计算机研究与发展
Journal of Computer Research and Development
ISSN 1000-1239/CN 11-1777/TP
49 (12): 2702-2712, 2012

操作系统对象语义模型（OSOSM）及形式化验证

钱振江 刘 苇 黄 皓
（南京大学软件新技术国家重点实验室 南京 210046）
（南京大学计算机科学与技术系 南京 210046）
（zhenjiang. qian@gmail. com）

OSOSM: Operating System Object Semantics Model and Formal Verification

Qian Zhenjiang, Liu Wei, and Huang Hao
(*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing* 210046)
(*Department of Computer Science and Technology, Nanjing University, Nanjing* 210046)

Abstract The complexity of the operating system makes its security problems become increasingly prominent. Many research works verify the correctness of the existing operating systems using formal methods, and they mainly focus on verifying the implementation of functions on code level with programming formal logic. In this paper, from the view of system design, we recently propose an operating system object semantics model (OSOSM) based on the higher order logic and type theory. OSOSM adopts a hierarchical structure, including essential effectiveness layer, implementation layer, and optimization layer. OSOSM abstracts the execution subjects and resources as objects of the operating system, and establishes the domain of the discourse for the operating system. We denote the state of the operating system by mapping from the set of operating system object variables to the domain of discourse. OSOSM describes the semantics of system calls, and the security properties with the predicate formulae. This paper also elaborates on the formal methods of verifying that during execution the operating system maintains the security tactics and properties. Finally, we use the self-implemented and the verified trusted operating system (VTOS) as an example to illustrate the semantics correctness of OSOSM, and verify the consistency between the design and safety requirements with Isabelle theorem prover and show that VTOS has the expected security properties.

Key words operating system object; semantics model; formal design; security verification; Isabelle theorem proving

摘 要 操作系统的复杂性使得其安全性问题日益突出. 有不少的研究工作采用形式化的方式对现有的操作系统进行了正确性的验证，这些工作主要是采用程序形式逻辑验证代码级的功能实现性. 从系统设计的角度，以高阶逻辑和类型论为基础，提出了操作系统对象语义模型（OSOSM）. OSOSM 采用分层结构，包括基本功效层、实现层和优化层. OSOSM 将操作系统中的行为主体和资源抽象为操作系统对象，建立操作系统的论域，利用以操作系统对象变元集合为定义域到论域的映射表示操作系统的状态，描述操作系统系统调用等行为的语义，使用逻辑系统的谓词公式表达操作系统的安全属性，给出如何验证操作系统在运行过程中保持安全策略和属性的形式化描述方法. 以实现并经过形式化验证的可信操作系统（VTOS）为例，阐述 OSOSM 的语义正确性. 使用 Isabelle 定理证明工具验证设计和安全需求的一致性，以说明 VTOS 具有预期的安全属性.

收稿日期：2011-05-10；**修回日期**：2011-11-16
基金项目：**国家“八六三”高技术研究发展计划基金项目**（2011AA01A202）；**江苏省科技支撑计划基金项目**（BE2008124）；**国家自然科学基金创新研究群体项目**（60721002）；**江苏省“六大人才高峰”高层次人才项目**（2011-DZXX-035）；**江苏省高校自然科学研究项目**（12KJB520001）

关键词 操作系统对象；语义模型；形式化设计；安全性验证；Isabelle 定理证明

中图法分类号 TP316

随着操作系统（operating system，OS）的规模不断扩大，系统的安全问题正日益地得到广泛的重视. 就安全威胁来说，目前存在着大量的针对 OS 内核和功能模块的攻击. 很多学者对各种攻击方式进行了有针对性的深入研究，取得了很好的效果，如 Abadi 等人的控制流完整性（CFI）方法[1]和 Chen 等人的 ROP 检测方法[2-3]. 利用形式化方法对 OS 进行验证的研究[4-10]也在不断地发展，Bevier 实现了一个小型的 OS 内核 KIT[4]，并对其进行了形式化验证，KIT 内核主要提供了进程调度、错误处理、消息传递、字符 I/O 设备驱动等功能. 文献［5］介绍了 seL4 项目的形式化验证工作，Klein 等人对 8 700 行的 C 语言代码和 600 行的汇编代码进行了验证；Paul 带领的 Verisoft 项目[6]对他们开发的微内核 OS 进行了分层设计，为每层分别进行验证，目的在于对整个系统进行普适形式化证明（pervasive formal verification）. Shao 带领的 Flint 项目组使用 VeriML[7]编程语言以及 λHOL^{ind}[7]逻辑系统实现了可验证的 OS 内核[11]，并提出了开放逻辑框架 OCAP[12]，将 OS 不同模块的验证逻辑结合起来，形成一个完整的验证系统.

现有的 OS 设计和验证往往是对已有的 OS 代码进行形式化证明，主要是验证代码的正确性. 本文认为，要构建一个安全的 OS，需要在系统设计和开发过程中，建立一个完善的 OS 形式化模型，在该模型的基础之上再对系统的功能进行形式化描述，提出 OS 的安全性目标，并对形式化模型和安全性目标的一致性进行形式化验证. 该模型需要具有以下这些特点：

1）该模型构建在完善的元逻辑（meta-logic）的基础之上. 这种元逻辑作为形式化验证过程的可信计算基（trusted computing base，TCB），提供良好的可计算性支持.

2）对系统中的各种对象和元素具有丰富的表达能力，便于形式化验证. OS 中含有各种不同的模块，这些模块往往采用不同的程序逻辑，并且处在不同的抽象层次. 对象语义模型需要能表达这些不同模块的逻辑，并且提供良好的抽象.

3）该模型具有很好的扩展性，便于和后期的系统更新相衔接. OS 随着应用的深入，需要加入更多的功能模块. 模型需要提供可扩展的描述能力，对系统的更新进行抽象表达.

本文提出一个 OS 对象语义模型（operating system object semantics model，OSOSM），该模型以高阶逻辑（higher-order logic，HOL）[13]为元逻辑，并以类型论（type theory）[14]作为类型系统的基础，提供对类型化的 λ 演算（typed lambda calculus）的支持. 我们从系统逻辑的角度阐述该模型，将 OS 中的行为主体和资源抽象为 OS 对象，建立 OS 的论域，利用以 OS 对象变元集合为定义域到论域的映射表示 OS 的系统状态，描述 OS 系统调用等行为的语义，使用 OS 逻辑系统的谓词函数表达 OS 的安全性目标，并且给出如何验证 OS 在运行过程中保持安全策略和属性的形式化描述方法.

就我们所知，OSOSM 是第 1 种采用形式化逻辑建立的 OS 对象模型，使用逻辑系统的谓词公式表达 OS 的安全属性，并且提出采用 Isabelle/HOL[15]定理证明工具并借助形式化推理来验证 OS 在运行过程中保持安全策略和属性的方法.

1 OSOSM 框架

本节我们阐述 OS 对象和 OSOSM 的基本框架.

1.1 OS 对象

OS 是一个服务系统，为用户提出的服务请求和设备产生的事件提供相应的服务. 实际上 OS 可以看成是一个对系统调用请求事件或中断事件进行处理的服务系统. OS 对一个系统调用的处理主要包括对内核数据对象的检索、读取、创建或改写等操作. 例如用户请求创建进程时，OS 会为该新进程分配内存等资源，装载镜像，将其列入相应的进程队列. 又例如用户请求分配动态内存时，OS 将根据伙伴（buddy）系统[16]的对象、进程的页表对象、进程的虚拟内存区域对象等来提供服务.

系统中各种动作的执行效果，可以看成是对系统状态的改变或者迁移. 我们将动作的主体和客体作为对象来看待，同时将系统状态之间的转换看成是动作的主客体对象相互作用的结果. 在这样一个场景下，下面的章节重点阐述 OSOSM 的整体框架.

1.2 OSOSM

从纵向来看，我们的对象语义模型 OSOSM 采用分层结构，分成 3 个层次：基本功效层（essential

effectiveness layer)、实现层 (implementation layer) 和优化层 (optimization layer). 基本功效层描述 OS 以及模块的关键数据对象. 与指称语义[17]的概念相同，我们使用这些关键对象集合上的函数来描述 OS 系统调用的功能语义. 实现层阐述实现系统调用功能所涉及的其他数据对象和相关的函数. 优化层阐述为了优化实现层描述的函数而引入的新的数据对象和相关的函数. 从横向来看，OS 可以按照功能分类进行模块化描述，如图 1 所示，可以分成内核模块、进程管理模块、虚拟内存模块以及文件系统模块等. 每个功能模块在纵向上包含各自的基本功效层、实现层和优化层.

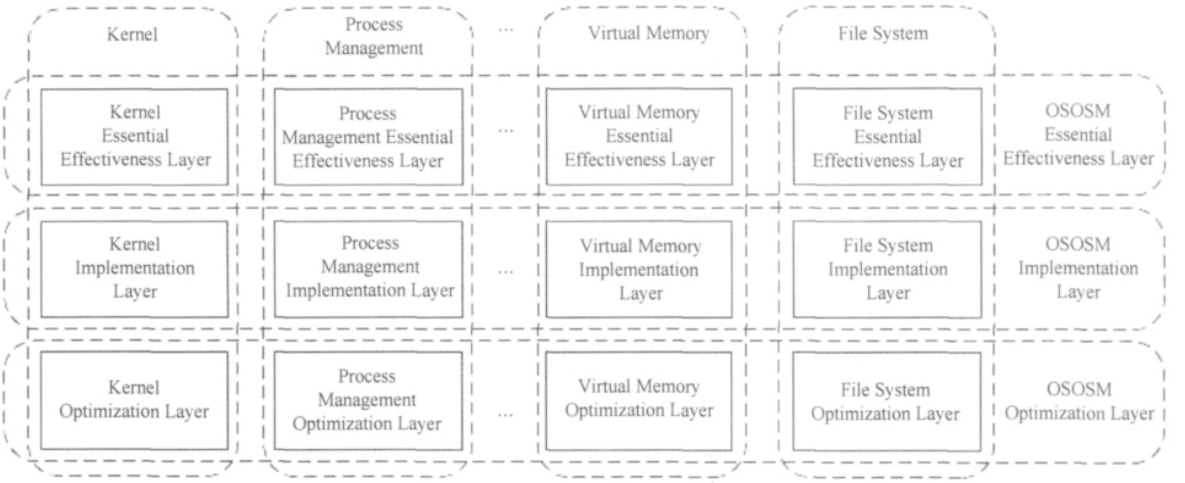

Fig. 1 OSOSM architecture.

图 1 OSOSM 框架

我们以实现的原型系统 VTOS 为例，阐述 OSOSM 模型. VTOS 是按照我们预定的安全目标而设计的微内核 OS，采用了 OSOSM 模型进行形式化设计，其安全性和正确性经过严格的形式化逻辑验证. VTOS 拥有安全核 (secure kernel) 和访问控制安全机制，实现了虚拟内存管理 (VMM)、多线程管理 (MTM)、多核调度管理 (MCSM)、进程管理 (PM) 和文件系统 (FS) 等功能. 限于篇幅，下面我们以 VMM 为例阐述 OSOSM 分层模型的设计思路.

VMM 提供的服务接口包括为用户进程分配与释放内存的函数 do_brk、映射区分配函数 do_mmap 和映射区释放函数 do_munmap. 存储空间以页面为单位，进程的虚拟地址空间由从 0 开始的连续页面组成. 内存管理的根本任务是将物理内存划分成一系列以页为元素的集合，并且它们互不相交. 每个进程都对应一组分配给它的物理页. 剩余的未分配的页称为空闲页. 空闲进程 (idle process) 对应的页面就是空闲页的集合. 为了完成虚拟内存管理，需要的 OS 对象包括：基本功效层中的页表对象 (PT)、页表是由形如 (虚拟地址页号，物理地址页号) 的 2 元组所组成的集合，还包括实现层中的虚拟地址块链表对象 vma_list 和空闲物理页对象 $free\_area$ 以及优化层中的用于指示空闲物理页的位图 (bitmap) 和快速查找虚拟地址块的红黑树结构 (red-black tree)，如图 2 所示：当 VMM 为一个进程设置好页表，该进程就可以直接通过处理器的 MMU 机制获得物理内存资源，这是内存分配的本质功效. 但是为了给一个进程设置页表，VMM 必须借助辅助的 OS 对象，如上述的数据对象 $free\_area$ 和 vma_list 等. 实现层利用 vma_list 来确定用户进程新分配的地址在虚拟地址空间的位置，也就是 2 元组 (虚拟地址页面号，物理地址页面号) 中的第 1 个元. 同时，实现层利用 $free\_area$ 来确定将哪些物理页面分配给用户进程，也就是 2 元组 (虚拟地址页面号，物理地址页面号) 中的第 2 个元.

VMM 必须高效地实现内存分配，例如尽可能将连续的物理页面分配给用户进程的一次内存分配请求，使得用户进程能够更加快速地访问内存. 为此，我们引入优化层. 优化层中的 buddy 系统借助 $free\_area$ 对象记录的所有未使用的连续页的信息和位图 bitmap 对象，将相邻的空闲页块合并成大的页块. 同时在优化层，利用红黑树结构快速定位用户进程的虚拟地址块 vma 对象.

图 2 说明了 VMM 功能的整体框架，我们从功能需求的角度阐述 OSOSM 分层模型.

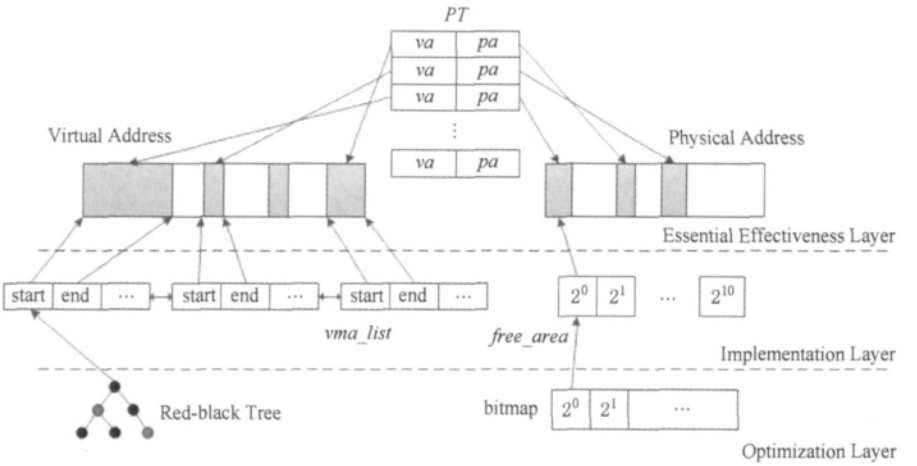

Fig. 2 Hierarchical object semantics model of VTOS virtual memory management.

图 2 VTOS 虚拟存储管理的分层对象语义模型

1）基本功效层. 该层的任务是当用户进程申请内存空间时，确定 OS 需要哪些最根本的数据对象，以及在这些对象上允许的操作集合，使得用户进程和 OS 都能理解该任务的完成效果. 在这一层并不涉及如何实现对这些对象操作的效果. 例如，在 VTOS 中 *do_mmap* 系统调用为用户进程在映射区分配页面，它在进程的页表中增加若干表项. *do_munmap* 为用户进程在映射区释放页面，它在进程的页表中减少若干表项.

2）实现层. 该层描述基本功效层中各种语义操作的具体实现. 为了实现基本功效层的语义函数引起的对象状态的转变，实现层加入了所需要的新的数据对象和相关的语义函数. 例如，在 VTOS 中为了实现系统调用 *do_brk*，*do_mmap* 和 *do_munmap* 所要求的数据对象值的转变，我们增加了 *free_area* 和 *vma_list* 对象. 以 *do_brk* 为例，如果 *do_brk* 的调用参数表明用户进程需要增加堆地址空间，则该系统调用的效果是在 *free_area* 中选出若干页，其大小总和是用户请求的空间大小，并在页表对象中增加这些页面映射的 2 元组，在 *vma_list* 中修改堆空间所在的 vma 区域的终止地址.

3）优化层. 为了尽可能地给用户进程分配连续的地址空间，VTOS 分配连续的物理页，称为页块，大小为 2 的幂，如 2^i. 在 VTOS 中，当页面 $k\times 2^i$，$k\times 2^i+1$，…，$(k+1)\times 2^i-1$ 页块都是空闲时，就将这 2^i 个页面组成 1 个页块，称为“2^i-页块”，其编号从 0 开始. 如果共有 $2n$ 个“2^i-页块”，当编号为 $2t$ 和 $2t+1$ 都是空闲时，组成 1 个“2^{i+1}-页块”，编号为 t. 对于物理内存有 $2n$ 个“2^i-页块”的情况，我们需要 n 个位记录这些页块的空闲状态，其中第 t 个位对应 $2t$ 号和 $2t+1$ 号“2^i-页块”，称为它们的对应位；第 t 个位为 1 时，两个页块中只有 1 个是空闲的；第 t 个位为 0 表示两个页块都已被分配或都未被分配. 当 1 个“2^i-页块”被释放时，如果对应位为 1，则与相邻的“2^i-页块”合并成 1 个“2^{i+1}-页块”，并列入

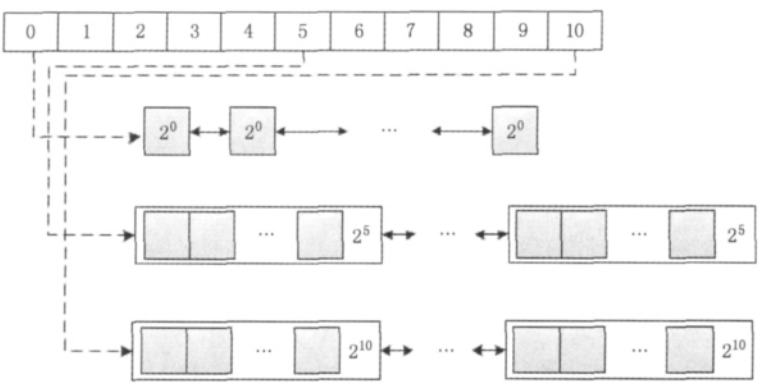

Fig. 3 Bitmap object structure.

图 3 Bitmap 对象结构

“2^{i+1}-页块”列表．如图3所示，VTOS将空闲页块按连续2^0，2^1，2^2，…，2^{10}个页面共11种不同的大小排列成11个列表，每次分配内存时，从满足需求的最小的页块列表中取出1块，分配之后若有剩余，则列入其他更小的列表中．同时，VTOS创建红黑树数据对象，以便快速检索当前处理的内存所在的虚拟地址块vma对象．

2 OSOSM论域和形式化模型

第1节对OSOSM采用自然语言的方式进行了说明，本节对OSOSM的形式化概念进行阐述．

由于OS的复杂性，我们采用高阶逻辑HOL作为OSOSM的元逻辑，高阶逻辑是在谓词逻辑的基础上进行了扩展：变量可以表示函数和谓词，任何变量都具有类型，谓词公式可以用布尔类型的项表示．同时由于在OSOSM的形式化描述中需要丰富的类型和抽象表示，为此我们引入类型论[14]以及类型化的λ演算．

定义1．OSOSM-HOL逻辑．OSOSM-HOL逻辑如下定义：

1）定义域（domain）D.

$$D::=B\mid\Omega\mid D\to D,$$

其中，B为基本域，如“OS对象的所有可能的取值集合”；Ω表示命题（propositions）域，在OSOSM中表示所有OS安全策略和属性的集合，命题如“两个进程空间互相隔离”等．

2）项（terms）的类型（types）定义．

按类型论的观点，“命题”、“集合”和“类型”是同构等价的[18]，定义域D的构造子B，Ω中的元素可以作为“类型”来对待．任意的$\sigma\in D$，我们对类型σ的项的集合$term_\sigma$归纳定义如下：

① 常元c_1^σ，c_2^σ，…是$term_\sigma$中的元素，符号c_1^σ表示c_1的类型为σ；

② 变量x_1^σ，x_2^σ，…是$term_\sigma$中的元素，符号x_1^σ表示x_1的类型为σ；

③ 如果φ：Ω，变量$x^\sigma\in term_\sigma$，那么$(\forall x^\sigma.\varphi)$：$\Omega$，符号$\varphi$：$\Omega$表示项$\varphi$的类型是$\Omega$；

④ 如果φ：Ω，ψ：Ω，那么$(\varphi\to\psi)$：Ω，符号$\varphi\to\psi$表示由φ推导出ψ；

⑤ 如果M：$\sigma\Rightarrow\tau$，N：σ，那么(MN)：τ，符号M：$\sigma\Rightarrow\tau$表示M为函数，其函数类型为$\sigma\Rightarrow\tau$．MN表示M作用于N得到的值；

⑥ 如果M：τ，变量$x^\sigma\in term_\sigma$，那么$(\lambda x^\sigma.M)$：$\sigma\Rightarrow\tau$.

利用上述定义，我们可以表达OSOSM中安全策略和属性的逻辑命题．

在OSOSM-HOL的基础上，我们来定义OSOSM的论域OSOSM-DD.

定义2．OSOSM-DD论域．OSOSM-DD是一个数学系统，主要由以下3个部分组成：

1）OSOSM-HOL中的基本域B，表示OS对象的所有可能的取值的集合；

2）一个B上的非空的函数集合，其中的每一个函数以1个B或者多个B的curry算子[17]为定义域并以B为值域，如B上的2元函数f可以表示为f：$B\Rightarrow(B\Rightarrow B)$；

3）一个关于B的非空命题集合，即OSOSM-HOL中的命题域Ω，每个命题表示B的元素之间、函数之间以及元素和函数之间的逻辑关系．

定义3．OSOSM-I解释．

OSOSM-HOL和OSOSM-DD之间需要有对应的映射关系，称为解释（interpretation），我们以OSOSM-I来表示．此映射将OSOSM-HOL中的每一个常元符号解释为OSOSM-DD中的一个元素，将每一个n元函数符号解释为OSOSM-DD中的一个n元函数，并将每一个n元谓词符号解释为OSOSM-DD中的一个n元关系．

定义4．OSOSM-S结构．

OSOSM-DD和OSOSM-I合称为结构（structure），我们以OSOSM-S表示，即OSOSM-S是一个2元组，我们可以表示为OSOSM-S＝（OSOSM-DD，OSOSM-I）．

定义5．赋值．

OS在某一时刻的状态是一个定义域由OS对象构成的变元集合V，值域为OSOSM-DD的映射，我们称为一个赋值（assignment），用γ表示，即γ：$V\Rightarrow$OSOSM-DD．γ把每个OS对象变元o赋以论域OSOSM-DD中的一个元素a，记为$\gamma(o)=a$.

定义6．OSOSM模型．

给定OSOSM-HOL逻辑，以及结构OSOSM-S和一个赋值γ，2元组（OSOSM-S，γ）称为一个模型，这就是我们定义的OS对象语义模型OSOSM，我们可以表示为OSOSM＝（OSOSM-S，γ）.

定义7．系统安全状态．

我们用OSOSM-HOL的逻辑公式来表达OS的安全策略和属性，如“进程A与进程B的当前内存地址空间不相交”、“当前进程的程序状态字的特权级位为0”等，并称这些逻辑公式为OSOSM

的安全策略和属性集，用 Γ 表示. 如果给定一个模型（OSOSM-S，γ），Γ 中的每一个公式关于该模型都为真，记为 OSOSM-S$\models_\gamma \Gamma$，表示 OS 在当前状态下满足安全策略和属性，系统处于安全状态.

定义 8. 保真性.

OS 的一个执行动作 f（如系统调用等）改变系统的状态，可以看成是将 OS 从一个模型（OSOSM-S，γ_1）映射到另一个模型（OSOSM-S，γ_2）的算子，记为 f：OSOSM $\Rightarrow$ OSOSM，f［（OSOSM-S，γ_1）］=（OSOSM-S，γ_2）. 对于执行动作 f，如果 OSOSM-S $\models_{\gamma 1}\Gamma$ 成立，那么 OSOSM-S$\models_{\gamma 2}\Gamma$，我们称执行动作 f 满足安全策略和属性集 Γ，记为 $f\Theta\Gamma$，也称为保真性.

定义 9. 系统安全性.

我们对 OS 的安全性定义：OS 的执行动作集合 $FUNCTION$，对于任意的动作 $f \in FUNCTION$，$f\Theta\Gamma$，则 OS 保持安全性.

我们通过证明 OS 的所有的行为 f 产生的模型集合上的算子在公式集 Γ 上都具有保真性来说明我们期望的 OS 在运行过程中始终能满足安全属性. 在第 3 节中，我们以 VMM 为例，定义 VMM 的安全属性为：“任意两个进程的地址空间不相交”，即进程隔离性. 利用 OSOSM 将 VMM 的操作 f 解释为 OS 对象从一个状态映射到另一个状态，即操作 f 的语义，并将它解释成对应的等效模型集合上的映射算子，然后我们证明它具有保真性.

3 OSOSM 语义分析和安全性验证

在第 2 节 OSOSM 论域和形式化模型的基础上，本节阐述 OSOSM 的语义分析，并使用 Isabelle/HOL 方式建立语义验证系统.

3.1 Isabelle 的符号表示

本节介绍下面将用到的符号系统. 我们直接使用 Isabelle 中的符号表示，这些符号表示和标准的数理逻辑表示基本是一致的. Isabelle 主要用于实现对计算机系统的形式化逻辑验证，是一种定理证明器（theorem prover）. 其中 Isabelle/HOL 是 Isabelle 中的高阶逻辑实现.

HOL 是一种类型化的逻辑，与函数式编程语言（如 ML，Haskell 等）中的类型系统类似. 类型变量可以用$'a$，$'b$ 等符号表示. 对于类型的项，可以用如 $x::'a$ 来表示. Isabelle 支持类型的构造子运算，例如 int list 表示由整形变量组成的列表，nat set 表示由自然数组成的集合.

在本文中我们使用 3 种方式进行新类型的构造：datatype，types 和 record. 我们使用 types 来表示类型的简化记号，如 types pid=nat，定义了新的类型 pid，它的基本类型为 nat，表示进程的标识符.

datatype 用于定义新的代数数据类型，例如，我们对于 VMM 的系统操作定义为

datatype $sysOPs = do\ \_brk\ addr\ len$
$\quad | do\ \_mmap\ addr\ len$
$\quad | do\ \_munmap\ addr\ len$,

其中，$addr$ 为操作所需的虚拟地址参数，len 为操作内存空间大小参数.

我们使用 record 来表示带名称的元组类型，例如，我们对于页表项信息定义为

record $pageitem =$
$\quad va::$int
$\quad pa::$int.

新类型 $pageitem$ 表示页表项信息，它含有 2 个成员：va 表示虚拟地址，pa 表示物理地址. 对于 record，引用成员信息可以表达为如 $va\ pageitem$，表示引用 $pageitem$ 中的 va 成员. 假设 $pageitem$ 拥有值（$|\ va=4,\ pa=5\ |$），更新操作可以表达为如 $pageitem$（$|\ pa := 6\ |$），表示将 $pageitem$ 中的 pa 成员修改成 6，但 va 成员保持不变.

对于函数，我们使用“$\Rightarrow$”表示函数定义域到值域的映射关系，这与定义 1 中函数类型的概念是一致的. 函数更新操作使用如 $g(x := y)$ 来表达. 函数在集合上的值域可以表达为如 $g'B \equiv \{y \mid \exists x \in B.\ y = g\,x\}$，表示函数 g 以集合 B 为定义域时的值域.

下面我们以 VTOS 的 VMM 模块为例，阐述 OSOSM 的语义分析以及使用 Isabelle/HOL 方式进行语义验证的过程.

3.2 VMM 基本功效层的语义

3.2.1 基本功效层论域

1）论域的元素集 M_1

① 常量的集合 $C = \{MAX\_PROCESS, MAX\_VPAGE, MAX\_PPAGE, PAGE\_SIZE\}$. $MAX\_PROCESS$ 表示进程数量的上限，MAX_VPAGE 表示最大虚拟页数，MAX_PPAGE 表示最大物理页数，$PAGE\_SIZE$ 表示页的大小.

② 进程 ID 集合 P 定义为

definition $P::$nat set where “$P \equiv \{p::\text{nat}.\ 1 \leqslant p \wedge p \leqslant MAX\_PROCESS\}$”.

对于 P 的幂集，我们用 2^P 来表示，是 P 的所有子集组成的集合.

③ 虚拟地址页的集合 $VM=\{a_1, \cdots, a_{MAX\_VPAGE}\}$. 进程的虚拟地址页的集合为 VM 的幂集，即 2^{VM}. 我们也用 VM 表示函数映射：$P \Rightarrow VM$，$VM(p)$ 表示进程 p 的已经分配的虚拟地址页的集合.

④ 物理内存页的集合 $PM=\{m_1, \cdots, m_{MAX\_PPAGE}\}$. 进程的物理地址页的集合为 PM 的幂集，即 2^{PM}. 我们也用 PM 表示函数映射：$P \Rightarrow PM$，$PM(p)$ 表示进程 p 的已经分配的物理地址页的集合.

如果系统当前已创建的进程的集合为 P_{curr}，则已分配的物理页集合：$PM_{curr}=\bigcup_{p\in P_{curr}} PM(p)$.

⑤ 所有页表的集合：$PT=\{pt \mid pt \in 2^{VM\times PM}, \forall (a_1, m_1), (a_2, m_2) \in pt. a_1 \neq a_2 \wedge m_1 \neq m_2\}$. 我们也用 PT 表示函数映射：$P \Rightarrow PT$，$PT(p)$ 表示进程 p 的页表.

⑥ 进程定义为：

record *process*=
  pid::nat
  pagetable:: "*pageitem* set".

在 Isabelle/HOL 描述中，我们将进程简化为包含进程标识符 *pid* 和页表 *pagetable* 的 record 类型，其中 *pagetable* 是页表项 *pageitem* 的集合，*pageitem* 是虚拟地址 *va* 和物理地址 *pa* 组成的 record.

⑦ 系统状态 *state* 定义为：

record *state*=
  heap:: "*pid* ⇒ *process*"
  cur_id::*pid*
  next_pid::*pid*,

其中，*heap* 存储系统所有的进程个体，通过进程标识符 *pid* 可以找到对应的进程 *process*. *cur_id* 表示当前处于运行态的进程个体. *next_pid* 表示下一个可以分配的进程标识符.

2) 论域的命题

对于基本功效层，VMM 的安全属性为进程的隔离性，即 $\forall p_i, p_j \in P. PT(p_i) \cap PT(p_j)=\varnothing$，表示进程的页表项交集为空.

3.2.2 基本功效层语义

下面阐述 VMM 基本功效层 3 个接口函数 *do_brk*，*do_mmap*，*do_munmap* 的语义.

1) *do_brk* 系统调用调整进程的堆空间的大小，它有 2 个参数 *addr* 和 *len*. 当 $len>0$ 时，它在进程堆空间的上边沿开始往上继续分配长度为 *len* 的动态地址空间，当 $len<0$ 时，在进程堆的上边沿开始往下回收长度为 $|len|$ 的动态地址空间. 当 $len>0$ 时，*do_brk* 可以理解为将进程 p 的原有页表 $PT(p)$ 映射为新的页表 $PT'(p)$ 的语义函数，可以用如下的逻辑公式来表达：

$$PT'(p)=PT(p) \cup \{\langle a_i, m_i\rangle \mid a_i=i+addr/PAGE\_SIZE;\ i=0, 1, \cdots, len-1;\ a_i \notin VM(p);\ m_i \notin PM_{curr}\},$$

其中物理地址项 m_i 是从空闲的物理页选择出来的，具体是哪个物理页以及如何选择这些物理页由实现层和优化层确定. 从上述 $PT'(p)$ 的逻辑公式可以看出，因为增加的所有的 2 元组 $\langle a_i, m_i\rangle$ 中，$m_i \notin PM_{curr}$，所以经过 *do_brk* 操作后进程的隔离性仍然满足，具体验证过程在 3.4 节中阐述.

我们用 Isabelle/HOL 对 *do_brk* 的操作语义定义如下：

definition *do_brkOperation* :: "*state* ⇒ *addr*⇒*len*⇒*state*" where
"*do_brkOperation s a l*≡
let *a_p_c*= {(| *va*=*a*/*PAGE_SIZE*+ (*i*::nat), *pa*=−1 |) | *i*. (0≤*i*) ∧ (*i*≤*l*−1)};
  p_t= (*pagetable* ((*heap s*) (*cur_id s*))) ∪ *a_p_c*;
  p= (| *pid*=*cur_id s*, *pagetable*=*p_t* |);
  h= (*heap s*)(*cur_id s* := *p*)
in *s* (| *heap* := *h* |)".

其中 *a_p_c* 为增加的页表项. 由于物理地址项由实现层和优化层决定，我们以 "−1" 表示物理地址项暂时为空，具体的值将由实现层和优化层的语义操作确定.

2) *do_mmap* 系统调用为进程创建或扩充一个映射区. 它的主要功能是将可执行文件的镜像映射到虚拟内存中，或将别的进程中的内存信息映射到该进程的虚拟空间中. 它的操作语义和 *do_brk* 系统调用类似.

3) *do_munmap* 系统调用用于释放进程空间，具体操作语义是指释放从 *addr* 开始，长度为 *len* 的一段虚拟地址空间，可以用如下的逻辑公式来表达：

$$PT'(p)=PT(p) \setminus \{\langle a_i, m_i\rangle \mid a_i=i+addr/PAGE\_SIZE;\ i=0, 1, \cdots, len-1\}.$$

我们用 Isabelle/HOL 对 *do_munmap* 的操作语义定义如下：

definition *do_munmapOperation*:: "*state*⇒ *addr*⇒*len*⇒*state*" where
"*do_munmapOperation s a l*≡

let *a__p__c*＝｛(｜*va*＝*a*/*PAGE__SIZE*＋(*i*::nat)，…＝()｜)｜*i*.(0≤*i*) ∧ (*i*≤*l*－1)｝；

p__t＝(*pagetable*((*heap s*) (*cur__id s*)))－

a__p__c；

p＝(｜*pid*＝*cur__id s*，*pagetable*＝*p__t*｜)；

h＝(*heap s*)(*cur__id s* := *p*)

in *s*(｜*heap* := *h*｜)”.

由于 *do__munmap* 从操作语义上是按照虚拟地址来进行释放操作，为此物理地址项在 *do__munmap* 的语义定义中可以不进行考虑，我们使用“…＝()”来表达.

对于 *do__munmap* 操作，所有进程的页表没有增加，所以进程的隔离性仍然满足.

执行 *do__brk*，*do__mmap*，*do__munmap* 操作之后系统状态的变化，我们定义如下：

fun *step*::“*sysOPs*⇒*state*⇒*state*” where

“*step*(*do__brk a l*) *s*＝*do__brkOperation s a l*”｜

“*step*(*do__mmap a l*) *s*＝*do__mmapOperation s a l*”｜

“*step*(*do__munmap a l*) *s*＝*do__munmapOperation s a l*”.

全函数 *step* 定义了执行一步操作后系统状态的转变，根据系统的当前状态和系统操作决定系统的后续状态.

下面将对 OSOSM 中 VMM 的实现层和优化层语义进行阐述. 由于优化层中红黑树（red-black tree）和位图（bitmap）的引入是为了提高实现层的效率，考虑到方便对实现层和优化层的形式化描述，我们下面将 VMM 的实现层和优化层的语义结合起来进行描述.

3.3 VMM 实现层和优化层的语义

3.3.1 实现层和优化层的论域

1）论域的元素集 M_2

① M_2 包含 M_1 中的元素.

② “2^i-页块”的集合 $free\_area = \bigcup_{i=0}^{MAX\_ORDER} free\_area(i)$. 其中 $free\_area(i) = \{ms \mid ms = \{k\times 2^i, k\times 2^i+1, \cdots, (k+1)\times 2^i-1\} \subset PM\}$. MAX_ORDER 为“2^i-页块”的最大幂次，即最大的空闲区块的上限.

③ 虚拟地址块链表对象 $vma\_list = \{[s_i, t_i] \mid s_i<t_i, i=1, 2, \cdots, k; \forall i, j\in[1, \cdots, k]. [s_i, t_i]\cap[s_j, t_j]=\varnothing\}$. vma_list 是进程互不相交的虚存段 vma 的集合，s_i 和 t_i 分别表示 vma 的起始和结束位置. 每个进程通常占用几个 vma 段，分别用于代码区、静态数据区、堆区、栈区和若干文件映射区.

2）论域的命题

对于实现层和优化层，VMM 的安全属性仍为进程的隔离性，即 $\forall p_i, p_j\in P. PT(p_i)\cap PT(p_j)=\varnothing$.

3.3.2 实现层和优化层语义

下面以物理内存管理接口函数 *alloc__pages* 和 *do__brk* 系统调用为例来阐述实现层和优化层的语义.

alloc__pages(*order*) 系统调用提供物理页面的分配功能. VTOS 在分配时只分配 2 的幂次个数的页，参数 *order* 指明分配 2^{order} 个页. *alloc__pages* 一般被缺页异常（pagefault）处理程序、*do__brk* 和 *do__mmap* 调用. *alloc__pages* 的主要操作是从“2^{order}-页块”连续空闲物理页块集合 *free__area*(*order*) 中选择 1 块 2^{order} 个页的空间分配给请求的进程. 如果没有 2^{order} 大小的页块，则从更大的空闲连续物理页块中分配，分配的剩下部分加入到 *free__area* 中更小的空闲连续物理页块的集合中. *alloc__pages* 的语义可以通过它对 OS 对象的操作来描述. *alloc__pages* 对 *free__area* 对象进行操作，将 *free__area* 映射到 *free__area*′，可以用如下逻辑公式描述：

$$free\_area' = \begin{cases} free\_area(i)\cup\{k\times 2^n+2^i \mid i = order, \cdots, n-1\}, \mu_{free\_area}(order)=n; \\ free\_area, \mu_{free\_area}(order) = MAX\_ORDER+1; \end{cases}$$

其中辅助语义函数 $\mu_{free\_area}(order)$ 定义如下：

$$\mu_{free\_area}(order) = \begin{cases} n, free\_area[i]=\varnothing \wedge free\_area[n]\neq\varnothing \wedge order\leqslant i\leqslant n-1 \wedge n\leqslant MAX\_ORDER; \\ n_1, free\_area[i]=\varnothing, order\leqslant i\leqslant n_1. \end{cases}$$

do__brk 系统调用为进程在堆中分配内存或从堆中回收内存，它的核心功效是在 3.2 节中阐述的对进程的页表增加或减少若干表项. 虽然核心功效反映在进程的页表上，但是要确定页表中映射的物理地址还需要借助 *free__area* 对象. 从实现层和优化层来看，*do__brk* 分别对 *vma__list*，*free__area* 和 $PT(p)$ 进行了操作，分别映射到 *vma__list*′，*free__area*′和 $PT'(p)$. 其语义可以使用以下的逻辑公式表示：

$$order = \mathrm{lb}\ len;$$

$$vma\_list' = vma\_list \cup \{[addr, addr+2^{order}]\};$$

$$free\_area' =$$

$$\begin{cases} free\_area(i) \cup \{k \times 2^n + 2^i \mid i = order, \cdots, n-1\},\ \mu_{free\_area}(order) = n; \\ free\_area,\ \mu_{free\_area}(order) = MAX\_ORDER + 1; \end{cases}$$

$$PT'(p) = PT(p) \cup \{\langle addr/PAGE\_SIZE + i,\ k \times 2^n + i\rangle \mid i = 0,1,\cdots,2^{order} - 1\}.$$

我们用 Isabelle/HOL 对 *do _brk* 的操作语义具体细化如下：

```
definition do _brkOperation :: "state ⇒ addr⇒len⇒state" where
"do _brkOperation s a l≡
let a _p _c= {( | va=a/PAGE _SIZE+ (i ::nat), pa=k * (2^n) +i | ) | i. (0≤ i) ∧ (i≤
2^(log 2 l) −1)};
 p _t= (pagetable ( (heap s) (cur _id s))) ∪
    a _p _c;
 p= ( | pid=cur _id s, pagetable=p _t | );
 h= (heap s) (cur _id s := p)
in s ( | heap := h | )".
```

从上述可以看出，实现层解决了需求层没有解决的细节问题，确定了用户进程的虚拟区域的具体位置，也确定了具体的物理页面. 为了优化运行的性能，可以通过红黑树对象和位图对象提高用户进程虚拟地址空间和空闲物理页面的管理性能.

3.4 系统安全性验证

本节主要阐述系统安全性的验证. 我们使用“$\bowtie s$”表示在状态 s 下，进程个体符合隔离性安全属性，即系统处于定义 7 的安全状态. “$\bowtie s$”的隔离性定义如下：

```
definition isolation:: "state⇒bool" ( " ⋈ " )
where " ⋈ s≡∀ p::pid. ∀ q::pid. p◇s ∧ q◇ s ∧ p ≠ q → (pagetable (heap s p)) ∩ (pagetable (heap s q)) = {}",
```

其中“$p◇s$”表示进程标识符 p 在状态 s 下存在，其定义如下：

```
definition isProcessOf :: "pid⇒state⇒bool" (infixl "◇" 100)
where "p◇s≡p<next _pid s".
```

引理 1. *do _brk* 隔离性. *do _brk* 操作不改变系统的进程隔离性，即如果执行 *do _brk* 操作之前系统状态符合进程隔离性，那么执行 *do _brk* 操作后系统状态仍然满足隔离性：

```
lemma do _brk _iso: " ⋈ s→ ⋈ (step (do _brk a l) s)".
```

证明. 由 3.2 节我们可知，*do _brk* 操作增加的页表项 $\langle a_i, m_i\rangle$ 中，物理地址 $m_i \notin PM_{curr}$，因此增加的物理内存块不会和系统已分配的任何内存块重叠，所以在执行 *do _brk* 操作之前系统状态符合进程隔离性的情况下，那么执行 *do _brk* 操作后系统状态仍然满足隔离性. 我们可以对 *do _brk* 操作的定义和隔离性定义展开进行验证：

```
apply (simp add: isolation _def
  do _brkOperation _def)
apply (blast)
done
```

simp 表示采用经典化简器 (simplifier) 进行前向推导，并借助隔离性定义 (isolation _def) 和 *do _brk* 的操作语义 (do _brkOperation _def) 以及经典推理器 (blast) 进行验证[15]. 证毕.

对于 *do _mmap*，*do _munmap* 操作，我们有相应的引理，证明过程类似.

引理 2. *do _mmap* 隔离性.

```
lemma do _mmap _iso: " ⋈ s→ ⋈ (step (do _mmap a l) s)"
apply (simp add: isolation _def
  do _mmapOperation _def)
apply (blast)
done
```

引理 3. *do _munmap* 隔离性.

```
lemma do _munmap _iso: " ⋈ s→ ⋈ (step (do _munmap a l) s)"
apply (simp add: isolation _def
  do _munmapOperation _def)
apply (blast)
done
```

总的来说，*do _brk*，*do _mmap*，*do _munmap* 操作符合定义 8 的保真性，即 *do _brk*Θ*isolation*，*do _mmap*Θ*isolation*，*do _munmap*Θ*isolation*.

下面我们证明 OSOSM 的系统安全性，按照定义 9，即需证明对于 *do _brk*，*do _mmap*，*do _munmap* 操作，系统始终保持进程隔离性.

定理 1. 系统安全性.

```
theorem VM _Isolation: " ⋈ s→ ⋈ (step cmd s)"
```

证明. 由引理 1、引理 2 和引理 3 可知，系统对于 *do _brk*，*do _mmap*，*do _munmap* 操作符合保真性，因此我们对 *cmd* 分情况讨论，并借助引理 1，引理 2 和引理 3 得到系统的安全性，Isabelle 验证过程如下：

```
apply (simp add: isolation_def)
apply auto
apply (case_tac cmd)
apply (rule do_brk_iso, do_mmap_iso,
  do_munmap_iso)
apply (blast)
done
```

其中 case_tac *cmd* 表示对操作 *cmd* 进行分类展开验证，rule 表示采用已有的规则或者结论进行验证. 证毕.

我们的验证环境配置为 Studio XPS 9100 台式电脑，2.8 GHz Intel i7 930 处理器，3 GB 内存，操作系统采用 openSUSE Desktop 11.3 版本，Isabelle 采用 Isabelle2009-2_bundle_x86-linux 版本. VTOS 的整个 Isabelle 验证工程代码量大概在 59 K SLOC 左右，完整的验证耗时 85 min 左右.

Isabelle 的验证结果如图 4 所示. “No subgoals” 说明 Isabelle 验证逻辑完整，不存在任何未证明的子目标.

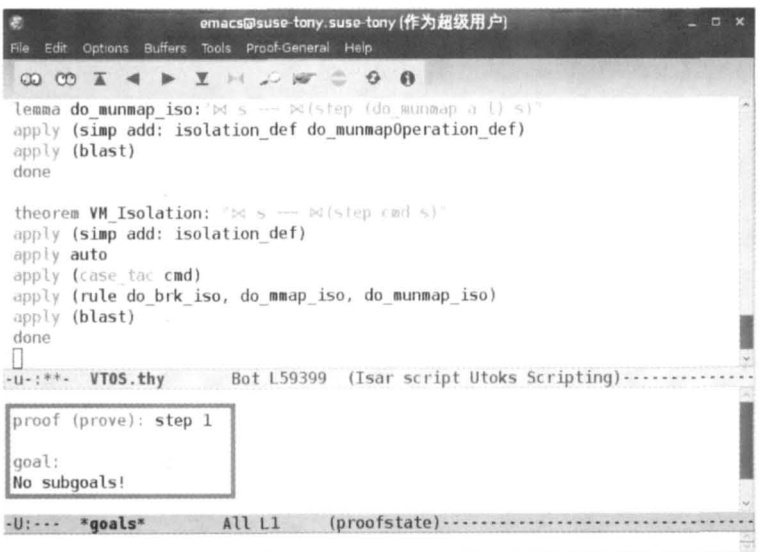

Fig. 4 Isabelle verification.

图 4 Isabelle 验证结果图

4 结论和进一步研究

本文以高阶逻辑为元逻辑，提出 OS 对象语义模型 OSOSM. OSOSM 通过对 OS 系统调用动作的语义进行描述，实现对安全 OS 的形式化设计. 我们描述了 OS 的逻辑系统，建立 OS 的论域模型，使用逻辑系统中的谓词公式表达 OS 的安全属性，并阐述了通过验证所有由 OS 系统调用引起的 OS 状态转变都保持系统的安全属性来说明系统安全性的方法. 同时，本文以我们实现的可信操作系统 VTOS 为例，阐述 OSOSM 的语义正确性，并使用 Isabelle 定理证明器实现对 OSOSM 设计和安全需求的一致性验证. 从验证过程可以看出 VTOS 系统调用始终满足系统保真性的要求.

我们接下来的工作将对 OS 系统动作的并发性问题和多核多线程环境进行形式化的描述和分析，以求实现对整个 OS 的安全性的形式化验证.

致谢 本文作者感谢《计算机研究与发展》所有本文的匿名审稿人，感谢他们对本文提出的宝贵意见！

参 考 文 献

[1] Abadi M, Budiu M, Erlingsson U, et al. Control-flow integrity principles, implementations, and applications [J]. ACM Trans on Information and System Security, 2009, 13(1): 1-40

[2] Chen Ping, Xiao Hai, Shen Xiaobin, et al. DROP: Detecting return-oriented programming malicious code [C] //Proc of the 5th Int Conf on Information Systems Security (ICISS'09). Berlin: Springer, 2009: 163-177

[3] Chen Ping, Xiao Xing, Hao Han, et al. Efficient detection of the return-oriented programming malicious code [C] //Proc of the 6th Int Conf on Information Systems Security (ICISS'10). Berlin: Springer, 2010: 140-155

[4] Bevier W R. A verified operating system lernel [D]. Austin: University of Texas at Austin, 1987

[5] Klein G, Elphinstone K, Heiser G, et al. seL4: Formal verification of an OS kernel [C] //Proc of the 22nd ACM Symp on Operating Systems Principles (SOSP'09). New York: ACM, 2009: 207-220

[6] Gargano M, Hillebrand M, Leinenbach D, et al. On the correctness of operating system kernels [G] //LNCS 3603: Proc of the 18th Int Conf on Theorem Proving in Higher Order Logics (TPHOLs'05). Berlin: Springer, 2005: 1-16

[7] Stampoulis A, Shao Z. VeriML: Typed computation of logical terms inside a language with effects [C] //Proc of the 15th ACM SIGPLAN Int Conf on Functional Programming (ICFP'10). New York: ACM, 2010: 333-344

[8] Shapiro J, Doerrie M S, Northup E, et al. Towards a verified, general-purpose operating system kernel [C] //Proc of the 1st NICTA Workshop on Operating System Verification. Sydney: NICTA, 2004: 1-19

[9] Shapiro J S, Smith J M, Farber D J. EROS: A fast capability system [C] //Proc of 17th ACM Symp on Operating Systems Principles (SOSP'99). New York: ACM, 1999: 170-185

[10] Tews H. Micro hypervisor verification: Possible approaches and relevant properties [C] //Proc of the NLUUG Spring Conf 2007. Holland: NLUUG, 2007: 1-14

[11] Shao Zhong, Ford B. Advanced development of certified OS kernels, YALEU/DCS/TR-1436 [R]. New Haven: Yale University, 2010

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

[13] Benthem J V, Doets K. Higher-order Logic [G] //Handbook of Philosophical Logic: Volume 1. Dordrecht: Kluwer Academic Publishers, 2001: 189-243

[14] Nanevski A, Pfenning F, Pientka B. Contextual modal type theory [J]. ACM Trans on Computational Logic, 2008, 9 (3): 23: 1-49

[15] Nipkow T, Paulson L C. Isabelle/HOL: A Proof Assistant for Higher-Order Logic [M]. Berlin: Springer, 2002

[16] Tanenbaum A S, Woodhull A S. Operating Systems Design and Implementation [M]. 3rd ed. Englewood Chiffs, NJ: Prentice Hall, 2006

[17] Zhou Chaochen. Introduction to Formal Semantics [M]. Changsha: Hunan Science and Technology Press, 1985 (in Chinese)
(周巢尘. 形式语义学引论[M]. 长沙: 湖南科学技术出版社, 1985)

[18] Howard W A. The formulae-as-types notion of constructions [G] //To H B. Curry: Essays on Computational Logic, Lambda Calculus and Formalism. Waltham: Academic Press, 1980: 479-490

Qian Zhenjiang, born in 1982. PhD candidate and lecturer. Member of China Computer Federation. His current research interests focus on operating system security, formal verification and embedded system.

Liu Wei, born in 1986. Postgraduate. His current research interests focus on virtual machine surveillance and formal verification.

Huang Hao, born in 1957. PhD, professor, and PhD supervisor. His current research interests focus on system software and information security.