Hyperkernel:push-button verification of an OS kernel

本文介绍了一种设计，实现和形式化验证一个OS内核的功能正确性的方法--hyperkernel，高度证明自动化且证明负载低。在教学操作系统xv6上设计了hyperkernel的接口。hyperkernel引入了3个重要的idea来实现证明自动化：有限化内核接口来避免无限的循环和迭代。分离用户和内核地址空间简化对虚拟内存的reasoning. 在LLVM中间表示层实现了验证来避免建模复杂的C的语义。 使用Z3 SMT solver验证了hyperkernel的实现，检查了总共50个系统调用和trap handler.实验证明hyperkernel可以避免类似的bug，而且hyperkernel可以以较小的验证负载实现。

文章内容

介绍

内核实现了进程之间隔离，允许多个应用分享物理资源。内核中的bug可能导致程序的不正确执行以及恶意软件攻陷整个系统。 之前的工作有做形式化验证的，代价较高。 本文旨在提出方法和工具证明内核的正确性。不是使用交互式的理论证明器如Isabelle或Coq来手工写证明，我们重新设计了xv6内核接口来满足自动推理工具SMT解析器。验证hyperkernel一个关键挑战是接口设计，需要打破可用性和证明自动化来打破平衡。一方面，内核维护了一个丰富的数据结构和不变量来管理进程，虚拟内存和设备。Hyperkernel的接口需要支持specification and verification of high-level properties，这是证明正确性的基础。另一方面，这一接口必须实现使用SMT solver实现自动化验证。 第二个挑战：内核代码的虚拟内存管理。第三个挑战是：hyperkernel是C写的。不适合做形式化的推理。由于指针等很难准确建模C的语义。 hyperkernel使用3个核心idea来解决这些挑战：1.其内核接口设计成有限的。所有的针对系统调用的处理程序没有无限循环和递归，使其可能使用SMT进行编码和验证。2.hyperkernel运行在一个独立的地址空间，为了加好地实现隔离，hyperkernel使用了Intel的虚拟化技术VT-X，以及AMD-V。3.hyperkernel在LLVM中间层实现验证，与C相比有更简单的语义同时足够高层来避免对机器细节的reason. 对hyperkernle的接口进行了验证：1.we have developed a specification of trap handlers in a state-machine style，描述其行为。2.为了提高状态机描述行为的正确性的可信性，进一步开发了声明式的高层specification.其描述了状态机必须满足的端到端之间的必须满足的specification。

2大贡献：按键式的方法来构建验证的OS内核，以及可用于SMT solving的内核接口设计。内核接口的仔细设计是实现验证自动化的主要因素。直接将hyperkernel的方法用在已有的OS内核上不一定能够成功。

overview

这部分介绍了hyperkernel开发工作流程。设计->specification->以及对一个系统调用的验证。 首先使用状态机式的说明和声明式语言的说明。C实现系统调用；验证器将说明和实现减少到一个SMT查询，调用Z3来实现验证。验证代码与信任代码连接在一起形成最终的kernel image 使用SMT的一个优点是便于找到错误。 有

2个假设：内核运行在单核系统上，并且关闭中断。确保每个系统调用的原子性。内核与用户空间在隔离的地址空间，对虚拟内存使用identity映射。 有限接口：我们使内核接口有限，通过确保每个trap handler的语义可以表示为一系列有限长度的traces.有限接口有2个主要的好处：尽管使用SMT验证POSIX DUP也是可能的，这对于较大参数的表来说扩展性不好；没有限制内核状态的大小；

规范

给定一个有限的接口，程序员描述内核想要实现的行为，这是通过实现有限状态机来完成的。规范包含2部分：抽象内核状态的定义；trap handler的定义；程序员也可以提供可选的声明式的规范。然后hyperkernel的验证器将会检查这些特性是否满足。程序员使用固定宽度的整数和映射来定义抽象内核状态。具体的定义方法文中给出了范例。对于状态迁移规范：大多系统调用的规范遵循以下常见的模式：验证系统调用参数，如果验证通过了将内核传到下一状态，否则，系统调用返回错误代码并且内核状态不发生变化。每个系统调用规范提供了验证条件和新状态。声明式规范：状态机规范是抽象的，没有未定义的行为，也不引入具体的实现细节。为了提高其正确性的可信度，我们也开发了一个高层的声明式规范，来更好地获取内核行为。声明式的规范获得编程者的意图，确保状态机规范以及实现满足想要的切边属性。

实现

dup(oldfd,newfd)在hyperkernel中的C实现与XV6和Unix V6很像。关键的区别是:不是搜索未使用的FD，代码只检查是否一个给定的newfd未被使用。 与之前的内核验证项目不同，我们对C的使用的限制更少，因为验证是在LLVM的中间表示进行。 表示不变量：内核显式地检查来自用户空间的数值的有效性，因为这些数据是不被信任的。内核内的数值通常被认为是有效的。检查有2种：动态检查和静态检查。

验证

hyperkernel的验证证明了2个主要的理论： 定理1（refinement）：内核实现是对状态机规范的一种精炼。定理2（横切）：状态机规范满足声明式规范。 对定理1的证明：需要程序员写一个等价函数来建立LLVM IR中的内核数据结构和抽象内核状态的相关性。使用等价函数，验证器可以证明定理1.将状态机规范和LLVM IR实现转化成SMT。 定理2的证明：验证器将状态机规范和声明规范转换成SMT，检查声明式规范在每次状态转换后依然成立。

测试生成：验证器在证明定理发生失败后发现两类错误：实现上的bug和状态机规范的bug.在这2种情况下，验证器会生成具体的测试用例。如果Z3不能找到任何的反例。定理1保证了内核实现的被验证部分没有低级bug，同时进一步保证功能正确性。定理2保证了状态机规范相对于声明式规范是正确的。文章通过使用dup系统调用的例子说明了如何设计有限接口。证明其正确性的负载较低。可信计算基（trusted computing base）包括规范，定理（包括等价函数），内核初始化和glue code,验证器，以及依赖的验证工具链。同时假设了硬件的正确性。

验证器

为了证明hyperkernel的2个正确性定理，验证器编码内核实现为SMT用于自动验证。为了使验证可扩展，验证器限制了SMT的使用。

建模内核行为

将内核行为建模为状态机有一整套标准的方法。hyperkernel中的trap handler是原子的。如何确保trap handler的原子性：单核；关闭中断；限制DMA到专用的内存区域。在该模型的基础上，做了一些形式化的工作：在此基础上提出了2个定义：定义1:规范-实现精炼：定义2：状态机规范正确性。为了证明上述的介绍的定理：验证器计算对应的抽象对象的SMT编码，然后调用Z3进行验证。验证器假设内核初始化的正确性，将这一部分的正确性交给了boot checker.

对LLVM IR的推理

LLVM IR是一种广泛用于构建编译器和找bug的代码表示。选择它作为验证目标的2个原因：1.相比于C来说其语义较为简单；2.相比于x86汇编，其保留了高层信息，如类型等。为了对每个trap handler构建SMT表达，验证器在其LLVM IR上实现符号执行。符号执行由2步组成：预先检查来排除在LLVM IR中未定义的行为；映射LLVM IR到SMT

编码crosscutting属性

为了实现可扩展的验证，验证器限制了SMT的使用到有效的一阶逻辑。LLVM IR和状态机规范生成的编码大量的由无量词规则生成。编码方式的意外情况是在高层使用量词。使用量词来阐明关于资源管理的2个常见的模式。资源只由一个object拥有和共享资源一直是引用计数的。编码量词规则会造成解析器遍历搜索空间以至于不能在给定的时间内完成任务。接下来描述扩展性好的SMT编码方式。详见文章。

THE hyperkernel

本部分介绍如何按照上述的一些规则来实现一个便于自动化验证的OS内核。 为了使内核接口有限，hyperkernel组合了来自Dune，exokernel and seL4的一些设计想法。分别有以下几个方面：通过硬件虚拟化的进程；显式资源管理；类型页；在设计有限接口方面：这种设计导致了几种常见的设计模式：通过引用计数来实现资源生命周期。对于任何进程P，其引用计数值nr_children的数量和将p作为父进程的数量相同。如果一个进程P被标记为free，没有进程将P指定为其父进程。强制细粒度保护：有一些posix系统调用有复杂的语义。形式化地阐明这些系统调用的行为是不容易的。Hyperkernel提供了一种外核风格的原语系统调用的进程创建。该原语系统调用较容易进行specify，实现和验证。一个进程P的页表根的页是只被P拥有的。类似的，hyperkernel向外暴露了细粒度的虚拟内存管理。验证链接数据结构的有效性；

用户空间的库包括：bootstrapping，文件系统，网络和Linux用户仿真。

限制：硬件虚拟化技术的使用简化了隔离用户和内核空间的验证。hyperkernel的数据结构都是专门为SMT而设计的。需要有限接口。工作在LLVM IR之上；内生继承了xv6的一些缺点。

检查器：hyperkernel定理保证了trap handlers的正确性。验证代码组件需要构建机器级别的规范，这项工作的复杂性较高。解决方法：boot checker：stack checker;link checker.

实验

这部分介绍hyperkernel的开发以及验证以及运行时的性能。

bug讨论

为了验证验证器和检查器在预防bug方面的有效性，检查xv6的git commit记录。作者手工分析了这些bug判断这些bug可不可能出现在hyperkernel中。声明式规范表明在内核接口的设计过程中暴露边角情况尤其有用。高层的声明式的规范捕捉到了跨系统多个部分的属性。

开发工作

开发工作耗费了一个研究者将近一年的时间。如下几个地方是时间花费较多的地方：系统设计，验证和调试；SMT编码

验证性能

使用Z3进行验证hyperkernel的2个定理时，在八核的机器上大概消耗15分钟。性能较为稳定，验证时间较为稳定。

运行时性能

为了评估hyperkernel的运行时性能，采用Dune中的benchmark,排除不适用于hyperkernel的内容。比较了hyperkernle和Linux在各方面的比较。实验结果：对于最坏的情况，hypercall的使用导致了5倍的负载，相比于syscall.在工作负载不同时，应用性能有可能从虚拟化获益。

对于硬件支持的想法

hyperkernel中虚拟化技术的使用简化了验证，但是使用hypercall代替了系统调用。尽管在X86上的hypercall要慢一个数量级，最近几年他们的性能已经得到了较大的提升。对于非X86架构啊，hypercall和系统调用的开销相近。

结论

hyperkernel是一个实现高度验证自动化的OS内核，并且验证负载小。通过有限化接口，硬件虚拟化来简化分析；hyperkenel可以防止很多的Bug，其为未来可验证的OS内核的开发设计提供了指导方向。

一些思考

验证OS内核的正确性和安全性，这一工作并不能直接用在现有的一些OS内核上，为了使OS便于进行自动化的验证，需要对OS进行适当的改变。尽管如此本文的工作，无论是从想法和工作量上看都是不错的。