上海交通大学

硕士学位论文

基于UEFI的虚拟机及其在数字签名上的应用

姓名：陈志峰

申请学位级别：硕士

专业：计算机应用技术

指导教师：张申生

20080101

基于 UEFI 的虚拟机及其在数字签名上的应用

摘 要

数字签名的安全性必需建立在密钥安全的基础上，存储在计算机系统中的密钥容易被计算机病毒等恶意程序窃取并伪造数字签名破坏数字签名方案的有效性。本文在分析比较当前几种数字签名的软件和硬件的实现方案优劣势的基础上，结合对 UEFI 和基于 UEFI 的虚拟机的研究，提出了利用虚拟机技术提高数字签名的密钥安全性的方法。本文在分析说明基于 UEFI 的虚拟机的结构框架以及将其应用于数字签名的方案模型的同时，还给出了数字签名服务、应用系统库程序以及消息传输协议等关键技术的实现算法及数据结构。最后本文综合分析了基于虚拟机的数字签名技术的功能特点并给出了综合性能评价。

关键词：UEFI，虚拟机，数字签名，信息安全，并行多处理计算

The Research of UEFI Based Virtual Machine and Its Application on Digital Signature

ABSTRACT

The security of digital signature is largely based on the safety of cipher key. In normal cases, a cipher key is stored in the computer system, which can be easily obtained by vicious computer programs like computer virus etc. These vicious computer programs can then fabricate another digital signature, and wreck the validity of digital signature system. This paper studied UEFI and UEFI based virtual machine, and then proposed an approach to improve the safety of digital signature with the application of virtual machine technology. At the same time this paper also gives detailed illumination of data structures and Algorithms of several critical techniques like the digital signature service, application support programs and message transmission protocols. Finally this paper systematically analyzed the characteristics of digital signature based on an UEFI virtual machine, and provides performance evaluations.

Keywords: UEFI, virtual machine, digital signature, information security, SMP computing

上海交通大学
学位论文原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：陈志峰

日期： 2008 年 02 月 17 日

上海交通大学
学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权上海交通大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

保密□，在___年解密后适用本授权书。

本学位论文属于

不保密√。

（请在以上方框内打“√”）

学位论文作者签名：陈志峰　　指导教师签名：张申生

日期： 2008 年 02 月 17 日　　日期：2008 年 02 月 16 日

Internet 飞速发展，信息化在社会中不断深入，电子商务，电子政务的应用日益广泛，技术的开放性与互联性带来的众多安全问题是始料未及的，系统自身的漏洞，病毒，木马给信息安全带来了莫大的挑战。

信息安全服务依靠安全机制来完成，而安全机制主要依赖于密码技术和密钥保护技术。一般来说，系统的保密性应遵循不依赖于加密体制或算法而只依赖于密钥的保密(Kerckhoff)原则[2]，也即是，加密和解密算法是公开的，密码分析者可以知道算法与密文，但由于他并不知道密钥，因此仍难于将密文还原为明文。为此，专用加密机、TPM(Trusted Platform Module 可信计算平台)应运而生。它们可将密钥存储，密码学运算放在独立的计算机或者单片机中进行，防止密钥被非法获取。

本文在对密码学基本概念介绍的基础上，对数字签名中的密钥保护技术进行研究，着重分析了基于虚拟机技术的数字签名技术的理论及其应用价值，并给出了基于 EFI-Tiano 和多核处理器的数字签名方案模型和关键技术。

第一章 基于虚拟机的数字签名

1.1 不安全的密钥

1.1.1 签名方案

签名方案又称为数字签名，是一种给以电子形式存储的消息签名的方法。一个数字签名方案包括两个部分：签名算法和验证算法。签名方能够使用一个（私有的）签名算法 sig 来为消息 x 签名，签名结果 sig(x)随后能使用一个公开的验证算法 ver 得到验证。给定数据对(x,y)，验证算法根据签名是否有效而返回该签名为“真”或“假”的答案[3]。

一个签名方案是一个满足下列条件的五元组(P,A,K,S,V)：

1．P 是由所有可能的消息组成的一个有限集合。

2．A 是由所有可能的签名组成的一个有限集合。

3．K 为密钥空间，它是由所有可能的密钥组成的一个有限集合。

4．对于每一个 $k \in K$，有一个签名算法 $sig_k \in S$ 和一个相应的验证算法 $ver_k \in V$。

对每一个消息 $x \in P$ 和每一个签名 $y \in A$，每一个 $sig_k : P \to A$ 和 $ver_k : P \times A \to |true, false|$ 都是满足下列条件的函数：

$$ver(x, y) = \begin{cases} true & y = sig(x) \\ false & y \neq sig(x) \end{cases}$$

由 $x \in P$ 和 $y \in A$ 组成的数据对 (x, y) 称为签名消息。

对每一个 $k \in K$，sig_k 和 ver_k 应该是多项式的时间函数。ver_k 是公开的函数，而 sig_k 是保密的。给定一个消息 x，除了 Alice 之外，任何人去计算试 ver(x,y)=true 的数字签名 y 应该计算上不可行(注意，对给定的 x，可能存在不止一个 y，这要看函数 ver_k 是如何定义的)。如果有另外一方能够计算出使得 verk(x,y)=true 的数据对(x,y)，而 x 没有事先被签名方签名，则签名 y 称为伪造签名。非正式地，一个伪造签名是由签名方之外的其他人产生的一个有效的数字签名。

按照定义，攻击者如果要对任何消息产生有效的签名即完全破译签名体制就必须确定签名者的私钥 sig_k。

攻击者要获得私钥有两种办法。一是利用签名算法本身的漏洞通过唯密钥攻击(已知公钥)和已知消息攻击(已知一系列以前的有效消息签名对)，间接计算获得。二是利用计算机系统的漏洞获得。其结果都是完全破译数字签名，对任何消息产生有效的签名。

为了提高数字签名的可靠性，人们不断提高签名方案的复杂性。发明出各种签名方案如 ElGamal, DSA, Schnorr,椭圆曲线 DSA 等等。签名算法的改进使得供给者通过唯密钥攻击和已知消息攻击来寻找可能的原文签名对变得困难。但这仅仅是从一个方面提高了签名体制的可靠性。一旦密钥被非法获取，攻击者仍然可以随意篡改原文并计算签名。

1.1.2 密钥存储漏洞

计算机系统进行加解密操作时，通常会需要一个密钥。而不论这个密钥是从服务器获得还是由用户输入，当前的计算机系统结构决定了要进行计算必须先把密钥存储在内存中(尽管可能是临时的)。这个暴露在内存中的密钥就可能成为攻击者可以利用的弱点。[4]

一个典型的例子是当 Web 服务器在处理 SSL 事务时，为了完成 SSL 私有密钥的操

作，需要有一份服务器的私有密钥保存在它的内存里。在 1999 年初，nCipher 发现了一个很有效的方法 — 通过扫描大量数据来寻找私有密钥。

在某些配置下，一些操作系统允许应用程序以同一个用户的身份运行，该程序可以读取其他进程的内存区域。由于 CGI 程序通常运行于这种方式，在同一个 Web 服务器上任何一个能够访问的 CGI 程序，潜在地都可以扫描该 Web 服务器的内存，已发现该服务器上所有安全站点的私有密钥。nCipher 示范了一个 CGI 程序，扫描内存并返回运行在服务器上的所有安全虚拟主机的私有密钥。

他们还提出了这个问题的解决方案：安全地将你的密钥写入一个外部硬件装置，比如说写入到一个硬件加速其中。然后加速器不但执行密钥的操作，而是它也有唯一的密钥拷贝。Web 服务器不需要向加速器提供密钥，实际上通常将在该装置内部产生密钥并且不会输出他们。然后 Web 服务器必须手动切断该硬件加速器上的全部密钥操作，因此每个 Web 服务器都需要具备自己附属的硬件加速器。

另外最普通的缓冲区溢出错误也可能导致密钥泄漏或是被非法修改。缓冲区是内存中存放数据的地方。在程序试图将数据放到机器内存的某一个位置的时候，因为没有足够的空间就会发生缓冲区溢出。而人为的溢出则是由一定企图的，攻击者写一个超过缓冲区长度的字符串，植入到缓冲区，然后再向一个有限空间的缓冲区中植入超长的字符串，这时可能会出现两个结果：一是过长的字符串覆盖了相邻的存储单元，引起程序运行失败，严重的可导致系统崩溃；另一个结果就是利用这种漏洞，可以执行任意指令。这种攻击手段对于无法动态分配缓冲区的语言(比如 C/C++)来说都是有效的。

尽管花了很长的时间使得人们知道了如何编写安全的程序，具有安全漏洞的程序依旧出现。特别是在软件开源的情况下，攻击者可以很清楚的知道程序的内存布局，也就更容易实现攻击。

因此，研究如何建立安全的计算机系统并阻止攻击者利用系统漏洞伪造签名成为计算机应用领域的一项重要课题。

1.1.3 TPM 加密机

为了增强计算机系统的安全性，人们通常会靠增加专门的安全硬件来保证。因为硬件设备相对比较独立，不容易被病毒木马等恶意程序的破坏。

TPM 即可信计算平台的计算模块，它通常就是具有密码运算能力和存储能力的芯片，在可信子系统中起核心的控制作用。[5]

可信子系统的功能主要就是由 TPM 来完成，它必须提供两个方面的功能。首先，它应能够衡量在这个平台上的软件环境的状态并且能够把数据封闭给一个特定的软件环境，应提供状态报告信息使别的实体能够据此推演在此平台上的计算环境的状态是否可接受，并且还能够与此平台执行某个数据交换计划，这个实体能够确保这些数据在秘密的形式下被使用，可信子系统必须提供一种方式去加密用于密码方面的关键信息(如密钥)，并在这些关键信息被解密前确认软件环境是可信任的。TPM 作为信息计算平台的核心安全控制和运算部件，它的工作要先于操作系统和 BIOS，不可能使用计算机的内存外存和处理器，因此必须内部实现一些公开的安全算法，以便与其它部件的接口标准化，和提供内部一些安全操作中的密码运算。

除了 TPM 这类比较简单的单片机加密装置以外，也有像加密机这样比较复杂，可以被多台计算机共享的加密装置，如加密机。

加密机是一个基于安全的操作系统平台，具有高级通信保密性、完整性保护功能的控制系统。它是专用软、硬件设备，可具有多个网络接口，可安装于内联网各局域网出口处，或安装于内联网与公共网络接口处，或集成于网络防火墙中，提供网络边界之间的加密，认证功能[6]。加密机与主机之间使用 TCP/IP 协议通信，所以加密机队主机的类型和主机操作系统无任何特殊的要求。从逻辑上看，加密机主要有四个功能模块组成：硬件加密部件、密钥管理菜单、加密机后台进程和加密机前台 API。其中硬件加密部件主要的功能是实现何种密码算法、安全保存密钥。而前台 API 给应用系统提供了加密开发接口。应用系统通过调用加密机前台 API 使用加密机的服务。

硬件加密技术的确是将系统安全性提升到了一个新的高度。把密码学运算和应用逻辑从物理层隔离，试恶意代码就无法取得密钥等关键信息。但在某些条件下，硬件方案也有其弱点或局限性。攻击者利用基于物理特征的分析技术，如电压分析技术，故障分析技术，差分分析技术和电磁辐射分析技术等，仍然可以获得密钥。单片机的运算速度通常较低，无法适应实时系统的需要。

1.2 虚拟机数字签名方案

虚拟机：一台中央处理机作为真实计算机，在此基础上构成一种具有同时运行若干不同操作系统功能的目标计算机，此目标计算机称为虚拟机。顾名思义，虚拟机有多干台，每台虚拟机可以有其自己的操作系统和用户程序。在虚拟机运行时，一台虚拟机的工作不依赖于与其并发运行的别的虚拟机。每一台虚拟机可以使用不同类型的操作系统[7]。

1.2.1 虚拟机结构

通过引入虚拟机，如图 1-1 所示的客户应用系统运行于图左方所示的操作系统中(称为系统 A)，而数字签名程序运行在另一个操作系统中(称为系统 B)。两个操作系统运行时互不干扰，互不依赖。数字签名系统的密钥等关键信息保存在系统 B 的内存中[8]。而运行应用系统的系统 A 中还可能运行着攻击程序，但由于其无法接触到系统 B，也就无法接触到数字签名系统的关键信息，起到了提高系统安全性的作用。

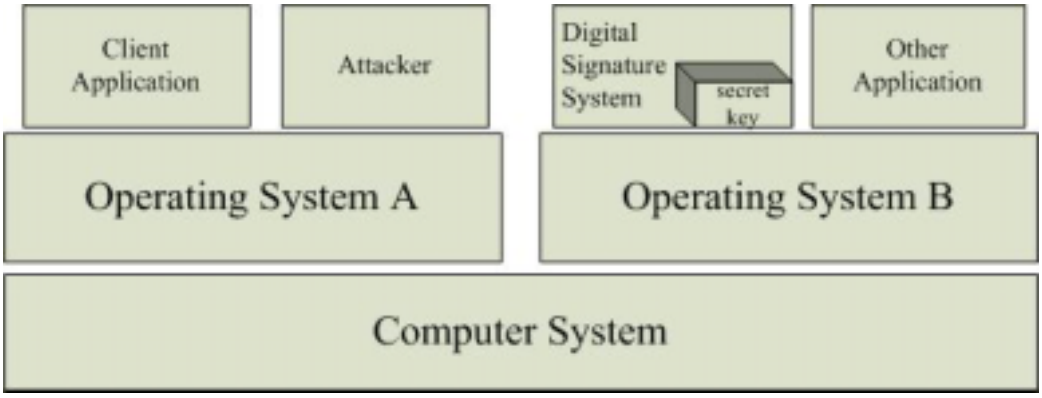


图 1-1 虚拟机方案结构图

Fig.1-1 Virtual Machine Framework

基于典型桌面计算机系统，虚拟机的实现采用的是逻辑分区的方式[9][10]，主操作系统(UEFI)和客户操作系统(Linux)分别使用各自独占的设备。Intel Dual-Core 处理器中的两个核心被划分给两个操作系统分别使用，同时内存也作了划分,Linux 使用高端的 128M 内存，而 EFI 使用低端内存，此外的设备如外存，输入输出设备都分配给了客户操作系统使用。所以从客户操作系统看来，它就像运行在一台普通的计算机上一样，完全不知道自己是一个虚拟机，而在另一个处理器上还有数字签名服务在工作。

将数字签名例程迁移到另一个操作系统中执行后，应用程序无法直接通过函数调用的方式使用另一个操作系统中的数字签名服务，这就需要在应用程序和数字签名服务中间建立一座桥梁，它不但要保证应用程序能够有效得使用数字签名服务，还要保证供恶意程序不能利用这个通道获取密钥等关键信息。共享内存是这座“桥梁”的实现手段之一。其实现必需遵循三条原则。1.虚拟机的存储区可以逻辑隔离。2.密钥等关键信息不得存放于共享区 3.要具有入侵抵抗与检测的能力。逻辑隔离要求操作系统仅能访问被分配到的内存区域(包含共享区)，不能访问超出这个范围的内存区域。操作系统应当认为它所拥有的内存是全部的物理内存。密钥仅能存放在操作系统 B 的非共享区域，这样才能阻止操作系统 A 中可能存在的恶意程序获得密钥。为了进一

步增强系统安全性，保证逻辑隔离的有效性，防止溢出攻击等攻击手段的破坏，运行签名程序的操作系统 B 还应当适当安排内存布局并增加入侵检测的能力[11]。

图 1-2 所示的是拥有 256MB 内存的虚拟机系统内存分配图。运行签名程序的操作系统 B 占据低端内存，运行客户程序的操作系统 A 拥有高端内存，其中设置有内存共享区可用于消息传递。密钥存放在操作系统 B 的内存中，远离共享区。

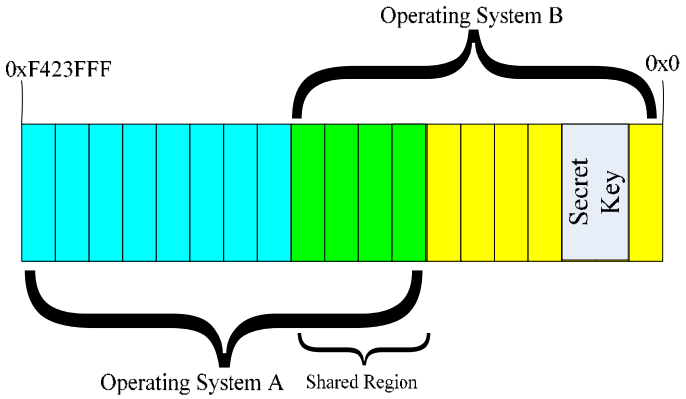


图 1-2 内存分配图

Fig.1-1 Memory distribution

1.2.2 应用程序接口

应用程序接口给客户程序提供使用数字签名服务的途径。客户程序一般工作在操作系统的用户态，处于保护模式，使用虚地址而非物理地址访问内存。因此应用程序无法读写共享区内存以设置签名数据以及读取签名结果。因此需要运行在核心态的应用程序接口来完成这些操作。应用程序接口的作用就是把用户请求中的数据按照物理地址复制到内存共享区中，签名完成后，又把共享区的结果按照虚拟地址复制到用户区域中。应用程序接口工作在内核态，可以同时按照虚拟地址或者物理地址访问内存。在如图 1-3 所示的合作图中，各部分按照以下顺序合作完成一次签名操作

1. 客户程序调用应用程序接口函数发送服务请求
2. 应用程序接口把请求中的数据按照物理地址复制到内存共享区
3. 签名服务程序读取共享区中的数据

4. 签名服务程序计算签名结果
5. 签名服务程序把签名结果复制到内存共享区
6. 应用程序接口把签名结果按照虚拟地址复制到用户空间
7. 客户程序读取签名结果

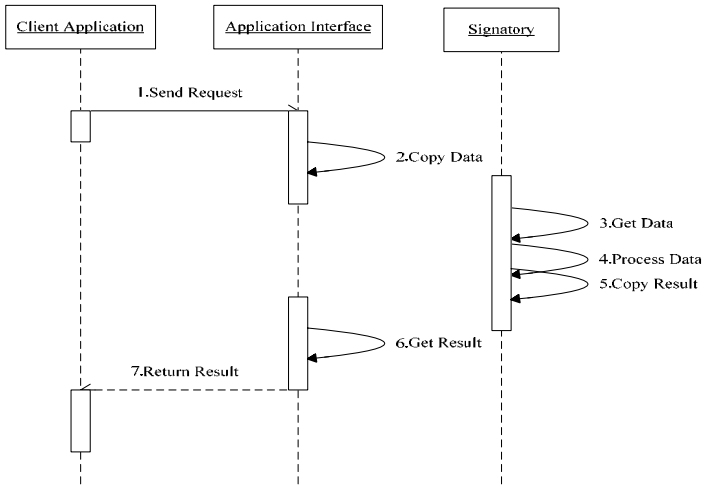


图 1-3 签名过程协作图

Fig.1-3 Signature Process Cooperation Diagram

1.2.3 消息传递协议

操作系统 A 中的应用程序接口和操作系统 B 中的数字签名系统利用消息传递来通信。不论是发送签名服务请求还是获取签名结果都要求使用统一的消息传递协议。在本文所述的消息传递协议中共定义了两类消息。第一类是命令消息，第二类是数据消息。命令消息用于传达应用程序接口给数字签名程序发送的指令如开始命令、结束命令等。数据消息既能传递需要签名的原始数据，也能传递签名运算的结果。消息传递协议的设计目标是传递服务请求，实现多个服务请求的并行处理，以及提高数据传输能力。

两类消息的格式如图 1-3 所示：

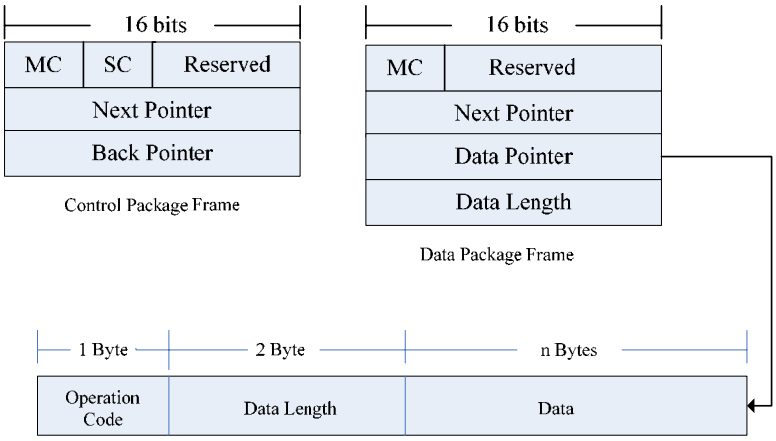


图 1-4 消息格式图

Fig.1-4 Message Format

消息格式

· 主类号(MC)：表示报文的主要类型，可以为数据报文或控制报文。

· 次类号(SC)：表示报文的次要类型，和主类号一起构成消息的完整类型信息。数据报文的次类号没有作定义，而控制报文的次类号可以表示连接初始报文或连接终止报文。

· 发送指针(Next Pointer)：指向下一报文的指针的地址。当报文在共享区中被构建起来的同时，一个指向下一报文的指针也会在共享区中被创建，这个指针地址作为消息的一部分传递给报文的接收者，作为下一个消息的发送地址。

· 接收指针(Back Pointer)：指向报文接收指针的地址，这一地址仅存在于建立连接请求报文中。当连接请求者欲建立连接时，应当预先在共享区创建指向接收报文的指针，在把这一指针的地址作为消息的一部分传递给接收方。接受方可以根据这个地址来回送报文。

· 数据指针(Data Pointer)：共享区中的字节数组指针，在这个数组中存放了待传输的数据。在报文被发送前，这个数组会首先被建立起来并且填充好数据，最后将数组的地址填入报文结构中。报文接收方再根据这个地址在共享区中取得这些数据。

· 数据长度(Data Length)：表示字节数组的长度，由于共享区中的数组不具有边界信息，这个长度提供给报文接收方确定数组边界的信息。

· 操作码(Operation Code)： 表示操作的类型，它是占用一个字节的枚举类型其定义如下：

{SelectAlgorithm = 0, FeedData = 1, RequireResult = 2,ResultData=3, ResultFinished=4 }

·数据长度(Length)：它是一个占用两个字节的无符号整数，用以表示数据段的长度，以字节数衡量。可以表示从 0 到 65535 字节的长度范围。

· 消息数据(Data)：在不同的消息中，数据段中数据的意义有所不同。例如在 SelectAlgorithm 消息中，数据段表示所选择的签名算法如 SHA1、SHA128、SHA256、椭圆曲线 DSA 等。在 FeedData 消息中数据段表示明文数据。

1.3 本章小结

本章对数字签名方案进行了简单介绍并且分析比较了几种数字签名的软件和硬件的实现方案，如 TPM 等，同时还给出了这些方案的优势及弱点。在综合分析这些技术优劣的基础上，提出了使用虚拟机技术实现安全的数字签名的概念设计。

在虚拟机系统中，一台虚拟机的工作不依赖于与其并发运行的别的虚拟机。每一台虚拟机可以使用不同类型的操作系统。使用虚拟机技术，可以将数字签名操作所用的密钥存放在主操作系统中，并用于主操作系统中的数字签名服务进行签名运算，客户操作系统通过使用服务的形式对目标明文进行签名。通过阻碍恶意程序获得密钥的途径，使得它无法对签名进行篡改，从而达到提高数字签名运算安全性的目的。

为了提供客户操作系统中的应用程序访问数字签名服务的途径，并保证签名运算的实现效率。本章最后还提出了一种基于共享存储区的消息传输模型和这种模型下的消息报文的结构及定义。这种建立在主操作系统和客户操作系统之间的点对点的通信模型，解决了多个应用程序并行使用数字签名服务的问题，保证了数据传输的稳定高效。

第二章 基于 UEFI 的虚拟机

2.1 UEFI 概述

2.1.1 设计目标

作为 UEFI 前身的 EFI 最初是由 Intel 于 1999 年开始开发并于 2004 年成功完成的新一代 BIOS，并在多方面的促进下最终形成了现在的 UEFI 规范。“ PC-AT ”引导环境如今已经成为了阻碍工业发展的一个障碍。由于“ PC-AT ”的限制，许多硬件方面的创新和进步都需要固件开发者去设计更加复杂的解决方案，同时操作系统开发者也需要修改启动代码来获得新硬件的支持。而这是一个耗时的过程，需要巨大的投入。UEFI 最主要的设计目标就是要减轻这样的负担[12]。

UEFI 规范的主要特点可以归纳为：

- 统一的可扩展的平台环境
- 支持操作系统的接口
- 独立于传统接口的设备抽象层
- 支持 Option ROM 接口
- 平台无关
- 进化而非革命
- 设计具有兼容性
- 使在操作系统外安装程序更容易
- 可安装在现有设备上

在符合 UEFI 规范的 BIOS 中建立虚拟机有如下优势：

✓ 有利于设计方案的系统集成，便于推广

在 BIOS 中固化签名服务，免去了软件的安装过程，使得安全数字签名成为了硬件系统的功能之一。使用 BIOS 固化软件比安装独立的软件更加便利，也更易受到用户的信任。

✓ 相对比较安全

对于安全系统来说，存储环境的安全性也是非常重要的。BIOS 相对其它存储器更加难以被非法程序篡改，特别是在 TPM 的验证下，BIOS 的安全性更加毋庸置疑。

✓ 更易于操作硬件资源

虚拟机需要使用各种硬件资源，而 BIOS 管理着整个系统的硬件资源。建立虚拟机需要做处理器，存储器以及多种硬件的再分配。通过使用 BIOS 内建的硬件驱动程序，使用硬件资源变得更加简单。

✓ 扩展性好，易于添加自定义应用程序

传统的 BIOS 很少支持程序扩展。UEFI 区别于传统 BIOS 的特点就是强大的扩展性，在 UEFI 中不仅能添加自定义的驱动程序，更能添加各种功能各异的应用程序以丰富 BIOS 的功能。

✓ 编程环境好，UEFI 工作在保护模式支持“C”语言编译的代码

传统 BIOS 使用汇编语言编程，运行在实模式下，可用内存空间仅 1M。而 UEFI 支持 C 语言编写的程序，工作在保护模式下，可以访问 4G 内存空间。UEFI 更加利于实现复杂的应用程序。

✓ 模块化设计，有定义清晰的规范

传统的 BIOS 历史久远，不具备模块化思想，结构混乱。因而在其上开发应用也困难重重。相反 UEFI 结构清晰，各个模块有相应的规范，即利于功能的重用也方便了软件在平台之间的移植，

2.1.2 系统结构

UEFI 在设计上主要基于以下几个基本元素：

- **重用现有的基于表的接口.**为了保留现有架构上的代码，不论是操作系统上的还是固件上的。UEFI 要求必须实现已经在现有的平台和处理器架构上广泛使用的那些规范。这些规范的实现使得 UEFI 能够兼容传统的操作系统。

- **系统分区.**系统分区定义了可以在不同用途的固件间实现共享的文件系统，这种分离的可共享的系统分区方式使得添加系统功能而不显著增加对非易失性存储器(NVRAM)的需求成为了可能。这一特性可以让 UEFI 加载存储在硬盘，闪存

等介质上的虚拟机应用程序。

- **启动服务(Boot Service).**启动服务为设备和启动时可用的系统功能提供了接口。通过“句柄”和“协议”，设备被抽象了出来。这种方式不强制保持底层实现必须符合规范从而允许重用传统的 BIOS 代码。启动服务是 UEFI 的核心，它给 UEFI 提供了强大的扩展性的，其中包含了映像文件服务，协议-句柄服务，内存管理，任务管理等功能。

- **运行时服务(Runtime Service).** 一系列运行时服务用来保证操作系统工作时可以获取正确的硬件资源信息。

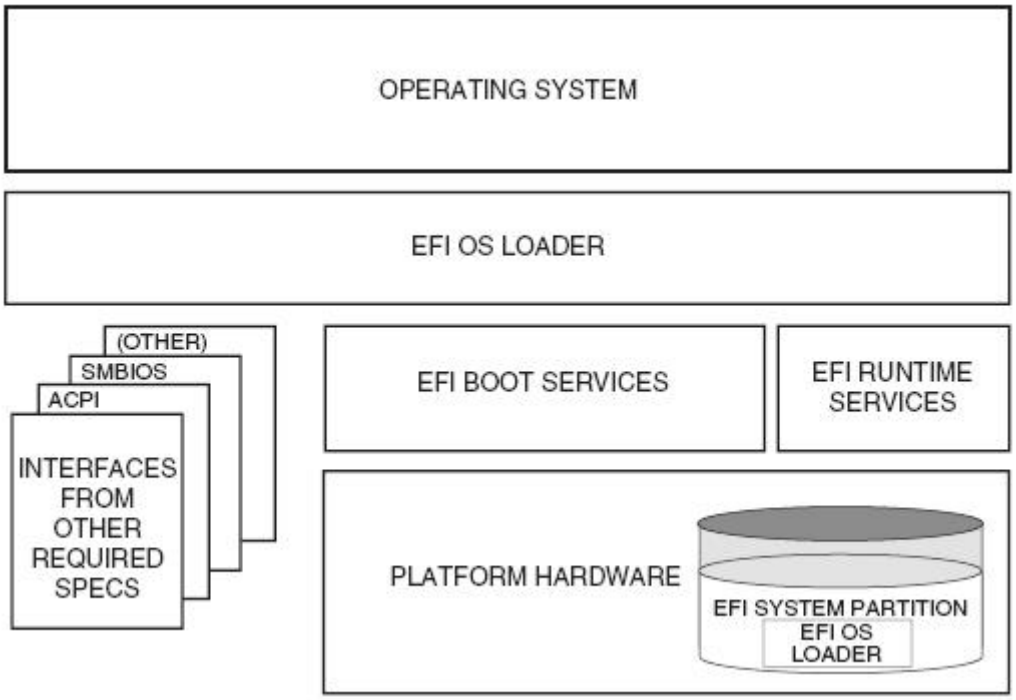


图 2-1 UEFI 结构总揽图

Fig.2-1 UEFI Conceptual Overview

图 2-1 展示了 UEFI 系统中各模块间的关系。UEFI 固件能够把虚拟机引导映像文件(VM loader image)从系统分区中读取出来。UEFI 规范提供了从多种类型的存储设备如磁盘、CD-ROM、DVD 以网络引导的能力。通过可扩展的接口协议，还可以添加可用于引导的介质类型。UEFI 应用程序如虚拟机引导程序能够使用启动服务去查询、分析以及初始化各种平台组件。值得注意的是，在引导阶段，就和启动服务一样，

运行时服务同样是可用的。

2.2 虚拟机

虚拟机，它作为主操作系统下的一个应用程序可以为运行于其上目标操作系统创建出一部虚拟的机器，目标操作系统像运行在一台独立的真实计算机上，丝毫察觉不到自己是处于主操作系统的控制之下[13]。在虚拟机上运行的应用程序认为自己独占整个计算机。由于虚拟机的处理器、存储器、I/O 控制器都只是全部计算机资源的某个子集，这也就使得运行在虚拟机上的应用程序难以分辨自己是否运行在真实计算机上，同时也无法使用这个子集以外的系统资源。总而言之是把虚拟机同计算机的其余资源通过逻辑隔离分割开来。

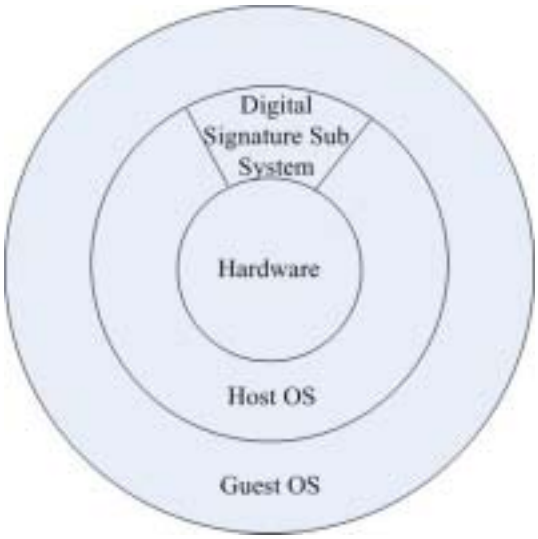


图 2-2 虚拟机数字签名方案逻辑图

Fig.2-2 Virtual Machine Digital Signature logic diagram

如图 2-2 所示，通过这样的设计，数字签名密钥的生成，存储和应用完全都在主操作系统内完成，目标操作系统仅能通过主操作系统提供的接口使用到数字签名子系统提供的签名功能，从而保证了密钥的安全，提高了数字签名的安全系数。在本文所论述的实现方案中主操作系统(Host OS)选用的是 UEFI，而目标操作系统(Guest OS)选用的是 Linux。

2.3 虚拟机的引导

UEFI 允许通过加载 UEFI 驱动和 UEFI 应用程序映像来扩展平台固件。当 UEFI 驱动和 UEFI 应用程序被加载以后就能访问所有在 UEFI 系统中定义的启动服务和运行时服务[15]。

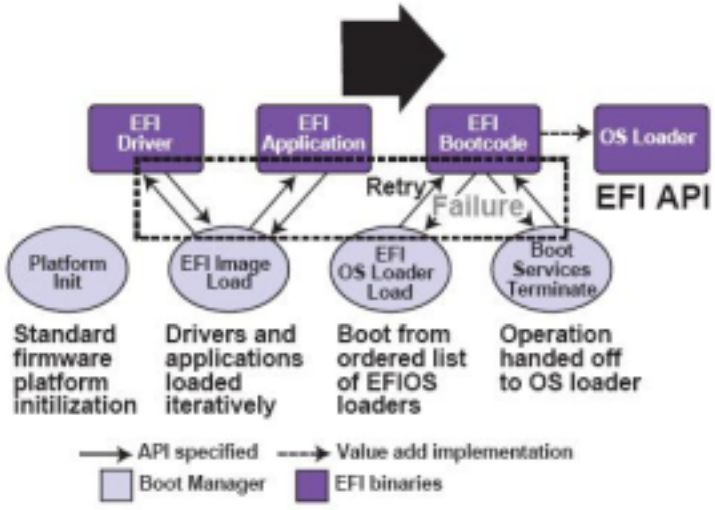


图 2-3 UEFI 启动顺序图

Fig.2-3 UEFI Booting Sequence

UEFI 允许把虚拟机启动菜单项和固件中的原始菜单项合并到一个菜单中。通过这个菜单，由 UEFI 启动服务支持，存储在任何介质的任何分区的虚拟及启动程序都能够被选择为启动项目。这个启动菜单同时还支持传统启动选项(比如从 A:或 C:驱动器启动)。

UEFI 可以支持从安装了 UEFI OS loader 的介质或者从 UEFI 系统分区启动。从块设备启动的方式要求该设备中有一个 UEFI 定义的系统分区。UEFI 不要求修改分区的首段，因此可以使传统的启动分区同时支持 UEFI 启动。

UEFI 启动管理器允许从任何 UEF 的文件系统的文件或者通过 UEFI 映像加载服务加载符合 UEFI 规范的应用程序和驱动程序。UEFI 通过在 NVRAM 中定义变量来指向要加载的 UEFI 应用程序。这些变量还包含了程序的入口参数和可以在菜单上显示应用程序的 Unicode 字符串。虚拟机启动程序就是一个符合 UEFI 定义的应用程序，通过启动管理器可以加载到内存并且运行。

UEFI 定义的变量还允许固件中包含可以指向各种操作系统的启动菜单。UEFI 的设计目标就是使得启动菜单可以被放在固件中。UEFI 仅仅定义了表示启动选项的

NVRAM 变量，这些菜单选项的实现统统放在独立的实现空间中。

对比当前的 PC-AT 系统的工作方式，UEFI 大大的扩展了启动的灵活性。UEFI 所支持的应用程序运行框架使得虚拟机运行在 UEFI 上成为了可能。当前的 PC-AT 系统仍然只允许从软驱、硬盘、CD-ROM、USB 或网卡启动系统。而且从一个硬盘引导不同操作系统还会导致多种兼容性的问题。

2.3.1 启动程序映像文件

UEFI 映像文件是一类符合 UEFI 定义且包含可执行代码的文件。UEFI 映像的一个特点是在文件的头部包含了表示这个可执行文件编码方式的字节序列[16]。

UEFI 使用 PE32+文件格式的子集并修改了头部签名。修改 UEFI 映像的头部签名可以使之和普通 PE32 执行文件有所区别。而‘ + ’号在这里表示该格式扩展了对 64-bit 的支持。

UEFI 镜像的头部签名中的 Subsystem 值定义如下：

```
// PE32+ Subsystem type for EFI images
#define EFI_IMAGE_SUBSYSTEM_EFI_APPLICATION             10
#define EFI_IMAGE_SUBSYSTEM_EFI_BOOT_SERVICE_DRVIER     11
#define EFI_IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER          12
```

不同的类型的映像主要的却别是固件会把它们加载到不同的类型的内存中，并且从映像入口函数退出的行为也有所不同。应用程序在程序执行从入口函数退出时被卸载。驱动程序映像当程序执行错误时才会被卸载。

表 2-1 UEFI 映像内存类型

映像类型	代码段内存类型	数据段内存类型
EFI_IMAGE_SUBSYSTEM_EFI_APPLICATION	EfiLoaderCode	EfiLoaderData
EFI_IMAGE_SUBSYSTEM_EFI_BOOT_SERVICES_DRIVDER	EfiBootServiceCode	EfiBootServicesData
EFI_IMAGE_SUBSYSTEM_EFI_RUNTIME_SERVICE_DRIVER	EfiRuntimeServicesCode	EfiRuntimeServicesData

PE 映像文件头中的 Machine 值表示了映像对应的机器类型码。UEFI 映像签名中的机器类型码是做如下定义的。

```
//PE32+ Machine type for EFI image
#define EFI_IMAGE_MACHINE_IA32          0x014C
```

```
#define EFI_IMAGE_MACGINE_IA64          0x0200
#define EFI_IMAGE_MACHINE_EBC           0x0EBC
#define EFI_IMAGE_MACHINE_x64           0x8664
```

UEFI 规定,一个特定平台的固件必须实现其直接对应的机器类型的映像以及 EFI 字节码(EBC)类型的映像。而其余类型码的映像则可以做选择性的实现。启动服务函数 LoadImage()函数可以把 UEFI 映像载入内存。启动服务要求 PE32+加载器把 PE32+映像文件的所有段同时载入。一旦映像被载入内存,并且完成了所有对应的准备动作,执行序列就会转向载入映像的入口函数地址(AddressOfEntryPoint)。

2.3.2 UEFI 应用程序

按照 UEFI 规范写的应用程序可以被启动管理器或其它 UEFI 应用程序调用。为了载入应用程序，固件首先要分配足够的内存空间以放置映像，然后把映像文件中的各段分别复制到对应类型的内存空间中，把各段内存空间标记成数据段或是代码段，最后把执行位置转向映像的入口地址。当应用程序从它的入口函数退出，或者当它调用启动服务的 Exit()函数后，固件把应用程序从内存中卸载掉，程序执行位置重新回到调用这一程序的 UEFI 组件。

当启动管理器载入应用程序时，映像的句柄还可以用来查找应用程序的“ 载入选项 ”。载入选项存放在非易失性存储器中并且作用于被启动管理启载入的特定应用程序。

映像类型分为驱动程序(Driver)和应用程序(Application)。GUID 是一个在所有句柄中具有唯一性的值，用于代表这一映像文件。入口函数是映像被载入内存后开始执行的位置。

虚拟机启动程序以及数字签名程序都属于应用程序。在它们的定义文件中需指定程序映像句柄的类型、GUID 和入口函数。如下所示，

```
FILE_GUID             = D565D560-65A6-43e4-AD10-56AF636D2D89
COMPONENT_TYPE        = APPLICATION
IMAGE_ENTRY_POINT     = InitializeVMApplication
```

2.3.3 虚拟机引导程序

2.3.3.1 UEFI OS loader

虚拟机引导程序调用 UEFI OS Loader 来加载操作系统。OS Loader 是一类特别的 UEFI 应用程序。它能够获得 UEFI 固件的所有控制权。OS loader 被载入时，它的行为和一般的 UEFI 应用程序相同，都需要使用固件分配的内存空间，并且只能使用 UEFI 服务和协议去访问固件允许访问的设备。如果 OS loader 要包含任何的驱动程序功能，它必须使用适当的 UEFI 接口访问总线资源。这也就是，必须和那些 UEFI 驱动程序一样调用正确的总线访问函数才能访问 I/O 和内存映射的设备寄存器。

如果 OS loader 遇到了问题并且无法正确载入它的操作系统，它可以释放它所分配到的资源并把控制权通过启动服务函数 Exit()交回固件。Exit()函数可以返回错误码和 *ExitData*。*ExitData* 包含 Unicode 字符串和 OS loader 的内部数据。如果 OS loader 成功载入了操作系统，它能够通过调用启动服务函数 ExitBootService()获取完整的系统控制权。调用函数 ExitBootService()成功后，包括内存管理器在内的系统中所有的启动服务都会终止，同时 OS loader 要保证系统继续运行。

虚拟机启动程序不同于一般的 OS Loader。它启动 OS，但并不调用启动服务函数 ExitBootService()来终止启动服务，而是在配置好运行环境后将 OS Loader 指定在某一个处理器上运行并随后返回 UEFI。它能够启动操作系统并使 UEFI 与操作系统并行不悖的运作，从而实现一台物理计算机上运行两个操作系统。

2.3.3.2 对称多处理(SMP)

对称多处理是基于多处理器单元的计算机结构。利用对称多处理技术能够简化虚拟机启动程序的复杂度，同时提高虚拟机的性能。

图 2-3 显示了基于对称多处理技术的计算机系统的一般结构。

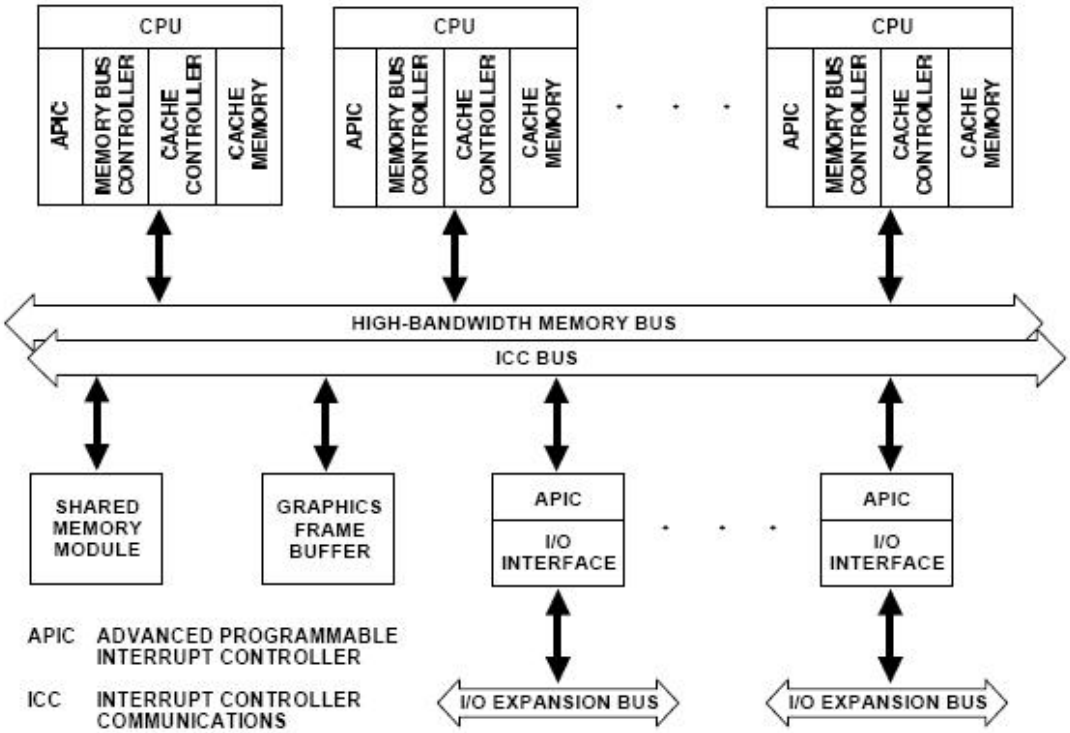


图 2-4 多处理器系统架构

Fig.2-4 Multiprocessor System Architecture

多处理系统表现为一个由可共享内存，分布式的处理器和分布式的输入输出中断器件紧密联结而成的结构。它是完全对称的；既是，所有的处理器在功能和配置上完全相同，每个处理器能够和其余处理器对等的通信。它们之间不存在继承和主从关系。系统对称性还表现在另外两个方面：

- **存储器对称** 存储器对称性表现在：处理器共享相同的存储器空间并且使用相同的地址访问这些空间。内存的对称性是一个重要特性，它能够允许所有的处理器运行同一个操作系统。任何已有的应用程序和操作系统都能无差别的运行在多处理系统上，不论处理的数量是否相同。
- **输入输出对称** 输入输出对称表现在：所有的处理器共享访问相同的输入输出子系统(包括输入输出端口和中断控制器)，所有的处理器都能从任何中断源接受中断。那些仅仅保证了存储器对称性的多处理架构实际上在中断控制上并不对称，因为他们仅允许一个处理器处理中断。输入输出的对称性消除了潜在的输入输出瓶颈，同时提高了系统的可扩展性。

多处理系统在启动时通常会首先启动其中一个处理器称为 bootstrap processor(BSP)，这个处理器负责系统的初始化，操作系统的引导。操作系统引导过

程中会初始化并启动其余的处理器，它们也被称为 application processor(AP)。BSP 和 AP 并无结构上的不同。每一个处理器既可以作为 BSP 也可以作为 AP[17]。

在对称多处理的系统上，首先由 BSP 运行 BIOS 的系统初始化代码，然后启动 UEFI 最后执行虚拟机启动程序。虚拟机启动程序首先会初始化所有的 AP，然后调用 OS loader 加载操作系统，最后将操作系统指定到其中一个 AP 上运行。在非 SMP 系统上要实现双系统并行需要软件模拟出两个处理器，包括软件模拟译码和软件模拟执行，而在 SMP 系统上双系统可以自然的并行工作在不同的处理器上[18] (当然存储器和输入输出由于共享导致的冲突依然需要解决)。

2.3.4 驱动程序

UEFI 驱动程序可以由启动管理器，UEFI 固件或其它 UEFI 应用程序载入。为了载入 UEFI 驱动程序，固件首先要分配足够的内存空间以放置映像，然后把各段复制到分配好的内存中并把各段内存空间标记成数据段或者代码段。然后执行转向驱动程序的入口函数。当从驱动程序入后函数返回时或者驱动程序调用启动服务 Exit()时，有选择地卸载驱动程序并将控制权交回调用程序的 EFI 组件。如果驱动程序入口函数返回 EFI_SUCCESS 那么它就不会被卸载。如果驱动入口函数返回错误代码那么它就会被卸载。

UEFI 驱动程序共有两类：启动服务驱动程序和运行时驱动程序。这两类驱动程序唯一的区别就是运行时驱动程序在调用 ExitBootService()并使操作系统控制整个系统以后仍然有效。启动服务驱动程序在调用 ExitBootService()函数后会被终止，并且由启动服务驱动程序占用的内存也会被释放。当操作系统调用 SetVirtualAddressMap()函数后类型为 EFI_IMAGE_SUBSYSTEM_EFI_RUNTIME_SERVICE_DRIVER 的运行时驱动程序可以获取到虚拟内存地址映射表。

UEFI 驱动程序给 UEFI 应用程序访问硬件提供了软件支持，但目标操作系统运行后也会加载一些驱动程序。如果硬件控制器仅支持独占式访问控制，那么当目标操作系统驱动程序和主操作系统驱动程序并行访问这些硬件时，就会产生错误。为了免共享冲突，在运行 OS Loader 之前需要卸载产生冲突的 UEFI 驱动程序或目标操作系统驱动程序[19]。

2.4 虚拟机存储器管理

虚拟机存储器管理可以通过 UEFI 服务来实现。UEFI 服务包括启动服务和运行时服务。

2.4.1 UEFI 服务

设计抽象的 UEFI 接口的目的是让 UEFI 映像可以调用一般性的启动环境。其中 UEFI 映像包括 UEFI 驱动，UEFI 应用程序和 UEFI OS loader。接口函数调用按照 64 位设计，因此可以兼容 64 位系统。这套标准的简洁的运行时服务还能为操作系统服务。

作为启动服务的基本实现方法之一，这一节所涉及的接口都支持使用标准的即插即用 Option ROM。UEFI 接口被这样设计以对应传统的接口而且不会受到传统 Option ROM 的任何束缚。因此 UEFI 服务能够很好的兼容现有的设备控制器。

UEFI 平台接口意图在硬件平台和其上的操作系统间建立一个抽象层。UEFI 规范也定义了硬件平台同诊断工具程序之间的抽象层。但是，它并非要实现一个完整的操作系统环境。它可以被想象成为一个小规模的类似于操作系统的环境，而且它易于建立在 UEFI 系统之上。

UEFI 平台接口可以这样分类：

- 运行时服务
- 启动服务
  - 全局启动服务
  - 基于设备句柄的启动服务
  - 设备服务协议
  - 协议服务

2.4.2 存储器分配服务

存储器分配服务是 UEFI 启动服务提供的一项功能。如表 2-1 所示，存储器分配服务包含了一系列用于分配与释放存储器单元的函数。

表 2-1 存储器分配函数

名称	类型	描述
AllocatePages	Boot	分配特定类型的页
FreePages	Boot	释放已分配的页
GetMemoryMap	Boot	返回当前内存映射表
AllocatePool	Boot	分配特定类型的内存池
FreePool	Boot	释放已分配的内存池

UEFI 中的页表是一个直接映射表，即页单元的虚地址等与其物理地址。基于以上事实，存储器分配服务一般使用指定的物理地址来分配和使用内存。

为了使主操作系统和目标操作系统在使用内存时不发生竞争和冲突，必须对两个操作系统的内存使用进行合理的分配。可以采取的方案是，主系统使用高端内存，目标操作系统使用低端内存，并且在中间规划出一块公共存储区专用于系统间的通信[20]。区域划分后，各系统只能严格按照约定范围使用内存，防止越界操作可能导致的错误，甚至系统崩溃。

2.4.3 运行时服务

运行时服务的主要作用是把硬件实现上的很多小的部分从操作系统抽离出来。运行时服务在启动阶段和运行阶段都是可用的。操作系统在运行时要运行时服务首先要切换到物理地址模式。但是如果 OS loader 或操作系统调用了运行时服务 SetVirtualAddressMap()，操作系统就只能在虚地址模式下调用运行时服务。所有的运行时服务都是无法被打断的，即使关闭中断后在调用它们也是可以的。

运行时服务使用的内存必须预先保留并且不能给操作系统使用。运行时服务的内存空间对 UEFI 函数是可用的而绝对不允许操作系统或其组件直接操作。UEFI 负责定义运行时服务可以使用的硬件资源，因此操作系统可以在调用运行时服务的同时使用那些硬件资源，或者保证操作系统绝对不使用那些硬件资源。运行时服务的功能主要有设置获取系统时间，设置获取变量，实虚地址转换等。

2.5 虚拟机 I/O 管理

2.5.1 UEFI 协议

系统输入输出接口由主操作系统和目标操作系统共享使用。为了避免共享可能带来的竞争与冲突，首先要合理安排输入输出接口的分配。对于无法共享的设备，仅将其分配给其中一个操作系统独占使用。

UEFI 在启动阶段能够发现与系统连接的设备总线，并通过总线发现总线上的设备控制器。根据设备信息，UEFI 为这些设备控制器加载驱动程序，并允许 UEFI 程序使用这些设备。目标操作系统被加载并开始运行后，同样也会发现这些设备控制器，并为它们加载操作系统使用的驱动程序。如果两个设备并行访问那些不支持并行的设备，就会发生硬件错误。因此，对于分配给 UEFI 使用的设备或设备总线，必须禁止操作系统使用它们，而对于分配给目标操作系统使用的设备则首先要在 UEFI 内卸载这些设备的驱动程序。

某个设备句柄或是所支持的协议可以通过 HandleProtocol()或者 OpenProtocol()启动服务查询。每个协议都包含以下信息：

- 协议的全局 ID (GUID)
- 协议函数接口定义
- 协议的实现

除非特别声明 1. 协议的成员函数仅在启动阶段可用 2. 协议成员函数的任务优先级等于或低于 TPL_NOTIFY(最高优先级)。3.协议成员函数不可重入不支持多任务。任何由协议成员函数定义的状态码都需要有其实现。如果返回额外定义的错误码，他们不会影响兼容性测试而且使用协议的时候不能依赖扩展的错误码。

为了确定设备是否支持某个特定的协议，必须把协议的 GUID 传给 HandleProtocol()或者 OpenProtocol()。如果设备支持所请求的协议就会返回一个指向协议的结构体的指针。通过协议结构体可以调用协议的各个成员函数以访问特定设备。

图 2-3 说明了协议的结构。 UEFI 驱动实现中的函数实现可以作为一个或多个协议的实例，然后通过调用启动服务 InstallProtocolInterface()把他们注册到设备句柄上。启动服务返回的协议结构体可以用于调用这些函数。 UEFI 驱动实现了服务于特定设

备的协议接口函数。

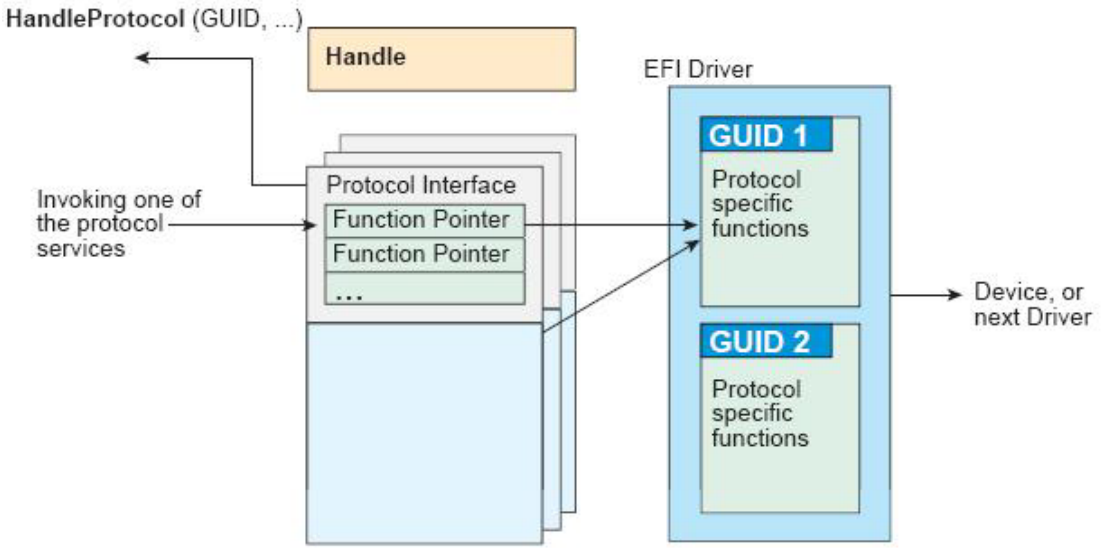


图 2-5 UEFI 协议结构图

Fig.2-5 Construction of a Protocol

```
// Definition of a protocol
typedef
EFI_STATUS
(EFIAPI * EFI_ ILLSTRATION_MAKEEFFECTS)(
ILLSTRATION_PROTOCOL                *this,
ILLSTRATION _EFFECT                 effect
);
typedef struct ILLSTRATION_PROTOCOL {
   EFI_ ILLSTRATION_MAKEEFFECTS    MakeEffects;
} ILLSTRATION_PROTOCOL;

// Get the protocol instance for device “EffectsDevice”
EffectsDevice.Handle = DeviceHandle;
Status = HandleProtocol (
             EffectsDevice.EFIHandle,
             &IllustrationProtocolGuid,
             &EffectsDevice.IllustrationProtocol);

// Invoke the device protocol
Status = EffectsDevice.IllustrationProtocol->MakeEffects(
      EffectsDevice.IllustrationProtocol,
```

```
    TheFlashyAndNpisyEffect
);
```

2.5.2 UEFI 驱动模型

UEFI 驱动模型意在简化设备驱动的设计和实现同时减少程序文件的尺寸。因此，一些比较复杂的实现已经被放到总线驱动或者固件库函数里。

当设备驱动程序需要提供特访问定设备句柄所需协议的实例时，它首先要等待并确定固件已经把驱动程序和对应的设备控制器连接好，然后，驱动程序负责产生用于读写对应设备句柄的协议的实例。而总线驱动不仅需要有一般设备驱动的功能还需要负责发现总线上连接的控制器并且产生对应每个设备的句柄。

对于系统结构的一个假设是 :系统可以看成是一个或多个处理器连接着一个或多个核心芯片组。核心芯片组又驱动一个或多个输入输出总线。UEFI 驱动模型的作用并非要处理器或是核心芯片组而是描述由核心芯片组驱动的输入输出总线和总线上的子节点。这些子节点既可以是设备也可以是另外的输入输出总线。这个结构可以被看成是一棵由总线和设备构成的树，而核心芯片组是树的根节点[21]。

图 2-4 展示了一个有着 4 条总线和 6 个设备的桌面系统。树状结构中的外围叶子节点负责输入输出，包括键盘、显示器、磁盘、网络等。非叶子节点是总线，他们设备和总线之间的数据传输。这种思想可以使 UEFI 驱动程序模型简单而且易于扩展。

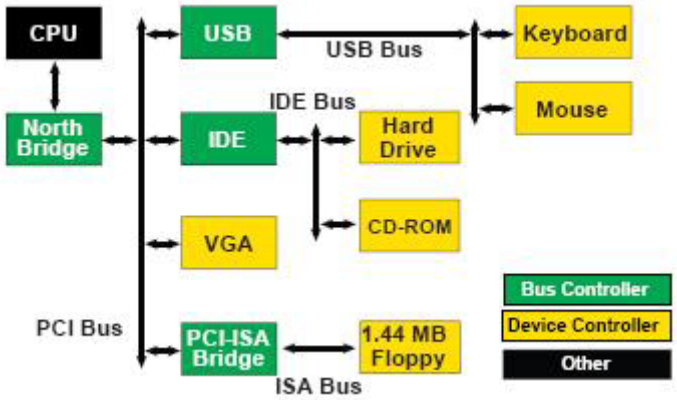


图 2-6 桌面系统结构拓扑图

Fig.2-6 Desktop System topological structure

2.5.3 驱动程序初始化

首先，驱动程序映像文件要从存储介质上载入。比如：ROM、FLASH、硬盘，软盘，网络等。找到驱动程序映像以后可以用启动服务 LoadImage()把它载入系统。LoadImage()会为驱动程序创建句柄，这个包含了协议实例的映像被称为*映像句柄*。仅仅这样，驱动程序还只是处在准备阶段。之后，它还必须通过启动服务 StartImage()来启动(这点对于所有 UEFI 映像都是相同的，不论是应用程序还是驱动)。依据 UEFI 驱动模型，驱动程序入口函数必须严格符合下列规范[22]。首先，不直接操作任何硬件。取而代之的是，驱动程序应该把协议的实例安装到它自己的映像句柄上。例如必须安装的 Driver Binding Protocol 以及选择安装的 Driver Configuration Protocol，Diagnostics Protocol，Component Name Protocol 等。安装了 Driver Binding Protocol 的映像句柄就是*驱动映像句柄*。图 2-5 表示了一个经过 StartImage()后的驱动映像句柄

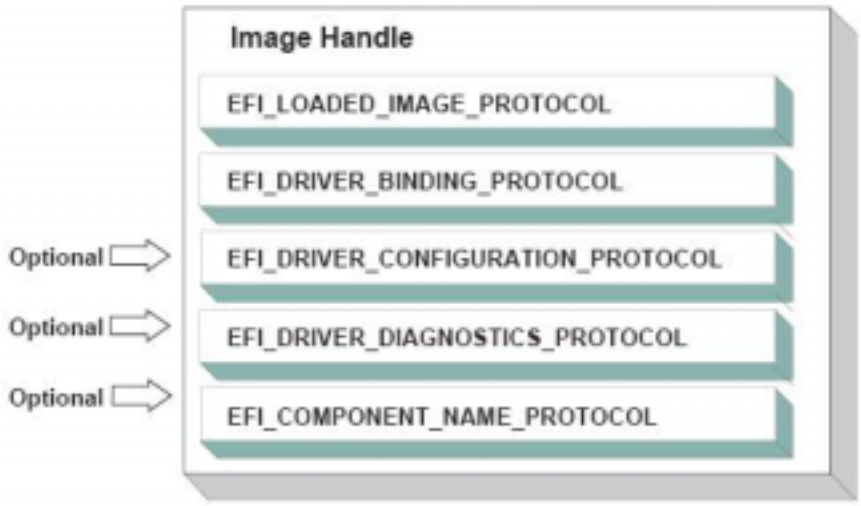


图 2-7 驱动映像句柄

Fig.2-7 Driver Image Handle

2.5.4 主总线控制器

驱动程序不允许在入口函数里操作硬件。因此，虽然驱动程序会被加载并运行，但是它们无法操作直到系统允许它们操作某些硬件控制器。启动管理器作为固件的一部分负责驱动和硬件的关联。然而要有一个机制来发现连接到系统上的控制器。主机总线控制器就是在初始化时发现其它控制器的控制器。

如图 2-6 所示计算机系统可以被看成是 n 个处理器和一个核心芯片组构成。核心芯片组连出一个或多个主总线。

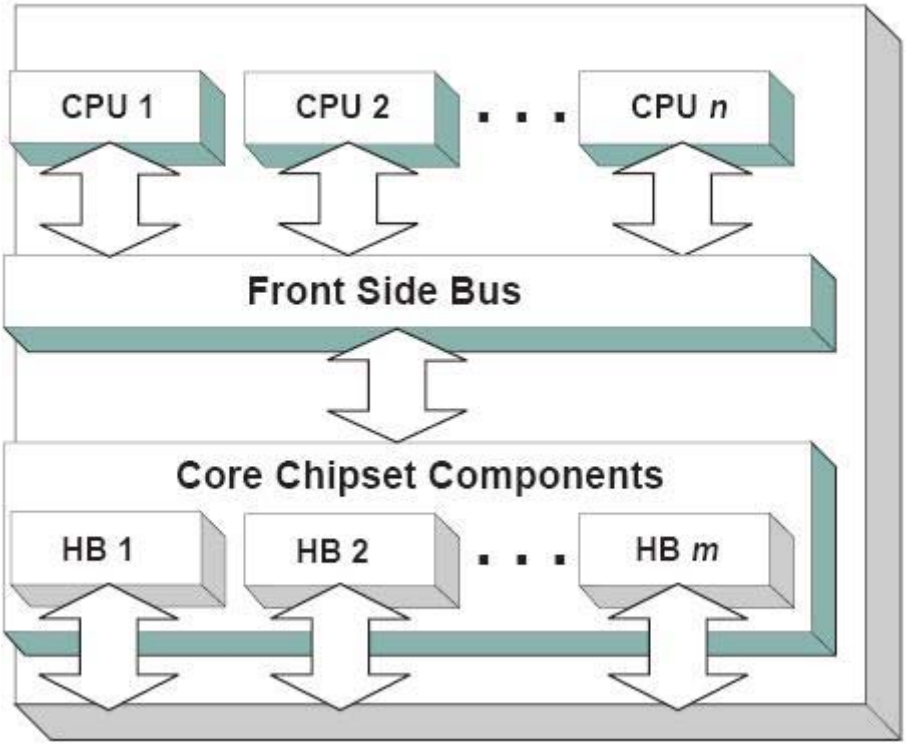


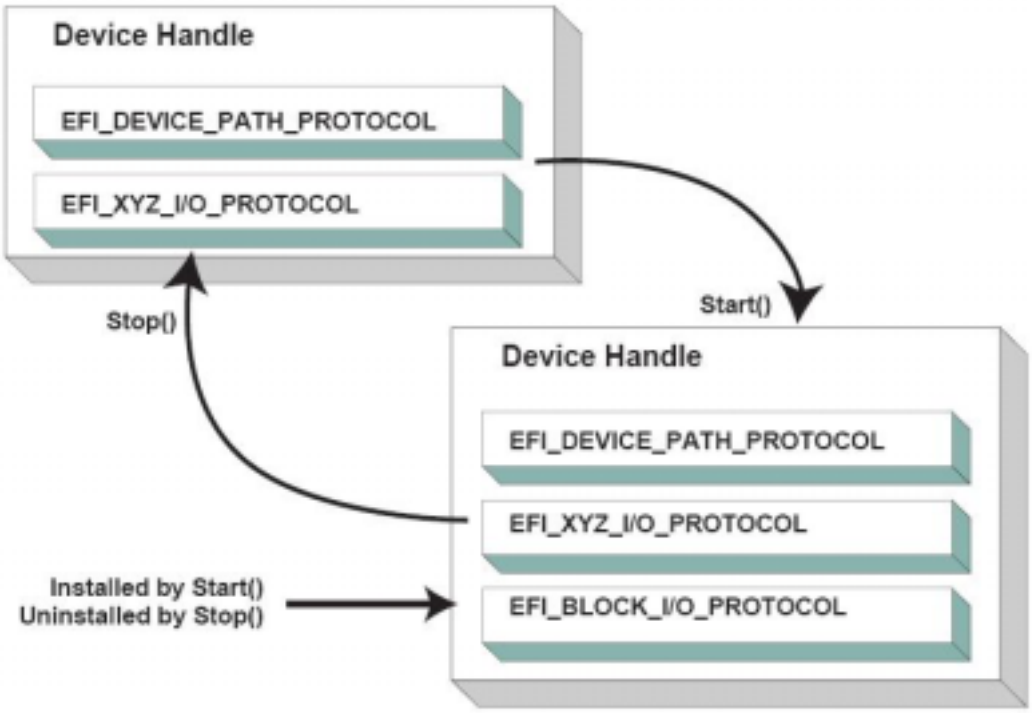
图 2-8 主总线控制器

Fig.2-8 Host Bus Controllers

每个主总线控制器在 UEFI 中都表现为一个包含 Device Path Protocol 实例的设备句柄。例如 PIC 总线驱动连接上 PCI 主桥句柄，然后为其上的每个 PCI 设备创建句柄，PIC 设备驱动之后才能够连接到这些句柄。

2.5.5 设备驱动程序

设备驱动并不能创建任何的设备句柄，而是在句柄上安装额外的协议。最一般的驱动程序会把输入输出协议附加到一个由总线驱动创建的设备句柄上。例如一些输入输出协议包含简单的文本输入输出，块的输入输出和网络协议。图 2-7 是一个设备句柄连接到一个设备驱动前后的状态对比。在这个例子中的设备句柄是 XYZ 总线的子节点，因此它包含了支持 XYZ 总线的 XYZ 输入输出协议。Device Path Protocol 不是所有设备句柄都需要安装的，只有实际连接到总线上的物理设备才需要而虚拟设备不



需要。

图 2-9 连接驱动程序

Fig.2-9 Connecting Device Drivers

连接到设备句柄的设备驱动程序必须在它的映像句柄上安装有 Driver Binding Protocol(如 2.5.1 节所述)。Driver Binding Protocol 包含三个函数：Supported()、Start()和 Stop()。Support()用来测试是否程序能支持某个设备控制器。在这个例子中，驱动程序会测试这个设备句柄能支持 Device Path Protocol 和 XYZ I/O Protocol。如果 Supported()测试通过，那么调用 Start()函数就能连接驱动和设备句柄。实际上 Start()函数只是把协议安装到设备句柄上。在这个例子中，它安装了 Block I/O Protocol。对应地，Driver Binding Protocol 的 Stop()函数使得驱动停止管理设备。它卸载任何 Start()函数安装在设备句柄上的协议。

2.5.6 总线驱动程序

从 UEFI 驱动模型的方面来说总线驱动程序和设备驱动程序是一样的。唯一的不同是总线驱动程序能从发现总线上连接的子设备控制器并创建与它们对应的设备句柄。因此，总线驱动程序明显比设备驱动更复杂，好处是简化了设备驱动的设计与实现的复杂度。总线驱动共有两类。第一类会在 Start()函数第一次调用时就把所有的

子设备控制器句柄创建出来。另一类可以允许多次调用 Start()函数来逐渐创建子设备控制器句柄。总线遍历设备的方式可以在系统启动阶段节省很多的时间。图 2-7 描述了总线控制器在调用 Start()函数前后不同的总线树形结构。进入总线控制器的虚线代表了连接到它的父总线控制的连接。如果这个总线控制器是主总线控制器，那么这根虚线就不存在。节点 A、B、C、D 和 E 代表了总线上的子控制器

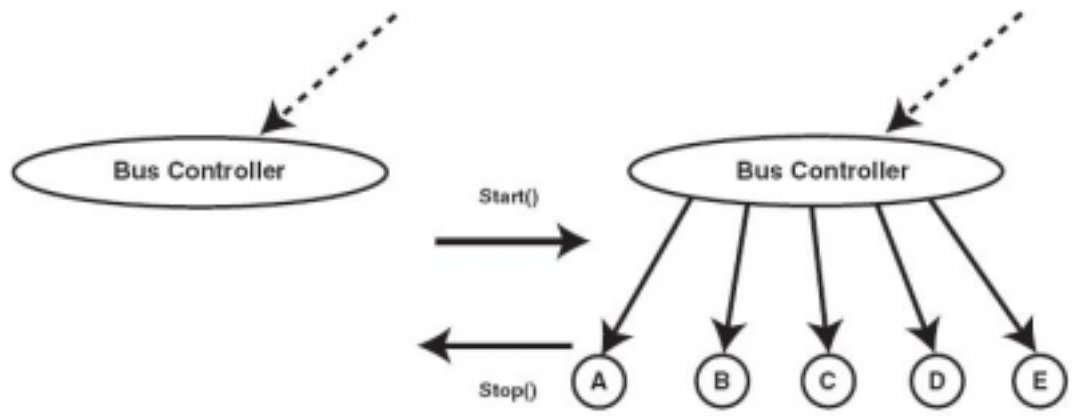


图 2-10 连接总线驱动程序

Fig.2-10 Connecting Bus Drivers

总线驱动程序必须给它创建每个子句柄安装传输协议。至少它必须给子控制器句柄安装输入输出总线服务协议。如果驱动程序创建的子句柄代表的是物理设备还需要给它安装 Device Path Protocol。总线驱动程序可能还会在子句柄上安装 Bus Specific Driver Protocol。这个协议在去子设备驱动程序连接到子设备句柄时会被用到。启动服务 ConnectController()按 UEFI 定义的方式去选择与某个设备句柄最佳匹配的驱动程序。Bus Specific Driver Override Protocol 比起 UEFI 默认的驱动搜索算法有着更高的优先级，但是比 Platform Driver Override Protocol 低。

2.5.7 驱动配置

在 UEFI 驱动模型中，建立和切断驱动和控制器之间的连接是平台固件控制的。UEFI 启动管理器中有其一般的实现，但是也可以增加其余的实现。启动服务 ConnectController()和 DisconnectController()可以让平台固件决定哪个设备控制器对应哪个驱动程序。如果平台要执行系统诊断或者安装操作系统，那它就应该把驱动和所有可能的设备做连接。如果平台要启动操作系统，它可以仅连接那些启动操作系统过程中需要用到的设备。通过启动服务 ConnectController()和 DisconnectController()，

UEFI 驱动模型可以支持这两种方式。另外，因为负责启动平台的平台部件必须要设备路径才能使同控制台设备和启动选项，所有的 UEFI 驱动模型中的服务和协议都为设备路径进行了优化。

因为平台固件可能会选择只连接启动控制台和访问启动需要的设备，操作系统提供的设备驱动不能假设某个设备的 UEFI 驱动已经运行了。系统固件和 option ROM 中的 UEFI 驱动程序并不能保证 UEFI 驱动会被加载，运行或是开始管理设备。所有操作系统的设备驱动必须能管理各种设备，不论它在 UEFI 中的驱动程序是否已经运行或起作用。

平台可以安装一个称为 Platform Driver Override Protocol 的协议。这个协议和 Bus Specific Driver Override Protocol 比较相似，只是优先级更高。这使得平台有更高的优先级去决定哪个驱动程序应该和哪个控制器连接。当 Platform Driver Override Protocol 被安装到系统中的某个句柄上之后，启动服务 ConnectController()就可以调用这个协议了。

虚拟机引导程序在运行目标操作系统之前首先调用 DisconnectController()以断开分配给目标操作系统的设备或者总线与它们的驱动程序间的连接，这时指定的设备控制器和指定总线上的字符设备控制器会脱离主操作系统的管理。随后客户操作系统会启动自身的驱动程序来管理这些设备。

2.6 本章小结

本章首先对 UEFI 进行了基础性的介绍，UEFI 是一种相对安全、功能强大、扩展性好的 BIOS 设计规范。UEFI 可以在计算机启动期间对计算机进行有效配置，和操作系统的引导。随后本章给出了在 UEFI 上运行虚拟机的一些重要功能，它们是基于现有的表的接口重用、系统分区、启动服务以及运行时服务。这些功能是引导虚拟机，实现存储器管理和设备管理的基础。

随后两节给出了 UEFI 上的虚拟机结构模型以及在 UEFI 上引导虚拟机的方法。在基于 UEFI 的 SMP 系统上实现虚拟机首先需要在 BSP 上启动 UEFI，在完成 UEFI 自身配置过程后，再通过 Shell 运行虚拟机引导程序，使虚拟机在 AP 上加载并运行。UEFI 提供了相应的库例程来实现这些功能。

最后本章还给出了系统的存储器管理和 I/O 管理的实现方法。UEFI 使用了纯页式分配的存储器管理模式。启动虚拟机首先要通过 UEFI 存储器管理器给客户操作系统分配一定的低端系统存储器，再把客户操作系统映像加载到分配好的存储器地址空间

中运行。I/O 管理主要是为了防止客户操作系统和主操作系统共享 I/O 可能出现的竞争冲突，I/O 管理给两个操作系统分配了不同的设备控制器从而有效避免了这些冲突的发生。实现客户操作系统和主操作系统在不同的地址空间上并行不悖的运行是建立基于虚拟机的数字签名服务的基本条件。

第三章 数字签名服务模型

3.1 模型概述

数字签名是指附加在数据单元上的一些数据，或是对数据单元所作的密码变换。这种数据或变换能使数据单元的接收者确认数据单元的来源和数据的完整性，并保护数据，防止被人伪造。签名机制的本质特征是该签名只通过签名者的私有信息才能产生，也就是说，一个签名者的签名只能唯一地由他自己产生。当收发双方发生争议时，第三方（仲裁机构）就能够根据消息上的数字签名来裁定这条消息是否确实由发送方发出，从而实现抗抵赖性安全服务[23]。

为了保证信息的确是发送方的原信息，没有被非法改动，可采取信息摘要算法：发送方运行一个单项函数（如 SHA 或 MD5 算法）从明文消息中产生一个固定长度的信息摘要，将其与消息一起发送给接收方。接收方收到消息后用发送方的公有密钥解密签名得到真正的摘要，同时他用原来的单项函数作用于收到的消息得到另一个信息摘要，通过比较这两条信息摘要来验证消息的完整性。其过程如图 3-1 所示：

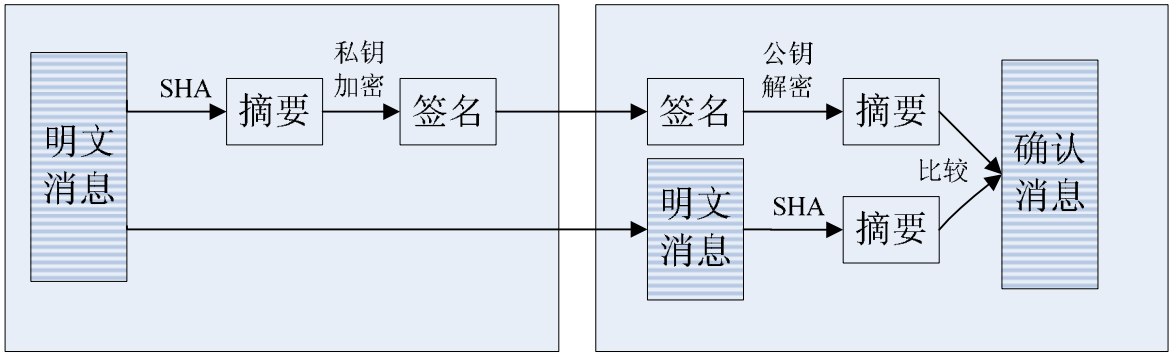


图 3-1 数字签名流程图

Fig.3-1 Digital Signature Process

3.1.1 设计目标

普通纯软件实现的数字签名方案是完全在操作系统进程中进行的，包括密钥的生成，存储和应用。尽管这样的密码系统在密钥存储中可能会采取分层加密的方式以提高用户密钥的保密性，但分层加密的保密性最终依赖的是一个根密钥，一旦此根密钥暴露，那受其保护的其它密钥也能够被逐层的解密[24]。

因此，必须把数字签名系统和操作系统从物理层或者逻辑层隔离开来才能较好的解决密钥保密性的问题。

现有的可用于数字签名的安全方案如加密机，TPM 等一般都采用物理隔离，即采用额外的硬件来使数字签名系统与操作系统保持分离[25] [26]。这些加密系统在物理结构上完全独立，有自己的运算器、控制器、存储器、I/O 端口等，并通过网络或总线与计算机系统相连。对计算机系统来说，它们就像黑盒一样，无法用通常的手段来获取其内部关键信息，如密钥等。

在基于 UEFI 虚拟机的数字签名系统中，数字签名程序在主操作系统中运行，在目标操作系统中运行的应用程序通过应用程序接口来使用数字签名服务。应用程序接口和数字签名服务间通过共享的存储区实现消息传递。由于数字签名所用密钥的存储区不在客户操作系统的访问范围内，可能存在于客户操作系统中的恶意程序也就无法获取密钥。因此，基于 UEFI 虚拟机的数字签名方案有如同加密机、TPM 等物理方案相同的安全效果。

基于 UEFI 虚拟机的数字签名方案的主要特点可以归纳为：

- 软件实现方案
- 以 UEFI 作为虚拟机主操作系统
- 密钥的安全性得到保证
- 支持多任务并行计算
- 不损耗客户机运行效率
- 主操作系统可访问网络，支持认证中心服务模式

3.1.2 系统结构

基于 UEFI 的虚拟机在设计上主要包含以下几个基本元素：

数字签名服务 数字签名的过程不再运行在目标操作系统进程中，而是运行在目标操

作系统外的主操作系统中。数字签名服务集密钥管理和各种散列加密功能于一身。运行在客户操作系统中的进程可以通过应用程序接口使用这些服务。相对目标操作系统的复杂并且开放的环境，主操作系统有一个比较封闭的环境，也更加安全。

应用系统接口 实现了客户操作系统进程与数字签名服务间的消息通信协议。应用程序可以不用关心与数字签名服务进行复杂的消息通信的过程，只需要调用简单的服务接口就能够获得签名服务结果。由于需要以物理地址操作内存单元，应用程序接口需要部分地工作在操作系统内核中。

支持 SMP 架构的虚拟机 在并行多处理(SMP)架构中，由于有多个处理器单元。除了运行主操作系统所需的一个处理器单元外，其余的处理器可以用于执行目标操作系统指令。在 SMP 架构的支持下，运行虚拟机而不需要模拟处理器。所有的指令都被处理器直接执行，指令的执行效率得到了保证。

基于共享区的消息传输引擎 以共享存储区为传输通道，实现了类似于计算机网络的跨系统的点对点消息传输。应用程序接口通过发送命令消息调用数字签名服务的函数并获取计算结果，通过发送数据消息向数字签名服务发送数据。

数字签名系统结构(如图 3-1 所示)包含：客户应用程序、应用程序接口、字符设备驱动程序、消息传输引擎、UEFI 数字签名服务。

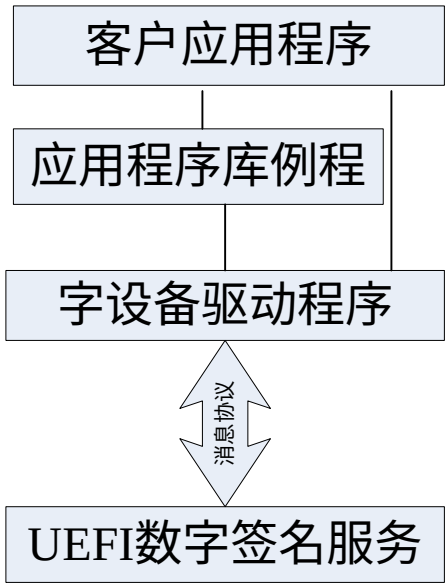


图 3-2 数字签名系统块结构图

Fig.3-2Digital Signature System Structure

3.2 数字签名服务

3.2.1 密钥管理

数字签名使用私钥加密来保证发送方的确定性并使用公钥加密来保证接受方的确定性，所用到的密钥存在产生、验证、传递、保管及销毁等过程。[27]采用公钥加密必须确定公钥持有者的真实身份，对信息发送者而言，它必须肯定用以加密的公钥是合法接收者的公钥，保证信息不被冒名者截获。对信息接收者而言，他必须确定所接受的信息的确是由意定的对方送来的，不是冒名者发来的。数字证书证明了公钥与公钥持有者真实身份之间的确定关系，是使用公钥加密的基础。一般采取的策略是提供一个可信任的第三方机构专门制造和发放证书，每个人在网络上的身份由他对应的数字证书来确定。用户自己产生密钥对，并通过 CA(Certificate Authority)认证中心获得数字证书，数字证书有计算机生成的纪录，包含有 CA 验证过的该证书的持有者的个人信息以及对应的公钥。CA 负责发放证书，并在每次通信过程中提供鉴别验证过程，但不涉及具体通信内容，其主要操作是针对用户的公钥进行的[28]。

数字签名服务密钥的产生过程如图 3-3 所示：

第一步 在 UEFI 中使用密钥生成程序产生公-私钥对。

第二步 连上 CA 取得 CA 的公钥并用这个公钥加密上面产生的公-私钥对。

第三步 将加密的公-私钥对通过网络发送给 CA。

第四步 CA 用自己的私钥解密上面发送的秘钥包，获得发送方的公-私钥对。

第五步 CA 根据发送方的身份信息和公-私钥对生成数字证书。

经过这个流程，所生成的私钥就可以存储在主操作系统内存中用于数据摘要的签名。

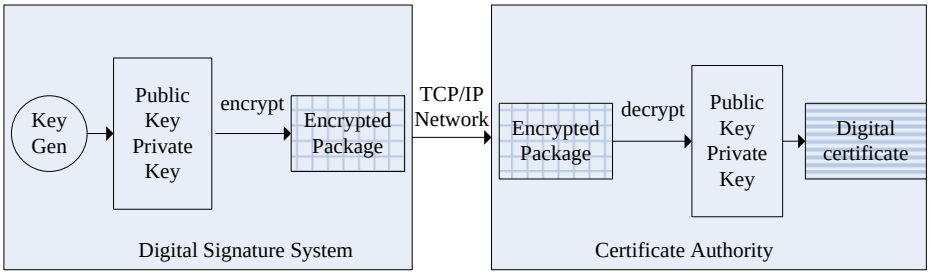


图 3-3 密钥生成流程图

Fig.3-2 Key Generation Process

使用私钥加密摘要信息保证了数据不能被冒名篡改[29]。主操作系统拥有独立的存储区保证了数字签名服务是私钥唯一使用者。因此，任何第三方都不能在有限的计算次数中伪造出另一份数据的签名，且使这份签名能够通过公钥的验证。但这还不能保证签名的数据只能被指定的接收者读取。假如明文信息同签名捆绑后直接发送，那么任何接收者都能得到明文的内容。要想保证签名接收者的唯一性，还需要使用接收者的公钥对明文和签名捆绑后的数据进行第二次加密。

保证公钥不被篡改是基于虚拟机的数字签名的另一重要作用。因为在这一方案中用于加密签名的公钥存在于独立的存储区中，任何第三方无法替换这一密钥。公钥获取的一般途径是从 CA 下载。假如没有对公钥的保护，如果第三方产生了一组公-私钥对，并用它产生的公钥替换了来自 CA 的真实公钥，那么所有的签名信息都被这个伪造的公钥加密，结果就是可以用伪造的私钥解密。最终第三方得到了明文的内容，数字签名方案失去了其本来的作用。

此外，私钥还可以写到主操作系统的非易失性存储器中。当系统下次启动时从中读取，避免了每次重复产生私钥。对于分配给主操作系统使用的设备控制器，客户操作系统是无法使用的，同时客户操作系统中的应用程序也无法读取属于主操作系统独占的非易失性存储器中的数据。

3.2.2 散列(Hash)运算引擎

签名方案几乎总是和一种非常快的公开密码 Hash 函数结合使用[30]。为 Hash 函数 $h:\{0,1\}^* \to Z$ 输入一任意长度的消息,它将返回一特定长度的消息摘要。产生的消息摘要用签名方案(P,A,K,S,V)签名，其中 $Z \subseteq P$。例如输入一长段信息 p,可以得到一小段固定长度的信息 H(p)，由于信息 p 的长度任意，而 H(p)的长度固定，因此函数 H() 是一个多对一的函数。对于同一输出，对应多种输入。

散列函数具有以下性质：

性质一：散列函数很难逆向求解，即给定 M，很难找到一段信息 $p'(p' \neq p)$，使得 $H(p') = H(p) = M$。

性质二：已知 p 和 H(p)，求 $p' \neq p$，使得 $H(p') = H(p)$ 是十分困难的。

性质三：想创建两段信息 p 和 $p'(p \neq p')$，使得 $H(p) = H(p')$ 是十分困难的。

性质四：已知 p，对任意字串 k 的情况下，求 $p'(p' \neq p)$，使得 $H(kop') = H(kop)$ 是很困难的。其中 o 表示两个字符串的串联。(例如：10011 o 011101 = 10011011101)

数字签名技术即是利用散列函数的上述性质来保证信息的完整性及不可伪造性。

主要的散列函数有 SHA,MD5 等等。

SHA-1 是一种安全散列算法，这是一个有 160 比特消息摘要的迭代 Hash 函数[31]。SHA-1 算法具体步骤如下：

第一步：填充消息。SHA-1 要求 $|x| \leq 2^{64} - 1$

$d \leftarrow (447 - |x|) \bmod 512$

$l \leftarrow |x|$ 的二进制表示，其中 $|l| = 64$

$y \leftarrow x \| 1 \| 0^d \| l$

第二步：变量初始化。初始化 5 个变量 $H_0 \sim H_5$ 为固定的常数

$H_0 \leftarrow 0x67452301$
$H_1 \leftarrow 0xEFCDAB89$
$H_2 \leftarrow 0x98BADCFE$
$H_3 \leftarrow 0x10325476$
$H_4 \leftarrow 0xC3D2E1F0$

第三步：算法的主循环。

将消息 y 分为 512 比特一组的数据 M_1~M_n 共 n 组 $y = M_1 \| M_2 \| \cdots \| M_n$

定义如下的操作 $f_0, \cdots, f_{79}$：

$$f_t(B,C,D) = \begin{cases} (B \wedge C) \vee ((\neg B) \wedge D) & 0 \leq t \leq 19 \\ B \oplus C \oplus D & 20 \leq t \leq 39 \\ (B \wedge C) \vee (B \wedge D) \vee (C \wedge D) & 40 \leq t \leq 59 \\ B \oplus C \oplus D & 60 \leq t \leq 79 \end{cases}$$

定义如下的常数 $K_0, \cdots, K_{79}$：

$$K_t = \begin{cases} 0x5A827999 & 0 \le t \le 19 \\ 0x6ED9EBA1 & 20 \le t \le 39 \\ 0x8F1BBCDC & 40 \le t \le 59 \\ 0xCA62C1D6 & 60 \le t \le 79 \end{cases}$$

进行如下循环：

for $i \leftarrow 1$ to n

{

令 Mi=$W_0 \| W_1 \| \cdots \| W_{15}$，其中每个 W_i 是一个字

for $t \leftarrow 16$ to 79

do $W_t \leftarrow ROTL^1(W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16})$

$A \leftarrow H_o$
$B \leftarrow H_1$
$C \leftarrow H_2$
$D \leftarrow H_3$
$E \leftarrow H_4$

for $t \leftarrow 0$ to 79

$$do \begin{cases} temp \leftarrow ROTL^5(A) + ft(B,C,D) + E + W_t + K_t \\ E \leftarrow D \\ D \leftarrow C \\ C \leftarrow ROTL^{30}(B) \\ B \leftarrow A \\ A \leftarrow temp \end{cases}$$

$H_0 \leftarrow H_0 + A$
$H_1 \leftarrow H_1 + B$
$H_2 \leftarrow H_2 + C$
$H_3 \leftarrow H_3 + D$
$H_4 \leftarrow H_4 + E$

}

Result = $H_0 \| H_1 \| H_2 \| H_3 \| H_4$

3.2.3 加密运算引擎

目前主要的签名方案的理论基础主要是大合数问题、离散对数问题、椭圆曲线问题、纠错码理论、有限自动机理论等。如著名的 RSA 数字签名方案的理论基础是大合数分解问题，它的安全性是建立在大合数分解因子的困难性上。ELGamal 签名方案的理论基础是离散对数问题，它的安全性是建立在求解离散对数的困难性上。DSS 是 ELGamal 签名方案的一个修改[32]。

从计算能力上来看，可将数字签名分为无条件安全的数字签名和计算上安全的数字签名。现有的数字签名大部分都是计算上安全的。诸如 RSA 数字签名、ELGamal 数字签名等等。所谓计算上安全的数字签名意指任何有足够计算能力的伪造签名者的签名。而无条件安全的数字签名的签名者和接收者都是无条件安全的。在理论上，它们在许多应用中能代替计算机上安全的数字签名，但在实际应用中是太不有效而不能被应用，这是因为在这种数字签名中需要一个复杂的交互密钥生成协议，而且签名很长。像公钥密码体制的情况一样，我们的主要用的还是计算上安全的数字签名方案。

DSA 算法是基于离散对数问题的数字签名算法[33]。DSA 的完整描述如下：

设 p 是长 L 比特的素数，在 Z_p 上其离散对数问题是难处理的，其中 L=0(mod64)且 $512 \le L \le 1024$，q 是能被 p-1 整除的 160 比特的素数。设 $\alpha \in Z_P^*$ 是 1 模 p 的 q 次根。

设 P={0,1}*，A= $Z_q^* \times Z_q^*$，并定义

$$K = \{(p, q, \alpha, a, \beta) : \beta \equiv \alpha^a (\mathrm{mod}\ p)\}$$

其中 $1 \le k \le q-1$，定义 $sig_k(x,k) = (\gamma, \delta)$，其中

$$\gamma = (\alpha^k \bmod p) \bmod q$$

$$\delta = (SHA1(x) + a\gamma)k^{-1} \bmod q$$

(如果 $\gamma = 0$ 或 $\delta = 0$，应该为 k 另选一个随机数)。

对于 $x \in \{0,1\}^*$ 和 $\gamma, \delta \in Z_q^*$，验证是通过下面的计算完成的：

$$e_1 = SHA-1(x)\delta^{-1} \bmod q$$

$$e_2 = \gamma\delta^{-1} \bmod q$$

$$ver_k(x,(\gamma,\delta)) = true \Leftrightarrow (\alpha^{e_1}\beta^{e_2} \bmod p) \bmod q = \gamma$$

例如 q=101，p=78q+1=7879。因为 3 是 Z_{7879}^* 中的一个本原元，因此我们取 α 是模 p 同余 1 的 q 次根。$\alpha = 3^{78} \bmod 7879 = 170$ 假设 a=75，那么 $\beta = \alpha^a \bmod 7879$ =4567 假设选择随机值 k=50，然后可以计算

$$k^{-1} \bmod 101 = 50^{-1} \bmod 101 = 99$$

$$\gamma = (170^{50} \bmod 7879) \bmod 101$$

= 2518 mod 101

= 94

$$\delta = (22 + 75 * 94)99 \bmod 101$$

= 97

对消息摘要 22 的签名(94,97)通过下面的计算加以验证：

$$\delta^{-1} = 97^{-1} \bmod 101 = 25$$

$$e_1 = 22 \times 25 \bmod 101 = 45$$

$$e_2 = 94 \times 25 \bmod 101 = 27$$

$$(170^{45} 4567^{27} \bmod 7879) \bmod 101 = 2518 \bmod 101 = 94$$

3.3 应用系统支持

3.3.1 字符设备驱动程序

字符设备驱动程序为客户操作系统中的应用程序与主操作系统的通信建立了通道[34]。

在 Linux 操作系统中，应用程序工作在用户态并且使用虚地址访问存储器。在应用程序中，一个指定的虚拟存储器地址经过内存管理处理器(MMU)的转换，可以转变为一个实际的物理地址。如图 3-4 所示，虚地址为 8200 的内存单元被映射到了实际物理地址为 24584 的内存单元。

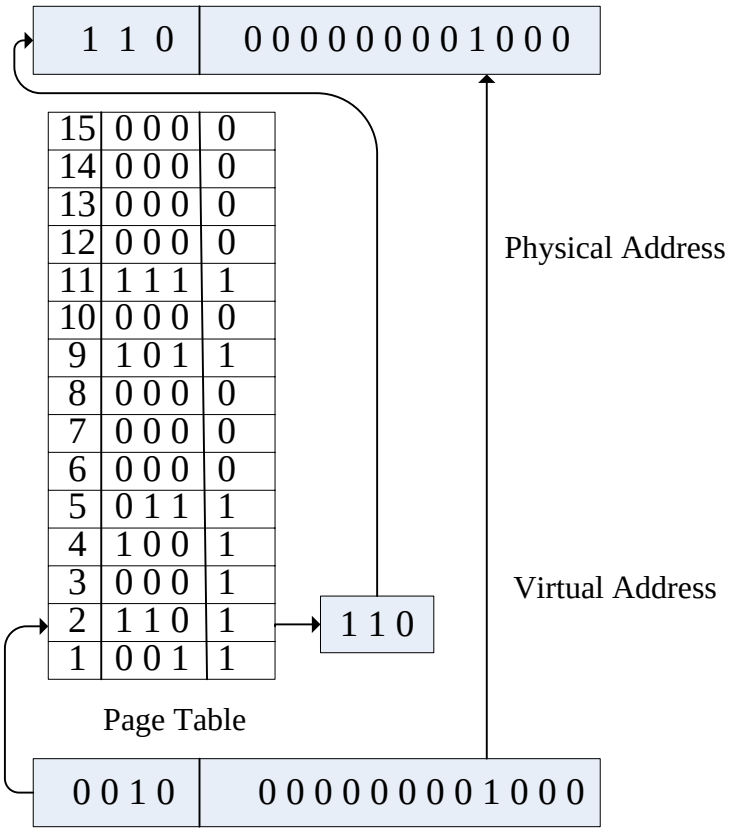


图 3-4 虚实地址转换图

Fig.3-4 Virtual Memory to Physical Memory Conversion

为了保护操作系统内核，操作系统不允许应用程序直接访问物理内存，而总是要求经过这样的转换才能访问相应的存储器单元[35]。

但是在我们的系统间通信模型中，可用共享交换区是一块连续的物理内存。一般来说，这块存储区在客户操作系统和主操作系统中会分别映射到不同的虚地址上，并且虚地址不一定连续。因此，必须找到一个能够让应用程序调用，又能够以物理地址访问存储器空间的中间层。

Linux 操作系统的 I/O 软件按照分层模型设计，低层的软件用来屏蔽硬件的具体细节，高层软件主要为用户提供一个简洁、规范的界面。

Linux 的 I/O 处理可以这么分为 4 层[36]：

1. 中断处理程序(底层)
2. 设备驱动程序

3. 与设备无关的操作系统软件
4. 用户层软件(高层)

中断处理程序 执行I/O操作的进程在等待返回结果时通常需要将自身进程挂起，以减少不必要的执行时间。而中断处理程序会在 I/O 操作结束并发生中断时唤醒挂起的 I/O 进程。进程自己阻塞的方法有：执行信号量 Down 操作；条件变量的 Wait 操作；或者信息的 Receive 操作。当中断发生时，中断处理程序执行相应的操作，以解除相应进程的阻塞状态。其中包括对信号量的 UP 操作，在管程中对条件变量执行 SIGNAL 操作或是向阻塞进程发一条消息。总之，其作用是将刚才被阻塞的进程恢复执行。

设备驱动程序 设备驱动程序中包括了所有与设备相关的代码。每个设备驱动程序只处理一种设备，或者一类紧密相关的设备。我们知道每个设备控制器都有一个或多个寄存器来接受命令。设备驱动程序发出这些命令并对其进行检查，因此操作系统中只有硬盘驱动程序才知道磁盘控制器有多少寄存器，以及怎么使用这些寄存器来控制磁头移动以正确的读取数据。一般而言，设备驱动程序的功能是从与设备无关的软件中接收抽象的请求，并执行之。设备驱动程序所执行的操作可分为两类，阻塞式和非阻塞式。在阻塞式的操作中驱动程序要等待控制器完成一些操作所以先将驱动程序阻塞，等待中断信号到达才解除阻塞。而非阻塞式的操作几乎没有延迟，因此无需阻塞。例如读写控制器寄存器。I/O 设备可粗略的分为块设备(block device)和字符设备(character device)。块设备的主要特征是能够独立地读写单个的数据块，例如磁盘。而字符设备可以发送或接收一个字符流。字符设备无法编址和寻址，如网络接口，鼠标等。

与硬件无关的 I/O 软件 在 Linux 中，多数设备无关软件属于文件系统。设备无关软件的基本功能是执行适用于所有设备的常用 I/O 功能，并向用户层软件提供一个一致的接口。通过设备无关 I/O 软件设备名称可以映射到相应的驱动程序。在 Linux 中一个设备名，如/dec/tty00 唯一的确定了一个 i 节点，其中包含主设备号(major device number)，通过主设备号就可以找到相应的设备驱动程序。i 节点还包含此设备号(minor device number)，它作为参数表示指定的某个具体设备。其次与硬件无关的 I/O 软件还提供了设备访问控制，独立于设备的块大小，数据缓冲，独占设备的分配和释放等功能。

用户空间的 I/O 软件 用户空间的 I/O 软件指不属于操作系统与用户程序链接在一起的库例程，甚至是在内核外的完整的程序。系统调用，包括 I/O 系统调用通常先

是库例程调用。如以下的 C 语句 count = write(fd, buffer, nbytes)；此类库例程的主要工作是给相应的系统调用提供参数并调用之。

如图 3-5 所示，设备驱动程序工作在内核态，因此允许使用物理地址访问存储器空间。另一方面，设备驱动程序又能够处理用户进程传来的服务请求。例如，用户空间的 I/O 软件可以将数字签名指令与数据通过操作系统中的设备无关 I/O 层写入一个特定的字符设备，这时字符设备驱动程序会将指令与数据按照一定的协议写入物理内存共享区。当用户读取字符设备时，它的驱动程序又能去共享区读取数字签名计算结果并返回给用户空间的 I/O 软件。

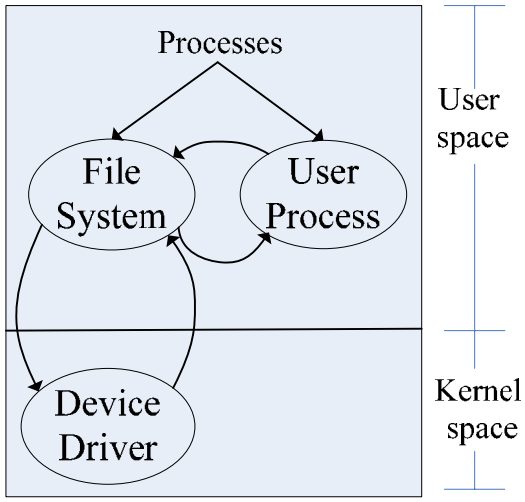


图 3-5 操作系统输入输出框架图

Fig.3-5 Operating System I/O Framework

为了使字符设备驱动程序能够工作，必须经过以下两个步骤

1. 创建字符设备文件

以 root 身份登陆 Linux 后执行命令 mknod /dev/memdev c 181 0

这一命令能够创建文件名为/dev/memdev 的字符设备文件，并且指定设备的主设备号为 181，次设备号为 0

2. 安装字符设备驱动程序内核模块

以 root 身份登陆 Linux 后执行命令 insmod memod.ko

这一命令将安装包含字符设备驱动程序的内核模块。

可装卸的内核模块(kernel module)是 Linux 设备驱动程序的一种运行载体。它是 Linux 操作系统内核的一种扩展机制，它允许操作系统在运行时把代码加入到内核运

行，以扩展内核功能。Linux 内核要求可扩展内核模块中必须有两个特定的接口函数 module_init 和 module_exit。它们分别在模块加载和卸载时被执行。module_init 中一般执行模块初始化操作，例如变量初始化，注册回调函数等。module_exit 中一般执行清理工作，例如销毁堆中分配的内存，卸载回调函数等。

在数字签名系统的字符设备驱动程序的 module_init 函数中，我们需要通过以下函数调用在字符设备上注册特定的接口函数。

```
int major = register_chrdev(MK_POINT, DEVICE_NAME,&fops);
```

register_chrdev 的作用是把主设备号为 MK_POINT 名称为 DEVICE_NAME 的特殊设备文件和它的接口函数结构 fops 连接起来。fops 的定义如下：

```
static struct file_operations fops={
    read: device_read,
    write: device_write,
    open: device_open,
    release: device_release,
 };
```

当用户打开、关闭或是读写该设备时，接口结构中对应的函数就会被调用。这些函数完全是在字符设备驱动程序中定义并实现的。

例如其中的写函数在驱动程序中是这样定义的

```
static ssize_t device_write(struct file * filp, const char *buf,size_t len,loff_t *off)
{
    char temp[len];
    copy_from_user(temp,buf,len);
    map<struct file*, Messenger *>::iterator m_iter = msgMap.find(filp);
    if(m_iter == msgMap.end())
       return 0;
    Messenger *msger = m_iter->second;
    MemMessage *msg  = new  MemDataMessage((byte*)temp,len);
    Msger->SendMessage(msg);
}
```

因为函数的入口参数 buf 指针是一个用户区指针，它的值是所指向内存单元的虚地址，而内核中的指针的值应该是所指内存单元的物理地址，因此首先要调用函数 copy_from_user()把数据从用户区拷贝到内核区。再调用系统间消息引擎 Messenger::SendMessage()把消息转送给主操作系统中运行的数字签名服务。其中 Messenger 类实现了客户操作系统和主操作系统间的消息传递协议。

综上所述，字符设备驱动程序作为操作系统内核服务程序，实现了用户空间和内核空间的数据转发，为客户操作系统的用户层软件和主操作系统的数字签名服务建立了消息传递通道。

3.3.2 应用程序库例程

字符设备驱动给了用户访问数字签名服务的通道，但应用程序必须十分清楚各种命令格式和消息格式。这给应用程序的编写带来了不便。

应用程序库例程的作用就是把应用的编写从各种复杂的命令、消息格式中解脱出来。通过隐藏底层协议，应用程序只需要调用库例程中简单且有意义的函数，就能够完成数字签名过程。库例程作为用户空间的 I/O 软件，一般和应用程序链接在一起。

例如：用于 DSA 数字签名的函数声明如下：

```
Public void BeginSignDSA();                              //开始 DSA 计算
Public void FeedData(byte* source, UINT length );        //提交明文数据
Public void EndSignDSA(byte* result, UINT &signLen); //结束计算，获取结果
```

其中 BeginSignDSA()的定义及实现如下：

```
Public void BeginSignDSA()
{
    dev_fnod = open("/dev/memdev", O_CREAT|O_WRONLY, 0);
    CmdMessage msg;
    msg.classID = ClassID.CMD;
    msg.subID = SubID.Begin;
    msg.param = SigFunction.DSA;
    write(dev_fnod,&msg,sizeof(msg));
}
```

BeginSignDSA 函数首先以写方式打开了字符设备“ /dev/memdev ”，建立一个命令消息 msg，再设置命令消息 msg 的主 ID,子 ID 以及操作码，随后将消息对象写入字符设备。之后文件系统会启动字符设备驱动程序，并把消息对象传给驱动程序的 device_wirte 函数处理。

如图 3-6 所示，使用库例程进行一次完整的 DSA 数字签名过程包括如下步骤：

1. 调用 BeginSignDSA()在主操作系统的数字签名服务中启动一个 DSA 签名请求
2. 若干次的调用 FeedData 函数把需要签名的数据提交到数字签名服务
3. 调用 EndSignDSA 函数取得签名结果。

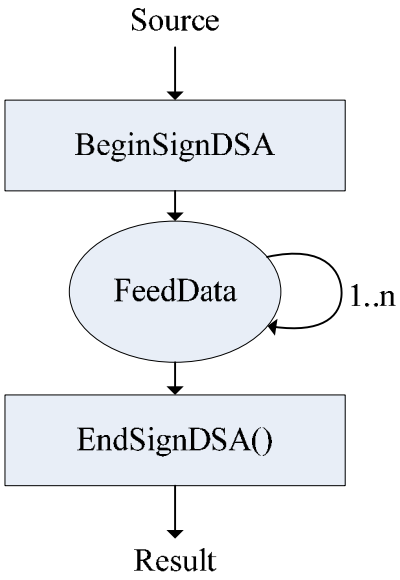


图 3-6 使用库例程进行 DSA 数字签名

Fig.3-6 DSA signature in library functions

例如，需要对一个 1024 个字节的明文数据 data 作 DSA 签名，可以这样来做：

```
Byte result[MaxSize];
UINT result_size;
BeginSignDSA();
FeedData(data,1024);
EndSignDSA(result, result_size);
```

注 :如果明文数据大小必须多次获取的,那么也可以使用 FeedData 多次提交数据。

3.4 消息传输机制

消息传输机制解决了运行在虚拟机客户操作系统中的应用程序和运行在主操作系统中的数字签名服务间的数据通信问题。

如图 3-7 所示，消息传输机制按功能可分为 3 层，分别是：存储层、传输层和应用层。在每一层上都定义了相应的协议，以使得数据可以正确、高效地在通信的双方间传输。

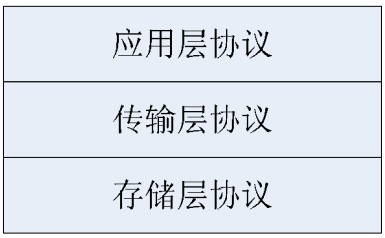


图 3-7 消息传输协议栈

Fig.3-7 Transmission Protocol Stack

如图 3-8 所示，协议栈在客户操作系统和主操作系统中的实现和分布方式各有不同。

首先是协议分布上的不同。在主操作系统中，UEFI 驱动程序实现了存储层协议和传输层协议，而数字签名服务程序实现了数字签名的应用层协议。这样做的好处是相同的 UEFI 驱动程序可以服务不同的应用，独立实现的传输层和存储层协议还有助于减少应用程序的复杂度。应该知道 BIOS 中的存储空间是非常珍贵的，而代码复用有助于减少代码的总长度。在客户操作系统中，字符设备驱动程序实现了存储层协议和传输层协议，而应用程序库例程实现了应用层协议。除了基于与主操作系统相同的设计上的考虑外，另一方面是 Linux 设备驱动程序工作在操作系统内核里，允许使用物理地址访问存储器。而后者对于实现基于物理内存共享区的存储层协议是必不可少的条件。

其次，在实现协议所用的语言方面。主操作系统中的协议栈由于开发环境的限制必须由 C 语言实现。而客户操作系统中的协议栈可以使用标准 C++语言实现。

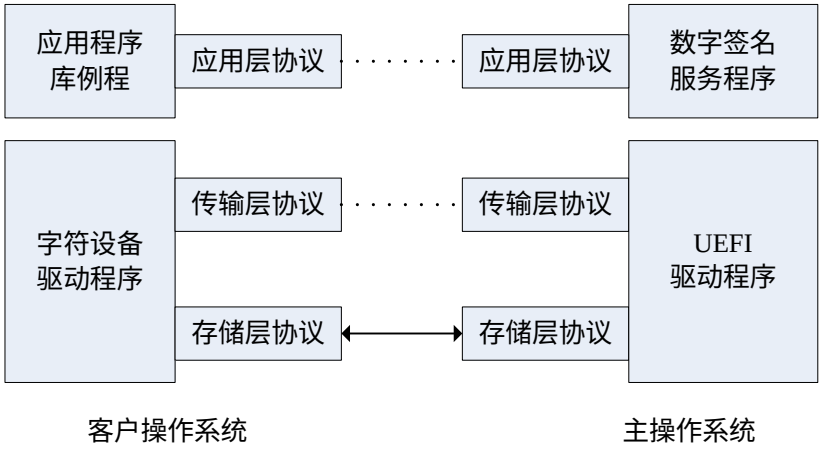


图 3-8 消息传输协议栈系统分布图

Fig.3-8 Transmission Protocol System Distribution

3.4.1 存储层协议

存储层工作在协议栈的最底层，在这里没有任何消息的概念。对于协议的实现者而言，它们所面对的仅仅是一块大小为一定字节的连续的存储区。由于客户操作系统和主操作系统都能够独立地访问这块区域，同时又有通信的需求，才需要有一个规范来描述存储区的状态，并且使得主操作系统和客户操作系统可以并行地对共享存储区进行操作。

因此，存储层协议的设计目标有两个：

1. 统一主操作系统和客户操作系统对存储层中数据对象的理解。
2. 防止多处理器并行操作可能带来的死锁以及竞争冲突。

3.4.1.1 基本管理协议

在最初的存储器中没有并没有变量或是对象的概念，所有的数据只是连续的字节序列。因此，首先要做的是对这块存储区进行管理，以使变量和对象可以在其中进行分配和释放。

一个常用的方法是自由链表法[37]。自由链表法维持的是一个关于分配和尚空闲内存段的链表。在这里，一个段或者是一个对象或者是两个对象间的一个空洞。图 3-8 表示了一块包含了 4 个数据对象和 4 个空洞的存储区的分配情况。其中 P 表示段为已分配空间，H 表示段为空闲空间。P/H 后跟随的数字表示数据块的字节数。P/H 标记和段长度所占的空间称为段头部。已知某个数据段的地址求它的下一数据段块地址，可以按照段地址+数据长度+段头部长度计算得到。

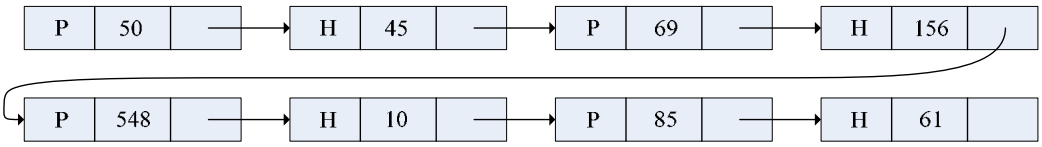


图 3-9 一个包含 4 个对象和 4 个空洞的内存区域

Fig.3-9 A memory block contains 4 objects and 4 holes

在分配和释放对象时需要更新链表。一个待释放的段一般有两个邻居(除非是在存

区的最低端或者最高端)，它们有可能是对象段也可能是空洞段，这时候如果相邻块是空洞，则需要合并两个或者三个空洞为一个大的空洞。为了提高释放空间操作的速度(释放空间时需要访问上一个段以确定它是否是空洞)，最好将链表做成双链。当链表是双链时释放操作的时间复杂度将从 O(n)级降为 O(1)级，n 表示整个空间中段的数目。 但是每一段需要增加一块空间来存储逆向指针。

因为分配操作仅需要遍历类型为空洞的段，所以为了提高分配空间时的速度，可以将空洞段单独形成一链。在平均情况下，即空白段和数据段均匀分布时，这一改动大约能节省一半的时间；而在空白段集中在空间的尾部时，这一改动能将分配操作的时间复杂度从 O(n)级降为 O(1)级。作为代价需要在每一段的头部保留下一空白段的指针。

3.4.1.2 协议中的锁机制

除了基本的共享区管理协议，另一个需要考虑的问题是并行多处理结构下，多 CPU 间的操作互斥问题。因为在本文讨论的虚拟机模型中，有两个或者更多的 CPU 可以同时访问这块共享区。内存的分配和释放操作可能需要增加、删除或是修改内存分配链的节点。假如不加限制，在两个 CPU 同时进行内存分配或者释放时，就极有可能导致分配链失效。假如当前分配链的状态如图 3-8 所示，而 CPU0 正在释放第三个数据段，即长度为 69 的数据段 sector01。同时 CPU1 要分配一个长度为 8 的数据段。CPU0 先运行，在释放数据段 sector01 后，发现它的前一个数据段也是空白段，即长度为 45 的空白段 sector02。根据内存释放算法 sector01 应当与 sector02 和并成为一个新的空白段 sector03。Sector03 的长度应该是 sector01 的长度和 sector02 的长度的和即 114。正当 CPU0 准备修改 sector02 的长度为 114 时，CPU1 发现在 sector02 上可以分配出长度为 8 的数据段，随后修改了 sector02 的属性为数据段，并把它的长度修改为 8。这时 CPU0 才把 sector02 的长度改为 114。这一操作会覆盖刚才分配的长度，并造成不可预测的结果。

可以正确处理以上用例的做法是，在 CPU0 完成了 sector01 的释放操作后，CPU1 才开始进行分配操作。为了保证存储器的分配和释放能正常工作，我们需要设置临界区并将共享区的空间的分配和释放操作放在临界区中，以次这些操作是互斥且不可重入的。

在 Linux 中，设置临界区的方法有很多，如信号量，管道，消息等。但这些手段只对 Linux 内运行的进程或者线程有效，而 UEFI 无法识别和使用 Linux 中这些设置

临界区的手段[38]。

为了解决 SMP 系统中的临界区设置问题，需要引入自旋锁(Spin Locks)[39]。自旋锁是为了实现保护共享资源而提出的一种锁机制。其实，自旋锁与互斥锁比较类似，它们都是为了解决对某项资源的互斥使用。无论是互斥锁还是自旋锁，在任何时刻最多只能有一个保持者，即是在任何时刻最多只能有一个执行单元获得锁。但是两者在调度机制上略有不同。对于互斥锁，如果资源已经被占用，资源申请者只能进入睡眠状体。但是自旋锁不会引起调用者睡眠，如果自旋锁已经被别的执行单元保持，调用者就一直在那里看是否该自旋锁的保持者已经释放了锁，“ 自旋 ”一词因此得名。

从自旋锁的工作方式可以看出，它是一种比较低级或原始的保护数据结构或代码片段的方式，这种锁可能存在两个问题：

- 死锁。试图递归地获得自旋锁：递归程序的持有实例在第二个实例循环，以试图获得相同的自旋锁时，将永远不会释放此自旋锁。在递归程序中使用自旋锁应遵守下列策略：递归程序决不能在持有自旋锁的时候调用自己，也决不能在递归调用时试图获得相同的自旋锁。此外如果一个进程已经将资源锁定，那么即使其它申请这个资源的进程不停地试图获得锁，也永远不会成功，从而进入死循环。
- 浪费 CPU 资源。如果不加限制，由于申请者在一直循环等待，因此自旋锁在锁定的时候，如果不成功，不会睡眠而会继续尝试。在单 CPU 的时候自旋锁会让其它进程得不到运行时间。因此，一般在实现自旋锁时会有一个参数限定持续尝试的最大次数。超出后，自旋锁放弃尝试，等待下一次机会。

由此可见，自旋锁比较适用于锁使用者保持锁时间比较短的情况。正是由于自旋锁使用者一般保持锁的时间非常短，因此选择自旋而不是睡眠是非常必要的，自旋锁的效率远高于互斥锁。信号量和读写信号量适合于保持时间较长的情况，它们会导致调用者睡眠，因此只能在进程上下文使用，而自旋锁适合于保持时间非常短的情况，它可以在任何上下文使用。自旋锁保持期间是抢占失效的，而信号量在保持期间是可以被抢占的。因此，自旋锁只有在内核可抢占或 SMP(多处理器)的情况下才真正需要。

在 SMP 系统中，对“ 锁 ”的要求和单处理器时有所不同。在单处理器环境中可以使用 SWAP(Bit Swap)指令或者 BTS(Bit Test and Set)指令实现进程互斥(SWAP 指令：交换两个内存单元的内容；BTS 指令：取出某一内存单元的值再给该单元赋一个新值)。这些指令设计对同一存储单元的两次或两次以上的操作，这些操作将在几个指令周期内完成，但由于中断只能发生在两条机器指令之间，而同一指令内的多个指令

周期不可中断，从而保证 SWAP 指令或 BTS 指令的执行不会交叉进行。

但在多处理机环境中情况有所不同，例如 BTS 指令包括“取”、“送”两个指令周期，两个 CPU 执行 BTS(&lock)可能发生指令周期上的交叉。如图 3-9 所示 lock 初始为 0,CPU1 和 CPU2 可能分别执行完前一个指令周期并通过检测(均为 0)，然后分别执行后一个指令周期将 lock 设置为 1，结果都取回 0 作为判断临界区空闲的依据，从而不能实现互斥。

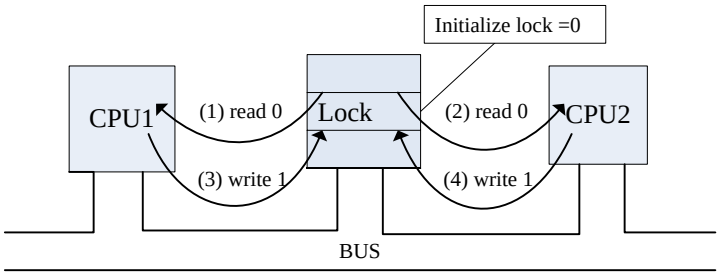


图 3-10 双 CPU 同时执行 BTS 指令时一种可能情况

Fig.3-10 A Possible Status when Execute BTS Instruction with dual CPUs

为在多 SMP 系统中利用 BTS 指令实现进程互斥，还需要硬件的进一步支持，以保证 BTS 指令执行的原子性。这种支持目前多以“锁总线” (Bus Locking)的形式提供。由于 BTS 指令对内存的两次操作都需要经过总线，在执行 BTS 指令前锁住总线，即可保证 BTS 指令执行的原子性，在 BTS 指令执行完后锁自动解开。使用锁总线机制后，进出临界区的算法如下：

```
enter_region:
        lock                                    //锁总线
        bts register, lock                      //复制 lock 到寄存器，并将 lock 置 1
        cmp register, #0                        //lock 是否等于 0
        jne enter_region                        //如果不等于 0，已上锁，再次尝试
        ret                                     //成功进入，返回调用程序

leave_region:
        move lock, #0                           //置 lock 为 0
        ret                                     //成功退出，返回调用程序
```

自旋锁是 SMP 系统中相当有效的机制，而在单处理器非抢占式的系统中基本没有作用。自旋锁在 SMP 系统中应用得相当普遍。在本文所论述的虚拟机数字千名方案中，自旋锁能够保证一次只能允许目标操作系统或主操作系统中的一者进入共享存储

区的分配或释放操作，从而保证了共享存储区管理协议的正确运行。

3.4.2 传输层协议

传输层协议提供的是客户操作系统和主操作系统间的数据包传输服务。利用存储层协议，客户操作系统和主操作系统已经可以准确地理解共享区中的结构与对象，但是它们还缺少使用这些共享区中的对象进行传输数据的方法。

因此，传输层协议的设计目标是：

1. 提供一种面向连接的传输协议，解决连接的建立与释放问题
2. 提供一个带宽高，并发度好的数据传输协议，解决数据包的发送与接收问题
3. 防止多处理器并行操作可能带来的竞争冲突。

3.4.2.1 数据结构与接口

报文

报文是传输层中数据传输的基本单位。按照功能的不同可以分为两类：数据报文和控制报文。

数据报文是一类携带数据的报文，传输的是上层用户交与传输层传递的信息

控制报文是一类传输控制信息的报文，用于连接的建立与释放

两类报文的结构如图所示

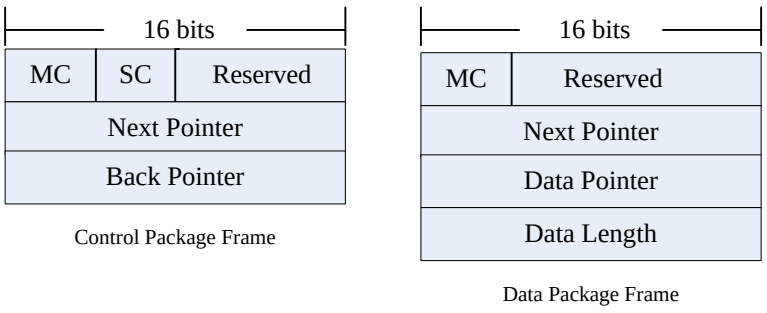


图 3-11 控制报文和数据报文结构图

Fig.3-11 structure of control package and data package

MC：主类号，表示报文的主要类型，可以为数据报文或控制报文。

SC：次类号，表示报文的次要类型，和主类号一起构成消息的完整类型信息。数据

报文的次类号没有作定义，而控制报文的次类号可以表示连接初始报文或连接终止报文。

Next Pointer：指向下一报文的指针的地址。当报文在共享区中被构建起来的同时，一个指向下一报文的指针也会在共享区中被创建，这个指针地址作为消息的一部分传递给报文的接收者，作为下一个消息的发送地址。

Back Pointer：指向报文接收指针的地址，这一地址仅存在于建立连接请求报文中。当连接请求者欲建立连接时，应当预先在共享区创建指向接收报文的指针，在把这一指针的地址作为消息的一部分传递给接收方。接受方可以根据这个地址来回送报文。

Data Pointer：共享区中的字节数组指针，在这个数组中存放了待传输的数据。在报文被发送前，这个数组会首先被建立起来并且填充好数据，最后将数组的地址填入报文结构中。报文接收方再根据这个地址在共享区中取得这些数据。

Data Length：仅表示字节数组的长度，由于共享区中的数组不具有边界信息，这个长度提供给报文接收方确定数组边界的信息。

套接字

套接字作为协议的实现者，具有建立连接、释放连接、发送报文和接收报文的功能。套接字的接口定义如下：

```
MemSocket Functions:

        void Send (MemPackage * msg);       //Send a package
        MemPackage* Receive();              //Receive a package
        static MemSocket* Accept();          //Response connection request
        void Connect();                      //Try to establish a connection
        void Disconnect();                   //Try to disconnect a connection
        bool Connected();                    //Check connection status
```

如图 3-10 所示，在传输层中，套接字在开始传输报文前必须首先建立连接。客户端套接字和服务器端套接字建立连接的方式有所不同。客户端套接字能够通过调用 Connect()函数主动与服务器套接字建立连接。服务器端套接字通过调用 Accept()函数侦测客户端发来的连接请求，如有未处理的连接请求则创建一个新的套接字与客户端建立连接，否则返回空值表示未发现连接请求。

已连接的套接字之间是对等的关系，它们可以通过调用 Send ()函数来发送报文，通过调用 Receive ()函数来接收报文。

一旦通信完成，通信双方中的任意一方都可以调用 Disconnect()函数断开连接。

Disconnect()函数会向对方套接字发送连接断开的控制报文，另一方收到连接断开报文后可以进行一些资源释放的工作。断开后的套接字还能够再次调用 Coneect()函数恢复连接。

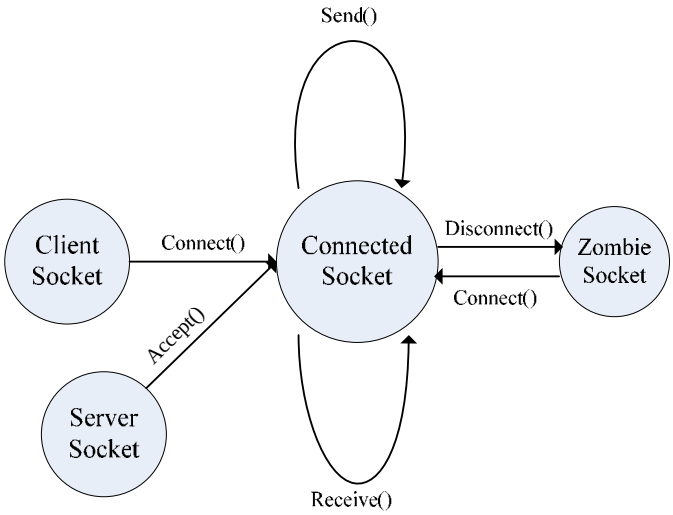


图 3-12 套接字生存期活动图

Fig.3-12 Activities in socket life time

客户端程序示例：

```
void Client()
{
    MemSocket socket();
    socket.Connect();
    MemDataPackage *package = new MemDataPackage(“message1”);
    socket.SendPackage(package);
    socket.Disconnect();
}
```

服务器端程序示例：

```
void Server()
{
    MemSocket * socket = null;
    while (socket == null)
    {
        socket = MemSocket.Accept();
        Sleep(100);
    }
```

```
    MemDataPackage *package;
    while (socket.Connected())
    {
        package = socket.ReceivePackage();
        if (package!=null)
        {
            //do something with the package;
        }
    }
}
```

3.4.2.2 协议的实现

同步传输与异步传输

一个简单的同步传输协议：

在这里假设参与传输的两方分别为 A 和 B。协议的执行过程为：

1. A 在共享区建立新的报文
2. A 通知 B 有新的报文到来
3. B 接收报文
4. B 通知 A 报文已经被接收
5. A 成功返回

在同步传输时，报文的收发要经历建立报文->发送报文->等待接收->完成接收->成功退出的几个过程。发送者在发送报文后必须循环等待报文被接收，并且只有收到报文被接收的信号才能退出。

考虑一个简单的异步传输协议：

1. A 在共享区建立新的报文
2. A 通知 B 有新的报文到来并成功返回
3. B 接受报文并返回

在异步传输时，发送者在发送报文后不再确认报文被接收而是直接退出。

从效率上考虑，同步传输协议的报文发送者必须要等待报文被接收才能退出，等待的时间里处理器只是在不断地检查是否有报文被接收的信号到来，而无法转而执行其它代码。异步传输协议将报文发送出去后立即返回，CPU 转而执行有效的代码例如发送下一个报文等等。考虑操作各步所花费的时间，假设在共享区建立一个报文需

要时间为 t_0，从共享区读取一个报文需要时间为 t_1，发送一个信号需要时间为 t_2，平均信号响应时间为 t_3。由于报文在存储层传输实际所花时间为 0，所以在平均情况下(发送报文的长度取平均值)，同步传输发送并接收一次报文使用所需时间为：

$$(step1)t_0+(step2)t_2+t_3+(step3)t_1+(step4)t_2+(step5)t_3=t_0+t_1+(t_2+t_3)\times 2$$

异步传输发送一次报文所需时间为：

$$(step1)t_0+(step2)t_2=t_0+t_2$$

接收一次报文所需时间为

$$(step3)t1+t3$$

在 SMP 系统中，工作在不同 CPU 上的收发双方可以并行操作。因而在传输大量报文时，异步传输的速度取决于工作速度较低的一方。因此报文的收发时间为

$$Max(t_0+t_2,t_1+t3)$$

异步传输比同步传输发送报文所用的时间少 $\mathrm{Min}(t_1+t_2+2t_3,t_0+2t_2+t_3)$，因此在效率上显然异步传输更高。受系统负载的影响，当计算负载变大时，信号的相应时间 t_3 也可能随之增加。对于异步传输，发送报文的速度受响应延迟影响较小，而同步传输则受此影响较大。

从空间使用率上考虑，同步传输采用的是发送-接受-再发送的策略，每个连接任意时刻最多仅有一个报文处于传输阶段–即存于共享区中。而异步传输采用的是 n 次发送与 n 次接收过程互相交叉的策略，因此每个链接任意时刻可能有 n 个报文存于共享区中(在连接中处于共享区的报文的总长度必须小于共享区的长度，平均情况下由于响应延时，每个连接有 N 个报文处于共享区中 N>=1)。假设报文平均长度为 P，共享区总长度为 L。在连接数为 M 的情况下，同步传输的平均空间使用率为 MP/L，而异步传输的平均空间使用率为 MNP/L。因此异步传输的空间使用率也大于同步传输。受系统负载的影响，当计算负载较大时，响应延时也可能会增加，在异步传输时，处于共享区中的报文个数 N 就会变大，异步传输的空间使用率也会进一步增加，而同步传输的空间使用率不随系统响应时间的变化而变化。

在错误处理方面，同步传输属于有确认的传输模式，伴随报文的确认信号，可以附加一定的校验信息，发送方根据校验信息判断发送是否成功。而异步传输属于无确认的传输模式，报文的发送的过程与接收过程是独立进行的，不存在校验的过程。

传输模式的选择

分析比较同步传输模式和异步传输模式在时间耗费上，空间利用率以及错误处理上的异同点，可以发现：异步传输模式在时间耗费上较低，速度较快，同时能够更加充分地利用共享区(实际上在异步传输模式中存储共享区起到了缓存的作用)。同步传输模式的优势是能够提供校验功能，这项功能可以保证数据传输的正确性。但是由于传输层底层协议是通过读写存储器共享区实现的，而不是通过线缆等易丢失数据的媒介，因此存储层实际上已经能够保证数据读写的正确性。综上所述，在存储层上，异步传输模式在各方面的性能均优于同步传输模式，也更适合传输层协议使用。

传输层协议

连接管理

连接的建立与释放过程如图 3-11 所示：

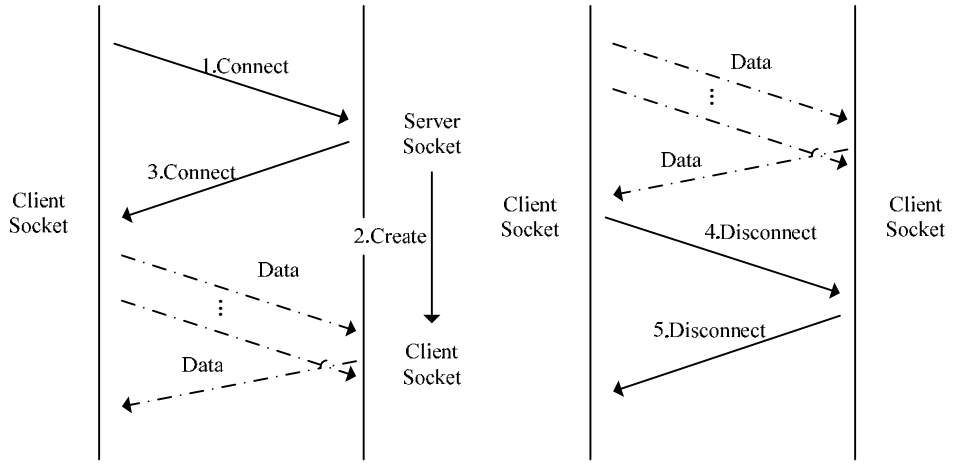


图 3-13 连接的建立与释放

Fig.3-13 establishes and releases a connection

首先是建立连接：

步骤 1：连接的建立首先必须基于一个固定的存储器位置，利用该位置服务器套接字可以有目的的检查是否有连接请求到来，而不需要搜索整个共享区。作为代价，同一时刻只能由一个客户端套接字发起连接，而其余套接字必须等待直到前面的连接成功建立或失败退出。不失一般性的，这个位置存储的可以是一个报文指针。在未有连接请求时，指针的值为空。当有连接请求时，指针则指向连接请求报文。当客户端要建立请求时，首先应该在共享区中任意位置建立连接请求报文，随后将该指针赋值为连接请求报文的地址。

步骤 2：服务器套接字通过检查共享区中固定位置的指针值，发现该指针值不为空。然后，服务器套接字根据该指针的值定位到由客户端套接字发送的连接请求报文。通过分析连接请求报文提供的信息，服务器套接字能初始化一个用于与客户端套接字对等连接的套接字，并使用这个新建的套接字维护与客户端的连接信息。

步骤 3：服务器套接字在共享区中建立同样一个连接请求报文，将报文发送给客户端套接字(发送报文的具体办法在下面一段会详细介绍)。在第一步客户端发出的连接请求报文中，会提供一个报文接收指针的地址。服务器套接字可以利用这个地址发送报文给客户端套接字。作为连接的确认报文，服务器套接字首先给客户端套接字发送一个连接请求报文。客户端收到确认报文说明连接已经建立，服务端和客户端就能够开始通信了。

这样一个步骤也称为连接握手，握手完成表示连接成功建立，此时通信的双方都可以往通道里写入报文或者读取报文。如果客户端套接字在等待一段时间以后没有收到服务器发来的连接请求报文则说明建立连接失败。此时客户端可以选择等待一段时间后重新建立连接。整个建立连接的过程只需要两次报文收发。

释放连接：

步骤 1：连接的任意一方可以给对方发送连接释放控制报文。在这里不妨称请求发起方为 A，相应方为 B。协议规定 B 方在收到这个报文后将认为 A 方已经断开连接并且立刻停止报文的传输，因此 A 方在发出连接断开报文后，不得继续向对方发送任何报文。释放连接的核心的问题是必须保证还处于共享区中的报文被全部被丢弃，任何未被丢弃的报文都会继续占用共享区，从而引起共享区内存泄漏。假如 A 方仍然继续发出报文，而 B 方却不能释放这些报文，那么就会引起这样的内存泄漏问题。

步骤 2：协议要求，B 方在收到请求报文后，必须立刻停止报文接收，回送连接释放请求并且销毁连接信息。由于在 A 方发出请求后，共享区中依然有 B 方发给 A 方的报文，如果不完全释放这些报文，依然会产生内存泄漏的问题。B 给 A 发送的请求报文就是为了解决上述问题, 因为B发出的释放连接请求一定是B发出的最后一个报文， A 在收到这个确认报文后可以断定，B 给 A 发送的所有报文都已经被释放，这时 A 就可以销毁连接信息并宣告连接断开了。

释放连接主要需要避免的问题就是连接断开后依然有报文未能得到释放而造成

的共享区内存泄漏问题。通过上述协议，就可以保证所有的报文都会在连接断开前得到释放。如果请求的发起方在等待一段时间后仍然未能收到释放连接的报文响应(这可能由接收方程序意外终止造成)，则可以认为连接已经断开并且摧毁连接信息。整个释放连接的过程只需要两次报文收发。

报文的传输

首先只考虑报文单向传输时的情况。如 3.4.1 节所述异步传输模式相对于同步传输模式更加适合于在存储层上实现，因此本文仅讨论异步传输算法。

基于存储层协议，间接链表法可以作为异步传输协议的一种实现方法。算法的实现基于一种如图 3-12 所示的间接链表结构[40]

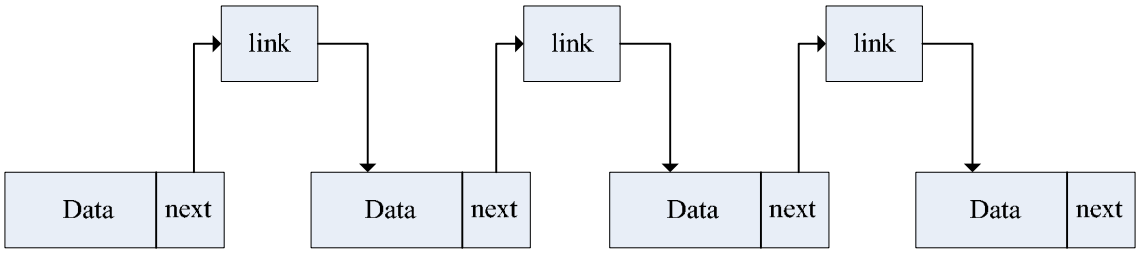


图 3-14 间接链表示意图

Fig.3-14 Indirect Linked List Diagram

在这种结构中，链表的 next 指针不再指向链表的下一个节点，而是指向一个指针类型的单独间接指针对象 link，再根据 link 内的指针找到链表的下一个节点。这种链表不直接使用链表节点内部的指针来找到下一个节点而是通过一个外部指针来间接达到这个目的，因此可以称其为间接链表。

连接建立以后，报文的发送方应当持有作为发送点(sendLink)的间接指针对象，而报文的接收方应当持有作为接收点(receiveLink)的间接指针对象。

报文的发送过程如下：

1. 在共享存储建立新的报文
2. 使 sendLink 中的指针指向新建报文
3. 用新报文的 next 指针所指向的间接指针对象覆盖当前的 sendLink

报文的接收过程如下：

1. 根据 receiveLink 找到下一个报文，如果 receiveLink 中的指针为空则返回空值

2. 读取新报文中的数据
3. 用新报文的 next 指针所指向的间接指针对象覆盖当前的 receiveLink

情形 A：当前报文链如图 3-13(1)所示，同时发送方要发送一个报文

发送方首先在共享区中任意位置建立报文 package3 和 link4，其中 package3 中有指针指向 link4。然后再根据 SendLink 找到并修改 link3 使其指向 package3。最终将 SendLink 指向 link4。报文发送完成后的报文链如图 3-13(2)所示。

情形 B：当前报文链如图 3-13(2)所示，同时接收方要接收一个报文

接受方首先根据 ReceiveLink 通过 link1 可以找到 package1，随后读取 package1 中的报文内容，最后把 ReceiveLink 指向下一个指针 link2，随后释放对象 link1 和 package1。报文接收完成后的报文链如图 3-13(3)所示。

使用基于存储层的异步传输协议，报文的发送与接受可以不受对方干扰，独立进行且操作的空间复杂度及时间复杂度都是 O(1)级。

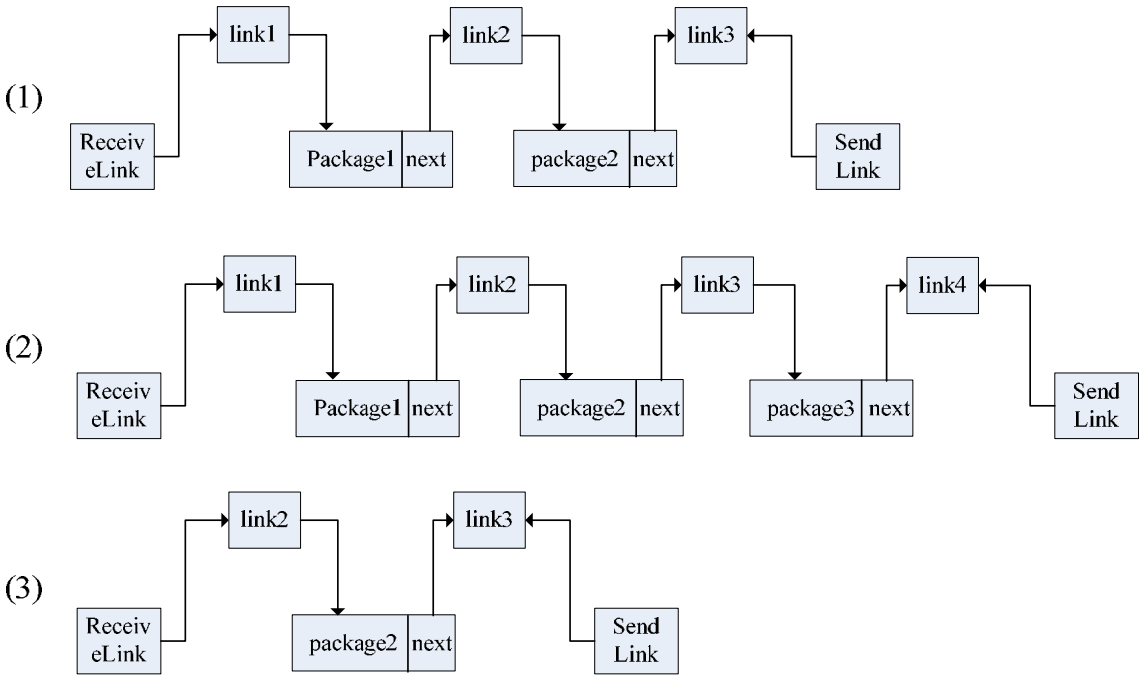


图 3-15 报文传输算法

Fig.3-15 Package Transmission

需要双向传输时，双方套接字在建立连接后各持有一对 SendLink 及 ReceiveLink 从而

形成两条方向相反，独立传输的报文链。通过这两条独立的报文链，通信的双方可以实现无差别的双向异步传输。

3.4.2.3 共享与互斥问题

在 SMP 系统中，两个系统异步得通过读写共享存储区传输报文。共享与互斥问题是必须考虑的问题，假如客户操作系统和主操作系统同时读写同一块内存区域，将很有可能导致操作失败或者操作发生错误。例如：主操作系统发送报文时需要首先在共享存储区建立报文，如果这时客户操作系统也在发送报文，那么就有可能两个系统申请到了同一块存储区，同时后面写入的报文数据部分或全部得覆盖前面的数据，从而出现传输失败。又比如主操作系统发送报文时在共享存储区分配了一定的空间，并开始往里面填充报文数据。假如这时客户操作系统已经开始读取报文信息，便可能读取到不完整的报文，从而出现传输错误。

为了消除这类由于并行访问带来的传输错误或者传输失败，本文所使用的传输协议从两个方面进行了保证。

1. 存储层互斥机制

存储层互斥指的是,在存储层的分配操作之间、释放操作之间或者分配操作与释放操作之间是互斥的，或者说是不可重入的。假如保证了共享区分配与释放操作的不可重入性，如第一例中的错误就不可能出现了。

存储层的分配操作和释放操作如下所示：

```
spinlock loc;
Malloc(Size_t size)
{
    Lock(&loc);
    //分配操作
    Unlock(&loc);
}
```

```
Free(void *ptr)
{
    Lock(&loc);
    //释放操作
    Unlock(&loc);
}
```

加入互斥机制之后，当主操作系统开始为报文分配空间时，客户操作系统无法同时进入 Malloc 函数而必须保持等待，直至主操作系统完成分配操作。这样就保证了报文的分配和释放操作不会因为共享而发生操作错误。

2. 传输层共享机制

通过对传输层和存储层协议的了解，可以知道报文实际上只是共享存储区中的对象。而传输层协议必须保证这些对象被读取的时一定是完备的报文，否则就会发生第二个例子中的传输错误。

因此传输层协议中规定了间接链表这样的数据结构来保证报文被读取时一定是完备的，因为在给报文分配存储空间和填充数据时，报文并没有被加入报文链表。按照传输层协议，报文的接收放必须根据链表末尾的间接指针来找到下一个报文，当下一个报文尚未到来时，间接指针应当为空值。当报文完成建立之后，发送方才会使间接指针指向新建报文，此后这个报文才可能被接收方读取。

综上所述，存储层的分配释放操作和传输层的传输操作共同保证了报文可以在 SMP 系统中多操作系统间正确无误的传输

3.4.3 应用层协议

应用层协议工作在传输层协议之上，其作用是给客户操作系统中的应用程序提供途径来调用主操作系统中的数字签名服务。

数字签名服务需要获得如下信息以完成一次数字签名操作：签名算法及明文数据。因此应用层协议需要定义适当的消息类型，来传输这些信息并且返回签名结果。

为了完成数据签名操作，定义了如图 3-15 所示的应用层消息格式：

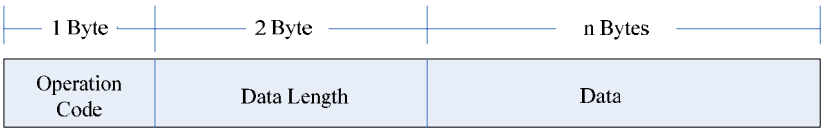


图 3-16 应用层消息格式

Fig.3-16 Application Layer Message Format

操作码(Operation Code)：

表示操作的类型，它是占用一个字节的枚举类型其定义如下：

{SelectAlgorithm = 0, FeedData = 1, RequireResult = 2,ResultData=3, ResultFinished=4 }

数据长度(Length)：

它是一个占用两个字节的无符号整数，用以表示数据段的长度，以字节数衡量。可以表示从 0 到 65535 字节的长度范围。

消息数据(Data)：

在不同的消息中，数据段中数据的意义有所不同。例如在 SelectAlgorithm 消息中，数据段表示所选择的签名算法。在 FeedData 消息中数据段表示明文数据。

客户操作系统通过如图 3-14 所示的步骤来完成一次签名操作

1. 建立系统间的连接
2. 选择签名算法
3. 分次提交明文数据
4. 完成明文数据提交
5. 分次获取签名结果
6. 断开连接

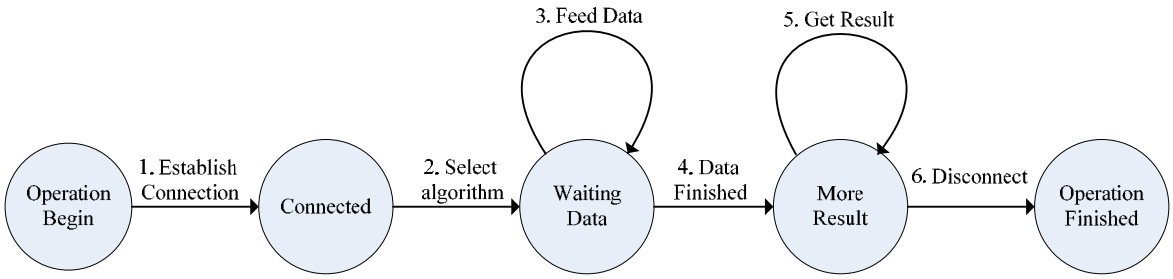


图 3-17 数字签名状态转换图

Fig.3-17 Digital Signature Status Conversion

连接建立后，客户操作系统首先发送操作码为 SelectAlgorithm 的消息通知主操作系统签名服务所需要的签名类型如 DSA 等，并且把签名类型写入 消息的 Data 段中。随后通过 1 次或多次的发送 FeedData 消息将明文数据提交给数据签名服务。由于消息的传输实际上是通过在共享区中分配空间来实现的，而共享区中的自由空间长度是有限的。如果消息长度太大，空间分配就可能失败，因此根据共享区的长度一般将消息应该限制在共享区的 1/10 以内，例如有 16MB 的共享区则可以选择最大 1M 的数据长度。同时数据长度又不应该太小，否则性能将浪费在传输层报文的传输操作上。明文提交完毕后则发送 RequireResult 消息告诉签名服务明文提交完毕，等待获取签名结果。签名服务在完成签名计算后可以发送 ResultData 消息将签名结果返回给客户操作系统应用程序并且最后发送 ResultFinished 消息表示结果返回完毕从而完成一次完整的签名流程。

3.5 本章小结

本章主要论述了在虚拟机系统上实现数字签名服务的规范和方法。本章首先给出了数字签名服务的模型概述，引出了模型中的 4 个基本元素：数字签名服务，应用程序接口， SMP 支持和消息引擎。

数字签名服务包括两部分内容，密钥管理和签名运算引擎。作为主操作系统的应用模块，数字签名服务在主操作系统内完成密钥的产生、存储、应用以及销毁。签名运算引擎部分以 SHA-1 散列算法和 DSA 签名算法为例介绍了数字签名算法及其实现。

应用程序接口作为客户操作系统的一部分，屏蔽了系统间的数据传输等底层实现，给客户操作系统中的应用程序提供了具有良好可读性的访问接口。应用程序接口包含了两部分，字符设备驱动程序和应用程序库例程。在以 Linux 为客户操作系统的实现方案中，字符设备驱动程序区别于应用程序具有访问物理存储器的能力，而应用程序库例程则把复杂的消息传输协议封装成为可读性好的数字签名库例程供应用程序调用。

消息引擎规范了客户操作系统和主操作系统传输数据的算法和数据结构。共包含三层协议。存储层协议、传输层协议和应用层协议。存储层协议在共享存储区上维护了一个堆结构，解决了共享区空间分配的问题。传输层协议定义了一系列的传输算法和报文数据结构，使得系统间可以异步并发的传输数据。应用层协议则是把数字签名的服务功能定义加入到报文中，便于应用程序和数字签名服务使用。

第四章 功能及性能分析

4.1 功能分析

4.1.1 密钥管理

数字签名使用私钥加密来保证发送方的确定性并使用公钥加密来保证接受方的确定性，所用到的密钥存在产生、验证、传递、保管及销毁等过程。因此在本文论述的基于虚拟机的数字签名服务内部除了能够实现这些功能，并且相对传统的软硬件实现方案还具有功能上的优势。

在普通的纯软件数字签名方案中，私钥的存储往往是一个困难的问题。不论将密钥存储在非易失性存储器中还是在非易失性存储器中，假如不加保护，系统中的恶意程序极容易将其盗取。假如使用加密方式存储密钥，也一定有某个根密钥能够解密被保护的密钥，一旦恶意程序获得了这个根密钥，数字签名的安全性也将完全丧失。

为了提高计算机的安全防护能力，Intel,Microsoft,IBM,HP 早在 1999 年 10 月共同发起了 TCPA 可信计算联盟，并与 2003 年 4 月更名为 TCG(Trusted Computing Group)可信计算集团。TCG 制定出了安全硬件和软件技术的授权条款及许可证政策，目标是将可信计算(Trusted Computing)技术用于从个人电脑(Personal Computer),PDA(Personal Digital Assistants)以及手机的等各种计算平台上。

基于密码体系和 TCG 规定的硬件平台规范，在计算机硬件中植入唯一标志使计算平台的可信硬件 TPM(Trusted Platform Module)由此产生密钥进而实施各种验证，依赖认证与加密的结果，保证计算机系统中各种动作的有序执行。

TPM 提供两种基本的服务：授权根和加密这两种服务往往总是联系在一起的。一个授权根服务监控计算机中运行的系统软件并通过加入硬件日志审计的功能保证其按预定的方向进行。而加密服务就包含对称加密和非对称加密。TPM 中的密钥可以分为存储密钥和签名密钥。密钥都是在 TPM 内部产生的，TPM 的功能之一就是创建 RSA 密钥对和对称密钥，他们的详细密钥信息在创建后将保存在 TPM 中。

由于 TPM 的存储空间有限，必要时将他们以加密的形式放到外部存储区中。当使用这个密钥的时候：首先，在密钥缓冲池中查找并判断次密钥是否已经存在；如果存在，则说明此密钥信息已经存在于 TPM 中，不需要重新加载，直接就可以在 TPM 中使用；如果不存在，则说明此密码信息不在 TPM 中，需要进行加载。

内部密钥和外部密钥的主要区别在于密钥在 TPM 内部是以明文存储的，而在外部是以密文存储的。密钥在外部是以一个树形结构进行存储的，其中树的根节点是存储根密钥。它负责管理一个易挥发的内存空间，这个内存空间存放用于签名和解密操作的密钥。一旦存储根密钥被泄漏，那么整个可信平台都将很容易的被解密。

而基于虚拟机的数字签名服务将密钥置于主操作系统存储器区域中，并且与客户操作系统存储器区域在逻辑上保持隔离。客户操作系统及其中的应用程序认为它们能够访问全部的存储器空间，而实际上它们所能访问的仅是全部存储器空间的一个部分。同时，由于可以使用主存作为数字签名服务的存储器，就避免了 TPM 中存储器空间不足的问题。对于易挥发的密钥可以全部统一存储在主操作系统存储区中，而需要永久存储的密钥可以加密以后存放在闪速存储器(Flash Rom)中。闪速存储器除了作为 BIOS 的载体以外，还可以作为密钥等平台安全信息的载体。

除了能够保持和 TPM 同样的密钥存储安全性以外，基于虚拟机的数字签名服务还能为密钥生成提供更多的底层支持。数字签名方案使用非对称加密技术给摘要签名。为了证明消息与它的发送者之间存在映射关系，因此需要产生数字证书将公钥公开。这样消息的接收者可以使用公钥来验证签名，如果消息摘要和用公钥解密签名的结果匹配成功，就能够证明消息的确是由持有对应私钥的发送者签署的。

相对于纯软件方案和 TPM 方案，虚拟机方案对数字证书的生成能够提供网络的支持。因为不论是纯软件方案还是 TPM 方案，数字证书的生成过程都必须将通过操作系统，并由操作系统通过网络将密钥对发送给安全中心。为了防止传输中的密钥对被非法窃取，需要使用安全中心(CA)的公钥将密钥对加密。假如操作系统中存在恶意程序，完全可以使用预先生成的的公钥来假冒安全中心的公钥，之后就可以用预先生成的私钥来解密加密过的密钥对。从而获得数字签名的私钥并拥有伪造签名的能力。而虚拟机的主操作系统自身就拥有访问网络的能力，这样不论是获取安全中心的公钥还是把密钥对提交给安全中心都不需要经过客户操作系统，这就避免了在制作数字证书的过程中密钥被客户操作系统中的恶意程序伪造的可能。

4.1.2 计算能力

密码学运算通常计算复杂度较高，而其中数字签名方案所用的非对称加解密运算相对于对称加解密运算复杂度又更高。因此，大量的、实时的数字签名计算需要强大的运算能力作为支持。计算机的中央处理器不但拥有高性能，同时在功能上可以根据用户的配置来充当通用处理器或数字签名协处理器。

基于虚拟机的数字签名服务的性能优势在于，它能够充分利用计算机系统中中央处理器的强大运算能力来满足实时数字签名运算的需要，同时又保证了实现上的安全性。相对于普通数字签名芯片如 TPM，加密狗中采用的嵌入式处理器，同期的计算机中央处理器计算性能要更强。一般来说，计算机中央处理器的运算速度是嵌入式芯片的 20~100 倍，因此从计算能力上可以完全超越普通的硬件实现方案。

4.1.3 安全性

在数字签名方案中，密钥是最重要的关键信息。因为攻击者的目标是伪造签名也就是获得对任意明文产生对应密文的能力，而简单的获得明文或是获得密文对攻击者的帮助都不大。因此即使攻击者能在应用程序进程中找到某段明文的签名，想要修改这段明文并通过验证还需要获得对应的密文，因为签名方案已经保证了根据已知的明密文对获得未知的明密文对在计算上是不行的，所以保证签名方案在实现安全关键是要保证密钥的安全[41]。

在基于虚拟机的数字签名方案中，密钥被隐藏在客户操作系统无法访问的存储区中，并且由主操作系统负责密钥相关的运算。客户操作系统通过一段共享存储区交换明文和密文使用主操作系统提供的数字签名服务。由于客户操作系统无法访问密钥，运行在客户操作系统中的恶意程序就同样无法获得密钥。尽管客户操作系统中存在各种漏洞和不安全因素，但既然密钥本身就不在操作系统中，任凭恶意程序怎样盗取也不可能获得本来就不存在的数据。

和 TPM 等硬件方案的原理一样，基于虚拟机的数字签名方案同样是将密码运算和操作系统的常规应用分离，使得恶意程序无法得到内存中的密钥[42]。与 TPM 不同的是，虚拟机方案属于软件方案，由于虚拟机的应用，才使得密钥安全得到了保证。软件方案具有灵活性好的特点，可以在不添加额外设备的前提下提高数字签名的安全性.。

此外，软件方案也能避免一些纯硬件方案的安全薄弱环节。

有一种攻击主要是通过分析电子设备执行计算过程中的能量消耗来寻找有关密钥的信息[43]。通常将这类攻击划分为简单能量分析攻击 SPA（Simple Power Analysis）和差分能量分析攻击 DPA（Differential Power Analysis）。DPA 攻击是通过分析泄漏信息进行攻击的主要形式。

在 SPA 攻击中，目标本质上来说是利用能量消耗的值来推测出相关的秘密信息甚至是密钥。在 DPA 攻击中，计算了两组平均能量消耗的差异，如果出现非常显著的差异就认为攻击成功。给人留下深刻印象的是虽然攻击者不了解而且也不试图找出该算法特定的执行部分的任何信息，DPA 攻击也同样可以找出密码算法（例如 DES 算法）的密钥。当前存在的算法中，有些能够防止 DPA 攻击，但不能防止 SPA 攻击；还有一些算法则相反，能够防止 SPA 攻击，不能防止 DPA 攻击；另外还有这两种攻击都能抵御的算法以及都不能抵御的算法。

从 Paul Kocher 于 1995 年公开发表 DPA 攻击的原理以来，现在已经出现一些相应的解决方案[44]：

（1）引进随机时间移动。这样计算方式不再与相同设施的能量消耗有关。

（2）替换一些关键设备，使它们很难被分析。

（3）对一个指定的算法提供一种明确的计算方式，以使 DPA 攻击对得到的执行可能无效。

如方案(2)所述，为了实现基于虚拟机的数字签名方案，可以采取用中央处理器的 AP 运行客户操作系统，BSP 运行主操作系统的运行模式。在这样的模式下，一个处理器同时拥有两条执行序列，在执行密码操作时，由于两个核心的物理特征图谱互相干扰，增加了仪器检测执行密码操作时的功率或辐射特性的难度，给旁路攻击造成了障碍。

4.2 性能评价

基于虚拟机的数字签名服务不仅从架构上保证了密钥不容易被泄漏，同时还能够拥有较好的性能。在需要实时加解密或频繁加解密的应用场合中，能够发挥其安全性上和性能上的双从优势。

衡量它的性能，具体可以从两个方面来分析：

一. 数据传输速度

数据传输速度具体指的是单位时间内客户操作系统可提交签名服务数据和主操作系统数字签名服务可返回签名结果数据的数据量的大小，一般用 KB/S 为单位进行量化。

由于使用计算机内部存储器作为数据传输的通道，其传输速率主要是由存储器的运行速率决定的。在存储器运行速率一定的情况下，数据传输速率可以逼近存储器的存储速率。

速度传输速度也和单个报文传输的数据量呈一定的比例关系。如图 4-1 所示：图中的横坐标表示报文长度指数。当报文的长度指数为 n 时，报文的大小等于 2^{2n+1} 字节。图中的纵坐标表示数据传输速度，它可以用一定量的数据和传输这些数据所用的时间的比值来计算。当单个报文数据量较小时数据的传输速率相对较低，当单个报文数据量较大时数据的传输速率相对较高。其主要原因是，当报文数据较少时，传输层和存储层本身的工作开销占用了整体传输时间耗费的较大部分，有效传输时间相对占空比较低。提高单个报文数据长度后，由于占空比提高，传输速率能够得到有效增长。当单个报文数据长度增长到一定长度之后，存储器的运行速度又成为了限制传输速率的主要因素。最好情况下，数据传输速度能够接近存储器的运行速度。

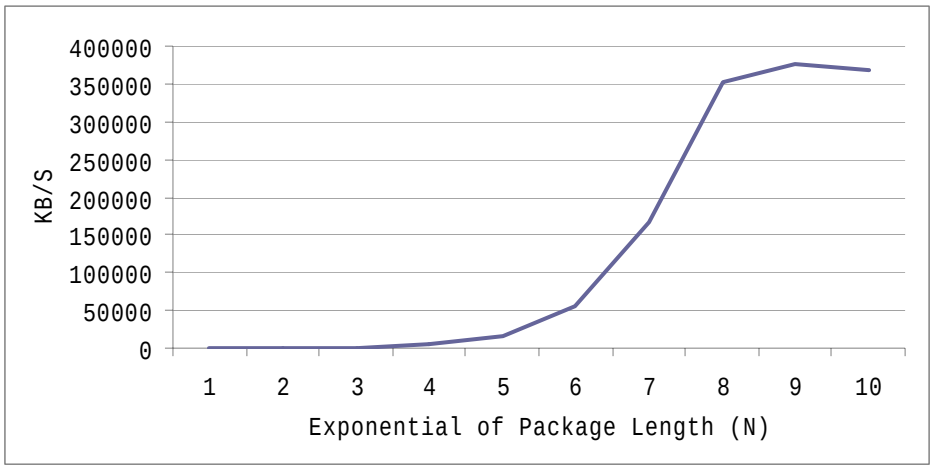


图 4-1 数据传输速度与报文长度关系图

Fig.4-1 the relationship between data transfer speed and package length

传输不同长度的报文 1024 次的时间耗费和有效数据传输速率如表 4-1 所示：

表 4-1 时间耗费和有效数据传输速率

Package length (Byte)	Time elapse (second)	Speed (KB/S)
8	0.12852	62.24712
32	0.127612	250.7601
128	0.1285668	995.5914
512	0.128604	3981.214
2048	0.1330056	15397.85
8192	0.1486756	55099.83
32768	0.1950436	168003.5
131072	0.371128	353172
524288	1.392084	376621
2097152	5.706176	367523.2

由此可见当单个报文长度为 13KB 时，传输速率接近最大值，达到并超过 350MB/S。

二. 签名运算速度

签名运算速度指的是单位时间内，某一数字签名算法可处理明文数据的长度。一般用 KB/S 为单位进行量化。

签名运算主要考验的是处理器的整数运算性能。在基于虚拟机的数字签名服务方案中，负责进行数字签名运算的是系统的中央处理器。区别于一些硬件实现的数字签名解决方案采用的嵌入式处理器，系统中央处理器集成电路规模更大，运算性能也更强。同时，由于其不需要增加额外的硬件，因此通用性较好，在一般的计算机系统上都能采用。

图 4-2 显示了数字签名服务使用主频为 1.86GHZ 的 Intel E6300 处理器，进行 DSA 数字签名运算的性能图谱。从图中可以看出，数字签名服务的数据处理速度还和用于签名的数据长度有关。当数据长度较小时，由于 SHA-1 散列运算需要将明文数据填充到 512bit，所以任何小于这个长度的数据进行 DSA 签名的时间耗费都是相同的，即此时数字签名速度与数据长度成正比。又由于处理短数据时额外的运算所占比例较大，所以当数据长度增加时，有效数据处理时间的比值也在增大，因此数据处理速度有相应的提升。从图中可以发现，当数据长度为 8Byte 时，签名运算速度为 2.7KB/S，当数据长度增加到约 840KB 时，签名运算速度几乎达到其峰值性能 220MB/S。

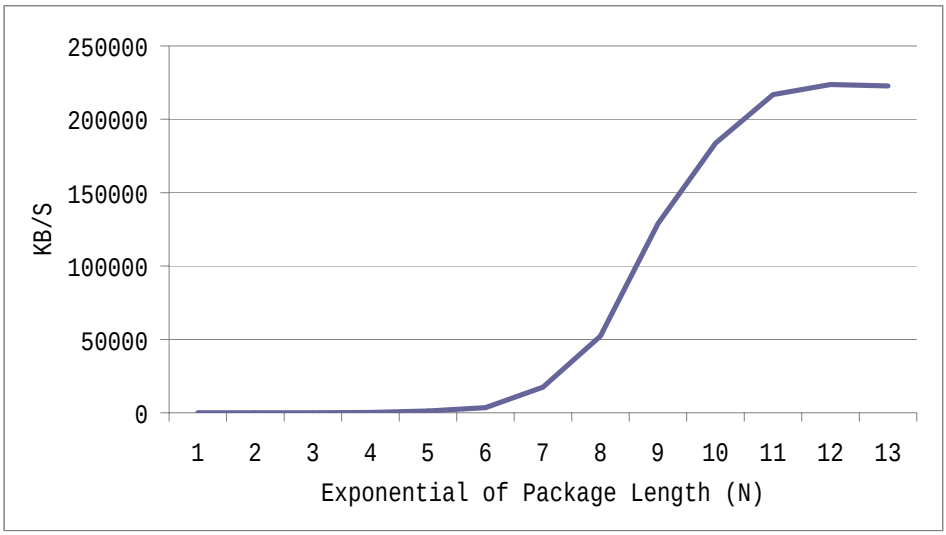


图 4-2 数字签名速度与数据长度关系图

Fig.4-2 the relationship between digital signature speed and data length

对于不同数据长度，进行 1024 次 DSA 签名运算的时间耗费及速度如表 4-2 所示：

表 4-2 签名运算的时间耗费和速度统计

Data length (Byte)	Time elapse (second)	Speed (KB/S)
8	2.953844	2.708335
32	3.395646	9.423833
128	2.548803	50.21965
512	2.126028	240.8247
2048	1.728456	1184.873
8192	2.374848	3449.484
32768	1.874949	17476.74
131072	2.507613	52269.63
524288	4.065967	128945.5
2097152	11.420003	183638.5
8388608	38.678512	216880.3
33554432	150.002254	223692.9
134217728	602.527675	222757.78

因此在正常情况下，应用程序提交给数字签名服务 840KB 的明文数据进行 DSA 签名，以有效数据传输速率 350MB/S，签名处理速度 220MB/S 计算。

传输明文 + 计算数字签名+ 返回结果

需要 840/350000 + 840/220000 + 0.16/1000 (秒)

= 0.00637818 秒

因此在明文数据长度满足大于等于 840KB 情况下，使用数字签名服务进行 DSA 数字签名的数据处理能力可达 132MB/S。因此可以满足大部分安全性要求高，实时性要求强的数字签名的应用需求。

4.3 本章小结

本章主要从功能及性能两个方面来研究基于虚拟机的数字签名技术的作用和意义。从功能上看，基于虚拟机的数字签名服务在保证密钥私密性的前提下，突破了 TPM 在存储空间上的限制，简化了密钥的存储。同时由于主操作系统本身可以访问网络，还避免了通过客户操作系统网络功能连接信任中心创建数字证书的过程中可能遭到的恶意程序攻击。从性能上看，基于虚拟机的数字签名服务可以使用计算能力更强的中央处理器来进行签名操作的运算，而 TPM 等硬件加密芯片由于受到成本等因素的限制，使用的处理器运算能力相对较弱。因此基于虚拟机的数字签名技术特别适合运算频繁，数据量大或是对速度和安全性都要求较高的应用，如实时数字签名等。

本章最后还从数据传输和签名运算两方面对系统的性能进行了测试，给出了实际平台下的性能参考数据。

第五章 全文总结

5.1 主要结论

本文在分析比较当前几种数字签名的软件和硬件的实现方案优劣势的基础上，提出了一种基于虚拟机的数字签名技术，并给出了这种技术的基本原理和实现方法。

一般的软件实现方案虽然有着成本低，灵活性好，性能较高的优势，但是安全性却又有比较大的不足，极易受到恶意程序的攻击。而硬件实现方案虽然安全性较好，但是却存在成本高，运算能力不足的缺点。不但如此，硬件方案还受到旁路攻击等基于芯片物理特征的攻击技术的威胁。

数字签名使用私钥加密来保证发送方的确定性并使用公钥加密来保证接受方的确定性。对信息接收者而言，由于私钥是由意定的发送方唯一持有的，所以私钥加密保证了他所接受的信息的确是由意定的对方送来的，不是冒名者发来的。签名算法保证了即使攻击者获得了明文和签名对也无法在有效的时间内修改明文并计算出有效的签名，因此只有密钥安全得到了保证，数字签名方案的安全性才能得到保证。

基于虚拟机的数字签名技术，将虚拟机技术运用在数字签名的实现上。阻止了恶意程序对密钥的攻击，有效保护了密钥等关键信息的存储、计算安全。而且签名运算由于在系统中央处理器的强大运算能力的支持下，签名方案的计算性能也得到了有力保证。此外，由于软件方案的灵活性，使用虚拟机技术无需引入额外硬件，也使得应用成本可以有效的降低。

5.2 研究展望

随着互联网技术的发展，信息化在社会中的作用日益突出。电子商务，电子政务的发展对数字签名技术的应用又具有广泛而巨大的需求。而虚拟机作为当前计算机应用技术研究领域中研究热点和发展重点，正经历着飞速发展的过程。随

着虚拟机技术的进一步推广和普及，基于虚拟机的数字签名技术也会有更加广泛的应用基础。可以预计，基于虚拟机的数字签名技术作为硬件数字签名技术的补充和替代会对经济发展、社会进步以及国防建设等各方面产生促进作用。

基于虚拟机的数字签名技术的应用前景虽然光明，但目前依然存在很多的课题等待计算机技术研究者去解决。

伴随着虚拟机技术的发展，恶意程序也开始将目标指向虚拟机，它们能够窃取计算机运行权限，运行特权指令以获得超出正常范围的访问能力。因此，未来的研究方向还应该包括如何完全隐藏虚拟机自身，并且提高虚拟机自身防范攻击的能力[45]。为了防止恶意程序冒充正常应用程序使用数字签名服务，还需要建立一套有效的验证机制来检查服务使用者的身份，并过滤非法的使用者。

参 考 文 献

[1] 秦志光. 密码算法的现状和发展研究. 计算机应用. 2004.02 24(2):1-4

[2] 杨茂磷，葛勇．密码算法及其在军事通信中的应用．火力与指挥控制 2006.03 31(3):68-71

[3] 冯登国．密码学原理与实践(第二版) 北京：电子工业出版社 2005: 233-236

[4] 程若非. Apache 与电子商务. 开放系统世界. 2002(7): 47-49

[5] 孔维广．TPM 的工作模型．武汉科技学院学报 2005.01 18(1):45-47

[6] 赵泽良，沈昌祥 IPSec 与 IP 加密机设计中要注意的问题. 信息安全与通信保密. 2001(11): 36-39

[7] 魏仲山. 多虚拟机研究. 计算机工程与应用. 1983(9): 39-46

[8] E.Bugnion. Running commodity operating systems on scalable multiprocessors. Sixteenth ACM Symposium on Operating System Principles. 1997.10:78-86

[9] J.E. Smith, Ravi Nair. An Overview of Virtual Machines Architectures. Elsevier Science(USA) 2003.11

[10] 谢文砚．虚拟技术现实全景．网络世界 2006.12 22

[11] 王汝传， 黄良俊. 基于虚拟技术的入侵检测系统攻击仿真平台的研究和实现. 电子信息学报. 2004.10 26(10)

[12] Intel Corporation. Extensible Firmware Interface Specification Version 1.10. 2002(12)

[13] 曾宪伟，张智军. 基于虚拟机的启发式扫描反病毒技术. 计算机应用与软件. 2005.09 22(9):125-126

[14] Puketza N, A software platform for testing intrusion detection system. IEEE Software Magazine 1997, 14(5): 43-51

[15] Unified EFI, Inc. Platform Initialization Specification version 1.0

[16] W A Arbaugh, D J Farber. ASecure and Reliable Boot-strap Architecture. Preceedings of the IEEE Symposium on Security and Privacy. 1997: 78-80

[17] Intel Corporation. Multi-Processor Specification Version 1.4. 1997. 5

[18] T. Borden, J.Hennessy. Multiple operating systems on one processor complex. IBM Systems journal 1989, 28(1) 104-123

[19] E.Bugnion, S.Devine. Running Commodity operating system on scalable multiprocessors. In proceedings of the 16th ACM Symposium on Operating Systems Principles. 1997.10 143-156

[20] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In Proceedings of the 5th Symposium on Operating Systems Design and Implementation 2002.11

[21] 吴松青，王典洪. 基于 UEFI 的 Application 和 Driver 的分析与开发. 计算机应用与软件.2007 24(2): 98-100

[22] Intel Corporation Mobile Platform Usage of UEFI and the Framework Technology 2006.9

[23] 蒋艳凰，白晓敏，杨雪军. 数字签名技术及其发展动态. 计算机应用研究. 2000 (9):1-3

[24] 张淼，杨昌. 可信计算平台中的密钥管理. 楚雄示范学院学报 2006.9 21(9):45-48

[25] 曲英杰. 可编程移动电脑加密机的设计方法. 计算机工程与应用 2007.05(43):99-101

[26] 秦中元，胡爱群. 可信计算系统及其研究现状. 计算机工程 2006.07 32(1):111-113

[27] FIPS PUB February 1, 1993, Digital signature Standard

[28] Rivest R L, Shamir A, Adleman L. A method for obtaining digital signatures and public-key cryptosystem. Common ACM 1978(21): 120-126

[29] M. O. Rabin Digitized Signatures and public-key functions as intractable as factorization. MIT Laboratory for Computer Science Technical Report LCS/TR-212 1979

[30] Douglas R.Stinson Cryptography Theory and Practice Publishing House of Electronics Industry 2005.2: 236-237

[31] 杜谦，张文霞. 多语言可实现的 SHA-1 散列算法. 武汉理工大学学报. 2007.7 29(7)

[32] 张卫清，张红南，谢冬青. 实用的数字签名方案进展. 计算机技术与自动化. 1997.1: 76-79

[33] 方妮, 郭超. DSA 数字签名算法研究及软件实现. 中国工程物理研究院科技年报. 2001.7

[34] 贺纲. Linux字符设备驱动程序. 信息工程大学学报. 2001.7

[35] Denning, P.J. Virtual Memory. Computing Surveys, vol2 1970.9 153-161

[36] 王鹏, 尤晋元. 操作系统设计与实现. 北京：电子工业出版社 1998.8: 116-120

[37] Andrew S. Tanenbaum Albert S. Woodhull, Operating Systems Design and Implementation, publish House of Electronics Industry 228-229

[38] 李晋, 葛敬国. Linux 下互斥机制及其分析. 计算机应用研究. 2005.8

[39] 彭正文. 基于 SMP 的 Linux 内核自旋锁分析. 江西教育学院学报. 2005.7

[40] Cheriton, D.R. An Experiment on Fast Message-Based Interprocess Communication. Operating System Review 1984 vol.18 12-20

[41] Andrew Grissonnanche. Security and Protection in Information Systems. Published by Elsevier Science Publishers 1989 37-39

[42] TCG.. TPM Design Principles V1.2 004.06

[43] 陈开颜, 赵强. 集成电路芯片信息泄漏旁路分析模型. 微计算机信息. 2006 22(6): 74-76

[44] 李欣, 范明钰. 能量分析攻击及其防御策略. 信息安全与通信保密 2005.7 :103-107

[45] Professor Jaege. Virtual Machine Security www.cse.psu.edu/~tjaeger/cse497b-s07/

致　谢

本文是在张申生教授的悉心指导下完成的。在三年多的学习生涯中，导师在学习、生活等方面均给予了我无私的帮助与支持。在我学位课程的拟定、研究课题的落实、论文撰写等方面给予了悉心指导和热情关怀，使我从中受益匪浅。可以说本文得以完成，倾注了导师大量的心血和汗水。在此谨向我的导师张申生教授致以崇高的敬意和衷心的感谢！

两年多来，电子信息和电气工程学院和研究生院的领导、老师在学习、生活上给予了我大量的指导和帮助，使我得以顺利地完成硕士研究生阶段的学习。特别张申生教授，不管在研究课题上，还是在实验工作方面，都给予了耐心指导，使我取得了长足进步。他的敬业精神、脚踏实地的作风、平和谦虚的为人、团结奋进的精神风貌为我树立了良好的榜样，这种潜移默化的作用将对我今后的工作、学习起着不可估量的影响。

在课题研究的过程中，实验室的陈老师给予了极大的支持，共同参与项目开发的博士研究生张熙哲师兄，给予了许多学术上的指导和帮助。同在一个项目组的硕士研究生张朝华、谢勇和博士研究生邓子健参加了数字签名安全性研究，兢兢业业、任劳任怨，为本论文做了大量的前期工作。在此一并向他们表示最衷心的感谢！

为本文提供设备及技术支持的因特尔亚太软件研发中心的各位领导和给予我们宝贵技术支持的各位因特尔的工程师。是你们的倾力支持，使我的实验得以顺利完成，可以说没有他们就没有这篇论文的诞生！

感谢电子信息与电气工程的老师教授们 7 年来，始终对我的学习和工作给予了热情的帮助和指导，师恩难忘！除了心中深深的感激，我目前无以为报，唯有继续努力！

我要特别感谢我的父母和多年来对我的学习和生活的支持和鼓励，感谢我的女朋友对我学习的理解和支持。在我困难的时候，是他们给予了我战胜困难的信心和力量，使我得以顺利完成了学业。本文凝结了他们的心血、教诲和温暖！

最后感谢评阅论文和出席博士论文答辩委员会的诸位专家、教授在百忙中给予的悉心指导！同时也对曾给予我帮助和关心我的老师、同学和朋友们表示衷心的感谢!

攻读硕士学位期间已发表或录用的论文

[1] 陈志峰，张申生，张熙哲. 基于虚拟机技术的密码系统的研究与实现. 计算机机应用与软件（已录用）