软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2023,34(8):3507−3526 [doi: 10.13328/j.cnki.jos.006866]

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

基于精化的 TrustZone 多安全分区建模与形式化验证*

曾凡浪[1,2], 常 瑞[1,3], 许 浩[1,2], 潘少平[1,2], 赵永望[1,3]

[1](浙江大学 计算机科学与技术学院, 浙江 杭州 310027)

[2](浙江大学 杭州国际科创中心, 浙江 杭州 311200)

[3](浙江省区块链与网络空间治理重点实验室, 浙江 杭州 310027)

通信作者: 常瑞, E-mail: crix1021@zju.edu.cn

摘 要: TrustZone 作为 ARM 处理器上的可信执行环境技术, 为设备上安全敏感的程序和数据提供一个隔离的独立执行环境. 然而, 可信操作系统与所有可信应用运行在同一个可信环境中, 任意组件上的漏洞被利用都会波及系统中的其他组件. 虽然 ARM 提出了 S-EL2 虚拟化技术, 支持在安全世界建立多个隔离分区来缓解这个问题, 但实际分区管理器中仍可能存在分区间信息泄漏等安全威胁. 当前的分区管理器设计及实现缺乏严格的数学证明来保证隔离分区的安全性. 详细研究了 ARM TrustZone 多隔离分区架构, 提出一种基于精化的 TrustZone 多安全分区建模与安全性分析方法, 并基于定理证明器 Isabelle/HOL 完成了分区管理器的建模和形式化验证. 首先, 基于逐层精化的方法构建了多安全分区模型 RMTEE, 使用抽象状态机描述系统运行过程和安全策略要求, 建立多安全分区的抽象模型并实例化实现分区管理器的具体模型, 遵循 FF-A 规范在具体模型中实现了事件规约; 其次, 针对现有分区管理器设计无法满足信息流安全性验证的不足, 设计了基于 DAC 的分区间通信访问控制, 并将其应用到 TrustZone 安全分区管理器的建模与验证中; 再次, 证明了具体模型对抽象模型精化的正确性以及具体模型中事件规约的正确性和安全性, 从而完成模型所述 137 个定义和 201 个定理的形式化证明(超过 11 000 行 Isabelle/HOL 代码). 结果表明: 该模型满足机密性和完整性, 并可有效防御分区的恶意攻击.

关键词: 可信执行环境; 安全分区; 定理证明; 精化; 安全性分析

中图法分类号: TP311

中文引用格式: 曾凡浪, 常瑞, 许浩, 潘少平, 赵永望. 基于精化的 TrustZone 多安全分区建模与形式化验证. 软件学报, 2023, 34(8): 3507–3526. http://www.jos.org.cn/1000-9825/6866.htm

英文引用格式: Zeng FL, Chang R, Xu H, Pan SP, Zhao YW. Refinement-based Modeling and Formal Verification for Multiple Secure Partitions of TrustZone. Ruan Jian Xue Bao/Journal of Software, 2023, 34(8): 3507−3526 (in Chinese). http://www.jos.org.cn/1000-9825/6866.htm

Refinement-based Modeling and Formal Verification for Multiple Secure Partitions of TrustZone

ZENG Fan-Lang[1,2], CHANG Rui[1,3], XU Hao[1,2], PAN Shao-Ping[1,2], ZHAO Yong-Wang[1,3]

[1](College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China)

[2](ZJU-Hangzhou Global Scientific and Technological Innovation Center, Hangzhou 311200, China)

[3](Key Laboratory of Blockchain and Cyberspace Governance of Zhejiang Province, Zhejiang University, Hangzhou 310027, China)

Abstract: As a trusted execution environment technology on ARM processor, TrustZone provides an isolated and independent execution environment for security sensitive programs and data on the device. However, making trusted OS and all trusted applications run in the

* 基金项目: 浙江省重点研发计划(2022C01165); 国家自然科学基金(62132014); 浙江省尖兵计划(2022C01045); 中央高校基本科研业务费(NGICS)专项资金

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: 2022-09-04; 修改时间: 2022-10-13; 采用时间: 2022-12-14; jos 在线出版时间: 2022-12-30

same trusted environment may cause problems—the exploitation of vulnerabilities on any component will affect the others in the system. Although ARM proposed S-EL2 virtualization technology, which supports multiple isolated partitions in the security world to alleviate this problem, there may still be security threats such as information leakage between partitions in the actual partition manager. Current secure partition manager designs and implementations lack rigorous mathematical proofs to guarantee the security of isolated partitions. This study analyzes the multiple secure partitions architecture of ARM TrustZone in detail, proposes a refinement-based modeling and security analysis method for multiple secure partitions of TrustZone, and completes the modeling and formal verification of the secure partition manager based on the theorem prover Isabelle/HOL. First, a multiple security partitions model named RMTEE is built based on refinement, an abstract state machine is used to describe the system running process and security policy requirements, and the abstract model of multiple secure partitions is established and instantiated. Then the concrete model of the secure partition manager is implemented, in which the event specification is implemented following the FF-A specification. Secondly, in view of the problem that the existing partition manager design cannot meet the goal of information flow security verification, a DAC-based inter-partition communication access control is designed and applied to the modeling and verification of TrustZone security partition manager. Finally, the correctness of the concrete model's refinement of the abstract model and the correctness and security of the event specification in the concrete model are proved, thus completing the formal proof of 137 definitions and 201 lemmas in the model (more than 11 000 lines of Isabelle/HOL code). The results show that the model satisfies confidentiality and integrity, and can effectively defend against malicious attacks of partitions.

Key words: trusted execution environment; security partition; theorem proving; refinement; security analysis

随着计算设备广泛应用和种类的多样化, 计算系统变得愈发复杂, 给病毒软件和恶意攻击提供了更大的利用空间. 软件提供商和用户需要可靠的防御机制来保护自己的代码和数据的安全. 基于 ARM TrustZone[1]实现的可信执行环境[2]利用硬件隔离机制为代码和数据提供有效防护. 然而, 在实际应用中, 基于 TrustZone 的隔离机制也暴露了一些不足. Pinto 等人[3,4]分析了主流的商用 TrustZone 系统中存在的安全问题, 在安全监视器、可信操作系统和可信应用中发现了数量可观的漏洞和错误. 最新的 CVE 漏洞也持续披露了 TrustZone 中存在的问题, 比如: CVE-2021-34389[5]发现, 英伟达 TEE 的 TA 中不正确的边界检查导致本地用户可以通过恶意客户端访问 TrustZone 中的堆内存; CVE-2021-44149[6]发现, OPTEE-OS 的 CSU 驱动中的漏洞导致普通世界可以绕过 TrustZone 任意读写安全世界内存.

ARM 为了缓解上述问题, 在 v8.4 新架构中提出了 S-EL2 虚拟化架构[7]及 FF-A (firmware framework for arm a-profile)[8]分区管理器 ABI 接口规范, 将安全世界划分为多个隔离分区(secure partition, SP), 不同的 Trusted OS 和 TA 运行在不同的分区中, 并由更高特权级别的分区管理器(secure partition manager, SPM)来管理, 系统 TCB 为分区管理器和安全监视器. 分区管理器还提供分区间的消息通信接口, 以支持分区间的功能调用和数据传输.

然而, 该架构设计缺乏严格的数学证明保证其设计与实现的一致性, 仍存在非法内存访问和恶意接口调用等安全威胁, 具体表现为:

(1) 分区管理器是保证分区安全隔离的关键组件, 如果其实现不当, 有可能会被恶意程序利用, 从而导致系统级的安全风险. 比如: CVE-2021-30278[9]发现, 高通 TrustZone 中内存转让接口的输入验证不当, 可能导致多款系统程序的信息泄露;

(2) 在 ARM 的多分区系统设计中, 分区间可以任意通信, 意味着信息在分区间是任意流动的, 分区隔离也就失去了意义, 无法满足信息流安全形式化验证的要求.

为了解决上述问题, 本文提出一种基于精化的 TrustZone 多安全分区建模和验证方法, 构建了 TrustZone 多安全分区模型 RMTEE (refinement-based multiple secure partitions trusted execution environment), 并在 Isabelle/HOL 定理证明器中, 以机器可检查的方式证明了其功能正确性和信息流安全性. 本文借鉴程序开发中的精化方法[10], 在形式化模型中使用精化演算[11], 从抽象的高层描述出发, 对系统功能和数据逐步增加细节, 每一步精化都得到一个更贴近底层实现的规约, 逐步得到底层规约. 同时, 为实现安全的分区间通信, 本文提出了基于自主访问控制(discretionary access control, DAC)的分区间通信访问控制, 在系统配置中定义分区间通信访问控制矩阵(access control matrix, ACM), 并在接口调用时查询访问控制矩阵, 验证调用的合法性,

确保系统满足信息流安全要求.

本文的主要贡献如下:

(1) 提出一种基于精化的 TrustZone 多安全分区形式化建模方法. 本文构建了 TrustZone 多安全分区模型 RMTEE, 该模型包含抽象模型和具体模型: 抽象模型由抽象参数状态机和安全属性组成, 具体模型包括执行模型和事件规约. 执行模型使用具体变量和函数实例化了抽象模型, 事件规约依据 ARM FF-A 规范所描述的分区管理器标准实现了具体事件接口. RMTEE 模型包含 137 个定义, 1 462 LOC 的 Isabelle/HOL 代码;

(2) 设计了分区间调用安全增强机制. 通过形式化验证, 发现了 FF-A 规范中存在的信息流安全隐患, 针对该不足设计了分区间调用自主访问控制安全增强机制, 应用到模型开发和验证中, 并分析了该机制可以抵御的两种攻击类型;

(3) 验证了 RMTEE 模型的正确性和安全性. 使用定理证明方法验证 RMTEE 抽象模型与具体模型间的精化关系及具体模型中事件接口的正确性和信息流安全性. 证明工作使用了 201 个定理, 9 715 LOC 的 Isar 代码. 结果表明, RMTEE 模型符合机密性和完整性.

本文第 1 节介绍与本工作研究内容相关的工作. 第 2 节分析威胁模型及安全假设. 第 3 节提出本工作的形式化建模和验证方法. 第 4–6 节描述系统建模和安全属性验证过程, 分别阐述抽象模型、具体模型规约和具体模型证明. 第 7 节对本工作进行安全性分析和评估. 第 8 节总结全文.

1 研究现状

1.1 TrustZone多安全区隔离

学术界和工业界已经有一些工作利用 TrustZone 的隔离特性建立安全分区. 在有些工作中, 分区也称为虚拟机(VM). TZ-RKP[12]将 Hypervisor 运行在安全世界来提供高特权级别和隔离, 它在运行时拦截并检查来自普通世界的系统调用, 防止目标系统运行未授权的特权代码. TEEv[13]设计了一种在安全世界内运行多个 VM 的方法, 由于安全世界没有专门用于运行 Hypervisor 的特权级别, 它将专门实现的 Hypervisor 与 VM 运行在相同的特权级别 S-EL1. 为了保证 Hypervisor 的高特权以管理 VM, 需要修改 VM 内核, 限制其对某些特权指令的访问. 此前的工作由于没有硬件支持, 都需要通过扫描二进制文件或软件拦截的方式来检查或修改 VM 指令, 带来实现和运行的开销. ReZone[14]利用现有 ARM 平台上的辅助硬件设计了一种安全架构, 将安全世界划分为隔离分区, 并且由硬件保障的内存访问控制限制分区对其他分区以及普通世界的访问权限, 以此来削减 S-EL1 的非必要特权, 进而缓解 S-EL1 被劫持所引发的系统级威胁. ARMv8.4 架构引入了 S-EL2 硬件隔离机制[7], 在安全世界隔离出多个分区, 使得单个分区出现安全漏洞被利用后产生的影响范围更小. TwinVisor[15]利用 S-EL2 扩展实现了安全世界内的隔离分区, 但它的设计与 ARMv8.4 标准有所不同, 它在安全世界内运行一个实现简单的 Hypervisor 来保证 VM 的安全性, 而复用普通世界的 Hypervisor 来管理硬件资源. TwinVisor 设计了一些方案来支持安全世界 Hypervisor 和普通世界 Hypervisor 协同管理 VM 资源以及减少系统开销, 但是它没有像 ARM 标准设计那样提供 VM 之间的通信机制. 表 1 比较了不同 TrustZone 多分区隔离机制之间的区别.

表 1 TrustZone 多安全区机制比较

多分区隔离机制	硬件支持	无系统修改	分区间通信	性能开销
TZ-RKP	×	√	×	运行时拦截并检查分区指令, 开销高
TEEv	×	×	√	纯软件方案, 开销高
ReZone	√	√	×	安全分区运行时, 处理器其他核心空转, 开销高
TwinVisor	√	√	×	需要两个世界的 Hypervisor 协作, 开销中等
ARM S-EL2	√	√	√	硬件支持的虚拟化, 开销低

ARMv9 提出了机密计算架构(confidential compute architecture, CCA)[16−18], 增强了 ARM 对机密计算的支持, 以应对大型计算密集型任务. CCA 基于硬件 RME (realm management extension)扩展和软件 RMM (realm

management monitor). CCA 在 TrustZone 的安全基础之上构建了 realm 的概念, realm 是由 RMM 管理的可以动态创建的安全分区. 此外, RME 支持向安全世界动态分配和回收物理内存. 由于 ARM 尚未公布 CCA 的细节规范, 其实际落地商用还需要一段时间.

1.2 分区隔离系统的形式化验证

在分区隔离系统的形式化验证领域存在着数量可观的成果. seL4 项目[19]针对一种高性能操作系统微内核展开了全面的形式化验证. seL4 是世界上第一个经过形式化验证的操作系统内核, 而且仍然是唯一经过验证的具有细粒度的基于能力的高安全性和高性能操作系统. 该项工作成功地使用逐层精化的建模方法, 为后来的大量形式化验证工作提供了借鉴意义, 也是本工作所用方法的来源之一. 耶鲁大学邵中团队主持的 CertiKOS 项目[20]验证了一个运行在多核处理器上的操作系统. 传统操作系统内核中, 各个部分互相联结, 一个漏洞就会影响整个操作系统的安全, 难以设计和验证. 该团队创新性地提出以抽象层为基础的组合规约, 在提高可扩展性的同时, 解决了并发内核难以验证的难题. Zhao 等人[21,22]在 Isabelle/HOL 中实现了一个符合 ARINC 653 隔离内核的顶层规范, 对 ARINC 653 的分区管理、分区调度和通信服务形式化建模, 并验证了 ARINC 653 隔离内核的一个工业实现和两个开源实现, 发现了 ARINC 653 标准及其实现中存在的 6 个可能导致信息泄露的安全缺陷. 本文的建模和验证主要参考了此项工作的框架. SeKVM[23,24]首次对商用 Hypervisor KVM 进行了形式化验证, 它将 KVM 拆分为一个核心部分和一组不可信服务, 使用 Coq 通过逐层求精的方法证明了 KVM 核心的机密性和完整性, 并使用 Rely-Guarantee 框架证明了其并发安全性, 此方法为我们后续工作对 TrustZone 多分区系统并发安全的验证提供了一个参考方案. 大多数分区内核的形式化验证的研究工作的目标都是证明数据隔离和信息流安全, 采用定理证明和精化的方法.

1.3 TEE的形式化验证

目前, 针对可信执行环境安全隔离证明的工作较少. Jin 等人[25,26]针对 TrustZone 提出了一种基于精化的内存隔离机制安全性验证方法, 用于对内存隔离机制进行细粒度的形式化验证. 该模型建模了 TrustZone 内存隔离机制的关键硬件和软件, 包括地址空间控制器、MMU、TLB 和安全监视器等, 在定理证明器 Isabelle/HOL 中验证了该模型满足无干扰、无泄露、无影响等信息流安全属性. 该项工作主要聚焦在验证传统 TrustZone 架构下, 普通世界与安全世界之间的隔离; 而本文工作验证了 ARMv8.4 及其之后的新架构中, 安全世界内多分区间的隔离和安全通信. Miao 等人[27]对利用内存标签技术在可信执行环境中实现的细粒度的内存隔离和访问控制机制提出了通用的形式化模型框架, 并提出了一种基于模型检测的访问控制安全性分析方法, 利用形式化语言 B, 设计并实现了该框架的抽象机模型, 并验证了可信执行环境中访问控制机制的正确性和安全性. 此工作所针对的 RISC-V 平台的 TIMBER-V 所采用的内存标签隔离技术与本文所验证的 ARM S-EL2 隔离机制存在较大差异, 其使用的模型检查方法也与本文的定理证明方法不同. 在本文之前, 还没有针对 TrustZone 安全世界内多隔离分区的形式化建模和验证工作.

2 威胁模型与假设

本文针对安全世界内的分区隔离, 不考虑普通世界与安全世界间的隔离. 由于安全世界内的 TA 和 OS 中存在为数不少的漏洞, 它们有可能被恶意用户利用, 从而成为恶意程序. 因此, 不信任安全世界内的分区, 如图 1 所示. 分区对自身的漏洞负责, 形式化验证的分区管理器确保对分区所提供接口中不存在可被利用的漏洞, 保证分区的私有数据不会被其他恶意分区窃取或篡改. 本文不考虑侧信道攻击和复杂的物理攻击.

此外, 本工作对 TrustZone 多分区系统的验证基于以下假设.

(1) 硬件和安全监视器是可信的;

(2) 保存在持久化存储中的分区和分区管理器的镜像未被篡改(通过安全启动验证其完整性).

攻击者通过读写任意内存地址以及发起恶意的分区间调用, 达到窃取、篡改其他分区的敏感数据的目的. 本文对系统威胁模型的形式化定义如下.

定义 1(威胁模型). $T=\langle S,E\rangle$, 其中,

- S 是系统状态集合;
- E 是敌手事件集合, $E=\{mem\_read,mem\_write,ipc\_call\}$, mem_read 表示内存读操作, mem_write 表示内存写操作, ipc_call 表示分区间调用, 且 E 中包含的事件均不符合系统安全约束, 即:

$$\forall e\in E,\ (e\in\{mem\_read(s,mem,p),mem\_write(s,mem,p)\}\wedge mem.owner\neq p)\vee$$
$$(e\in\{ipc\_call(s,p_1,p_2)\}\wedge(p_1,p_2,ipc\_call)\notin ipc\_acm).$$

s 表示系统状态, mem 表示一块内存, p 表示一个分区, $mem_read(s,mem,p)$表示分区 p 在状态 s 下读取某块内存 mem, 而 $mem.owner\neq p$ 表示分区 p 不是内存 mem 的所有者. $ipc_call(s,p_1,p_2)$表示在状态 s 下, 分区 p_1 向分区 p_2 发起接口调用, ipc_acm 是分区间调用访问控制矩阵, $(p_1,p_2,ipc\_call)\notin ipc\_acm$ 表示访问控制配置不允许分区 p_1 向分区 p_2 发起分区间调用.

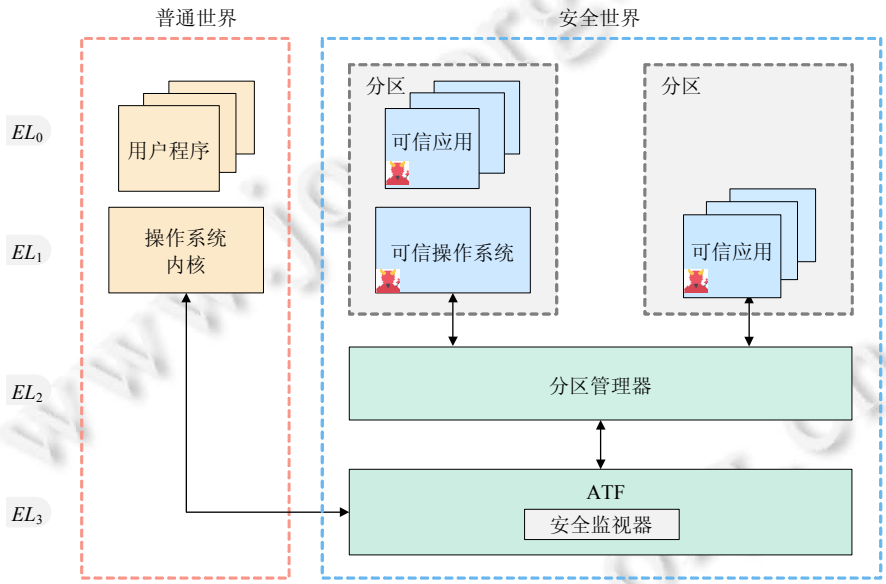


图 1 TrustZone 多分区架构的攻击面

3 整体验证思路

由于分区系统信息流安全的自动分析受到状态空间大小的限制, 使用模型检测方法来进行安全分析比较困难, 因此, 本工作使用基于定理证明的方法. 本工作选择 Isabelle/HOL 定理证明器, Isabelle 中简单而强大的函数式语言提供了形式规约所需的形式规约语言, 数学计算、数学定理证明及计算机软硬件系统都可以通过这种函数式语言来描述和实现, 并且 Isabelle 内置的逻辑系统也为形式规约所需要的形式验证提供了严格推理和证明的能力. 而 Isabelle/HOL 的高表达性、高度的证明自动化、强大的库等, 使它可以用于真实世界商用系统的大规模形式化验证. 此外, 大多数分区系统使用 Isabelle/HOL 的成功验证案例, 比如 seL4 和 PikeOS, 这也是支撑本工作使用 Isabelle/HOL 的一个因素.

图 2 所示是本文的建模和验证框架. 首先定义 TrustZone 分区系统的安全资产和安全策略, 抽象出分区间的干扰性, 基于干扰性定义信息流安全属性, 即机密性和完整性. 使用参数化抽象的状态机来描述抽象模型, 状态机中的定义的状态参数和函数都使用抽象类型, 与系统具体实现无关. 然后, 将抽象模型实例化为一个具体模型, 得到具体模型规约. 通过这种方式, 抽象模型中定义的属性、约束和安全性质都可在具体模型中保留和重用. 具体模型规约分为两个部分: 执行模型和事件规约. 执行模型是抽象模型的实例, 事件规约定义了 ARM FF-A 规范所定义的分区生命周期管理接口、分区间通信接口以及维护分区间隔离所需的内存管理接

口. 事件规约中的具体接口会被执行模型调用. 接着证明具体模型对抽象模型的精化关系及具体模型中所有事件接口的正确性, 并且事件接口都满足抽象模型中定义的安全属性. 最后对任意地址映射和未授权分区间通信这两种攻击行为建模, 验证模型能够防御分区的攻击行为.

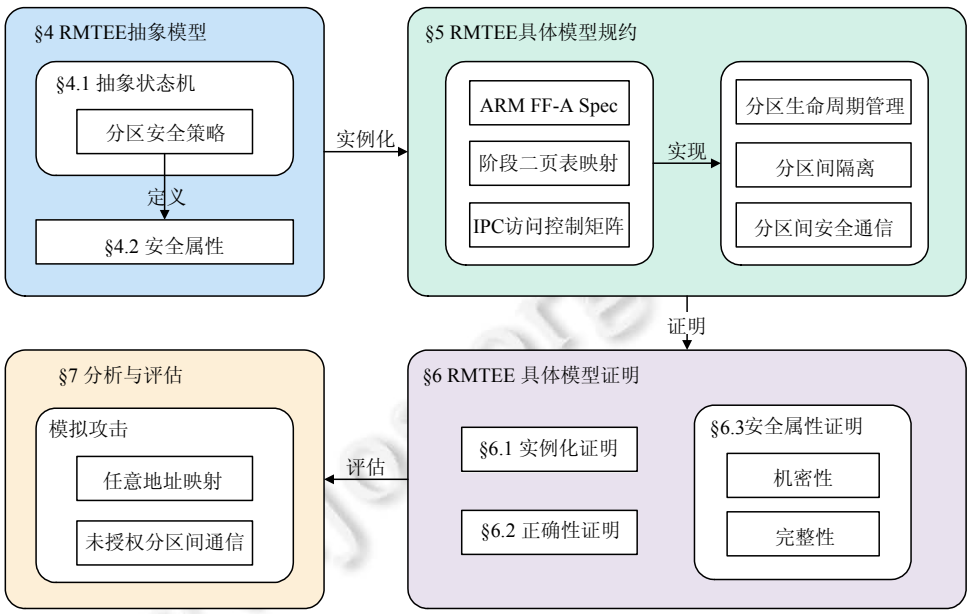


图 2 基于精化 TrustZone 多安全分区建模与验证

4 RMTEE 抽象模型

本节定义 RMTEE 抽象模型, 它由状态机、安全属性以及安全属性的推导关系组成. 在状态机模型中, 状态变量表示系统的状态, 转移函数和辅助函数用以描述状态变量的变化过程. 状态转移函数是系统对外暴露接口的抽象, 它们描述了系统状态如何变化. 安全属性是系统安全需求的形式化描述, 在形式化的安全模型中, 最为关键的部分就是安全性证明, 它的核心证明为对安全属性的可满足性证明. 抽象模型使用 Isabelle 的 *locale* 关键字定义, 其中的参数都是抽象类型, 而不涉及具体的数据结构和函数实现, 以便聚焦于系统性质的描述, 这些抽象参数在具体模型中被精化为具体实现.

4.1 抽象状态机

本文使用基于状态-事件的抽象状态机来描述系统模型, 系统状态机模型包含确定模型的要素(变量、函数、操作规则等)、安全初始状态与安全策略. 系统状态机是一个由抽象数据类型组成的元组, 定义如下.

定义 2(系统状态机). $M=\langle \mathcal{S},\mathcal{D},\mathcal{E},\varphi,\rightsquigarrow,\sim,\mathbb{D},s_0,SPM\rangle$, 其中,

- $\mathcal{S}$表示系统所有状态的集合;
- $\mathcal{D}$表示所有安全域的集合. 每个分区是一个安全域, 分区管理器也是一个安全域;
- $\mathcal{E}$表示系统有限的事件集合. 事件是系统接口的抽象, 由于系统提供确定数量的接口, 因此, 事件集合是有限的;
- φ是状态转移函数, $\varphi(e)\in\mathcal{S}\times\mathcal{S}$, 表示系统在一个状态下执行单个事件 e 后到达一个新的状态;
- $\rightsquigarrow$表示安全域间的干扰关系, $\rightsquigarrow\subseteq\mathcal{D}\times\mathcal{D}$, $d_1\rightsquigarrow d_2$ 表示允许信息从分区 d_1 流向分区 d_2. 不可干扰记为$\backslash\rightsquigarrow$;
- $\sim$表示状态等价, 两个状态 s 和 t 对于一个安全域 d 等价记为 $s\sim d\sim t$, 意味着在状态 s 和状态 t 下, 安全域 d 的私有数据完全相同. 在一些文献中, 等价(equivalence)也表述为不可区分(indistinguishable), 即分区无法区分两个状态;

- $\mathbb{D}$表示事件的执行域. 事件是由分区发起接口调用产生的, 在状态 s 下产生的事件 e 关联一个执行分区, 记为 $\mathbb{D}(s,e)\in\mathcal{D}$;
- s_0 表示系统初始状态, $s_0\in\mathcal{S}$;
- SPM 表示分区管理器, $SPM\in\mathcal{D}$, 区别于普通分区,拥有最高权限.

分区的干扰关系描述了系统所允许的信息流向, 表示授予后一个分区对前一个分区的数据读权限以及前一个分区对后一个分区的数据写权限. 干扰关系和状态等价是描述系统状态变化特征的基础, 基于它们的定义和系统安全策略, 定义以下 6 条安全策略规约.

(1) $\forall s, d, (s\sim d\sim s)$;

(2) $\forall s, t, d, (s\sim d\sim t)\rightarrow(t\sim d\sim s)$;

(3) $\forall s, t, r, d, (s\sim d\sim t)\wedge(t\sim d\sim r)\rightarrow(s\sim d\sim r)$;

(4) $\forall d\in\mathcal{D}, (d\rightsquigarrow d)$;

(5) $\forall d\in\mathcal{D}, (SPM\rightsquigarrow d)$;

(6) $\forall d\in\mathcal{D}, (d\rightsquigarrow SPM)\rightarrow d=SPM$.

前 3 条规约分别描述了状态等价的自反性、对称性和传递性, 后 3 条规约描述了系统安全策略, 即分区间信息流动的规则: 分区可以干扰自身; 分区管理器可干扰任意分区; 普通分区不能干扰分区管理器. 由于分区管理器拥有最高权限, 因此允许分区管理器向任意分区的信息流动.

4.2 安全属性

本文在状态机模型的基础上定义系统安全属性. 此前, 已有工作使用无干扰(noninterference)和无泄漏(nonleakage)来定义信息流安全性. 无干扰[28]是指系统中不能干扰分区 d 的那些分区所产生的事件不会影响分区 d 的状态. 无泄漏[29]的含义是在执行事件的过程中不向其他分区传递期望外的信息. 文献[29]证明了无干扰和无泄漏的结合等价于无影响(noninfluence)属性. 文献[22]对这些安全属性进行了扩展, 并推导论证了所有这些属性之间的蕴含关系, 然后证明了基于一组解旋条件可以推导出系统的无影响属性. 本文遵循文献[22]的推理框架, 在抽象模型中定义系统的机密性和完整性来描述信息流安全属性.

定义 3(事件的机密性).

$$\begin{aligned}&\textbf{\textit{confidentiality}}(\boldsymbol{e})\equiv\forall d, s, t, s', t',\\&(s\sim d\sim t)\wedge\\&(\mathbb{D}(s,e)=\mathbb{D}(t,e))\wedge\\&((\mathbb{D}(s,e)\rightsquigarrow d)\rightarrow(s\sim\mathbb{D}(s,e)\sim t))\wedge\\&(s,s')\in\varphi(e)\wedge(t,t')\in\varphi(e)\rightarrow\\&(s'\sim d\sim t')\end{aligned}$$

机密性保证不存在未预期的信息泄漏. 分区调用系统接口向其他分区发起分区间通信, 产生分区间的信息流动. 机密性定义要求在任意两个状态 s 和 t 下, 执行同个事件后,对于其他分区产生的影响是相同的, 即向其他分区传递的信息相同, 从而保证分区没有泄露期望外的信息. 如果事件执行时依据其他分区的数据产生不同的结果, 就会导致结果状态 s'和 t'不等价, 这样就泄漏了其他分区的数据.

定义 4(事件的完整性).

$$\begin{aligned}&\textbf{\textit{integrity}}(\boldsymbol{e})\equiv\forall d, s, s',\\&(\mathbb{D}(s,e)\backslash\rightsquigarrow d)\wedge\\&(s,s')\in\varphi(e)\rightarrow\\&(s\sim d\sim s')\end{aligned}$$

完整性防止未授权的数据篡改. 任何分区执行任意事件的过程中, 不允许修改在系统安全策略中定义的

该分区所不能干扰的其他分区的私有数据, 事件执行前后的状态, 对于该分区不能干扰的那些分区是等价的. 换言之, 分区只能向系统允许其干扰的分区发送数据.

总而言之, 机密性定义了对分区的读访问控制, 完整性定义了对分区的写访问控制. 基于单个事件的机密性和完整性, 可以定义系统的机密性 $confidentiality \equiv \forall e.confidentiality(e)$和完整性 $integrity \equiv \forall e.integrity(e)$.

5 RMTEE 具体模型规约

RMTEE 具体模型由执行模型和事件规约组成. 执行模型是抽象模型的实例, 使用 *interpretation* 关键字将抽象模型 *locale* 中的抽象参数和函数实例化为具体实现. 事件规约将事件具体化为一组 FF-A 接口函数, 用于建模目标模块的功能和服务. *locale* 类似于面向对象语言中的接口, 为本文实现由抽象到具体的精化模型提供支持; *interpretation* 则对应接口的实现, 使用具体参数将 *locale* 中的信息解释到当前上下文, 同时, 借助 Isabelle 的自动定理证明器证明具体化的模型仍然满足抽象模型中定义的规约.

5.1 自主访问控制

ARM S-EL2 规范定义了分区间通信与调用的标准接口, 但没有定义具体消息内容, 任意分区间可以预先定义通信协议设计消息体结构, 然后使用同一个标准接口完成不同的功能调用. 然而, 这种宽松的管理模式存在潜在风险: (1) 如果某个分区的具体功能实现没有对消息体中的输入参数进行合法性检查, 就有可能产生除零、整数溢出或命令注入等错误; (2) 恶意分区可以无限制地向特定分区发起请求, 占满其 vCPU, 从而实现拒绝服务攻击. 由于分区管理器不干涉消息体的结构和内容, 因此无法在分区管理器中完成消息内容的合法性检查.

为此, 本文设计了分区间调用的自主访问控制, 分区在其配置文件中定义其所信任的分区, 以及它们可以向本分区发起哪些接口调用. 通过访问控制机制定义系统的信息流安全策略, 为分区增加一层安全保障, 缓解漏洞利用和恶意攻击的风险. 分区间调用访问控制在逻辑上它可以表示为一个访问控制矩阵, 即:

定义 5(分区间调用访问控制矩阵). $IPC\_ACM=\langle S,O,A\rangle$, 其中,

- $S=\mathcal{D}$, 表示主体集合, 代表发起分区间调用的分区;
- $O=\mathcal{D}$, 表示客体集合, 代表接受调用的分区;
- A 是访问控制矩阵, $A[s][o]\in P(\mathcal{E})$, $s\in S$, $o\in O$, P 是幂集运算, $A[s][o]$是事件集合的一个子集, 表示主体分区可以向客体分区发起哪些接口调用.

表 2 是访问控制矩阵的一个示例, 其中, $\mathcal{E}$表示所有事件集合, ∅表示空集.

表 2 分区间调用访问控制矩阵示例

分区	P_1	P_2	P_3	P_4
P_1	$\mathcal{E}$	{*FFA_MSG_SEND_DIRECT_RESP*}	{*FFA_MSG_SEND*$_2$}	{*FFA_RUN*}
P_2	{*FFA_MSG_SEND_DIRECT_REQ*}	$\mathcal{E}$	∅	{*FFA_MEM_RELINQUISH*}
P_3	{*FFA_MSG_SEND*$_2$}	∅	$\mathcal{E}$	∅
P_4	∅	{*FFA_MEM_DONATE*, *FFA_MEM_LEND*, *FFA_MEM_SHARE*}	∅	$\mathcal{E}$

表 2 所示的访问控制矩阵规定, 分区 P_2 可向 P_1 发起直接消息通信, P_2 可向 P_1 返回结果; P_1 和 P_3 之间可以发起异步消息通信; P_1 可以调用 P_4 运行; P_4 可以将内存块转让或共享给 P_2, P_2 可以返回 P_4 共享的内存访问权限.

5.2 执行模型

执行模型使用具体参数类型和函数代替抽象模型中的抽象参数: 具体化的参数包括安全域(d)、状态(s)、事件(e), 具体化的函数包括干扰性($\rightsquigarrow$)、状态等价($\sim$)和事件的执行域($\mathbb{D}$). 它们在具体模型中的精化实现如表

3 所示.

其中, 安全域(*domain*)和状态(*state*)使用 Isabelle 的 *record* 关键字定义, 类似于结构体类型, 由多个字段组成. 通过 *record* 的扩展语法(类似于面向对象的继承), 又进一步定义了分区类型(*partition*)和分区管理器类型(*SPM*), 它们继承了安全域中的字段,同时又包含各自独有的扩展字段. 系统状态描述系统全局的数据,包括分区和分区管理器的运行时状态数据. 事件(event)通过 *datatype* 关键字定义, 类似于构造体. 这些具体化参数的数据结构见表 4.

表 3 状态机抽象参数的精化实现

状态机参数	抽象模型	具体模型
安全域	d	*record* Domain
状态	s	*record* State
事件	e	*datatype* Event
状态转移	φ	*definition fn_step*
干扰性	$\rightsquigarrow$	*definition fn_interference*
状态等价	$\sim$	*definition fn_equiv*
事件的执行域	$\mathbb{D}$	*definition fn_kdom*

表 4 具体模型中精化的参数数据结构

参数类型	字段	字段类型	描述
Domain	*id*	*Domain_ID*	分区的唯一标识
	version	*Major_Version*×*Minor_Version*	分区所实现的 FF-A 版本, 由主版本号和次版本号元组组成
Partition	*vcpu_ids*	*vCPU_ID* set	分区的 *vcpu id* 集合
	boot_order	*nat*	分区启动优先级, 分区管理器在初始化时按优先级顺序启动各个分区
	rx_buffer	*IPA*	分区的 RX Buffer 的起始虚拟地址和大小
	tx_buffer	*IPA*	分区的 TX Buffer 的起始虚拟地址和大小
SPM	*reserved*	*nat*	保留字段, 分区管理器没有额外的配置参数
State	*currents*	*CPU_ID*⇀*Domain_ID*×*vCPU_ID*	每个物理 CPU 上正在运行的分区 *vCPU* 信息
	partitions_state	*Domain_ID*⇀*Partition_State*	每个分区运行时状态数据
	buffers	*Domain_ID*⇀*RXTXBuffer*	每个分区的 RX/TX Buffer 数据
	registers	*Domain_ID*×*vCPU_ID*⇀*Register*	每个分区的每个 *vCPU* 中的寄存器数据
	memories	*Mem_Block* set	所有内存块的集合
Partition_State	*vcpus*	*vCPU* set	分区的 *vCPU* 集合
	vmmu	*IPA*⇀*PA*	分区的页表
Event	*hyperc*	*Hypercall CPU_ID*	提供给分区的系统调用接口
	intlc	*Internal_Call CPU_ID*	分区管理器内部函数
Sys_Conf	*spm*	*Domain_ID*	分区管理器的 *id*
	partitions_conf	*Domain_ID*⇀*Partition*	每个分区的配置清单
	cpus	*CPU_ID* set	物理 *CPU id* 的集合
	memories	*Mem_Block* set	初始物理内存块的集合, 表示页框粒度(如 4KB)
	ipc_acm	(*Domain_ID*×*Domain_ID*×*Event*) set	分区间调用访问控制矩阵

Domain 类型包含分区的唯一标识 *id* 和记录分区所实现的 FF-A 版本信息. *Partition* 类型是分区的静态配置清单, 用于分区初始化,其中的 *IPA* 表示一块虚拟地址空间, 由起始地址和大小组成. *State* 类型记录所有分区的运行时状态数据, 包括分区的 RX/TX Buffer、分配给分区的内存、分区的 *vCPU*、页表以及分区中每个 *vCPU* 的寄存器数据. 其中的 *memories* 字段代表系统中所有内存块的集合, *Mem_Block* 中有一个 *owner* 字段标识该内存块属于哪一个分区. 状态类型中的 *currents* 字段记录了当前状态下每个物理 *CPU* 上所运行的 *vCPU* 信息. *record* 类型变量中的字段通过(字段名 *record* 变量)的方式取出. 表中的⇀符号表示偏函数, 它在定义域的某些值上可能没有对应的值, 比如: 若在 *s* 状态下 *cpu*0 处于空闲状态, 则((*currents s*) *cpu*0)的取值就是 *None*. 符号×表示二元组, 比如 *currents* 的取值若存在, 则是一个由分区 *id* 和 *vCPU id* 组成的二元组. *Event* 类

型分为提供给分区的系统调用(*Hypercall*)和分区管理器内部函数(*Internal_Call*). 这两种事件类型又进一步枚举为一系列具体的事件接口, 在 *fn_step* 中映射到 FF-A 规范所定义的标准接口. 状态转移函数 ***fn_step*** 使用具体的状态和事件类型精化事件执行, 伪代码定义如下.

定义 6(状态转移).

fn_step e≡case e of

$Hypercall.FFA\_MSG\_SEND_2 \Rightarrow (s, ffa\_msg\_send_2(s,\ldots))$

$Hypercall.FFA\_MSG\_WAIT \Rightarrow (s, ffa\_msg\_wait(s,\ldots))$

…

$Internal\_Call.SP\_INIT \Rightarrow (s, sp\_init(s,\ldots))$

$\_ \Rightarrow (s,s)$

fn_step 依据事件类型的不同, 调用不同的事件接口, 最后返回由起始状态和结束状态所组成的二元组. 其中的 $ffa\_msg\_send_2$ 等函数是事件接口的具体实现, 它们组成了事件规约, 在第 5.3 节详细介绍. 实例化的状态等价函数 ***fn_equiv*** 表示: 对于分区管理器而言, 若在两个状态下, 分区管理器拥有的内存相同, 则它们等价; 对于普通分区而言, 状态等价意味着分区的 RX/TX Buffer、内存、*vCPU* 和寄存器数据相等. 事件类型中包含 *CPU_ID*, 标识发起事件的 CPU, 在结合状态类型中的 *currents* 字段定义, 可查询到该 CPU 上正在执行的分区,从而定位到事件的执行域, 这就是事件的执行域函数 ***fn_kdom*** 的实现.

执行模型中的干扰性函数 ***fn_interference*** 伪代码定义如下.

定义 7(干扰性).

fn_interference d_1 d_2≡

if (d_1=d_2) **then** True

else if $is\_spm(d_1)$ **then** True

else if $is\_spm(d_2)$ **then** False

else if $is\_valid\_ipc(d_1, d_2)$ **then** True

else False

fn_interference 是使用 *definition* 关键字定义的函数, d_1, d_2 的类型是 *Domain_ID*, 分别表示源分区和目标分区. 首先判断 d_1 是否等于 d_2: 若是, 则返回 True, 表示分区可以干扰自身. $is_spm(d_1)$判断 d_1 是否为分区管理器: 若是, 则返回 True, 表示分区管理器可以干扰任意分区. 接着判断 d_2 是否为分区管理器(此分支下隐含前提 d_1 不是分区管理器): 若是, 则返回 False, 表示任意分区不能干扰分区管理器. $is_valid_ipc(d_1,d_2)$判断(d_1,d_2)所组成的元组是否在访问控制矩阵中, 表示分区之间能否干扰由系统配置中的分区间调用访问控制矩阵决定. 最后一个分支表示系统定义之外的分区不能干扰任何分区.

具体模型中还定义了系统配置(*Sys_Conf*)类型, 它是由一组静态变量组成的 *record* 结构体. 系统配置中的参数在运行过程中保持不变, 用于系统初始化或验证事件执行的合法性. 系统配置的数据结构如表 4 所示: *spm* 字段指定了分区管理器的 *id*; *partitions_conf* 字段是一个从分区 *id* 到 *Partition* 类型的映射,记录所有分区的配置清单; *cpus* 是物理处理器 *id* 的集合; *memories* 是初始物理内存块的集合; *ipc_acm* 是⟨分区,分区,事件⟩三元组的集合, 等价于第 5.1 节所述的分区间调用访问控制矩阵. 由于系统配置决定了系统的初始状态和运行时行为, 模型中定义了 5 条规则来约束系统配置.

(1) 任何普通分区都不是分区管理器:

$$\forall p, (sys\_conf.partitions\_conf[p] \neq None) \rightarrow (p \neq sys\_conf.spm);$$

(2) 分区管理器不是普通分区:

$$\forall p, (sys\_conf.spm = p) \rightarrow (sys\_conf.partitions\_conf[p] = None);$$

(3) 分区配置中的分区 *id* 必须正确配置:

$$\forall p, (sys\_conf.partitions\_conf[p] \neq None) \rightarrow ((sys\_conf.partitions\_conf[p]).id = p);$$

(4) 任意两个物理内存块的地址互不重叠:

$\forall x, y, (x\neq y\wedge x\in sys\_conf.memories\wedge y\in sys\_conf.memories)\rightarrow\neg(is\_memory\_conflicted(x,y))$;

(5) IPC 访问控制矩阵中的配置的都是有效分区.

$\forall(d_1,d_2,e)\in sys\_conf.ipc\_acm, (sys\_conf.partitions\_conf[d_1]\neq None)\wedge(sys\_conf.partitions\_conf[d_2]\neq None)$.

只有符合这些规则的系统配置才是合法的, 以保证系统正确运行. 这些规约通过 Isabelle 的 *specification* 关键字定义, 它用于对特定常量定义一系列公式来描述其约束, 并提供推导过程, 证明满足约束的常量是存在的.

5.3 事件规约

如第 5.2 节所述, 具体模型的事件规约定义了两类事件: 系统调用和内部函数, 它们描述了 FF-A 标准所定义的系统行为和接口, 并在执行模型的 *fn_step* 状态转移函数中被调用作为实际执行体. 系统调用是分区管理器提供给分区调用的系统接口, 用于实现分区间通信等功能. 内部函数实现分区管理器的内部功能, 不暴露给分区. 事件规约中定义了分区生命周期管理、分区间隔离和分区间通信这 3 个模块的共计 36 个函数, 详见表 5.

表 5 RMTEE 事件函数

接口函数	功能描述
ffa_error	状态报告接口, 返回FF-A接口调用失败的错误代码
ffa_success	状态报告接口, 返回FF-A接口调用成功的结果
ffa_interrupt	状态报告接口, 切换到分区处理中断
ffa_version	获取框架版本号
ffa_features	查询指定*function*是否实现
ffa_rxtx_map	为分区的RX/TX Buffer建立映射
ffa_partition_info_get	查询指定分区的信息
ffa_id_get	分区获取自身ID
ffa_spm_id_get	获取分区管理器ID
ffa_msg_wait	分区结束自身运行, 进入READY状态
ffa_yield	分区阻塞自身运行, 进入BLOCKED状态
ffa_run	分区阻塞自身运行, 并唤醒目标分区的指定*vCPU*执行, 自身进入BLOCKED状态
*ffa_msg_send*2	异步消息通信接口, 将消息从*sender*的TX Buffer传输到*receiver*的RX Buffer
ffa_msg_send_direct_req	同步消息通信接口, *sender*唤起*receiver*的一个*vCPU*处理消息, 自身进入BLOCKED状态
ffa_msg_send_direct_resp	发送对*ffa_msg_send_direct_req*的响应消息, *sender*唤醒目标*endpoint*运行, 自身进入READY状态
ffa_mem_donate	分区把内存块的所有权和访问权转移给指定分区
ffa_mem_lend	分区把内存块的独占访问权转移给指定分区
ffa_mem_share	分区把内存块的共享访问权分享给指定分区
ffa_mem_retrieve_req	分区请求完成*donate*, *lend*或*share*这3种事务
ffa_mem_retrieve_resp	响应*ffa_mem_retrieve_req*的请求
ffa_mem_relinquish	分区归还*lend*/*share*的内存块访问权
ffa_mem_reclaim	分区恢复对内存块的独占访问权
ffa_mem_frag_rx	分区申请内存事务描述符的下一片传输
ffa_mem_frag_tx	完成并响应*ffa_mem_frag_rx*请求
mm_map	对指定内存区域建立页表映射
mm_unmap	对指定内存区域取消页表映射
mem_alloc	内存申请函数
mem_free	内存释放函数
sp_get_count	获取系统的分区数量
sp_find	使用ID查找对应分区
sp_get_vcpu	获取*vCPU*, 使用*vCPU ID*查找安全分区自身的*vCPU*作为返回值
vcpu_on	*vCPU*状态启动接口, 将*vCPU*状态从初始状态OFF修改为READY
vcpu_run	*vCPU*状态运行接口, 将*vCPU*状态从其他状态修改为RUNNING
vcpu_switch	*vCPU*切换接口, 切换到指定*vCPU*执行, 自身进入BLOCKED状态
get_s2_table_address	获取分区阶段二页表地址
sp_init	分区初始化接口, 读取分区配置, 为分区分配内存、RX/TX Buffer, 建立页表映射, 初始化*vCPU*集合

5.3.1 分区生命周期管理

分区生命周期管理维护分区正常运行, 包括分区初始化、分区切换和分区状态查询等. 分区镜像和分区配置清单保存在持久化存储中, 它们都通过固化在芯片中的证书签名, 并通过安全启动保证其完整性. 分区管理器的初始化函数从持久化存储加载分区镜像和配置清单, 然后依据配置清单描述为分区分配所需资源, 完成分区初始化; 之后, 分区进入等待状态. TrustZone 的安全分区不主动运行, 而是被动等待请求, 当分区管理器接收到针对某个分区的接口调用时, 才会切换到目标分区执行. 一个分区可以有多个执行实例, 它们的运行时上下文组织为 *vCPU* 数据, 由分区管理器维护. 分区 *vCPU* 的状态可以为关闭、开启、运行、等待、阻塞、被抢占中的任意一种, 它们之间的转换关系[8]如图 3 所示.

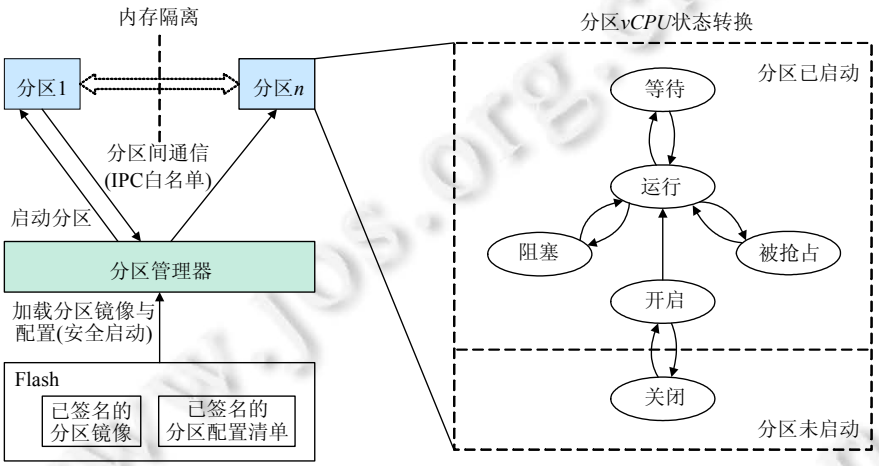


图 3 分区生命周期管理

分区管理器在执行分区 *vCPU* 切换时, 会将当前寄存器信息保存在当前分区 *vCPU* 结构中, 然后载入目标分区 *vCPU* 的信息, 切换到目标分区 *vCPU* 执行. 以 *vCPU* 阻塞式切换为例, 其伪代码描述的定义如下.

定义 8 (*vCPU* 切换函数).

vcpu_switch *s cpu_id dst_id dst_vcpu_id*≡

(*src_id*,*src_vcpu_id*)=*get_current*(*s*,*cpu_id*)

src_vcpu=*get_vcpu*(*s*,*src_id*,*src_vcpu_id*)

dst_vcpu=*get_vcpu*(*s*,*dst_id*,*dst_vcpu_id*)

s′= *update_vcpu_state*(*s*,*src_id*,*src_vcpu*,*state*=BLOCKED)

s′=*update_vcpu_state*(*s*′,*dst_id*,*dst_vcpu*,*state*=RUNNING)

s′=*update_current*(*s*′,*cpu_id*,(*dst_id*,*dst_vcpu_id*))

return *s*′

vcpu_switch 是使用 *definition* 关键字定义的函数, 它的输入参数包括当前状态 *s*、物理处理器 *id*、目标分区 *id* 和目标 *vcpu id*. 其中: *get_current* 获取当前状态下指定物理 CPU 上所运行的分区和 *vCPU* 的 *id*; *get_vcpu* 获取指定分区和 *vCPU id* 的 *vCPU* 对象; *update_current* 更新物理 CPU 上运行的 *vCPU*; *update_vcpu_state* 更新指定 *vCPU* 的状态. 函数将源分区正在执行的 *vCPU* 置为阻塞状态, 目标分区 *vCPU* 置为运行状态, 并更新当前所在物理 CPU 的上的运行 *vCPU* 为目标 *vCPU*, 最后返回更新后的状态.

5.3.2 分区间隔离.

分区间隔离通过分区管理器对分区的内存管理实现, 具体包括阶段 2 页表映射、内存池管理和内存共享. 传统操作系统的页表管理负责将虚拟地址转换成物理地址, 页表中记录虚拟内存和物理地址的映射关系. 随着地址空间范围的扩大, 一级页表可能扩展为 3 级乃至 4 级页表, 但是 MMU (memory management unit)的核

心功能并没有变化. MMU 对应用程序封装了内存的具体实现, 使得程序可以安全申请和使用内存, 而无需担心内存越界和覆盖. 在引入分区管理器后, 在虚拟地址和物理地址之间增加了中间物理地址(intermediate physical address, IPA). 如图 4 所示: 分区内的操作系统内核负责将虚拟地址转换为中间物理地址, 这个过程称为阶段 1 页表映射; 分区管理器将中间物理地址转换成实际的物理地址, 这个过程称为阶段 2 页表映射[7]. 通过阶段 2 页表映射, 分区管理器可以控制每个分区只能访问自己的内存, 并且分区操作系统没有权限修改阶段 2 页表来映射到属于其他分区的内存地址. 在分区管理器看来, 分区的 IPA 就是一个虚拟地址, 因此, 阶段 2 地址映射与普通的地址映射没有区别.

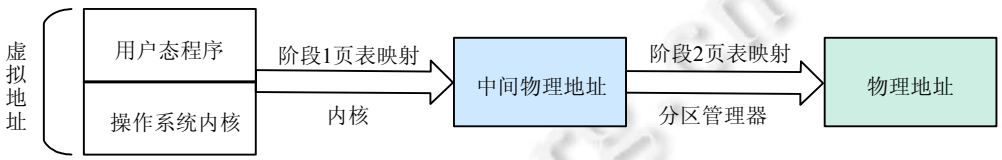


图 4 阶段 2 页表映射

页表映射函数的伪代码定义如下.

定义 9(页表映射函数).

mm_map *s cpu_id domain_id ipa pa*≡

if (*get_pa_owner*(*pa*)≠*domain_id*) **then**

return *s*

else

vmmu=*get_page_table*(*s*,*domain_id*)

vmmu[*ipa*]=*pa*

s′=*update_page_table*(*s*,*domain_id*,*vmmu*)

return *s*′

mm_map 是使用 *definition* 关键字定义的函数, 它的输入参数包括当前状态 *s*、物理处理器 *id*、分区 *id*、要映射的虚拟地址 *ipa* 和物理地址 *pa*. *mm_map* 首先通过 *get_pa_owner* 获取内存块的所有者, 验证所要映射的物理地址属于当前分区, 满足要求则再调用 *get_page_table* 获取分区的阶段 2 页表 *vmmu*, 它是一个从 *IPA* 到 *PA* 的映射; 然后, 将输入的 *ipa*→*pa* 的映射关系添加到 *vmmu* 中; 最后更新分区页表, 返回更新后的状态. 类似地定义了解除页表映射的函数 *mm_unmap*, 它完成相反的操作.

内存管理的另一个功能是内存分配和回收, FF-A 规范没有规定分区管理器如何实现内存池管理, 它是由具体实现决定的. RMTEE 具体模型使用了基于位图的动态内存分配算法. 初始状态下, 分区管理器维护一个空闲内存池, 并记录每个内存块的分配状态. 在分区初始化时, 根据分区配置为分区分配合适大小的内存块. 在分区运行过程中, 可以通过分区管理器提供的接口动态申请和释放内存. 内存分配函数的定义如下.

定义 10(内存分配函数).

mem_alloc *s sys_conf cpu_id mem_size*≡

unalloc_mem=*get_unallocated_mem_bysize*(*s*,*mem_size*)

if (*unalloc_mem*=*None*) **then**

return *s*

else

(*caller_id*,_)=*get_current*(*s*,*cpu_id*)

alloc_mem=*update_mem*(*unalloc_mem*,*owner*=*caller_id*,*access*={*caller_id*},*allocated*=True)

memories=*get_memories*(*s*)

memories=*insert*(*memories*−{*unalloc_mem*},*alloc_mem*)

s′=*update_memories*(*s*,*memories*)
if (∃*x*, *y*, *x*≠*y*∧*x*∈*get_memories*(*s*′)∧*y*∈*get_memories*(*s*′)∧*is_memory_conflicted*(*x*,*y*)) **then**
return *s*
else
return *s*′

mem_alloc 是使用 *definition* 关键字定义的函数, 它的输入参数包括当前状态 *s*、系统配置、物理处理器 *id* 和申请的内存大小. 函数首先通过 *get_unallocated_mem_bysize* 确认系统中存在满足所申请大小的空闲内存块 *unalloc_mem*. 然后, 通过 *update_mem* 修改其拥有者为当前分区、当前分区有访问权限(*access*={*caller_id*})、内存状态为已分配, 得到新的内存块对象 *alloc_mem*. 函数 *get_memories* 获取系统内存块集合, 接着, 从 *memories* 中删除旧内存块 *unalloc_mem*, 插入新内存块 *alloc_mem*, 完成分配操作, 得到新的系统状态 *s*′. 如果分配后导致任意两个物理内存块的地址互重叠, 则撤销此次分配, 回滚为分配前的状态 *s*; 否则, 返回状态 *s*′. FF-A 规范定义了分区间共享内存的接口, 分区可以将自己拥有的内存的访问权共享或暂时转让给另一个分区, 或者将内存所有权转移给另一个分区. 篇幅所限, 此处不再列举相关代码.

5.3.3 分区间通信.

分区管理器提供同步消息通信接口和异步消息通信接口. 在同步消息通信下, 发送方发起消息请求后, 会让出自身的执行权限, 唤起接收方处理消息. 接收方处理完成后, 再将执行权限返还给发送方. 如图 5 所示是同步消息通信的时序逻辑[11]. 在异步消息通信下, 发送方发起消息请求后,继续运行; 由外部调度器唤醒接收方执行来完成消息处理, 接收方完成请求后, 可以以相同的方式将处理结果返回给发送方, 发送方可以通过轮询的方式接受处理结果. 在两种通信模式下, 分区管理器都作为中间媒介转发消息. 在 RMTEE 具体模型中, 分区管理器还会验证通信双方是否合法, 即是否在分区间调用访问控制矩阵中.

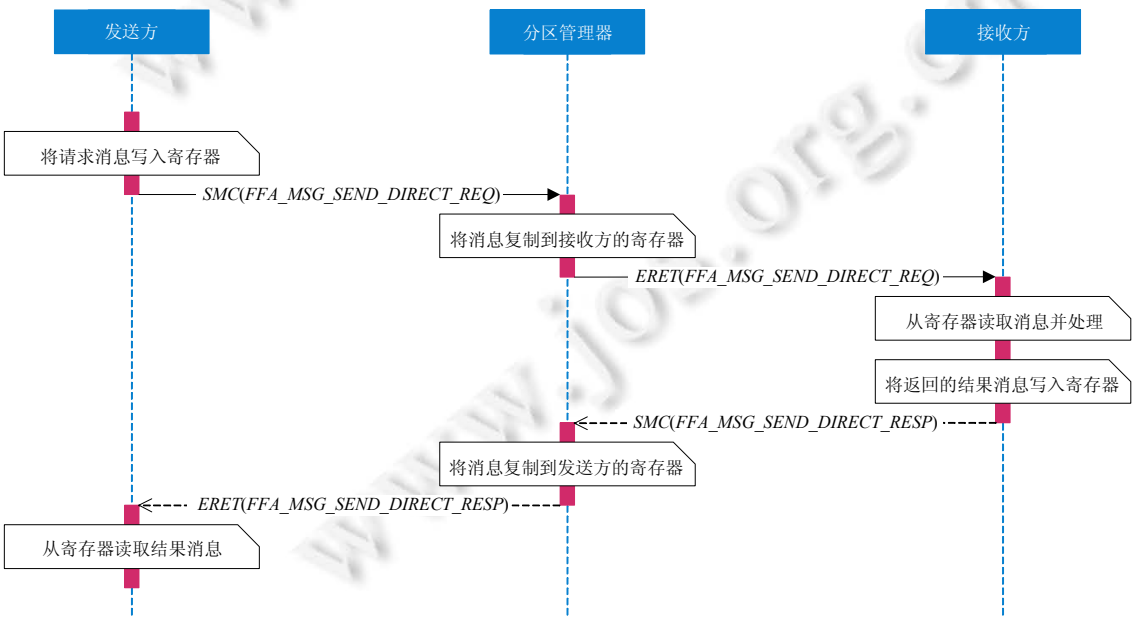


图 5 同步消息通信

消息传输函数的伪代码定义如下.

定义 11(消息传输函数).

transfer_msg *s msg*≡
src_id=*msg.sender_id*

dst_id=msg.receiver_id

s'=update_buffer(*s*,*src_id*,*content=None*,*owner=src_id*)

s'=update_buffer(*s'*,*dst_id*,*content=msg*,*owner=dst_id*)

return *s'*

transfer_msg 是使用 *definition* 关键字定义的函数, 它的输入参数包括当前状态 *s* 和要传输的消息 *msg*. 其中, *update_buffer* 函数更新指定分区 *buffer* 的内容和所有者. *transfer_msg* 从消息头获取发送方分区标识 *src_id* 和接收方分区标识 *dst_id*, 然后将消息复制到接收方的 RX Buffer, 清空发送方的 TX Buffer(设置为 *None*). 最后更新系统状态中保持的分区缓冲区数据, 返回更新后的状态. 消息通信接口(如 $ffa\_msg\_send_2$ 等)会在验证输入参数以及系统状态的合法性后, 调用 *transfer_msg* 函数完成消息转发.

6 RMTEE 具体模型证明

本节介绍对 RMTEE 具体模型的验证工作, 包括: 对执行模型的实例化证明, 保证具体模型是对抽象模型的正确精化; 对事件规约的正确性证明, 证明形式化描述的具体事件满足系统功能需求; 以及对事件的安全属性证明, 验证其符合上层抽象模型中所定义的机密性和完整性要求.

6.1 实例化证明

精化关系的证明即验证下层模型的行为与上层模型行为保持一致. 对于本文的具体模型和抽象模型而言, 就是要证明在将安全域、系统状态、事件、干扰性、状态转移、状态等价、事件的执行域等参数实例化后, 具体模型仍然满足抽象模型所定义的安全策略规约. RMTEE 具体模型中实现了一系列引理, 证明执行模型满足第 4.1 节所述的 6 条安全策略规约, 即状态等价的自反性、对称性和传递性以及分区间信息流动的规则. 例如, 普通分区不可干扰分区管理器在执行模型中的描述如下所示.

引理 1(普通分区不可干扰分区管理器).

fn_sp_non_inf_spm≡

$\forall d, fn\_interference(d,spm) \rightarrow (d=spm)$

引用干扰性定义 *fn_interference*, 可通过 Isabelle 自动定理证明器证明上述引理. 类似地, 可以得到其他 5 条规约的具体化引理及其证明, 因此可说明具体模型的执行模型是对抽象模型的正确精化.

6.2 正确性证明

事件规约的正确性证明, 分解为对事件规约中每个具体事件的正确性证明, 本文使用霍尔逻辑[30]来描述事件的正确性. 霍尔逻辑可表述为$\{P\}\ C\ \{Q\}$, 其中, *C* 代表一个事件, *P* 代表前置条件, *Q* 代表后置条件. 在 *P* 成立的条件下执行 *C* 之后, 要么 *C* 不终止, 要么 *Q* 成立. 由于本文使用 Isabelle 的 *definition* 关键字定义具体事件, 自动保证了事件的可终止性, 因此, 事件的正确性证明就转化为证明在给定前提下执行事件后到达期望的状态. 以 *vCPU* 切换函数为例, 其正确性定义如下.

引理 2(*vCPU* 切换函数的正确性).

vcpu_switch_correctness≡

$\{get\_vcpu(s,dst\_id,dst\_vcpu\_id) \neq None \wedge get\_vcpu\_state(s,dst\_id,dst\_vcpu\_id) \neq \text{OFF} \wedge$

$get\_vcpu\_state(s,dst\_id,dst\_vcpu\_id) \neq \text{RUNNING}\}$

$r=vcpu\_switch(s,cpu\_id,dst\_id,dst\_vcpu\_id)$

$\{get\_vcpu\_state(r,pid,dst\_vcpu\_id)=\text{RUNNING} \wedge get\_current(r,cpu\_id)=(dst\_id,dst\_vcpu\_id) \wedge$

$vcpus-\{vcpu\}=r\_vcpus-\{r\_vcpu\}\}$

其中, *get_vcpu* 获取指定的分区 *vcpu*, *get_vcpu_state* 获取指定分区 *vcpu* 的运行状态, *vcpu* 和 *vcpus* 分别是事件执行前所指定的 *vcpu* 和分区的 *vCPU* 集合, *r_vcpu* 和 *r_vcpus* 分别是事件执行后当前运行的 *vcpu* 和分区的 *vCPU* 集合. 前置条件保证所指定的 *vcpu* 存在、已启动且不处于运行状态. 后置条件表示函数正确执行后,

所指定的 *vcpu* 处于运行状态, 且该分区 *vCPU* 集合中的其他 *vcpu* 未被修改(如图 6(a)所示).

如果事件修改了 *vCPU* 集合中的其他元素, 比如删除了一个 *vcpu*(图 6(b))、增加了一个 *vcpu*(图 6(c))或者修改了其他 *vcpu* 的状态(图 6(d)), 那么 *vcpus*−{*vcpu*}或 *r_vcpus*−{*r_vcpu*}中必定存在其他被修改的 *vcpu*, 而不会等于两个集合的交集, 因此就不满足后置条件中 *vcpus*−{*vcpu*}=*r_vcpus*−{*r_vcpu*}的断言.

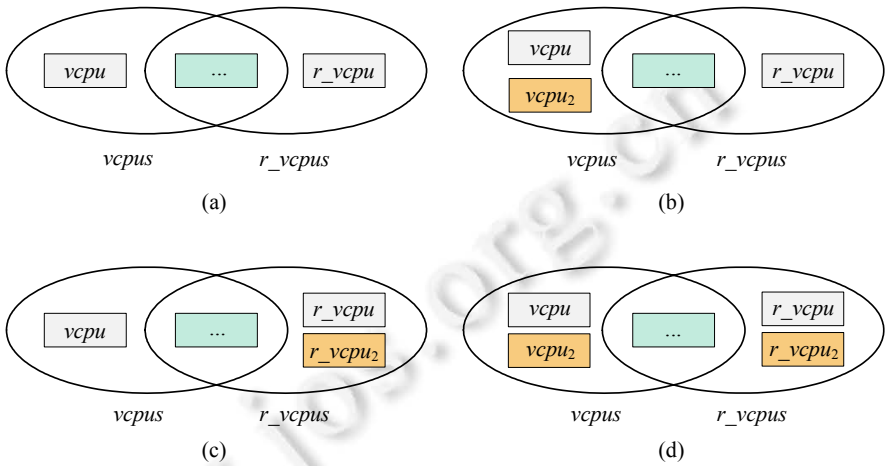


图 6 *vCPU* 集合修改前后的交集关系

6.3 安全属性证明

具体模型继承了抽象模型中定义的机密性和完整性, 并证明每个具体事件都满足这两个安全属性的要求. 为了简化具体事件的安全性证明, 定义了事件的弱机密性.

定义 12(事件的弱机密性).

$$\begin{aligned}&\textbf{\textit{weak\_confidentiality}}(e)\equiv\forall d,s,t,s',t',\\&\quad(s\sim d\sim t)\wedge\\&\quad(\mathbb{D}(s,e)=\mathbb{D}(t,e))\wedge\\&\quad(\mathbb{D}(s,e)\rightsquigarrow d)\wedge\\&\quad(s\sim\mathbb{D}(s,e)\sim t)\wedge\\&\quad(s,s')\in\varphi(e)\wedge(t,t')\in\varphi(e)\rightarrow\\&\quad(s'\sim d\sim t')\end{aligned}$$

弱机密性假定机密性定义中的前提 $\mathbb{D}(s,e)\rightsquigarrow d$ 成立, 从而得到 $s\sim\mathbb{D}(s,e)\sim t$ 也成立, 然后依据这两条前提得到结论. 对于 $\mathbb{D}(s,e)\rightsquigarrow d$ 不成立的情况, 通过完整性可以推导出要证明的结论, 即这几条安全属性间存在推导关系: ⟦*weak_confidentiality*, *integrity*⟧⇒*confidentiality*. 因此, 具体模型中只需证明事件的弱机密性和完整性, 相比于证明机密性和完整性, 减少了大量重复的证明工作.

对于每个具体事件, 其安全性定义就是将抽象模型的安全属性中的事件替换为该事件. 以异步消息通信接口 $ffa\_msg\_send_2$ 为例, 其弱机密性定义如下.

引理 3(*ffa_msg_send*$_2$ 接口弱机密性).

$$\begin{aligned}&\textbf{\textit{ffa\_msg\_send}}_2\textbf{\textit{\_weak\_confidentiality}}\equiv\forall d,s,t,\\&\quad e=Hypercall.FFA\_MSG\_SEND_2\wedge\\&\quad(s\sim d\sim t)\wedge\end{aligned}$$

$(\mathbb{D}(s,e)=\mathbb{D}(t,e))\wedge$

$(\mathbb{D}(s,e)\rightsquigarrow d)\wedge$

$(s\sim\mathbb{D}(s,e)\sim t)\wedge$

$s'=ffa\_msg\_send_2(s,cpu\_id)\wedge$

$t'=ffa\_msg\_send_2(t,cpu\_id)\rightarrow$

$(s'\sim d\sim t')$

其完整性定义 *ffa_msg_send*$_2$*_integrity* 也是类似地将完整性定义中的事件替换为 *ffa_msg_send*$_2$. 在具体接口的安全属性证明中, 将结果状态展开, 显式地向证明器提示接口所修改的状态变量, 结合 Isabelle 的自动推导能力, 可以证明结果状态的等价性. 在证明了所有事件的机密性和完整性后, 就可以对机密性和完整性定义中的事件应用归纳规则, 进一步证明整个事件规约的安全属性, 从而证明具体模型满足信息流安全要求.

7 分析与评估

本节对 RMTEE 模型进行全面地评估, 首先对本文的分区管理器模型进行安全性分析, 接下来给出形式化规约和验证的工作量评估, 最后讨论该模型的可扩展性.

7.1 安全性分析

本文考虑分区中的操作系统或应用程序存在恶意代码, 或者存在可被利用的漏洞, 导致分区成为恶意分区, 恶意分区会尝试窃取或篡改其他分区或分区管理器中的运行时数据, 本工作所验证的 RMTEE 模型可抵御分区的这种恶意行为. 为证明 RMTEE 模型的安全性, 本文建模了两种攻击场景: 任意地址映射和未授权的分区间通信. 由于分区所访问的内存地址都要通过分区管理器的阶段 2 页表进行 IPA 到 PA 的转换, 因此, 如果分区希望恶意访问属于其他分区的内存, 必须先通过分区管理器的内存映射函数在自己的阶段 2 页表建立映射到其他分区 PA 的页表项. 任意地址映射的形式化定义如下.

定义 13(任意地址映射).

random_addr_map$\equiv\exists s, cpu\_id, p_1, ipa, mem,$

$mem.owner\neq p_1\wedge$

$mm\_map(s,cpu\_id,p_1,ipa,mem)\neq s$

它表示恶意分区 p_1 在自己的阶段 2 页表映射一个不属于自己的物理内存地址, 并且映射函数 *mm_map* 返回状态不为 s, 表示调用成功, 修改了系统状态. 此后, 分区 p_1 就可以使用 *ipa* 访问这个物理地址. 未授权分区通信的定义如下.

定义 14(未授权的分区间通信).

unauthorized_ipc$\equiv\exists s, cpu\_id, p_1, p_2, msg,$

$(p_1,p_2,FFA\_MSG\_SEND_2)\notin(sys\_conf.ipc\_acm)\wedge$

$msg=get\_buffer(s,p_1).content\wedge$

$msg.receiver\_id=p_2\wedge$

$ffa\_msg\_send_2(s,cpu\_id)\neq s$

系统配置的分区间通信访问控制矩阵不允许 p_1 向分区 p_2 发起异步消息通信, 但恶意分区 p_1 通过成功调用异步消息通信接口与分区 p_2 通信. 利用条件-合取等价式 $p\wedge q\equiv\neg(p\rightarrow\neg q)$, 可将上述攻击模型中的命题转化成蕴含式. 例如, *random_addr_map* 转化为

$$\exists s, cpu\_id, p_1, ipa\ mem, \neg((mem.owner\neq p_1)\rightarrow(mm\_map(s,cpu\_id,p_1,ipa,mem)=s)).$$

在模型中将其证伪, 即其否定为真, 即可证明模型可以防御任意地址映射攻击. 其否定为

$$\forall s, cpu\_id, p_1, ipa, mem, ((mem.owner\neq p_1)\rightarrow(mm\_map(s,cpu\_id, p_1,ipa,mem)=s)).$$

引用 *mm_map* 函数的正确性证明引理, Isabelle 定理证明器可自动推导上述命题为真, 因此, 证明阶段 2

页表映射可以防止分区的任意地址映射. 对 *unauthorized_ipc* 执行相似的证明过程, 可证明 *unauthorized_ipc* 的否定为真, 其定义为

$$\begin{aligned}&\forall s, cpu\_id, p_1, p_2, msg,\\&\quad (p_1,p_2,FFA\_MSG\_SEND_2)\notin(sys\_conf.ipc\_acm)\wedge\\&\quad msg=get\_tx\_buffer(s,p_1).content\wedge\\&\quad msg.receiver\_id=p_2\rightarrow\\&\quad ffa\_msg\_send_2(s,cpu\_id)=s\end{aligned}$$

表示对任意不在访问控制矩阵中的消息通信调用, 必定无法修改系统状态. 即, 基于 IPC 访问控制矩阵的分区间通信接口可以防御未授权分区调用.

7.2 工作量评估

本文使用 Isabelle/HOL 实现 TrustZone 多安全分区规约和验证. 模型的信息流安全性是用 Isabelle 的结构化证明语言 Isar 证明的, 证明代码可同时被人类和计算机所理解, 证明推导通过 Isabelle 定理证明器完成. 如表 6 所示, 本文实现的形式化模型使用 137 个定义建立了 RMTEE 模型, 共 1 462 行 Isabelle 代码. 使用 201 个定理证明了模型的正确性和安全性, 共 9 715 行 Isar 代码. 整项工作花费了大约 18 人月.

表 6 RMTEE 形式化规约与验证代码统计

RMTEE 模型	规约		证明		合计(LOC)
	locale/definition	LOC	lemma/theorem	LOC	
抽象模型	14	108	12	117	11 177
具体模型	123	1 354	189	9 598	
合计	137	1 462	201	9 715	

具体来说, 使用 Isabelle 的参数理论 *locale* 定义了 RMTEE 抽象模型, 然后使用 *interpretation* 将参数模型实例化得到 RMTEE 具体模型, 并同时证明了实例化的正确性. 另外, 具体模型中的事件使用霍尔逻辑证明其正确性, 以半自动化的定理证明验证其满足机密性和完整性. 由于所有 Isabelle 代码都是机器可检查的, 可保证模型规约和证明的正确实现.

7.3 讨 论

本文基于精化方法建立了 TrustZone 多安全分区模型, 沿用本文的建模和验证框架, 可进一步将具体模型精化, 逐层建模和验证分区管理器的详细设计与实现. 对于不同的分区管理器设计和实现, 无需修改上层模型, 只要修改下层模型中的状态变量和事件规约, 并验证新引入的状态变量和事件的正确性和安全性. 因此, RMTEE 模型可以扩展到不同的分区管理器实现中. 对于其他架构, 本文的分层精化框架仍然适用, 但具体模型可能需要根据框架设计进行相应的修改. 以 ARMv9 机密计算架构为例, 其实现了一个运行机密虚拟机的执行环境, 对应到具体模型中, 就需要将执行模型的分区扩展为安全世界分区和机密虚拟机, 并遵循规范定义它们之间的干扰关系; 同时, 也需要依据机密计算架构所提供的接口修改事件规约和相关的正确性与安全性证明.

8 总结与展望

本文首次实现了对 ARM TrustZone 多安全分区架构的形式化建模和验证. 基于精化方法提出并构建了形式化模型 RMTEE, 包括抽象模型和具体模型, 并对分区生命周期管理、分区间隔离和分区间通信这 3 个模块的接口进行具体事件规约. 接着证明了具体模型对抽象模型精化关系以及事件规约中所有事件的正确性和安全性, 从而证明了该模型满足机密性和完整性. 最后, 本文通过威胁模型的两个模拟攻击对 RMTEE 模型进行了安全性分析, 对于任意地址映射, 可以通过阶段二页表映射的地址所有者检查限制此种非法操作; 对于未授权的分区间通信, 系统配置的 IPC 访问控制矩阵可以检查并拒绝未授权调用.

本文的具体模型包含了 FF-A 规范中的部分模块, 未来的工作进一步覆盖 FF-A 规范中的其他模块, 包括通知模块和中断管理模块等. 另外, 还将实现对具体分区管理器的详细设计以及代码实现的建模和验证. 有望将 RMTEE 模型扩展到 ARMv9 等其他架构下的多安全分区机制.

References:

[1] Ngabonziza B, Martin D, Bailey A, *et al*. TrustZone explained: Architectural features and use cases. In: Proc. of the 2nd Int'l Conf. on Collaboration and Internet Computing. Pittsburgh: IEEE, 2016. 445−451. [doi: 10.1109/CIC.2016.065]

[2] GlobalPlatform. Introduction to trusted execution environments. 2018. https://globalplatform.org/wp-content/uploads/2018/05/Introduction-to-Trusted-Execution-Environment-15May2018.pdf

[3] Pinto S, Santos N. Demystifying arm trustzone: A comprehensive survey. ACM Computing Surveys (CSUR), 2019, 51(6): 1−36. [doi: 10.1145/3291047]

[4] Cerdeira D, Santos N, Fonseca P, *et al*. Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems. In: Proc. of the 41st IEEE Symp. on Security and Privacy. San Francisco: IEEE, 2020. 1416−1432. [doi: 10.1109/SP40000.2020.00061]

[5] VulDB. NVIDIA jetson OTE protocol message parser memory leak. 2021. https://vuldb.com/?id.177340

[6] F-Secure. Security advisory: OP-TEE TrustZone bypass at wakeup on NXP i.MX6UL. 2021. https://github.com/f-secure-foundry/advisories/blob/master/Security_Advisory-Ref_FSC-HWSEC-VR2021-0002-OP-TEE_TrustZone_bypass_at_wakeup.txt

[7] ARM. Learn the architecture—AArch64 virtualization. https://developer.arm.com/documentation/102142/0100

[8] ARM. Arm firmware framework for arm a-profile. https://developer.arm.com/documentation/den0077

[9] VulDB. Qualcomm snapdragon auto TrustZone memory transfer interface information disclosure. 2021. https://vuldb.com/?id.189552

[10] Wirth N. Program development by stepwise refinement. Communications of the ACM, 1983, 26(1): 70−74. [doi: 10.1145/357980.358010]

[11] Back RJR, Sere K. Stepwise refinement of action systems. In: Proc. of the Int'l Conf. on Mathematics of Program Construction. Berlin, Heidelberg: Springer, 1989. 115−138. [doi: 10.1007/3-540-51305-1_7]

[12] Azab AM, Ning P, Shah J, *et al*. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In: Proc. of the 2014 ACM SIGSAC Conf. on Computer and Communications Security. New York: Association for Computing Machinery, 2014. 90−102. [doi: 10.1145/2660267.2660350]

[13] Li W, Xia Y, Lu L, *et al*. TEEv: Virtualizing trusted execution environments on mobile platforms. In: Proc. of the 15th ACM SIGPLAN/SIGOPS Int'l Conf. on Virtual Execution Environments. New York: Association for Computing Machinery, 2019. 2−16. [doi: 10.1145/3313808.3313810]

[14] Cerdeira D, Martins J, Santos N, *et al*. REZONE: Disarming TrustZone with TEE privilege reduction. In: Proc. of the 31st USENIX Security Symp. (USENIX Security 2022). Boston: USENIX Association, 2022. 2261−2279.

[15] Li D, Mi Z, Xia Y, *et al*. TwinVisor: Hardware-isolated confidential virtual machines for ARM. In: Proc. of the 28th SIGOPS Symp. on Operating Systems Principles. New York: Association for Computing Machinery, 2021. 638−654. [doi: 10.1145/3477132.3483554]

[16] ARM. Introducing arm confidential compute architecture. 2022. https://developer.arm.com/documentation/den0125/latest

[17] Mulligan DP, Petri G, Spinale N, *et al*. Confidential computing—A brave new world. In: Proc. of the 2021 Int'l Symp. on Secure and Private Execution Environment Design (SEED). Washington: IEEE, 2021. 132−138. [doi: 10.1109/ SEED51797.2021.00025]

[18] Li X, Li X, Dall C, *et al*. Design and verification of the arm confidential compute architecture. In: Proc. of the 16th USENIX Symp. on Operating Systems Design and Implementation (OSDI 2022). Carlsbad: USENIX Association, 2022. 465−484.

[19] Klein G, Elphinstone K, Heiser G, *et al*. seL4: Formal verification of an OS kernel. In: Proc. of the 22nd ACM SIGOPS Symp. on Operating Systems Principles. New York: Association for Computing Machinery, 2009. 207−220. [doi: 10.1145/1629575.1629596]

[20] 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 2016). Savannah: USENIX Association, 2016. 653−669.

[21] Zhao Y, Sanán D, Zhang F, *et al*. Reasoning about information flow security of separation kernels with channel-based communication. In: Proc. of the Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Berlin, Heidelberg: Springer, 2016. 791−810. [doi: 10.1007/978-3-662-49674-9_50]

[22] 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]

[23] Li S W, Li X, Gu R, *et al*. A secure and formally verified Linux KVM hypervisor. In: Proc. of the 2021 IEEE Symp. on Security and Privacy (SP). San Francisco: IEEE, 2021. 1782−1799. [doi: 10.1109/SP40001.2021.00049]

[24] Li S W, Li X, Gu R, *et al*. Formally verified memory protection for a commodity multiprocessor hypervisor. In: Proc. of the 30th USENIX Security Symp. (USENIX Security 2021). USENIX Association, 2021. 3953−3970.

[25] Ma Y, Zhang Q, Zhao S, *et al*. Formal verification of memory isolation for the trustzone-based TEE. In: Proc. of the 27th Asia-Pacific Software Engineering Conf. Singapore: IEEE, 2020. 149−158. [doi: 10.1109/APSEC51365.2020.00023]

[26] Jin CZ, Zhang QY, Ma YW, *et al*. Refinement-based verification of memory isolation mechanism for trusted execution environment. Ruan Jian Xue Bao/Journal of Software, 2022, 33(6): 2189−2207 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6577.htm [doi: 10.13328/j.cnki.jos.006577]

[27] Miao XL, Chang R, Pan SP, *et al*. Modeling and security analysis of access control in trusted execution environment. Ruan Jian Xue Bao/Journal of Software, 2023, 34(8): 3637−3658 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6612.htm [doi: 10.13328/j.cnki.jos.006612]

[28] Haigh JT, Young WD. Extending the noninterference version of MLS for SAT. IEEE Trans. on Software Engineering, 1987, (2): 141−150. [doi: 10.1109/TSE.1987.226478]

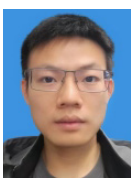
[29] Oheimb D. Information flow control revisited: Noninfluence=noninterference+nonleakage. In: Proc. of the European Symp. on Research in Computer Security. Berlin, Heidelberg: Springer, 2004. 225−243. [doi: 10.1007/978-3-540-30108-0_14]

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

附中文参考文献:

[26] 靳翠珍, 张倩颖, 马雨薇, 等. 基于精化的可信执行环境内存隔离机制验证. 软件学报, 2022, 33(6): 2189−2207. http://www.jos.org.cn/1000-9825/6577.htm [doi: 10.13328/j.cnki.jos.006577]

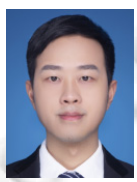
[27] 苗新亮, 常瑞, 潘少平, 等. 可信执行环境访问控制建模与安全性分析. 软件学报, 2023, 34(8): 3637−3658. http://www.jos.org.cn/1000-9825/6612.htm [doi: 10.13328/j.cnki.jos.006612]



曾凡浪(1994−), 男, 博士生, CCF 学生会员, 主要研究领域为可信执行环境, 形式化方法.

常瑞(1981−), 女, 博士, 副教授, 博士生导师, CCF 高级会员, 主要研究领域为硬件辅助安全, 形式化验证, 程序分析.

许浩(1999−), 男, 硕士生, 主要研究领域为可行执行环境, 形式化方法.



潘少平(1994−), 男, 硕士生, 主要研究领域为可信执行环境, 形式化验证.

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, 操作系统与安全, 编程语言与编译, 安全认证.