**我们能够使操作系统更安全更可靠吗?**

**微内核-----长久被遗弃而不被接受是因为与单块集成电路内核相比它们的拙劣表现。而后者则很可能在操作系统领域创造一番业绩。是因为他们潜在的高可靠性，而这点正是许多研究者认为比表现更为重要的东西。**

上次你的电视机坏了，迫使你从网站上下载一些紧急更新软件，也许不知道是什么时候了？然而，除非它是一次事故，它是一个带有一个CPU，一个大的显示器，一些相似的用于解码的电子元件，一对特殊的I/O装置------一个远程的控制器，一个内置的VCR或者DVD装置------以及一个在ROM中的启动程序。

这些修辞的问题指出了一个极粗俗的秘密，那就是在计算机领域，我们并不喜欢讨论：为什么电视装置、DVD录音机、MP3播放器、电池电话、以及其它一些软件支撑的电子装置是可信和安全，而计算机则没有呢？当然，有许多原因------比如说计算机是柔性的，用户可以更换软件，比如说IT产业是不成熟的等等。但是，当我们前进到一个时代，那时大量的计算机用户是非计算机专业用户，相应的这些借口对他们来说是站不住脚的。

消费者从计算机希望得到的和他们希望从电视装置得到的是相同的：你购买，安装，它工作起来，在接下来的10年中它工作良好。在IT领域，我们需要应对这种挑战并且使计算机能够象电视机一样可靠和安全。

当我们谈到安全和可靠性方面的时候，最大的冒犯者就是操作系统。虽然应用程序包含有许多的缺陷，若操作系统没有错误，则在应用程序中的错误仅能够带来有限的破坏，因此我们主要把焦点集中在操作系统方面。

然而，在我们详细讨论细节之前，关于可靠性与安全性之间关系的一些术语是明确的。这些领域的问题经常有着相同的根源：软件中的错误。一个缓冲器错误能够使一个系统毁坏（可靠性问题），但是它也能够允许一个狡猾的写入病毒或者蠕虫来接管计算机（安全性问题）。尽管我们把焦点主要集中在可靠性方面，提高可靠性同样能够提高安全性。

**为什么系统不可靠？**

当前的操作系统有两个特征使系统不可靠和不安全：他们非常巨大并且他们有非常有限的错误隔离措施。Linux内核有超过250万行的代码；而Windows XP内核则是Ｌinux两倍多。

一项关于软件可靠性方面的调查显示每1000行可执行代码中就有6---16个错误。（1）而另一项研究显示每1000行可执行代码中的错误密度为2---75个错误，（2）取决于模块大小。用一个相对保守的估计方法每1000行代脉中有6个错误，则Linux内核大概有15000个错误，WindowsXP则是Linux的双倍。

事情更糟糕的是，典型的，大约70 %的的操作系统包含有驱动程序，而其中含有的错误大概是普通代码的3—7倍，（3）上面引述的错误数很可能是一个巨大的错误估计。很显然，发现和修正所有的错误是不可行的，而且，错误修正经常会引入新的错误。

当前操作系统的巨大体积意味着没有一个人可以理解整个事情。很显然，当没有人真正理解它时，把一个操作系统工程做好则变的非常困难。

这促使我们转向第二个方面：错误隔离。没有一个单独的人理解航空器运载器是怎么工作的，但是航空器运载器的子系统却能够很好的分离。一个带有木质的盥洗器并不能影响导弹发射子系统。操作系统在各个组件之间并不具备这种分离。一个现代的操作系统包含成百上千个

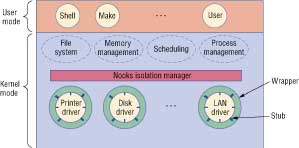
连接在一起的过程，组成为一个单一的二进制程序而运行在内核模式。这些百万行的内核代码中的每一个能够覆盖不相关组建使用的关键的数据结构，使用一种难以检测到的方法来破坏系统。此外，如果一个病毒或者蠕虫感染了一个内核程序，我们是没有办法快速的避免十它感染其他程序，从而阻止它控制整个机器。

回顾我们的轮船体系，现代轮船在船体内有许许多多的部件，如果一个部件有一个漏洞，仅仅是那部分漏水，而不是整个部分漏水。在划分发明之前，当前的操作系统和船的道理非常相似：一个部分漏水足以使整个船沉没。

幸运的是：形势并不是没有希望的。研究者正致力于研究产生更可靠的操作系统。这里，我们列举4个不同的方法，这正是研究者们用来使未来的操作系统更可靠和更安全的方法，下面我们从最次要的到最根本的方法逐一介绍：

**ARMORED 操作系统**

最保守的方法，Nooks，是设计来提高已存在系统比如说Windows和Linux的可靠性的。Nooks，保持了单块集成电路内核结构，在单一的内核模式空间中带有成千上万个连接在一起的过程，但是，它集中在制造驱动程序---问题的核心----更少的危险性。



***Figure 1. The Nooks model. Each driver is wrapped in a layer of protective software that monitors all interactions between the driver and the kernel.***

尤其，如图1显示，Nooks保护内核以避免出错装置驱动通过包裹起每一层中的保护软件来形成一个轻保护领域，一种叫做沙箱的保护技术。围绕在每个驱动程序周围的这个包裹装置监控着所有的在驱动和内核之间的交互作用。这种技术也能够用来对于内核的其他扩展部分也适用，比如装入文件系统。但是为简单起见，我们指的是驱动。

Nook工程的目标如下：

保护内核以防止驱动失败,

当一个驱动失败时能够自动恢复,

带有变化的，尽可能的对存在的驱动和内核做所有的事

保护内核反对恶意攻击并不是目标。最开始的执行是在Linux上实现的，但是这个观点对其它的也同样适用。

**分离**

用来防止错误的驱动程序破坏内核数据结构的主要工具是虚拟内存页面印象。当一个驱动运行时，所有的它外部的页面都转变为只读，从而实现了为每个驱动设置一个分离的轻保护措施。通过这种方式，这个驱动能够读出它所需要的内核数据结构，但是，任何试图直接修改内核数据结构的方式将会导致一个CPU异常的产生，该异常能够被Nook分离管理器捕获。而驱动程序的私有内存，它存储栈，堆，私有数据结构，以及内核对象的拷贝本都是可读写的。

**插入**

每一个驱动类输出一系列内核能够调用的函数。比如，声卡驱动程序能够提供一个调用来把一块声音样例写到卡中，另外一个则来调整声音等等。当驱动程序装载进时，一系列指向这个函数的指针就填充进来，因此，内核程序能够找到每一个程序。此外，驱动程序导入有由内核提供的一系列函数，例如，分配一个数据缓冲区。

Nooks为导入的和导出的函数都提供了包裹器。当内核调用一个驱动或者驱动调用一个内核程序,这个调用实际上进入一个包裹器去检查参量有效性以及控制这个调用。当封皮残余部分被显示在表1 当线黏附入和在外面司机自动地引起从他们的作用原型, 开发商必须handwrite 封皮身体。总计, 角落队写了455 个封皮: 329 为仁出口的作用并且126 为设备驱动程序出口的作用。当包裹器剩下部分---图1中嵌入的行数从函数原形中自动产生，开发者必须手写这个包裹器主题部分。总之，Nook团队写了455个包裹器，329个用于内核导出函数，126个用于驱动导出函数。

当一个驱动试图修改一个内核对象，它的包裹器复制对象入驱动的保护领域，即，它的私有读写页。驱动然后修改拷贝。一旦请求成功完成，分离管理器复制本修改内核对象到内核。通过这种方式，在一个调用中发生的驱动毁坏或失败总是以一种无效的状态离开内核对象。对导入对象跟踪路径是对象细节，因此，Nook团队不得不手写代码以跟踪Linux驱动所使用的43个对象类。

**恢复**

失败以后，用户模式恢复机构运行并咨询配置数据库看做什么。在许多情况下，释放许多资源并重新启动这个驱动是足够的，因为，大多数普通算法错误通常在测试中被发现，而留下许多耗时的和特殊的错误。这个技术能够恢复系统，但运行的应用可能失败。除了另外的工作，Nooks团队把阴影驱动这个概念加进来以便允许一个驱动失败以后应用继续运行。

简而言之，在正常运行期间，一个阴影驱动记录在每个驱动和内核之间的通信情况，如果它将作为需要的补救。在一个驱动开始后，这个阴影驱动从日志供养这个新启动的驱动，比如，重复这个I/O控制系统调用以设置参数，如音量。内核没有意识到把新驱动返回到老状态所处的相同状态的过程。一旦这个过程完成，驱动开始处理新的请求。

**局限性**

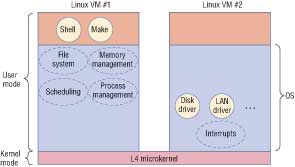
当实验显示，Nooks能够捕获99%的致命的驱动错误和55%的非致命错误，它并不是完美的。比如说，驱动能够执行他们不应该执行的优先的指令，他们能够写不正确的I/O端口，他们能够无限次的循环。而且，Nooks团队不得不用手写大量的包裹器，它们也可能包含错误。最后，驱动并不能从能够写入所有的内存被阻止。然而，这正是一种朝改进遗留内核可靠性的有用的方法。

**PARAVIRTUAL MACHINES**

第二种方法有它的根在虚拟机概念, 而这可以追回到60年代后期。(6) 简而言之, 想法是在光秃的硬件上运行一个特别控制程序, 称为一台虚拟机显示器,代替一个操作系统。虚拟机创造真实的机器的多个事例。各个事例可能运行任一软件，而裸机也可以。

这个技术是常用来允许二个或更多操作系统、比如Linux 和Windows, 同时跑在同样的硬件上，而每一个都认为它完全拥有整个机器对本身。虚拟机的用途对错误分离有一个很好的声誉。毕竟，如果虚拟机不知道其它的部分，一个机器里的问题是无法传播对其它地方的。

这里的研究将这种概念运用于保护在一个唯一操作系统之内, 而不是在不同的操作系统之间。(7) 此外, 因为奔腾不是充分地虚拟化的， 在虚拟机状态下，运行一个非限定的操作系统想法被做了让步。这种让步允许对操作系统进行改动以确定它不做无法被虚拟化的东西。与真实的虚拟化相比较, 这个技术叫做paravirtualization。



***Figure 2. Virtual machines. One of the virtual Linux machines runs the application programs while one or more other machines run the device drivers.***

具体地, 在90 年代, 一个研究小组在卡尔斯鲁厄大学建立了L4 微内核(8) 他们能够在L4的顶部运行修改过的版本Linux（L4Linux），而它能被描述作为一个虚拟机。(9) 研究员以后意识到, 而不是在L4上只运行Linux 的一个拷贝, 他们能运行多个拷贝。作为图2 显示, 这种观察导致了我们有这个想法：即真正Linux 机器的当中一个运行应用程序，而一个或更多其它机器跑设备驱动程序。

通过把设备驱动程序放置在一个或更多虚拟机里，从主要虚拟机分离出来，主虚拟机运行操作系统和应用程序的剩余部分。 如果设备驱动程序毁坏, 只有它的虚拟机下来, 而主机则不会停下来。这种方法另外的好处是, 设备驱动程序不必须被修改，当他们看一个正常Linux 环境。当然, Linux内核必须被修改达到paravirtualization, 但这是一次性变动, 并且它不是必要重覆它为各个设备驱动程序。

既然驱动程序运行在硬件的用户方式下，一个主要问题是它们怎么执行I/O和处理中断。物理I/O的处理是通过加入3000行的代码到Linux内核中，驱动程序运行以允许它们使用L4服务以应对I/O，代替自己来做它。另外的5000行代码处理被隔离的驱动盘、网络和PCI总线之间的通讯。虚拟机运行应用程序。

原则上, 这种方法比一个唯一操作系统应该提供更大的可靠性，因为当一台虚拟机包含一个或更多驱动毁坏时, 虚拟机能够被重新起动并且驱动能够返回到他们的初始状态。而在Nooks中没有采取任何试图回到以前未毁坏的状态的措施。 和在角落。这样，如果一个声卡驱动程序坏了，它将被恢复到默认的初始声音设置状态，而不是把它调整到比毁坏更优的状态。

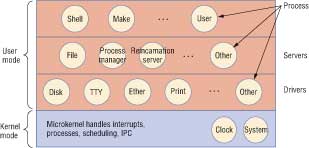
表现测量显示时下使用paravirtualized machines大约占3%---8%。

**MULTISERVER OPERATING SYSTEMS**

上述的两个操作系统主要集中在修补以前的操作系统。下面的两个主要集中在未来的系统上面。

这些方法的之一直接触及问题的核心：使整个操作系统当作一个巨大的二进制程序来运行在内核模式。反而，仅有微小的微内核运行在内核模式，而操作系统的其余的部分则做为完全分离的用户模式服务和驱动进程的集合而运行。

这样的想法已经存在大约有20年了，但是，一开始它并没有被完全探索出来，因为它的表现比一个单板内核差。在20世纪80年代，工作表现代表了一切，而安全性和可靠性并没有提上议事日程。当然，在那个时候，航空工程师没有担心每加仑能行使多少英里或者驾驶员坐舱们对承受武装攻击的能力。时代变了，人们认为什么是重要的想法也改变了。

****

***Figure 3. The Minix 3 architecture. The microkernel handles interrupts, provides the basic mechanisms for process management, implements interprocess communication, and performs process scheduling.***

**Multiserver architecture**

对现代的例子做一个回顾有助于我们对多服务的操作系统的概念有个更好的理解。如图3所示，在Minix3中，微内核处理中断，提供了基本的进程管理的机制，实现了跨进程的通信。实现进程安排。它同时提供一些小的内核调用以授权驱动和服务程序，比如说，读一些选择的用户地址空间或者写一些授权的I/O端口。这个时钟驱动共享微内核的地址空间，但是，它是作为一个分离的进程而分配的。其它的驱动程序不能运行在内核模式。

在微内核之上的是设备驱动层。（10）每一个I/O都有自己的驱动，它们作为一个分开的进程独自运行在自己的私有地址空间中，被内存管理单元(MMU)硬件保护着。这层包括为磁盘的驱动进程，中端（键盘和显示器），以太网，打印机，音响等等。驱动运行在用户模式，不能执行有权限的指令，不能读或写计算机I/O端口。它们必须能产生内核调用以得到这些服务。通过引进一些少量的上层程序，这种设计也能够提高可靠性。

在设备驱动层之上是服务层。文件服务（约4500行可执行代码）是一个小程序，它接受用户进程为Posix系统调用的请求以及与此相关的文件操作，比如说读，写，定位，状态及实现。并且，进程管理也在这一层实现，它处理进程和内存管理，实现Posix和其它系统调用，如fork,etec,以及brk。

一个有些异常的特点是再生服务器，它是所有其它服务和其他驱动的父进程。如果一个驱动或者服务毁坏了，退出了，或者对其他定期的pings无法响应了，再生服务器在有必要时把它消灭掉并且从磁盘或者RAM中重新启动一个copy。驱动可以通过这个方式来重新启动，但是在当前，只有那些不含有内部状态的服务程序能够启动。

其他服务包括网络服务，含有一个完整的TCP/IP栈，数据存储是其他服务使用的一个简单的名字服务和信息服务，有助于调试。

最后，位于服务层之上的是用户进程。位于这个和其他Unix系统之间唯一的区别是库程序是用来读，写而其他系统调用是通过发送消息来工作的。除此之外的另一个差别，隐藏在系统库中，他们是普通的用户进程，能有用于Posix API。

**Interprocess communication**

由于它允许所有进程合作,在一个multiserver 操作系统中跨进程通信是非常重要的。但是, 由于所有服务和驱动在Minix 3 上作为独立的进程而运行的, 他们无法直接地调用其他的函数和共享数据结构。 相反，Minix3通过传递固定长度的消息来执行IPC，使用了集合原则：当发送和接收都已经准备好了的时候，系统直接把消息从发送端拷贝到接收端。此外，一个异步事件通知机制是可用的。那些不可传送的事件加上标记，标记为在进程表中加一个图标。

Minix3集成了中断，带有消息传送系统。中断处理程序使用这个通知机制来标志I/O的完成。这种机制允许一个处理程序在驱动的设置一个位，标明是“迫近的中断”，并且不带阻塞的继续。当驱动准备好接收中断的时候，内核把它转变成一个正常的消息。

**Reliability features**

Minix 3's 可靠性来自多个来源头。首先, 只有大约4,000 个代码行运行在内核, 因此以每1000行六个错误的一个保守估计, 内核中的总的错误的估计数大概有15,000 个，这是对于Linux估计，而Windows操作系统则更多。既然除了闹钟，其他所有的驱动程序都是用户进程的，没有其他外部的代码运行在内核模式。内核的小容量同样能够使它验证代码变的更实际，无论是通过手工或者正式的技术。

Minix 3's IPC 设计不要求消息排队或缓冲, 这就消除了在内核对缓冲区管理的需要。此外, 因为IPC 具有强有力的构造能力, 各个服务器和驱动的的IPC 能力紧紧被限制。对于各个过程来讲, 可利用的IPC , 允许的目标, 和用户事件通知等被限制了。用户进程，比如，能够使用仅有的集合原则，并且能够发送出仅有的Posix服务。

另外，所有的内核数据都是静态的。所有的这些特点大大的简化了代码并且消除了与缓冲运行、内存泄露、不合适宜的中断、未信任的内核代码等等有关的内核错误。当然，移动大多数操作系统到用户模式并不能消除驱动和服务中不可避免的错误，它使它们力量更弱。一个内核错误能毁坏重要的数据结构，写垃圾文件到磁盘等等；驱动和服务中的一个错误并不能带来更多错误，是因为这些就进程并划分了，而且它们在它所能做的范围内被限制了。

用户模式下的驱动和服务并不作为超级拥用户来运行。它们并不能访问它们地址空间之外的内存，除非能够产生内核调用（这需要内核检查有效性）。更强的是，内核进程表内的位图和乏味控制一组被允许的内核调用，IPC能力以及一个继承基础上的I/O端口。例如，内核能够防止打印驱动写用户地址空间，防止触动磁盘I/O端口或者发送消息到自动驱动。在传统的单块集成电路系统中，驱动能做所有的事情。

另一个可靠性的特征是分离指令和数据空间的使用。一个臭虫或者病毒难道真的能破坏驱动或者服务缓冲，并且把外部代码放置到数据空间中吗？注入的代码并不能通过跳转或者用一个过程回调来执行它，因为内核不会执行代码除非它是在只读空间中。

在其它的旨在提高可靠性的具体措施之中，最关键的是自治疗特征。如果一个驱动通过一个无效的指针做了一个存储，进入一个无限循环，或者误操作。这个再生服务程序会自动替换它，而不会影响其它在运行的进程。

然而重新启动一个逻辑上不正确的驱动并不会去除臭虫，实际上，微妙的时间和相似的臭虫能造成许多问题，重新启动驱动会修复系统。此外，这种机制允许从失败中恢复，而这些错误是由被攻击引起的，比如“ping死亡”，它能通过发送一个不正确的复制的IP包来毁坏一个计算机系统。

**Performance considerations**

数十年来，研究员们由于集中的表现问题批评了基于微内核的多服务架构。然而，各种各样的工程已经证明，模块设计实际上能提供竞争表现。尽管Minix3并没有由于表现而被优化，系统是合理的快速的。用户模式下驱动损失的表现与内核驱动相比小于10%。在少于6秒

，并且是在2.2千兆赫Athlon处理器 上，系统能够建立自己，包括内核，普通驱动以及其他所有服务（112个编辑和11个链接）。

多服务架构，以一点点表现为代价，使提供一个高可靠性的，象UNIX的环境成为可能，这样一个事实使之更实际。为奔腾的Minix3可以免费下载，网址：<http://www.minix3.org/>。端口对其它架构以及对嵌入式系统正在发展中。

**LANGUAGE-BASED PROTECTION**

最根本的方法是来自一个意想不到的来源----微软研究员。实际上，微软方法抛弃了这样一种概念，即操作系统是运行在内核模式的一个单一程序加上运行在用户模式的一些用户进程的集合，取而代之的是用一个由新的类型安全语言写成的系统。而这样的语言并没有象c/c++的指针或者其他等等诸如此类的一些问题。就象前面的两种方法，这个方法已经存在了数十年了。

Burroughs B5000计算机使用了这种方法。唯一可利用的语言是Algol。保护措施并不是由MMU来处理的，机器没有MMU，而是由Algol的编译器来实施的，因为它拒绝产生危险代码。微软的研究方法为21世纪更新了这个观点。

**Overview**

这个系统，称为Singularity，基本上是由Sing#语言，这样一种新型的安全语言写成的。这个语言是基于C#，但是改进了消息传递方式。其语义学是由正式的书面合同定义的。因为语言安全紧紧的和把系统和用户进程联系在一起，所有的进程能够运行在单一的虚拟地址空间中。这种设计产生了安全（因为编译器不允许一个进程去触及另一个进程的数据）以及效率（因为它消除了内核陷阱和上下文转换）

此外，Singularity设计是弹性的，因为每一个进程是一个封闭的实体，因而有自己的代码，数据结构，内存布局，运行时系统，库，垃圾回收。MMU使能，但只映射页而不是为每一个进程建立一个分开的保护领域。

一个关键的Singularity设计原则是它禁止动态进程扩展。在其它后果中，这个设计不允许载入的模块，比如装置驱动和内置浏览器，因为它们引入了未验证的外部代码，这些代码能够毁坏母进程。相反的，这些扩展必须作为一个分离的进程而运行，完全被隔离起来并且由标准IPC机制通讯。

**The microkernel**

Singularity操作系统包括一个微内核进程和一组用户进程，都典型的运行在普通的虚拟地址空间中。微内核控制对硬件的通入，分配和取消内存，创造、销毁、分配线程；处理带有信号量的线程同步问题；处理带有频率的跨进程同步问题；监督I/O。每一个驱动装置做为一个分离进程而运行。

虽然大多数微内核是由Sing#写成的，一小部分端口是由C#、C++、或者汇编写成的，它们必须得到信任因为它无法被验证。这些被信任的代码包括硬件抽象层和垃圾回收器。硬件抽象层从系统中隐藏了较低的硬件。它通过隐藏诸如I/O端口、中断请求线、直接存储器访问、时钟来提供机器独立抽象于操作系统的其他部分。

**Interprocess communication**

用户进程采用点对点通过发送强烈的类型信号到微内核获得系统服务。实际上，所有的进程对进程的通信使用这个通道。不象其他的信息传递系统，在一些库里有发送和接收两个函数,Sing#在语言上完全支持这个通道，包括复制类型和协议规格。

为了使这一点更清晰，考虑下面的通信的例子：

contract C1 {

in message Request(int x) requires x > 0;

out message Reply(int y);

out message Error();

state Start:

Request? -> Pending;

state Pending: one {

Reply! -> Start;

Error! -> Stopped;

}

state Stopped: ;

}

这个例子声明这个通道接收三个消息：请求，接收，错误，第一个是正整形变量作为参数，第二个是任意整形变量作为参数，第三个是无参数。当用做为连接到服务器的一个通道时，请求信息从客户到服务器，其他两条消息走另一个通道。系统指定协议为渠道。

在启动状态，客户发送请求信息，把通道投入到即将发生的状态。服务器可能或者回复一个应答信息或者回复一个错误信息。应答信息使通道重回到初始启动状态，通信可以继续进行。错误信息使通道回到停止状态，终止这个通信。

**The heap**

如果所有数据，比如从磁盘读出的文件块，必须经过通道，系统会是非常慢的，因此，一个异常必须被加入到基本规则中去，即每一个进程数据是完全私有的和只对自己开放。Singularity支持一个共享的对象堆，但是在一个瞬时，堆里的每一个对象属于一个单一的进程。但是，对象的归属可能通过一个渠道传递。

作为一个对怎样工作的一个例子，我们考虑I/O。当磁盘驱动程序读一个块时，它把块放在一个堆上。以后，系统为块传递柄到用户，请求数据，保持唯一拥有原则，但是允许数据从磁盘通过零拷贝方式移动到用户。

**The file system**

**Singularity为所有服务保持有一个唯一的等级名字空间。跟名服务器处理树的上面，但其它名服务程序可能登上它的结点。特别是，文件系统，是个过程，登上fs，因此名字如/fs/users/Linda/foo可能就是用户的文件。文件被象按照B树来分类，用块号作为关键字。每当用户进程请求文件时，文件系统命令磁盘驱动把请求块置在堆上。归属就如所描述的传递了。**

**Verification**

**每个系统组建都有元数据描述它的独立性、输出、资源和行为。这些元数据是用来验证的。系统图象包括微内核、驱动和运行系统必要的应用程序以及他们的元数据。外在的检验器在系统执行它之前可以执行许多对图象的检查，比如说确信驱动没有资源冲突。**

**验证是一个三步过程：**

**编译器检查类型安全、对象归属、通道协议等等。**

**编译器产生微软中间语言，一个便携式象JVM的字节码，以便检查器检查。**

**MSIL通过一个后端编译器被编译成×86代码，能插入执行时代码检查（当前的编译器不能做这个）。**

**丰富的验证点的要点是在验证器里捕获错误。**

**这四个不同的试图都提高操作系统的可靠性的方法集中在阻止错误装置驱动毁坏系统。**

**在Nooks方法中，每个驱动都是单独的被手包裹在一个软件包中，以控制它和操作系统的其它部分交互。但是它把所有的驱动都遗留在内核中了。Paravirtual machine方法把这一步走的更远，它移动驱动到一个或更多的机器而不是主要的那个，从驱动拿走了更多的力量。这两种方法都是旨在改进现有的操作系统。**

**相反，其他两种方法用更加可靠和更加安全的方法替换那些遗产操作系统。Multiserver方法在一个分离的用户空间运行每个驱动和操作系统组建并且允许他们用微内核的IPC机制来运行。最后，Singularity，这个最根本的方法，使用一个类型安全语言，一个单一的地址空间，形成契约来仔细的限制每个模块所能做的。**

**四项研究中的三个---L4基于paravirtualization,Minix3和Singularity使用微内核。它不为人所知，不管怎样，从长远来看，这些方法将被广泛采用。但是，很有趣的注意到，微内核长久的被遗弃而不被接受是因为**与单块集成电路内核相比**它们的低劣的表现。**而后者则很可能在操作系统领域创造一番业绩。是因为他们潜在的高可靠性，而这点正是人们认为比表现更为重要的东西。再生的车轮翻转了。

**Acknowledgments**

**我们感谢Brian Bershad,Galen Hunt，以及Michael Swift，他们该了很好的饿评论和建议，这部分工作也得到了荷兰科学研究组织的大力支持，表示感谢。**

**References**

1. V.R. Basili and B.T. Perricone, "Software Errors and Complexity: An Empirical Investigation," *Comm. ACM*, Jan. 1984, pp. 42-52.
2. T.J. Ostrand and E.J. Weyuker, The Distribution of Faults in a Large Industrial Software System, *Proc. Int'l Symp. Software Testing and Analysis*, ACM Press, 2002, pp. 55-64.
3. A. Chou et al., "An Empirical Study of Operating System Errors," *Proc. 18th ACM Symp. Operating System Principles*, ACM Press, 2001, pp. 73-88.
4. M. Swift, B. Bershad, and H. Levy, " Improving the Reliability of Commodity Operating Systems," *ACM Trans. Computer Systems*, vol. 23, 2005, pp. 77-110.
5. M. Swift et al., "Recovering Device Drivers," *Proc. 6th Symp. Operating System Design and Implementation*, ACM Press, 2003, pp. 1-16.
6. R.P. Goldberg, "Architecture of Virtual Machines," *Proc. Workshop Virtual Computer Systems*, ACM Press, 1973, pp. 74-112.
7. J. LeVasseur et al., "Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines," *Proc. 6th Symp. Operating System Design and Implementation*, 2004, pp. 17-30.
8. J. Liedtke, "On Microkernel Construction," *Proc. 15th ACM Symp. Operating System Principles*, ACM Press, 1995, pp. 237-250.
9. H. Hartig et al., "The Performance of Microkernel-Based Systems," *Proc. 16th ACM Symp. Operating System Principles*, ACM Press, 1997, pp. 66-77.
10. J.N. Herder et al., "Modular System Programming in MINIX 3," Usenix; [www.usenix.org/publications/login/2006-04/openpdfs/herder.pdf](http://www.usenix.org/publications/login/2006-04/openpdfs/herder.pdf).