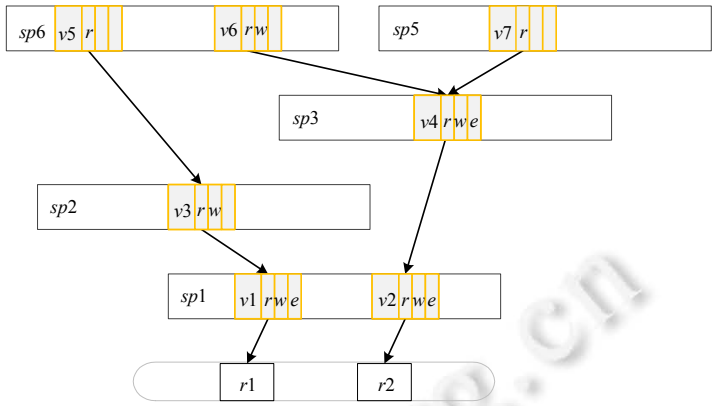


图 2　地址空间结构图

利用 *page_t* 定义页表类型. 地址空间每个虚拟页的映射存在 3 种情况: 映射到另一个地址空间的虚拟页、映射到物理页、无映射. 因此, 使用返回值为 *option* 类型的函数来定义虚拟页的映射, 即页表类型如下:

type_synonym mapping=vpage_t⇀page_t×“perms_t set”.

其中, 符号⇀与表示函数的符号⇒类似, 区别是前者必须保证返回值是 *option* 类型, 它将 *option* 关键字进行省略. *perms_t set* 是权限集合, 它定义了此映射中对虚拟页的访问权限(实质是对物理页的访问权限, 因为映射不为空的虚拟页一定能找到物理页).

除上述用于定义固定大小的页面的数据类型外, L4 提供了为提高系统灵活性而设计的数据类型 *Fpage*, 该类型由 3 个部分构成: 基地址、页面大小和访问权限. 对此, 本文将其定义为一个三元组如下:

type_synonym fpage_t=base×size×“perms_t set”.

其中, *base* 和 *size* 均为自然数类型. L4 要求页面大小至少是 1 KB, 即 *size* 的值至少是 10, *base* 以 1 K 作为单位进行偏移. 此外, 系统实际使用的有效基地址和 *base* 可能不同, 系统会依据 *size* 的值进行调整. 例如: 考虑基地址为 3 K, 若设置的页面大小 *size* 分别为 1 KB 和 2 KB, 则对应的有效基地址分别为 3 K 和 2 K, 即保证基地址数值大小是页面大小的整数倍. 此外, 采用 *Fpage* 定义的页面还包括完整页(页面大小为 4 GB)和空页. 因此,定义如下类型来区分这些页面:

datatype fpage_type=invalid_fpage|valid_fpage|complete_fpage|nil_fpage.

其中, *invalid_fpage* 和 *valid_fpage* 分别表示页面是无效的和有效的, 后两者分别表示完整页和空页. 通过函数 *fpage_trans* 实现 *fpage_t* 类型到 *fpage_type* 类型数据的转化.

硬件机制 TLB 由若干个表项(*entry*)组成. ARM 体系架构中的表项包括 4 个部分: 地址空间标识符(*ASID*)、虚拟页号、物理页号、访问权限. 其中, *ASID* 号长度限制在 8 位. 本文基于 ARM 架构定义 TLB 表项的数据类型 *tlb_entry* 为

datatype tlb_entry=PTE asid vpage_t rpage_t “perms_t set”.

其中, *asid* 和 *spaceName_t* 类型相同, 但鉴于两者位数的不同, 利用限制两者类型所对应数据的最大值来体现其区别. 例如, *asid* 类型下的数据最大值为 2^8-1. 在遍历 TLB 表项时, 通常会出现命中(*hit*)和未命中(*miss*)两种情况. 为防止实现 TLB 相关功能的代码出现错误, 添加 TLB 与页表数据不一致(*incon*)的情形, 后续通过不变式保证该情形不可能发生. TLB 的返回值类型 *lookup_type* 如下:

datatype lookup_type=Miss|Incon|Hit tlb_entry.

该类型中, *Hit* 携带 *tlb_entry* 这一参数, 表示命中了指定的 TLB 表项. 该表项是唯一的, 否则将出现 TLB 表项之间存在不一致. 依据上述所有类型, 可定义记录状态的类型 *state* 如下.

record state=

current::　　*spaceName_t*

space_mapping:: *spaceName_t*⇀*mapping*

heap:: *paddr_t*⇒*byte*

tlb:: *tlb_entry set*

其中,

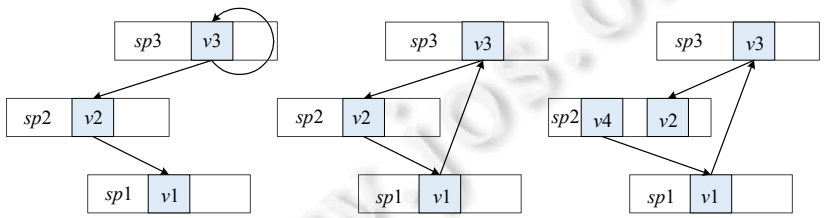
- *current* 表示当前系统所处地址空间;
- *space_mapping* 字段携带地址空间名这一参数, 并且返回值是元素为 *mapping* 的 *option* 类型, 记录了每个地址空间中的映射情况, 既实现了用于将已建立映射的虚拟页转换为物理页的页表的功能, 又实现了用于记录页面之间关系的 MDB 的功能. 当返回值为 *None* 时, 表示该地址空间还未被创建. 将 *space_mapping s space* 符号化为 *s@space*, 定义已创建的地址空间为 *spaces s*=*dom*(*space_mapping s*);
- *heap* 记录了内存数据, 其返回类型为字节类型;
- *tlb* 包含了一系列表项, 其数目在配置系统环境的静态变量 *SysConf*(后续称为系统配置)中通过 Isabelle/HOL 中的关键字 *specification* 约束.

结合上述定义, 可表示图 2 中页面之间的关系. 对此, 文献[9]提出了路径(*paths*)的概念. 首先给出了直接路径的定义: 状态 *s* 中页面 *x* 到页面 *y* 存在直接路径当且仅当满足 3 个条件: (a) *x* 为某个地址空间 *space* 的虚拟页 *v*; (b) *space* 的映射为 *m*; (c) *v* 在 *s* 状态经过 *m* 映射到 *y*. 本文在该定义基础上增加了访问权限字段以及该字段与其他字段之间的关系, 下面以归纳形式定义拓展的路径 *direct_path*:

$$s@space=Some\ m \Rightarrow m\ v=Some(y,perms) \Rightarrow s\vdash(Virtual\ space\ v)\rightsquigarrow^{1} y.$$

页面 *x* 直接表示为 *Virtual space v*. 基于直接路径, 归纳定义了路径的传递闭包和自反传递闭包, 分别符号化为 $s\vdash x\rightsquigarrow^{+} y$ 和 $s\vdash x\rightsquigarrow^{*} y$.

为便于后续页面关系的研究, 本文称直接路径中 *v*3 为 *v*1 的直接子页面, *v*1 为 *v*3 的直接父页面; 称路径(传递闭包)上能到达 *v*1 的所有页面为子页面(*v*3,*v*5), 路径上 *v*3 能到达的所有页面为父页面(*v*1,*r*1). 此外, 本文提出了“自环”和“闭环”的概念. 自环关系如图 3(a)所示, 地址空间 *sp*3 中的虚拟页 *v*3 映射到了本页面, 形成了针对 *v*3 自身的环状关系. 闭环关系存在两种情况, 分别如图 3(b)和图 3(c)所示: 图 3(b)中, 虚拟页 *v*1, *v*3, *v*2 依次连接, 形成完全封闭状态的结构图; 图 3(c)中, 虽然虚拟页 *v*1, *v*2, *v*3 和 *v*4 之间不构成完全封闭关系, 但 *v*2 和 *v*4 处于同一地址空间, 即封闭于同一地址空间, 该情况亦不允许出现在地址空间结构中.



(a) 自环 (b) 闭环 1 (c) 闭环 2

图 3 环状结构

2.2 映射机制相关操作

L4 映射机制为页面提供了 4 种操作: 回收(*unmap*)、释放(*flush*)、映射(*map*)和授予(*grant*), 图 4 展示了由图 2 执行 4 个操作后分别得到的新的页面关系. 本文对这些操作进行了形式定义, 完善了对访问权限字段的处理. 此外, 添加了 *map* 操作和 *grant* 操作的可执行条件, 修正了文献[9]中地址空间会形成自环的错误, 同时防止了闭环情况的发生.

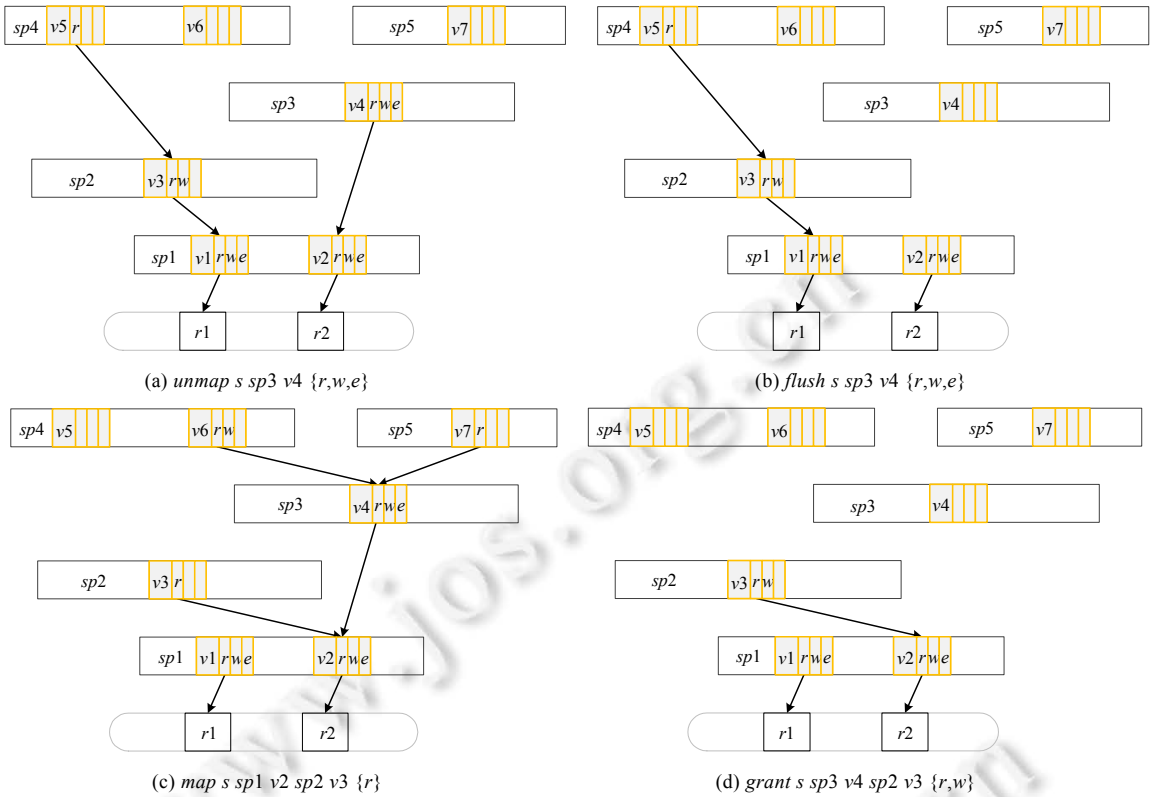


图 4 映射机制基本操作

unmap 操作完成以下任务之一: 或者回收地址空间中给定页面曾经映射出去的所有页, 或者删除这些页的部分访问权限. 当所有的访问权限均被删除时, 则执行前者, 即删除图 4(a)中路径 *v*4 后续的所有边; 否则, 不改变页面之间的关系, 仅删除权限即可. *unmap* 操作能够改变其他地址空间中的页面映射, 引入辅助函数 *clear* 来清除页面之间的关系. *clear* 函数详细描述为: 对于给定的地址空间 *space*、虚拟页 *vpage*、映射 *mapping* 以及删除的权限集 *del_perms*, *clear* 函数将依赖于 *space* 中 *vpage* 虚拟页的 *mapping* 进行更新, 当权限被删除为空时, 则 *mapping* 中的 *vpage* 映射更新为 *None*; 否则, 将 *mapping* 中 *vpage* 的访问权限更新为减去 *del_perms* 后的结果, 最后形成新的映射. *clear* 函数定义如下.

```
clear:: state⇒spaceName_t⇒vpage_t⇒mapping⇒perms_t set⇒mapping
clear s space vpage mapping del_perms=
  λv_page1. case mapping v_page1 of None⇒None
    |Some(page,perms)⇒
      if s⊢page⇝*(Virtual space vpage)
      then if (perms⊆del_perms) then None else Some(page,perms−del_perms)
      else Some(page,perms)
```

unmap 对每个地址空间中的映射均进行 *clear* 操作. 因此利用 *clear* 函数, 定义 *unmap* 函数如下.

```
unmap:: state⇒spaceName_t⇒vpage_t⇒perms_t set⇒state
unmap s sp v_page del_perms=
  s(|space_mapping:=
    λsp'. case s@sp' of None⇒None|
      Some mapping'⇒Some(clear s sp v_page mapping' del_perms)|)
```

flush 利用 *unmap* 处理地址空间中的虚拟页, 然后释放该虚拟页, 如图 4(b)中 *v*4 不再拥有到其直接父页面 *v*2 的映射了, 除非 *flush* 操作仅删除部分权限. 当指定的地址空间为特权空间 *Sigma*0*Space* 时, *flush* 操作无效. *flush* 函数定义如下.

```
1  flush:: state⇒spaceName_t⇒vpage_t⇒perms_t set⇒state
2  flush s sp vpage del_perms=
3     if sp≠Sigma0Space then flush_mapping(unmap s sp vpage del_perms) sp vpage del_perms else s
```

其中, *flush_mapping* 为上述释放虚拟页操作.

map 操作将指定地址空间中的虚拟页映射到目标地址空间, 前者地址空间映射不变, 将目标地址空间进行更新. 当然, 目标地址空间为 *Sigma*0*Space* 时, *map* 操作无效. 图 4(c)展示了将 *v*2 映射到 *v*3 后得到的页面关系图, 具体过程是: 首先, 将 *v*3 回收并释放, 即执行 *flush* 操作; 然后, 将 *v*2 映射给 *v*3. 为方便定义 *map* 函数, 引入有效页的概念, 用来判断虚拟页是否有映射, 具体定义 *valid_page s page*(符号化为 $s\vdash page$).

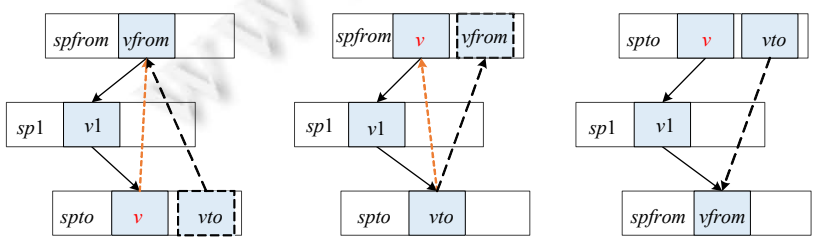
- 当页面 *page* 为虚拟页时, 如果状态 *s* 中虚拟页存在直接父页面, 则 *page* 为有效; 否则无效;
- 页面 *page* 为物理页时, *page* 有效:

$$s\vdash page\equiv \text{case } page \text{ of } Virtual\ space\ vpage\Rightarrow \exists p,\ s\vdash page\rightsquigarrow^{1} p\,|\,Real\ r\Rightarrow \text{True}.$$

其中, 符号≡表示恒等. 利用有效页的定义, 形式化 *map* 函数如下.

```
1  map:: state⇒spaceName_t⇒vpage_t⇒spaceName_t⇒vpage_t⇒perms_t set⇒state
2  map s spfrom vfrom spto vto perms=
3     if sp_to≠Sigma0Space
4     then if (s⊢(Virtual spfrom vfrom))∧
5             (perms≠{·}∧perms⊆get_perms s spfrom vfrom)∧
6             (∀v, ¬s⊢(Virtual spfrom vfrom)⇝+(Virtual spto v))∧
7             (∀v, ¬s⊢(Virtual spfrom v)⇝+(Virtual spto vto))∧
8             (∀v, v≠vto→¬s⊢(Virtual spto v)⇝+(Virtual spfrom vfrom))
9          then create_mapping s spto vto(Virtual spfrom vfrom) perms
10         else s
11    else s
```

在该定义中, *get_perms* 函数获取访问权限, *create_mapping* 为实际建立映射的函数, 第 4–8 行是执行映射需满足的条件. 文献[9]仅为 *map* 函数添加了第 4 行的条件判断, 即指定空间中的虚拟页是否为有效页.第 5 行的条件是本文为访问权限字段添加的必要约束: 映射时必须给予目标虚拟页访问权限, 但给予的权限不能超过指定虚拟页自身拥有的访问权限. 本文在后续证明功能安全性时无法通过定理证明器 Isabelle/HOL 的检查, 其原因是 *map* 函数中的条件不是完整的. 因此, 本文提出第 6–8 行中的 3 个关键条件, 分别防止了图 5 中的 3 种情况出现.



(a) *v*=*vto* 或 *v*≠*vto* (b) *v*=*vfrom* 或 *v*≠*vfrom* (c) *v*≠*vto*

图 5 禁止 *map* 操作的地址空间结构

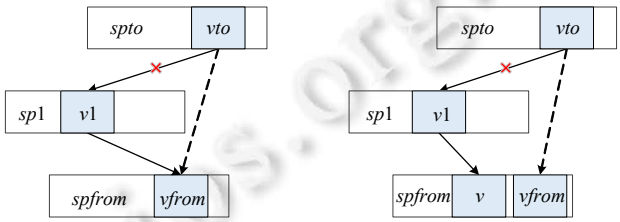
在图 5(a)中, 已经存在从映射方地址空间 *spfrom* 中的虚拟页 *vfrom* 到被映射方的地址空间 *spto* 中虚拟页

v 的路径, 若 *v* 正好是被映射虚拟页 *vto*, 执行 *map* 即完成橙色虚线操作, 形成图 3 中闭环 1 的情况; 若 *v* 是关于 *vto* 的其他虚拟页, 则完成黑色虚线操作, 形成闭环 2 情况. 图 5(b)表示已经存在从映射方地址空间 *spfrom* 中虚拟页 *v* 到被映射方地址空间 *spto* 中的虚拟页 *vto* 的路径, 执行操作和形成环状过程与图 5(a)类似. 图 5(c)表示已经存在从被映射方地址空间 *spto* 中关于 *vto* 其他的虚拟页 *v*(*v*≠*vto*)到映射方地址空间 *spfrom* 中的虚拟页 *vfrom* 的路径, 执行 *map* 后会完成黑色虚线操作, 该结构违背: 同一个地址空间中的页面不能被映射到同一个页面. 通过添加的第 6 行至第 8 行的条件, 分别避免了上述 3 种情况的发生.

容易混淆的是, 图 5(c)中 *v*=*vto* 表示已经存在 *vto* 到 *vfrom* 的路径, 即满足如下关系:

$$s\vdash(Virtual\ spto\ vto)\rightsquigarrow^{+}(Virtual\ spfrom\ vfrom).$$

此时, 将 *vfrom* 映射到 *vto* 是允许的, 无论是修改 *vto* 的访问权限还是将 *vto* 的直接父页面修改为 *vfrom*, 操作过程均如图 6(a)中所示, 执行黑色虚线操作的同时会删除 *vto* 到 *v*1 的映射.



(a) *v*=*vto* (b) *v*≠*vfrom*

图 6 已存在映射关系的地址空间结构

类似该处理过程的情况还包括图 6(b)所示的页面关系, 表示已经存在 *vto* 到 *vfrom* 所在地址空间 *spfrom* 的路径, 形式描述为

$$v\neq vfrom\wedge s\vdash(Virtual\ spto\ vto)\rightsquigarrow^{+}(Virtual\ spfrom\ v).$$

grant 操作将指定地址空间的虚拟页授予其他地址空间, 不再拥有到父页面的映射了. 完成该操作需要执行两步: (1) 首先, 将目标虚拟页映射到授予方的直接父页面; (2) 对授予方的虚拟页执行 *flush* 操作. *grant* 函数定义如下.

```
1  grant:: state⇒spaceName_t⇒vpage_t⇒spaceName_t⇒vpage_t⇒perms_t set⇒state
2  grant s sp_from v_from sp_to v_to perms=
3    if (s⊢(Virtual sp_from v_from))∧
4      (perms≠{·}∧perms⊆get_perms s sp_from v_from)
5      (∀v, ¬s⊢(next_page sp_from v_from)⇝+(Virtual spto v))∧
6      (∀v, ¬s⊢(Virtual(next_pre sp_from v_from) v)⇝+(Virtual spto vto))∧
7      (∀v, v≠vto→¬s⊢(Virtual spto v)⇝+fst(the(the(s@sp_from) v_from)))
8    then let page=next_page sp_from v_from
9         in flush (create_mapping s sp_to v_to page perms) sp_from v_from perms
10   else s
```

该定义中, 第 3–7 行中的条件与 *map* 函数中的条件类似, 区别为映射方为 *Virtual sp_from v_from* 的直接父页面, 采用 *next_page* 函数获得该页面, 并且 *next_pre* 获得该页面所在地址空间.

上述过程完成了对页面的 4 个基本操作的形式定义, 这些操作都仅对单个固定大小的页面进行操作. 为了实现灵活页面的处理, 本文分别对上述操作拓展了递归函数, 将每个大页面拆分成若干个小页面进行处理. 以 *unmap* 为例, 定义了递归函数 *Unmap_fpage* 如下.

```
1  Unmap_fpage:: state⇒spaceName_t⇒vpage_t⇒perms_t set⇒nat⇒state
2  Unmap_fpage s sp v perms 0=s|
```

3 *Unmap_fpage s sp v perms*(*Suc n*)=(*unmap*(*Unmap_fpage s sp v perms n*) *sp*(*v*+*n*) *perms*)

第 2 行是函数终止条件, 第 3 行表示递归调用 *unmap* 函数依次处理 *v* 至 *v*+*n* 范围内的页面. 由于不考虑性能, 一次映射大页面与多次映射拆分后的小页面得到的效果是相同的.

2.3 带TLB的MMU行为

总体来说, MMU 利用页表实现虚拟地址到物理地址的转换. 用户发起读写命令时, MMU 首先检查用户访问权限, 后续通过转换操作定位到指定的物理内存, 完成用户读写操作. TLB 是页表的部分缓存, 用于加快整个过程的效率. 带 TLB 的 MMU 具体转换过程如图 7 所示, 它将虚拟地址的页号同时送到 TLB 和页表中进行匹配. 由于速度上的差异, TLB 会优先得到该页号, 然后遍历 TLB 表项. 如果匹配结果为命中, 直接将转换后的物理页号与偏移地址连接, 得到物理地址; 如果匹配结果为缺失, 则执行遍历页表操作, 一旦匹配成功, 将匹配成功的表项加载到 TLB, 即图 7 中虚线过程, 接下来与 TLB 命中时过程一致. 当 TLB 中数据出现不一致或者页表未命中的异常时, 交由处理器去处理.

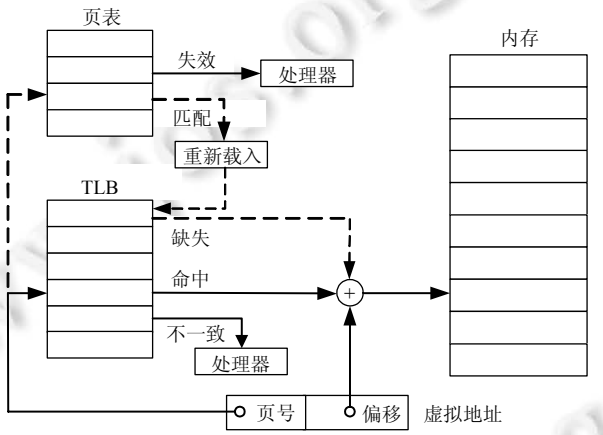


图 7 MMU 执行过程

本文借鉴文献[42]的部分建模思路, 形式化定义了带 TLB 的 MMU 行为, 具体包括: 利用 TLB 和页表实现虚实地址的转换、对用户读写操作的处理. 其中, 转换操作包括遍历 TLB 和页表. 此外, 上述异常通常被转发给相应线程处理, 不在本文考虑范围内.

首先给出 TLB 遍历过程的形式定义.

1 *lookup_tlb*:: *state*⇒*asid*⇒*vaddr_t*⇒*lookup_type*

2 *lookup_tlb s asid vaddr*=

3 let *S*=*entry_set*(*tlb s*) *asid vaddr*

4 in **if** *S*={·} **then** *Miss* **else**

5 **if** ∃*e*, *S*={*e*} **then** *Hit*(*the_elem S*) **else** *Incon*

其中, *entry_set* 函数根据 ASID 号和虚拟地址获取 TLB 中所有匹配的表项.

- 当获取的表项集合为空时, TLB 缺失;
- 当表项集合刚好仅有一个元素时, TLB 命中, 使用 *the_elem* 函数提取该元素; 否则, 表示 TLB 不一致.

其次, 页表遍历函数 *pt_walk* 定义如下.

1 *pt_walk*:: *state*⇒*spaceName_t*⇒*vpage_t*⇀*rpage_t*×*perms_t set*

2 *pt_walk s space vpage*=

3 **if** (*s*⊢(*Virtual space vpage*))

4 **then** *Some*((*SOME rpage*, (*s*⊢(*Virtual space vpage*)⇝$^{+}$(*Real rpage*))),*get_perms s space vpage*)

```
5    else None
```

该函数表示: 在给定状态 *s* 下, 指定地址空间 *space* 中虚拟页 *vpage* 为有效页时, 可获得相应的物理页号和访问权限, 即此时匹配成功; 否则, 返回值为空, 即页表查询失败. 在获取物理页时, 虽然匹配成功时物理页是唯一的, 但由于虚拟页与物理页是以路径相关联, 物理页 *rpage* 必须通过归纳方式得到, 即传递闭包公式, 该公式描述的物理页可能存在多个, 本文通过希尔伯特选择算子 SOME 随机得到其中的某一个(后续通过不变式保证物理页的唯一性).

利用遍历 TLB 函数 *lookup_tlb* 和遍历页表函数 *pt_walk*, 可完成 MMU 转换操作的定义.

```
1   mmu_translate:: state⇒asid⇒vaddr_t⇒state×(paddr_t option)
2   mmu_translate s asid vaddr=
3   case lookup_tlb s asid vaddr of
4     Miss⇒let v_page=vaddr_to_vpage vaddr;
5           pt_r=pt_walk s asid v_page
6       in if pt_r=None
7          then (s,None)
8          else let perms=snd(the pt_r);
9                   r_page=fst(the pt_r);
10                  entry=PTE asid v_page r_page perms
11              in (s(|tlb:=tlb s∪{entry}|),Some(vaddr_to_paddr vaddr entry))|
12  Hit entry⇒(s,Some(vaddr_to_paddr vaddr entry))|
13  Incon⇒(s,None)
```

上述定义中, 第 3 行执行了遍历 TLB 操作, 结果为缺失时, 先通过 *vaddr_to_vpage* 函数将虚拟地址转成虚拟页号; 然后, 在第 5 行进行页表遍历, 遍历结果保存到 *pt_r* 中; 第 6 行对其判断; 第 7 行表示找不到对应物理页; 第 8–10 行根据获得的页表项数据构建 TLB 的表项; 然后, 在第 11 行中加入到 *tlb* 中, 并返回物理地址; 第 12 行、第 13 行分别表示 TLB 命中和 TLB 不一致的两种情况.

对于用户读写操作的建模, 只需要在 MMU 转换函数基本上添加对访问权限的判断即可. 以写内存操作 *mmu_write* 为例, 形式定义如下.

```
1   mmu_write:: state⇒asid⇒vaddr_t⇒byte⇒state×bool
2   mmu_write s asid vaddr val=
3     if pfWrite∈get_perms s asid(vaddr_to_vpage vaddr)
4     then let paddr=snd(mmu_translate s asid vaddr)
5        in if paddr=None then (s,False) else (mem_write s(the paddr) val,True)))
6     else (s,False))"
```

第 3 行判断指定页面是否存在写权限; 第 5 行中, *mem_write* 函数直接更新物理内存 *heap* 中的值. *mmu_write* 返回值类型中 *bool* 类型指示写操作是否成功.

2.4 地址空间管理操作

L4 通过 3 个操作来对地址空间进行管理: 创建地址空间、初始化地址空间和删除地址空间. 创建地址空间操作建立新的地址空间, 并将其加入已创建的地址空间集合中. 此时, 该地址空间中虚拟页的映射均为空. 初始化地址空间操作将已创建(尚未初始化)的地址空间加入到初始化地址空间集合中, 该操作的目的是保证地址空间中的线程能够被激活. 本文不考虑线程部分的建模, 因此忽略了该操作. 删除地址空间操作将指定地址空间从两集合中移除, 并回收其他地址空间中依赖该地址空间中页面的所有映射, 最后释放自身映射. 下面以删除地址空间为例进行介绍, 该操作形式定义如下.

```
1   DeleteAddressSpace:: state⇒spaceName_t⇒state
```

```
2  DeleteAddressSpace s space=
3    if space∈spaces s∧¬dIsPrivilegedSpace space
4    then let s1=Unmap_fpage s space 0 UNIV page_maxnum
5      in s1(|space_mapping:=(space_mapping s1)(space:=None)|)
6    else s
```

在该定义中, 首先判断空间是否已经被创建, 并且删除的空间是否是特权空间; 其次, 采用 *Unmap_fpage* 函数对指定地址空间 *space* 从第 0 页开始到最大页 *page_maxnum* 依次回收(UNIV 表示删除的访问权限集合是全集, 因此是回收操作而不是减少权限); 最后, 将 *space* 移除两集合并删除映射.

3 功能正确性

功能正确性是指程序是否实现了问题所要求的功能. 如果一个程序满足其程序规范, 则该程序是正确的. 霍尔逻辑[43]表明: 要证明一个程序完全正确(total correctness), 需要保证该程序满足部分正确性(partial correctness), 并且该程序一定能终止(termination). 部分正确性通过霍尔逻辑中的霍尔三元组表示, 符号形式为$\{P\}c\{Q\}$, 其中, P 和 Q 是一阶逻辑公式, 分别表示前置条件和后置条件. 霍尔三元组的含义为: P 有效的情况下执行程序 c 后, Q 有效. 本文依据霍尔逻辑, 采用霍尔三元组来表示事件的部分正确性, 然后证明霍尔三元组成立以及该事件满足终止性. 以 *unmap* 函数为例, 部分正确性引理如下:

$$\{get\_perms\ s\ sp\ v \subseteq perms\}\ s'=unmap\ s\ sp\ v\ perms\{\nexists sp', v', s' \vdash (Virtual\ sp'\ v') \rightsquigarrow^{+} (Virtual\ sp\ v)\}.$$

该式前置条件通过 *get_perms* 函数得到地址空间 *sp* 中虚拟页 *v* 所拥有的权限, 并满足该权限是待删除权限 *perms* 的子集. 在执行 *unmap* 操作后, 子页面不再具备任何权限, 因此回收虚拟页 *v* 曾经映射出去的所有页. 结果需在后置条件保证新状态 s' 中不存在到虚拟页 *v* 的路径. 容易看出, 部分正确性的定义实际是对前后置条件的构造. 本文依据 L4 内核开发手册[3]和源代码[22]提取了映射机制相关操作和地址空间管理操作的前后置条件, 共构建了 20 条部分正确性引理.

证明: 部分正确性描述了事件在给定的前置条件下执行应满足后置条件, 产生正确的结果. 大多数部分正确性引理的证明主要依赖事件自身性质, 具体通过事件定义、Isabelle/HOL 理论库和自动证明策略完成证明. 对于映射机制中的 4 个基本操作以及对应的处理不同页面大小的递归函数, 证明过程较复杂, 其原因在于: 这些事件的定义中包含传递闭包或自反传递闭包表示的路径, Isabelle/HOL 提供的策略不支持该类问题的自动证明. 根据证明经验, 将带有闭包的路径情况分为两类.

- 第 1 类是带闭包的路径满足性质 P, 例如传递闭包即证明 $s \vdash x \rightsquigarrow^{+} y \Rightarrow P\ x\ y$, 需要手工引入证明策略 *induct* 或 *induction* 对路径进行归纳. 此时, 证明目标会被划分为如下两个子目标.
  1) 直接路径情况: $s \vdash x \rightsquigarrow^{1} y \Rightarrow P\ x\ y$;
  2) 传递路径情况: $s \vdash x \rightsquigarrow^{1} y \Rightarrow s \vdash y \rightsquigarrow^{+} z \Rightarrow P\ y\ z \Rightarrow P\ x\ z$;
- 第 2 类是证明不存在使得性质 P 成立指定的路径, 即$\nexists x, y, s \vdash x \rightsquigarrow^{+} y \wedge P\ x\ y$, 此时用 Isabelle/HOL 在定义归纳类型是自动产生简化规则 *tran_path.simps* 来简化子目标, 且需手工采用替换策略 *subst* 结合该规则替换目标(直接应用简化策略是不可行的), 进一步生成如下子目标.
  1) 直接路径情况: $s \vdash x \rightsquigarrow^{1} y \Rightarrow \neg P\ x\ y$;
  2) 传递路径情况: $s \vdash x \rightsquigarrow^{1} y \Rightarrow s \vdash y \rightsquigarrow^{+} z \Rightarrow \neg P\ x\ z$.

通过上述方式, 将两类问题的证明转换为直接路径情况和传递路径情况两类子目标的推理, 子目标的证明过程分别为: 展开$\rightsquigarrow^{1}$的定义即可证明直接路径情况; 传递路径情况的证明主要通过依赖性质 P 具有传递性, 结合前提 $P\ y\ z$ 和直接路径的结论 $P\ x\ y$ 推出 $P\ x\ z$. 通过证明所有事件满足部分正确性以及 Isabelle/HOL 自动对终止性的自动证明, 完成了功能正确性验证, 保证了本文形式模型中所有事件的定义符合功能需求. 证毕. □

4 功能安全

本文将功能安全相关性质定义为若干不变式, 因为不变式能够描述该系统在整个运行期间保持不变的关系. 不变式通常被定义形如 *invariant*: *state*⇒*bool*, 表示: 给定的状态是否满足指定的要求. 本文定义了 16 条不变式, 它们描述了页面关系、页表和 TLB 之间关系等, 覆盖了文献[9]中定义的所有属性, 同时还涉及更多新的不变式, 以提高功能安全相关性质的验证完整度. 下面仅介绍其中 7 条关键不变式.

不变式 1. 地址空间中的任何页面不会形成自环和闭环结构:

$$\forall s, space\ v1, (\nexists v2, s\vdash(Virtual\ space\ v1)\rightsquigarrow^{+}(Virtual\ space\ v2)).$$

对于任意地址空间 *space* 中的任意虚拟页 *v*1, 不存在一条能够到达 *space* 中任意页面的路径. 文献[9]已给出防止地址空间结构中出现图 3 自环结构的不变式, 本文提出的不变式 1 在自环结构基础上拓展了闭环 1 和闭环 2 两种情况, 保证了这 3 种情况均不会发生. 需要说明的是: 虽然上述公式仅仅涉及虚拟页面, 但仍然保证了物理页面也不会出现环状结构. 这是由于虚拟页面不可能被映射到物理页面中, 即地址空间结构中物理页面不会成为子页面.

不变式 2. 一个页面是有效的等价于该页面存在到物理页面的路径:

$$\forall s, x, (s\vdash x\leftrightarrow(\exists r, s\vdash x\rightsquigarrow^{*}(Real\ r))).$$

有效页会映射到物理页; 反之, 能映射到物理页的页面必然是有效页. 结合不变式 1, 保证了地址空间结构中任意一条路径均能终止于物理页面.

不变式 3. 任何页面的直接父页面唯一:

$$\forall s, x, y1, y2, s\vdash x\rightsquigarrow^{1}y1\wedge s\vdash x\rightsquigarrow^{1}y2\rightarrow y1=y2.$$

若 *x* 存在直接父页面 *y*1 和 *y*2, 则 *y*1 和 *y*2 一定相同. 该式确保了每个页面到达物理页面的路径的唯一性, 亦保证了页表遍历函数 *pt_walk* 中获取的物理页面最多存在一个. 此外, 不变式 3 借助于直接路径 *direct_path* 描述, 替代了文献[9]中采用传递闭包进行定义的不变式, 即:

$$s\vdash(Virtual\ n\ v)\rightsquigarrow^{+}(Real\ r)\wedge(s\vdash(Virtual\ n\ v)\rightsquigarrow^{+}(Real\ r'))\rightarrow r=r'.$$

该不变式缺点是: 必须通过手工方式归纳证明每个事件均满足该不变式, 显著提高了证明任务量. 本文中, 结合不变式 1 和不变式 3 完成了该公式的证明, 后续仅需通过 Isabelle/HOL 中已有策略自动证明不变式 3 即可.

不变式 4. 有效页面访问权限不为空:

$$\forall s, space\ v, s\vdash(Virtual\ space\ v)\rightarrow get\_perms\ s\ space\ v\neq\{\cdot\}.$$

当通过删除操作使得页面的访问权限为空时, 映射关系也将删除, 即防止了访问权限被完全删除但仍保留映射的情况发生. 此外, 该不变式避免了映射或授予其他页面时访问权限为空的无效操作.

不变式 5. 任何页面的访问权限是直接父页面访问权限的子集:

$$\forall s, sp1, sp2, v1, v2, s\vdash(Virtual\ sp1\ v1)\rightsquigarrow^{1}(Virtual\ sp2\ v2)\rightarrow get\_perms\ s\ sp1\ v1\subseteq get\_perms\ s\ sp2\ v2.$$

显然, 执行映射或授予操作时, 被映射方获得的访问权限不得超过映射方.

不变式 6. TLB 表项和页表项数据一致, 包括访问权限和转换后的物理页号:

$$\forall s, space\ va, \text{case } lookup\_tlb\ s\ space\ va \text{ of } Miss\Rightarrow \text{True}\mid Hit(\text{PTE } space\ vpage\ rpage\ perms)\Rightarrow$$
$$s\vdash(Virtual\ space\ vpage)\rightsquigarrow^{+}(Real\ rpage)\wedge get\_perms\ s\ space\ vpage=perms$$

TLB 表项和页表项一致是指缓存到 TLB 中的部分页表数据与对应的页表项数据是相同的, 即: 对于指定地址空间, 同一个页号通过 TLB 转换和页表转换得到的物理页号相等. 显然, 当 TLB 未缓存指定的页表项数据时, TLB 不存在不一致. 当 TLB 命中时, 得到的物理页号和权限应与页表查询到的结果均相同.

不变式 7. TLB 自身不会出现不一致的情况:

$$\forall s, space\ va, lookup\_tlb\ s\ space\ va\neq Incon.$$

该公式表明, TLB 中不会出现 ASID、页号均相同的两个表项.

证明: 功能安全性描述为不变式后, 证明目标转换为状态机中所有状态满足不变式, 而状态机中所有状态可用可达状态表示, 因此证明可达状态满足不变式即可, 符号表示为: $R\ s\Rightarrow Inv\ s$, Inv 是本文所有不变式的交集. 依据可达状态的定义, 可将该目标拆解为两个子目标, 分别是初始状态 $s0$ 满足不变式和每个事件的执行都保持不变式成立. 由于初始状态被静态定义, 采用自动证明策略 auto 即可证明. 对于后者, 需要推理任意事件的执行满足每一个不变式, 对于指定的不变式 I 和事件 e, 所构建的引理如下:

$$\{I\ s\}\ s'=step(s,e)\ \{I\ s'\}.$$

其中, $step$ 表示从状态 s 执行单个事件 e 的函数. 该公式的含义为: 状态 s 满足不变式 I, 执行事件 e 后得到的新状态 s'仍然满足不变式 I. 容易看出: 该公式同样符合霍尔三元组形式, 该公式前置条件和后置条件通常是相同的, 除非在前置条件中引入更多的不变式, 此时引理如下:

$$\{I_1\ s\wedge I_2\ s\wedge\ldots\}\ s'=step(s,e)\ \{I_1\ s'\}.$$

通过不变式 I_1 和 I_2 等共同推出事件 e 执行后的状态 s'满足 I_1. 本文构建的不变式引理涵盖上述两种形式的引理. 需要说明的是: 在部分正确性公式中, 前置条件和后置条件通常是不同的, 强调事件执行引起的状态变化, 进一步判断变化后的结果是否正确. 故两者证明方式不同. □

完成不变式引理的定义后, 使用 Isabelle/HOL 对其进行证明. 本文大部分工作在于完成映射机制中 4 个递归函数关于这些不变式的推理. 考虑递归函数 F, 采用子函数 f 依次操作固定大小的页面, 在操作当前页面 v 后, 不仅需要保证 v 与其他页面之间关系满足不变式, 还需要保证该操作不会影响后续待处理的所有页面的关系. 前者利用子函数 f 的性质进行证明. 后者的证明较复杂, 体现在两个方面: 1. 当不变式引理为上述第 2 种形式时, 需要证明操作当前页面 v 之后得到的新状态满足前置条件中的所有不变式, 即 I_1 和 I_2 等; 2. 需要通过归纳方式找到 v 所有的子页面, 然后证明这些子页面与后续页面的所有子页面不相交. 下面以 *Unmap* 函数为例, 对证明过程进行介绍. 如图 8 所示: 假设 *Unmap* 当前操作 $v1$ 页面, 它调用子函数 *unmap* 收回 $v1$ 的所有子页面. 此时, 需要证明 *unmap* 操作满足以下 3 个引理.

引理 1. *unmap* 操作不会影响 $v1$ 到直接父页面 $r1$ 的映射(如图 8 蓝色部分所示).

$$loop\_free\ s\Rightarrow the((unmap\ s\ sp1\ v1\ perms)@sp1)\ v1=the(s@sp1)\ v1.$$

loop_free 是根据不变式 1 的定义名称, 函数 *the* 获取 *option* 类型中 *Some a* 的 *a*.

引理 2. *unmap* 操作将 $v1$ 的所有子页面的映射关系全部删除, 即收回 $v3$, $v6$, $v7$ (如图 8 灰色部分所示).

$loop\_free\ s\Rightarrow s\vdash(Virtual\ sp\ v)\rightsquigarrow^{+}(Virtual\ sp1\ v1)\Rightarrow get\_perms\ s\ sp\ v\subseteq perms\Rightarrow$

$the((unmap\ s\ sp1\ v1\ perms)@sp)\ v=None$.

引理 3. *unmap* 操作不影响其他页面 $v2$ 相关的映射关系, 即收回的页面不包括 $v4$, $v5$, $v8$ 和 $v9$ (如图 8 橙色部分所示).

$loop\_free\ s\Rightarrow s\vdash(Virtual\ sp\ v)\rightsquigarrow^{+}(Virtual\ sp1\ v2)\Rightarrow v1\neq v2\Rightarrow$

$(unmap\ s\ sp1\ v1\ perms)\Rightarrow(Virtual\ sp\ v)\rightsquigarrow^{+}(Virtual\ sp1\ v2)$.

引理 1 是为简化 *Unmap* 函数关于不变式的证明而提出. 若证明 *Flush* 函数, 该引理应修改为: *Flush* 操作会删除 $v1$ 到直接父页面 $r1$ 的映射.

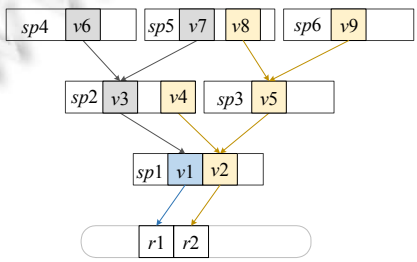


图 8 Unmap 关于不变式的证明过程

5 信息安全

信息安全通常指系统不因为偶然或者恶意的原因而遭到破坏、更改和泄漏, 所涵盖的内容包括信息的保密性、完整性和真实性等. 本文在信息安全方面主要考虑信息的保密性, 基于信息流安全属性中无泄漏的定义, 构建针对 L4 虚拟内存子系统的无泄漏定义, 保证内存数据之间的隔离性. 信息流安全属性的定义主要依赖于 4 个关键定义: 安全域、事件域、观察函数以及信息流策略. 由于 L4 未提供任何信息流策略或访问控制策略来保护用户信息, 本文依据 L4 的特点以及已有工作[35,36,39,40]的经验, 提出了针对 L4 地址空间之间的信息流策略, 该策略使得无泄漏得以证明. 需要说明的是: 本文实际上完成了无干扰的部分证明, 在此过程中, 发现了系统不满足该属性, 且很难通过修改信息流策略、修改内核设计或代码等方式使其具有对无干扰的可满足性, 这与 L4 自身追求灵活性、可靠性和性能而不考虑安全(security)的特点是符合的, 因此, 本文仅完成了无干扰的部分证明. 后续将上述关键定义称为安全配置, 并将针对本文定义的无干扰和无泄漏称为安全属性. 以下首先介绍针对 L4 虚拟内存子系统的安全配置, 其次借助于安全配置定义对安全属性形式化, 最后对安全属性中无泄漏进行证明, 同时简要介绍系统不满足无干扰的原因.

5.1 安全配置

- 安全域

安全域定义了系统中应用程序的安全等级, 不同等级的线程在不同的域中执行. 安全域的主要划分方式是: 分别为每个进程或分区定义成一个域, 即使等级相同的应用程序也在自身所在域中执行, 每个安全域保护自身可见的数据; 通常, 除应用程序的安全域外, 还需单独定义一个安全域, 用于表示内核代码所执行的区域. 在经典案例 seL4[35,36]的信息流安全验证中, 首先将安全域定义为进程(process), 然后分别为每个进程定义一个安全域, 内核域定义为 *sched*; 在 ARINC 653[39,40]的验证案例中, 安全域定义方式和上述类似, 区别在于安全域定义为分区(partition). 本文关于安全域的定义依据 L4 中两个设计原则: (1) 采用地址空间将线程划分到不同区域, 使得线程之间能够保持隔离; (2) 采用 IPC 机制使得不同地址空间中的线程能够通信, 并且通信策略是以地址空间为对象进行配置. 因此, 本文将地址空间(space)作为应用程序部分的安全域(以下称为用户域). 具体来说, L4 地址空间包括特权地址空间 *Sigma*0*Space*, *RootServerSpace* 和后续被创建的普通地址空间, 每个地址空间作为一个用户域, 构成用户域集合 $\boldsymbol{S}$; 其次, 本文定义了内核域 *Sched*, 用于描述切换地址空间(scheduling)这一事件的执行所在域. 因此, L4 安全域集合 $\boldsymbol{D}=\boldsymbol{S}\cup\{Sched\}$.

- 事件域

事件域是一个函数, 它定义了每个事件执行时所在的安全域. 在传统的事件域定义中, 一个事件执行所在域依赖于事件本身, 事件一旦确定, 安全域随之确定. 例如在文献[32]中, 将事件域定义为 *dom e*. 然而, 当系统执行事件时, 依据不同状态能够得到不同的安全域, 上述公式已不满足事件域的要求. 事件域发展为 *dom s e*, 该公式在原有基础上增加了状态参数, 表示获取 *s* 状态下事件 *e* 执行所在的域, 在文献[35,36]中得到了应用. L4 中的事件大多发生在当前地址空间(即 *current s*)中, 符合后者的定义方式, 因此, 本文定义事件域函数 *dom* 如下:

$$dom\ s\ e=\text{case } e \text{ of } eSchedule\Rightarrow Sched|eSpace\_\Rightarrow RootServerSpace|\_\Rightarrow current\ s,$$

其中, *eSchedule*, *eSpace* 是事件的标签, 分别标识调度和地址空间管理操作. L4 调度操作在内核域中执行, 地址空间管理操作在 *RootServerSpace* 中执行, 其他事件发生在当前地址空间中.

- 观察函数

观察函数定义了每个安全域可见的数据, 安全域中数据对于其他安全域而言是私有的, 因此安全域保护数据的作用通过该函数体现. 对于观察函数 *observe*, 公式 *observe s d t* 表示在 *s* 状态和 *t* 状态安全域 *d* 可见的数据是否一致, 符号化为 *s*~*d*~*t*. 进一步地, 利用 *observe* 可定义观察等价性关系 *obs_equiv s t es d*, 表示两状态 *s*, *t* 执行事件序列 *es* 后, 得到的新状态中安全域 *d* 中的数据是否等价, 符号化为 $s\triangleleft es\cong t\triangleleft es@d$, 它用于判断安全域被保护的数据是否被篡改. 考虑到安全属性的推理, 需保证 *observe* 满足自反性、对称性和传递性. 观

察函数的具体内容, 依据每个安全域所保护的数据进行定义. L4 虚拟内存子系统中, 内核作为调度机制管理地址空间的切换, 实现方式是从已创建的地址空间中选择其一并设置为当前地址空间, 因此内核域可见当前地址空间和所有已创建的地址空间, 即 *current* 字段和 *space_mapping* 字段的定义域. 对于用户域, 每个地址空间包含已映射到的内存数据和页面映射关系. 此外, 地址空间能够将自身页面映射到其他地址空间, 映射方和被映射方在拥有读写权限的前提下均能够修改所映射到的内存数据(共享内存). 容易看出, 被映射方必然信任映射方. 因为当被映射方不信任映射方时, 前者将拒绝后者发起的映射操作从而不建立映射关系. 这种信任关系使得父页面所在地址空间能够观察到所有子页面地址空间中的映射, 因此用户域可见的数据包括自身安全域所拥有内存数据、自身映射、所有子页面所在安全域及其映射. 综上所述, 本文定义观察函数 *observe* 如下:

$s\sim d\sim t \equiv$ **if** $d=Sched$ **then** $current\ s=current\ t \wedge spaces\ s=spaces\ t$ **else**

$mem\ s\ d=mem\ t\ d \wedge s@d=t@d \wedge child\ s\ d=child\ t\ d \wedge (\forall d' \in child\ s\ d,\ s@d'=t@d')$,

其中, 函数 *child* 求出所有子页面所在地址空间.

- 信息流策略

信息流策略明确了安全域之间的访问策略, 即安全域之间是否存在信息流. 对于 n 个安全域, 访问策略可用 $N\times N$ 的二维矩阵描述, 矩阵中第 i 行 j 列(i,j)的值为 True, 表示允许有安全域 i 至安全域 j 的信息流, 安全域之间访问策略关系是有向的, 且不保证该矩阵对称. 该矩阵关系形式化为返回值为 *bool* 类型的策略函数 *policy i j*, 符号化为 $i\rightsquigarrow j$. 在带有内核域的系统[35,36,39,40]中, 通常需要保证以下 3 条关系.

1) $\forall d, Sched\rightsquigarrow d$;

2) $\forall d, d\rightsquigarrow Sched \rightarrow d=Sched$;

3) $\forall s, t, a, s\sim Sched\sim t \rightarrow dom\ s\ a=dom\ t\ a$.

公式 1)表示内核域存在到任意域的信息流; 公式 2)表示存在对内核域有信息流的安全域只能为内核域自身; 公式 3)直观含义为: 若从状态 s 和 t 观察到内核域的信息是等价的, 则任何事件在这两状态中的执行所在域是不可区分的. 该公式确保用户域的切换只能由内核来完成. 事件域和观察函数已经保证了公式 3)成立, 本文将公式 1)和公式 2)添加到信息流策略中. 此外, 在定义观察函数时已提到, L4 通过映射方式来分配内存, 映射方和被映射方能够互相影响对方数据, 即信息流是相互的, 并且映射方通过路径能够影响所有的子页面所在地址空间, 因此, 用户域之间的信息流满足对称性和传递性关系. 综上所述, 本文提出了传递性类型的信息流策略, 如图 9 所示: 内核域 *Sched* 存在到任何域的信息流, 且用户域均不能流向 *Sched*, 满足上述关系 1)和关系 2); 用户域之间的信息流被静态定义, 且满足对称和传递关系, 例如, 图 9 中用户域 $Space_1$ 与 $Space_2$ 能相互通信, $Space_2$ 与 $Space_N$ 两者均不能与对方通信; 信息流策略满足自反关系, 每个安全域能影响自身数据. 信息流策略形式定义如下:

$d1\rightsquigarrow d2 \equiv$ **if** $d1=d2 \vee d1=Sched$ **then** True **else if** $d2=Sched$ **then** False **else** $CConf\ SysConf\ d1\ d2$,

其中, *CConf* 函数定义了用户域之间的信息流, 该函数定义于系统配置 *SysConf*. 如下是函数 *CConf* 满足对称性和传递性的两个约束.

- $CConf\ SysConf\ d1\ d2=CConf\ SysConf\ d2\ d1$;
- $CConf\ SysConf\ d1\ d2 \wedge CConf\ SysConf\ d2\ d3 \rightarrow CConf\ SysConf\ d1\ d3$.

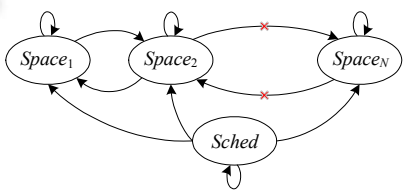


图 9 信息流策略

5.2 安全属性

在形式化安全属性之前, 引入两个核心辅助函数, 分别是源函数(*sources*)和清除函数(*ipurge*). 公式 *sources*(*s*,*es*,*d*)求出从当前状态 *s* 执行事件序列 *es* 过程中对安全域 *d* 存在信息流的安全域集合, 存在信息流包括直接($dom(s,e)\rightsquigarrow d$)和间接($dom(s,e)\rightsquigarrow dom(s_1,e_1)\rightsquigarrow dom(s_2,e_2)\rightsquigarrow d$)这两种方式. *sources* 函数定义如下:

$$sources(s,[\cdot],d)=\{d\}$$

$$sources(s,e\#es,d)=\begin{cases}sources(step(s,e),es,d)\cup\{dom(s,e)\}, & \text{如果}\exists v\in sources(step(s,e),es,d)\wedge dom(s,e)\rightsquigarrow v,\\ sources(step(s,e),es,d), & \text{否则}\end{cases}$$

其中, *step* 为执行单个事件的函数. 公式 *ipurge s es d* 清除事件序列 *es* 中与安全域 *d* 无关的所有事件, 从而构建一个新的事件序列. 显然, 被清除的事件的执行所在域不在 *sources s es d* 集合中. *ipurge* 函数形式定义如下:

$$ipurge(s,[\cdot],d)=[\cdot]$$

$$ipurge(s,e\#es,d)=\begin{cases}e\#ipurge(step(s,e),es,d), & \text{如果}dom(s,e)\in sources(s,e\#es,d),\\ ipurge(step(s,e),es,d), & \text{否则}\end{cases}$$

- 无干扰

借助上述安全配置和辅助函数, 可定义无干扰如下:

$$noninterference\equiv s_0\triangleleft es\cong s_0\triangleleft ipurge(s_0,es,d)@d.$$

即: 从初始状态 s_0 执行事件序列 *es* 和执行 *ipurge*(s_0,*es*,*d*)之后产生的新状态, 对于 *d* 来说是不可区分的.

- 无泄漏

一个带内核域的系统满足无泄漏需保证: 系统中任意两状态 *s* 和 *t*, 在关于内核域和执行 *es* 过程中所有影响 *d* 的安全域等价的前提下, 从 *s* 和 *t* 分别执行 *es* 所产生的新状态, 对于 *d* 来说是不可区分的. 本文定义无泄漏如下:

$$nonleakage\equiv R\ s\wedge R\ t\wedge s\approx sources(s,es,d)\approx t\wedge s\sim Sched\sim t\rightarrow s\triangleleft es\cong t\triangleleft es@d.$$

对于 $s\approx D\approx t$ 表示: 对于 $d\in D$, 有 $s\sim d\sim t$.

5.3 证 明

安全属性通过展开条件进行证明. 展开条件指单步一致性(step consistent)和局部不变性(local respect)这两条性质.

- 单步一致性表示: 对于任意的可达状态 *s* 和 *t* 以及域 *d*, 若 *s* 和 *t* 观察到 *d* 中数据相同、待执行的事件能影响 *d* 中数据、*s* 和 *t* 观察到待执行事件的域中数据相同, 那么在 *s* 和 *t* 状态下执行该事件后, *d* 中数据仍然相同. 在本文中定义如下:

$$step\text{-}consistent\equiv\forall e,d,s,t,s',t',R\ s\wedge R\ t\wedge(dom(s,e)\rightsquigarrow d\rightarrow s\sim dom(s,e)\sim t)\wedge s\sim Sched\sim t\wedge$$
$$s\sim d\sim t\wedge s'=step(s,e)\wedge t'=step(t,e)\rightarrow s'\sim d\sim t'.$$

- 局部不变性表示域中数据只受到能够向其发送消息的域的影响:

$$local\text{-}respect\equiv\forall e,d,s,s',R\ s\wedge dom(s,e)\not\rightsquigarrow d\wedge s'=step(s,e)\rightarrow s\sim d\sim s'.$$

上述定义中, 符号$\not\rightsquigarrow$是$\rightsquigarrow$的否定.

文献[35]将 *step-consistent* 定义为保密性(*confidentiality-u*), 并完成了 *nonleakage* 与 *confidentiality-u* 两者等价的可靠性(soundness)和完备性(completeness)证明, 因此, 无泄漏的证明只需保证系统状态机满足 *step-consistent* 即可. 而无干扰的证明需要保证 *step-consistent* 和 *local-respect* 两者成立, 因为无干扰和展开条件之间满足关系: *step-consistent*$\wedge$*local-respect*$\Rightarrow$*noninterference*. 系统状态机关于展开条件的可满足性证明可拆分为每个事件的可满足性证明. 对此, 本文为每个事件定义了满足单步一致性和局部不变性的引理, 以创建地址空间 *Create* 这一事件为例, 引理定义如下.

定理 1. 创建地址空间满足单步一致性:

$$R\ s\wedge R\ t\wedge(RootServerSpace\rightsquigarrow d\rightarrow s\sim RootServerSpace\sim t)\wedge s\sim d\sim t\wedge s\sim Sched\sim t\wedge$$

$$s'=Create\ s\ space \wedge t'=Create\ t\ space \rightarrow s' \sim d \sim t'.$$

定理 2. 创建地址空间满足局部不变性:

$$R\ s \wedge RootServerSpace \neq d \wedge s'=Create\ s\ space \rightarrow s \sim d \sim s'.$$

创建地址空间事件只能在 *RootServerSpace* 中执行, 因此, 事件域为 *RootServerSpace*. 对于定理 1 的证明, 主要依赖不变式和 Isabelle/HOL 提供的自动化证明策略如 auto, blast 等完成推理工作, 最终证明系统满足无泄漏属性. 然而, 定理 2 被证明不成立, 其原因为: 用户域中事件的执行改变了内核域可观察到的数据, 即创建地址空间操作改变了已创建的地址空间.

6 验证结果与评估

本节首先统计了代码情况, 然后对验证过程中发现的问题进行描述, 并给出一些解决办法或建议.

- 代码统计

本文在 Isabelle/HOL 中完成了 L4 虚拟内存子系统的形式建模和验证工作, 证明了形式模型满足功能正确性、功能安全和信息流安全部分属性. 信息流部分属性是指针对该系统定义的无泄漏, 同时, 本文对无干扰进行了证明, 发现该系统关于无干扰是无法满足的, 这一点与 L4 不考虑 security 的设计目标相符合. 本文证明代码采用结构化语言 Isar[44]编写, 该语言具有可读性强、复用性高、易于读者理解等特点. 使用该语言实现了共计约 6 000 行代码, 其中, 建模部分 1 500 行左右, 包括 30 个关键定义, 花费了约 1 人月; 其余为证明代码, 功能正确性、功能安全和信息安全这 3 个部分的证明分别约 700 行、1500 行和 2 300 行, 涵盖了 90 条关键引理, 证明阶段花费了 9 人月. 建模和验证代码均通过了 Isabelle 的检查, 证明了本文构建的模型满足 3 类性质.

- 已发现的问题

在建模和验证 L4 虚拟内存子系统过程中, 发现该系统的设计以及源代码存在 3 点问题: 问题 1 在建模过程中发现, 问题 2 和问题 3 在验证时发现. 以下对其详细介绍, 同时对每个问题提出了解决办法或建议.

1. 在授予页面时, 源代码(pistachio-0.4/kernel/src/generic/mapping.cc, 第 208–210 行)[22]处理方式是: 授予方仅删除待授予页面 *v* 到自身的映射, 保留 *v* 的子页面的映射关系, 然后将 *v* 映射到新的页面. 由于 *v* 的子页面不清楚 *v* 映射到新的页面了, 获取到的内存数据与原来不一致. 该操作未按照功能需求将 *v* 后续所有子页面映射关系全部删除. 对此, 本文建议在执行授予操作时, 先对 *v* 执行 *flush* 操作, 然后将 *v* 映射到指定页面. 此外, 源代码(pistachio-0.4/kernel/src/generic/linear_ptab_walker.cc, 第 99–761 行)允许 *Sigma*0*Space* 授予页面给其他地址空间, 当其他地址空间释放这块内存后,系统中不存在一个地址空间到该内存的映射, 因此, 该内存后续无法被映射. 依据本文模型, 不允许在 *Sigma*0*Space* 中执行授予操作;
2. 内核域和用户域之间违背了无干扰这一属性. 创建地址空间和删除地址空间操作执行所在安全域为 *RootServerSpace*, 该域属于用户域, 而内核域可见已创建的地址空间, 即 *spaces s*, 用户域不存在对内核域的信息流, 但能影响内核域可见数据. 例如, 成功创建地址空间后, 使得新状态中已创建的地址空间和原来的不一致. 这导致 *local-respect* 这一展开条件不成立, 进一步可断言无干扰不成立. 本文建议在地址空间的调度策略上为时间片轮转的前提下, 配置好所有地址空间, 不依赖地址空间必须被创建这一条件, 使得内核域仅能观察到当前地址空间字段; 若应用场景为仅需保证数据的机密, 而不严格要求动作的保密性, 则不需修改;
3. 在构建信息流策略时发现: L4 虚拟内存子系统中, 映射机制导致用户域之间均存在信息流, 因为物理内存被 *Sigma*0*Space* 占据, 后续由该地址空间映射到其他地址空间. 由于信息流策略的对称性以及自反性, 任何地址空间中的数据能够通过中间媒介 *Sigma*0*Space* 流向其他任何地址空间. 信息流动过程如图 10 所示.

该情况使得用户域之间不具备任何隔离性. 对此, 本文提出两个解决方案: 1) 限制 *Sigma*0*Space* 中内部数据流动, 该空间中的 *sigma*0 仅作为服务线程用于管理内存分配, 然后考虑更细粒度的安全域, 例如页面,

规定类似于 *Sigma*0*Space* 的服务线程所在地址空间中的页面之间不存在信息流; 2) 每次创建新地址空间的同时创建管理该地址空间内存分配的 *pager*, 该 *pager* 也拥有自身的地址空间, 然后, *Sigma*0*Space* 执行授予操作将需要的内存赠送给 *pager* 所在地址空间, 进一步由 *pager* 执行映射或释放操作. 通过上述两种方式, 能保证更高的信息安全相关性质, 使系统既满足无泄漏又满足无干扰. 需要说明的是: 后者付出了额外的内存开销, 并且信息流策略是动态的, 验证过程较复杂.

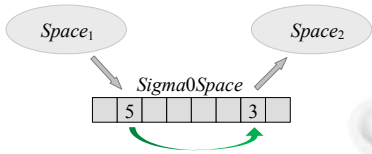


图 10 信息流传递过程

7 总 结

本文提出了关于 L4 虚拟内存子系统的形式模型, 该模型考虑了系统中复杂的数据结构, 建模了映射机制所有操作、带 TLB 的 MMU 行为和地址空间管理操作, 弥补了已有模型中灵活页面类型、访问权限字段以及对不同大小页面的处理未定义的不足, 并修正了部分函数允许执行的条件, 完善了 L4 虚拟内存子系统所涉及的行为形式建模. 为提高 L4 虚拟内存子系统正确性和可靠性, 本文形式化了功能正确性、功能安全和信息安全这 3 个方面的属性, 并采用机器检查方式证明了模型满足这 3 类属性. 通过形式建模和验证, 发现了源代码中存在不处理覆盖映射以及破坏功能安全的 3 个问题, 本文在形式模型中均对其进行了处理, 同时给出了修正源代码的具体措施. 此外, 虽然 L4 目标在于提高灵活性、可靠性以及性能, 但是通过指定的策略能够实现部分安全性, 根据模型关于安全属性的证明结果表明, 本文提出的策略能够保证地址空间占有的内存满足隔离性.

本文采用手工方式建模和验证了 L4 虚拟内存子系统, 在未来工作中, 考虑提出一个自动化验证工具: 在该工具中, 通过人工方式构建子系统的抽象模型; 然后, 使用该工具将实现子系统的源代码自动转为具体模型; 接下来, 自动完成抽象模型关于预期属性的证明以及抽象模型和具体模型的精化证明; 最终, 在源代码层次实现对 L4 虚拟内存子系统的形式验证.

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. Towards real microkernels. Communications of the ACM, 1996, 30(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, *et al*. Mach: A new kernel foundation for UNIX development. In: Proc. of the 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): 1−29. [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 (SOSP). ACM, 2013. 133−150. [doi: 10.1145/2517349.2522720]

[7] Härtig H, Hohmuth M, Liedtke J, *et al*. The performance of μ-kernel-based systems. In: Proc. of the 16th ACM Symp. on Operating System Principles (SOSP). 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. ACM, 2009. 25−30. [doi: 10.1145/1519130.1519135]

[9] Tuch H, Klein G. Verifying the L4 virtual memory subsystem. In: Proc. of the NICTA Formal Methods Workshop on Operating Systems Verification. National ICT Australia, 2004. 73−97.

[10] Klein G, Tuch H. Towards Verified Virtual Memory in L4. TPHOLs Emerging Trends, 2004.

[11] Saraswat VA, Jagadeesan R, Michael MM, *et al*. A theory of memory models. In: Proc. of the 12th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP). ACM, 2007. 161−172. [doi: 10.1145/1229428.1229469]

[12] Leroy X, Blazy S. Formal verification of a C-like memory model and its uses for verifying program transformations. Journal of Automated Reasoning, 2008, 41(1): 1−31. [doi: 10.1007/s10817-008-9099-0]

[13] Gallardo MM, Merino P, Sanán D. Model checking dynamic memory allocation in operating systems. Journal of Automated Reasoning, 2009, 42(2): 229−264. [doi: 10.1007/s10817-009-9124-y]

[14] Mansky W, Garbuzov D, Zdancewic S. An axiomatic specification for sequential memory models. In: Proc. of the 27th Int'l Conf. on Computer Aided Verification. Cham: Springer, 2015. 413−428. [doi: 10.1007/978-3-319-21668-3_24]

[15] Ševčík J, Vafeiadis V, Nardelli FZ, *et al*. CompCertTSO: A verified compiler for relaxed-memory concurrency. Journal of the ACM (JACM), 2013, 60(3): 1−50. [doi: 10.1145/2487241.2487248]

[16] Tews H, Völp M, Weber T. Formal memory models for the verification of low-level operating-system code. Journal of Automated Reasoning, 2009, 42(2): 189−227. [doi: 10.1007/s10817-009-9122-0]

[17] Vaynberg A, Shao Z. Compositional verification of a baby virtual memory manager. In: Proc. of the 2nd Int'l Conf. on Certified Programs and Proofs (CPP). Berlin, Heidelberg: Springer, 2012. 143−159. [doi: 10.1007/978-3-642-35308-6_13]

[18] Alkassar E, Schirmer N, Starostin A. Formal pervasive verification of a paging mechanism. In: Proc. of the 14th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Berlin, Heidelberg: Springer, 2008. 109−123. [doi: 10.1007/978-3-540-78800-3_9]

[19] Blanchard A, Kosmatov N, Lemerre M, *et al*. A case study on formal verification of the Anaxagoros hypervisor paging system with Frama-C. In: Proc. of the 20th Int'l Workshop on Formal Methods for Industrial Critical Systems (FMICS). Cham: Springer, 2015. 15−30. [doi: 10.1007/978-3-319-19458-5_2]

[20] Bolignano P, Jensen T, Siles V. Modeling and abstraction of memory management in a hypervisor. In: Proc. of the 19th Int'l Conf. on Fundamental Approaches to Software Engineering (FASE). Berlin, Heidelberg: Springer, 2016. 214−230. [doi: 10.1007/978-3-662-49665-7_13]

[21] Zhao Y, Sanán D. Rely-guarantee reasoning about concurrent memory management in zephyr RTOS. In: Proc. of the 31st Int'l Conf. on Computer Aided Verification (CAV). Cham: Springer, 2019. 515−533. [doi: 10.1007/978-3-030-25543-5_29]

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

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

[24] Klein G, Norrish M, Sewell T, *et al*. seL4: Formal verification of an OS kernel. In: Proc. of the 22nd ACM SIGOPS Symp. on Operating Systems Principles (SOSP). Montana: ACM, 2009. 207−220. [doi: 10.1145/1629575.1629596]

[25] Gu R, Shao Z, Chen H, *et al*. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In: Proc. of the 12th USENIX Symp. on Operating Systems Design and Implementation (OSDI). Savannah: USENIX Association, 2016. 653−669.

[26] Fang B, Sighireanu M. Hierarchical shape abstraction for analysis of free list memory allocators. In: Proc. of the 26th Int'l Symp. on Logic-based Program Synthesis and Transformation (LOPSTR). Cham: Springer, 2016. 151−167. [doi: 10.1007/978-3-319-63139-4_9]

[27] Fang B, Sighireanu M. A refinement hierarchy for free list memory allocators. In: Proc. of the 2017 ACM SIGPLAN Int'l Symp. on Memory Management (ISMM). ACM, 2017. 104−114. [doi: 10.1145/3092255.3092275]

[28] Fang B, Sighireanu M, Pu G, *et al*. Formal modelling of list based dynamic memory allocators. Science China Information Sciences, 2018, 61(12): 1−16. [doi: 10.1007/s11432-017-9280-9]

[29] Marti N, Affeldt R, Yonezawa A. Formal verification of the heap manager of an operating system using separation logic. In: Proc. of the 8th Int'l Conf. on Formal Engineering Methods (ICFEM). Berlin, Heidelberg: Springer, 2006. 400–419. [doi: 10.1007/11901433_22]

[30] Su W, Abrial JR, Pu G, *et al*. Formal development of a real-time operating system memory manager. In: Proc. of the 20th Int'l Conf. on Engineering of Complex Computer Systems (ICECCS). IEEE, 2015. 130−139. [doi: 10.1109/ICECCS.2015.24]

[31] Yu D, Hamid NA, Shao Z. Building certified libraries for PCC: Dynamic storage allocation. In: Proc. of the 12th European Symp. on Programming (ESOP). Berlin, Heidelberg: Springer, 2003. 363−379. [doi: 10.1007/3-540-36575-3_25]

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

[33] 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]

[34] Von Oheimb D. Information flow control revisited: Noninfluence=noninterference+nonleakage. In: Proc. of the 2004 European Symp. on Research in Computer Security (ESORICS). Berlin: Springer, 2004. 225−243. [doi: 10.1007/978-3-540-30108-0_14]

[35] Murray T, Matichuk D, Brassil M, *et al*. Noninterference for operating system kernels. In: Proc. of the 2nd Int'l Conf. on Certified Programs and Proofs (CPP). Berlin, Heidelberg: Springer, 2012. 126−142. [doi: 10.1007/978-3-642-35308-6_12]

[36] Murray T, Matichuk D, Brassil M, *et al*. seL4: From general purpose to a proof of information flow enforcement. In: Proc. of the 2013 IEEE Symp. on Security and Privacy (S&P). IEEE Computer Society, 2013. 415−429. [doi: 10.1109/SP.2013.35]

[37] Costanzo D, Shao Z, Gu R. 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 (PLDI). ACM, 2016. 648−664. [doi: 10.1145/2908080.2908100]

[38] Sigurbjarnarson H, Nelson L, Castro-Karney B, *et al*. 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. 287−305.

[39] Zhao Y, Sanán D, Zhang F, *et al*. 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 (TACAS). Berlin, Heidelberg: Springer, 2016. 791−810. [doi: 10.1007/978-3-662-49674-9_50]

[40] Zhao Y, Sanán D, Zhang F, *et al*. Refinement-based specification and security analysis of separation kernels. IEEE Trans. on Dependable and Secure Computing, 2017, 16(1): 127−141. [doi: 10.1109/TDSC.2017.2672983]

[41] Hawblitzel C, Howell J, Lorch JR, *et al*. Ironclad apps: End-to-end security via automated full-system verification. In: Proc. of the 11th USENIX Symp. on Operating Systems Design and Implementation (OSDI). USENIX Association, 2014. 165−181.

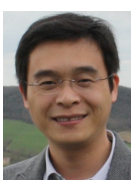
[42] Syeda HT, Klein G. Program verification in the presence of cached address translation. In: Proc. of the 9th Int'l Conf. on Interactive Theorem Proving (ITP). Cham: Springer, 2018. 542−559. [doi: 10.1007/978-3-319-94821-8_32]

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

[44] Wenzel M. The Isabelle/Isar reference manual. 2021. http://isabelle.in.tum.de/doc/isar-ref.pdf

章乐平(1994－), 男, 博士生, 主要研究领域为形式化方法, 自动化验证, 信息流安全.

李悦欣(1998－), 女, 硕士生, 主要研究领域为形式化方法, 程序验证.



赵永望(1979－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为形式化方法, 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.

冯潇潇(1997－), 男, 硕士生, 主要研究领域为形式化方法, 操作系统设计.

王布阳(1996－), 男, 硕士生, 主要研究领域为形式化验证.