分类号：_____ 密级：_____

U D C：_____ 编号：_____

工学硕士学位论文

# 基于 UEFI 体系的虚拟 TPM 研究


硕士研究生 ：朱东亮

指导教师 ：顾国昌 教授

学位级别 ：工学硕士

学科、专业 ：计算机应用技术

所在单位 ：计算机科学与技术学院



论文提交日期 ：2010 年 1 月

论文答辩日期 ：2010 年 3 月

学位授予单位 ：哈尔滨工程大学

Classified Index :
U.D.C:

A Dissertation for the Degree of M. Eng

# The Research of Virtual TPM Based on UEFI

Candidate : Zhu Dongliang
Supervisor : Prof. Gu Guochang
Academic Degree Applied for : Master of Engineering
Specialty : Computer Applied Technology
Date of Submission : January, 2010
Date of Oral Examination : March, 2010
University : Harbin Engineering University

# 哈尔滨工程大学
# 学位论文原创性声明

本人郑重声明：本论文的所有工作，是在导师的指导下，由作者本人独立完成的。有关观点、方法、数据和文献的引用已在文中指出，并与参考文献相对应。除文中已注明引用的内容外，本论文不包含任何其他个人或集体已经公开发表的作品成果。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。

作者（签字）：朱东亮

日期：2010 年 3 月 18 日

# 哈尔滨工程大学
# 学位论文授权使用声明

本人完全了解学校保护知识产权的有关规定，即研究生在校攻读学位期间论文工作的知识产权属于哈尔滨工程大学。哈尔滨工程大学有权保留并向国家有关部门或机构送交论文的复印件。本人允许哈尔滨工程大学将论文的部分或全部内容编入有关数据库进行检索，可采用影印、缩印或扫描等复制手段保存和汇编本学位论文，可以公布论文的全部内容。同时本人保证毕业后结合学位论文研究课题再撰写的论文一律注明作者第一署名单位为哈尔滨工程大学。涉密学位论文待解密后适用本声明。

本论文（□在授予学位后即可 ☑在授予学位 12 个月后 □解密后）由哈尔滨工程大学送交有关部门进行保存、汇编等。

作者（签字）：朱东亮　　导师（签字）：顾国昌

日期：2010 年 3 月 18 日　　2010 年 3 月 18 日

# 摘　要

UEFI（Unified Extensible Firmware Interface）是由 Intel 推出的一种在计算机系统中替代 BIOS 的升级方案。同传统 BIOS 相比，它具备了极大的可扩展性和可定制性。UEFI 采用高度模块化设计，用户只要符合 UEFI 标准，即可方便快捷地开发出应用程序和驱动程序。

TPM（Trusted Platform Module）可信平台模块被认为是可信计算领域的核心模块。它是新式计算机主板上嵌入的一枚重要安全芯片，内部包含处理器单元、存储单元、密钥管理单元等丰富资源。限于目前很多旧式主板缺少 TPM，硬件 TPM 无法在平台上同时运行多个保护实例，并且 TPM 的硬件调试和可扩展性得不到满足，因此采用软件方式实现 TPM 成为目前研究领域的热点问题。

本文主要阐述了 UEFI、TPM 等工业标准的发展和研究现况，详细讨论了 UEFI 架构技术和 TPM 核心技术。以此为基础，本文详细讨论 UEFI 下虚拟 TPM 的驱动程序设计和其实例应用。在虚拟 TPM 驱动程序设计中，首先在详细分析了标准 UEFI 驱动模型基础上，按照驱动模型规范，本文提出一种虚拟 TPM 核心驱动程序框架设计方案。其次，根据虚拟 TPM 模块需要实现了 UEFI 系统下的随机数发生器。最后，在分析驱动绑定协议和设备路径协议基础上，实现了用于满足虚拟 TPM 核心功能的用户自定义协议—虚拟 TPM 协议。实现过程中，本文提出一种 UEFI 驱动程序与 UEFI Shell 应用程序数据通信的解决方案。为了进一步应用虚拟 TPM 设计，本文通过特有的数据结构和算法设计并实现了两个基于 UEFI Shell 的安全领域方面的应用—文件加密系统和文件监控系统。实验在基于 Windows 平台下的 UEFI 模拟器完成，实验结果完全达到了预期效果。

**关键词：** UEFI；TPM；Shell；虚拟；协议

# Abstract

UEFI(Unified Extensible Firmware Interface) is introduced as a upgrade program which is used to substitute traditional BIOS in a computer system by Intel. Compared with the traditional BIOS, it provides great scalability and customization. UEFI is designed by a high modular. Once users meet the UEFI standard, they can be convenient to develop applications and drivers program.

TPM(Trusted Platform Module) is considered as the core module in trusted computing area .It is an important security chip in the new computer's motherboard. It includes processor unit, memory unit, key management unit and other rich resources. But a lot of old motherboards is lack of TPM ,the TPM hardware can not run multiple instances of protection on the platform and the TPM hardware debugging and scalability are not satisfied. So the software TPM becomes a hot topic in the current research area.

This thesis mainly described UEFI,TPM and other industry-standard development and present status, and then discussed UEFI framework and TPM core technology. On this basis, this thesis focused on UEFI virtual TPM driver program design and its application instance design. In the virtual TPM driver designing, based on detailed analysis of standard UEFI driver model ,this paper firstly presented the framework design of a virtual TPM core driver. Secondly, the random number generator under the UEFI system was achieved according to the need of the virtual TPM module. Finally, based on the analysis of the driver binding protocol and the device path protocol, a virtual TPM protocol which was user-defined and was used to meet the core functionality was achieved. In the process, this paper presented data communication solutions between the UEFI driver and UEFI Shell application. In order to use the virtual TPM design, this paper designed and implemented two kinds of application of security which were based on the UEFI Shell – the file encryption system and the document monitoring system through the specific data structures and

algorithms.The experiment was completed under the UEFI simulator which was based on Windows platform and the experimental results fully met the expected results.

**Key words**: UEFI; TPM; Shell; Virtual; Protocol

# 目　录

# 第1章 绪论

## 1.1 课题研究背景与意义

近年来,随着计算机技术与网络通信技术的飞速发展，计算机的应用已渗透到社会各个领域。如何提供可信的计算平台来保证数据信息的安全性问题，已经成为计算机学科领域研究的热点。传统的信息安全技术，如防火墙、入侵检测和病毒防护，都是以保护服务器防止外来入侵为重点，以被动防御为技术路线，对企图越权访问的非法用户进行封堵，在防外上具有一定的效果。但随着网络防范外延的不断扩大，防火墙越垒越高、补丁越补越多、病毒特征库越建越大，在面对新产生的网络病毒时常常出现无能为力的局面；而且对于来自内部的安全威胁，如因BIOS代码的恶意窜改导致计算机不能正常启动等故障，传统的信息安全技术则无法解决。

基于此情况，2003年由Intel、IBM、微软等IT界巨头成立可信计算组织（Trusted Computing Group，TCG[1]），提出可信计算的基本思想，即在计算平台上引入“可信根”并建立“可信链”以实现信息的可信传递。对于PC平台而言，TCG定义了从计算平台加电EFI（extensible firmware interface，可扩展固件接口）执行开始，到引导代码的执行，再到OS启动和上层应用程序的执行的一系列过程，信任将通过这个过程一直传递下去，直到整个计算环境的建立。

TPM[2]（Trusted Platform Module）诞生于上世纪90年代末，随着电脑和互联网的日益普及，用户对电脑加密等安全措施的要求也越来越高。为了适应不断发展变化的形势，1999年10月TCPA（信任运算平台联盟，Trusted Computing Platform Alliance）成立，加入的厂商有Compaq、HP、IBM、Intel、Microsoft，以共同推广PC的识别技术。TCPA专注于从计算平台体系结构上增强其安全性，并于2001年1月发布了可信计算平台标准规范。到了2003年3月，TCPA决定将推广范围扩大，改组成TCG（可信任运算集团，Trusted Computing Group），从此吸引了PC行业之外的厂商积极参与响应，如Nokia、Sony等，并提出TPM规范。目前TCG已发展为成员超过190家，遍布全球各大洲主力厂商，在IT和通信业拥有广泛影响力的大型组织。TPM的出现使计算机合法用户的数据和隐私不被破坏和泄密，对计算机的信息安全提供可靠保障，因此对TPM的研究和扩

展具有很重要的实际意义和应用价值。

EFI[3]，可扩展固件接口是由Intel推出的一种在计算机系统中替代BIOS的升级方案。同传统BIOS相比，它提供了极大的可扩展性和可定制性。相对于完全用汇编实现的传统BIOS，EFI大部分采用C语言实现。这导致EFI更容易被破译，因此这对EFI的安全性提出了更高的要求。基于EFI的TPM实现是目前对EFI安全研究的重要方面。

在开机启动后，EFI启动阶段取得系统控制权，进行初始引导，引导过程需要TPM进行平台的安全验证，从而逐步确保整个平台可信。随着EFI体系的不断成熟，基于EFI体系的TPM研究受到业界的广泛关注，TPM标准依然在不断更新和完善，而通过软件模拟TPM是扩展和调试TPM的有效方法。目前，EFI已经支持TPM硬件芯片，而本课题选择在EFI平台上研究虚拟TPM，具有以下重要意义：

（1）目前很多旧式主板不含有TPM安全芯片，导致平台固件在安全性方面存在隐患。

（2）采用软件方式虚拟能够方便调试和扩展TPM接口，这大大减少了开发和调试的难度。

（3）在平台上虚拟 TPM 可以同时运行多个保护实例，这在安全性要求不高的场合极为重要，而硬件 TPM 每次只能运行一个保护实例。

## 1.2 概况和技术特点

### 1.2.1 国内外研究现状

由于某些国家的法律法规影响，TPM 并没有推广到全球。中国拥有自主知识产权的可信计算规范被称为可信密码模块（Trusted Cryptography Module，TCM），与之对应的国际可信计算的规范 TPM。TCM 与 TPM1.2 有很多的相同点，TCM 是借鉴了 TPM1.2 的架构，替换了其核心算法后的产品。同时 TCM 中也按照我国的相关证书、密码等政策提供了符合我国管理政策的安全接口。从安全战略方面分析，如果采用国外的 TPM 技术，我国的安全体系就会控制在别人手上，中国将来的标准计算机上产生的所有信息对外国人来说将不存在秘密，这样安全技术的主导权、产业的主导权就更谈不上了。因此国内产业界、

学术界发出共同的心声：必须要建立独立自主的可信计算技术体系和标准。只有我们拥有独立自主的可信计算技术体系，为国家信息安全基础建设打下坚实基础，才能保证未来我们有能力、有办法保护秘密，保护主权。只有掌握这些关键技术，才能提升我国信息安全核心竞争力。

虽然我国的信息化技术同国际先进技术相比，存在一定的差距。但是中国和国际上其他组织几乎是同步在进行可信计算平台的研究和部署工作。其中部署可信计算体系中，密码技术是最重要的核心技术。具体的方案是以密码算法为突破口，依据嵌入芯片技术，完全采用我国自主研发的密码算法和引擎，来构建一个安全芯片。

这是按照我国密码算法自主研制的具有完全自主知识产权的可信计算标准产品[4]。有业内人士表示，中国错过了发展具有自主知识产权的 CPU 和操作系统的机会，TCM 是我国信息安全最后的防线。

TCM 由长城、中兴、联想、同方、方正、兆日等十二家厂商联合推出，得到国家密码管理局的大力支持，TCM 安全芯片在系统平台中的作用是为系统平台和软件提供基础的安全服务，建立更为安全可靠的系统平台环境。TCM 并不能完全兼容 TPM，像 windows Vista bit locker 和 Wave 指纹识别模块这样的应用程序就不能得到 TCM 的兼容。

国际TPM标准采用最新的TPM1.2版本，目前英特尔及微软两大巨头在全世界力推TPM规范，微软的Vista操作系统已经支持这一规范，所以国际PC厂商更愿意支持TPM，这给国产PC厂商带来了新的机会。

### 1.2.2 传统 BIOS 方案的局限

BIOS[5]（Basic Input/Output System，基本输入输出系统），是一组固化到计算机主板上的只读存储器芯片中的程序，要功能是为计算机提供最底层的、最直接的硬件设置和控制。传统 BIOS 是传统 PC 机的体系结构，主要功能包括：加载操作系统，设置系统配置，硬件中断处理、上电自检和初始化。BIOS 中主要存放以下程序：

自诊断程序/（加电自检程序）：在系统初始化前读取 CMOS 中的内容以便识别计算机系统硬件配置[6]；

CMOS 设置程序：引导过程中，用特殊热键启动，进行设置后，存入 CMOS RAM 中；

系统自举装载程序：在自检成功后将磁盘的 0 道 0 扇区上的引导程序装入内存，让其加载操作系统；

基本外围设备的驱动程序：例如键盘显卡等一些基本必要的设备驱动程序，提供基本的输入输出功能。

每当新的平台功能或者新的硬件出现时，固件开发商都需要不断开发复杂的解决方案，同时还经常需要 OS 开发商也做出修改以配合他们新的启动代码，在此之后，消费者们才能真正受益到新的发明。这个过程非常耗时，且需要投入很大的资源。传统 BIOS 的局限性在于：

（1）传统 BIOS 在编程上比较复杂。由于传统 BIOS 采用低级语言编程，而熟悉低级编程语言的程序员的数量有限，这导致 BIOS 程序开发受限。

（2）没有统一的标准接口。由于缺少统一标准，不同厂商生产的 BIOS 也是不同的，这给用户在配置 BIOS 过程中带来了许多不便和困难。而该问题自计算机系统出现以来一直没有得到很好的解决。

（3）不提供对网络的支持，网络安全性得不到有效保障。

（4）采用纯文本界面，与用户交互比较复杂。

## 1.3 EFI 的引入

2000年，Intel向业界展示BIOS的新一代接口程序EFI（Extensible Firmware Interface），并在安腾服务器平台上采用EFI技术。EFI是由Intel推出的一种在未来的电脑系统中替代BIOS的升级方案。传统的BIOS采用汇编语言编写，这大大限制了BIOS的发展。而EFI的引入则解决了传统BIOS的不足之处，首先，它采用更通用的编程语言C语言作为开发语言，这保证更多的程序员参与到EFI的开发和扩展工作中，因此EFI的功能相比传统BIOS要强大许多。其次，EFI是由intel提出的标准统一的可扩展固件接口规范，这使得EFI程序可以运行在不同架构的CPU上，达到平台兼容的目的。最后，EFI可以在IPF、X64等高端架构的服务器上顺利运行，这是传统BIOS无法实现的。

从核心来看，EFI很像一个被简化的介于硬件设备和高级操作系统之间的

操作系统。由于EFI内置了图像驱动功能，因此它能够提供一个高分辨率的彩色图形环境，并且支持鼠标点击操作，明显有别于传统BIOS单调的纯文本界面。与传统BIOS的另一显著不同点是，EFI使用全球最广泛的高级语言C语言进行编写，摆脱了传统BIOS复杂的16位汇编语言代码编写方式。这意味着有更多工程师可参与EFI的开发工作，有利于平台创新快速发展。目前，EFI的应用已由服务器领域扩展至PC领域，苹果在其x86 PC机上已采用了EFI。与此同时，EFI技术向消费电子、家用设备领域的延伸也从未停止。例如，通过EFI技术，当计算机未进入操作系统前，就可接入互联网。

无疑，EFI 在安腾服务器平台上的应用和它的前景，让更多人看到了 EFI 的魅力。2005 年，在工业界达成共识的基础上，Intel 将 EFI 规范交给了一个由微软、AMD、惠普等公司共同参与的工业联盟进行管理，并将实现该规范的核心代码开源于网站上。与此同时，EFI 也正式更名为 UEFI[7]（统一可扩展固件接口）。UEFI 联盟将负责开发、管理和推广 UEFI 规范。

### 1.3.1 EFI 的优点

EFI 可以解决 BIOS 无法克服的许多问题。从核心来看，EFI 或许更像一个被简化的操作系统，介于硬件设备以及高级操作系统（比如 Windows 或者 Linux）之间。它提供了一个支持鼠标的图形界面，与纯文本、界面单调的普通 BIOS 明显不同。传统的 BIOS 受到容量的限制，需要使用相当紧凑的低级语言进行编程；而 EFI 却使用高级语言 C 或 C++语言进行编写，从而为有更多的工程师参与 EFI 的开发工作提供了可能。为了保证充裕的容量，EFI 彻底抛弃了 ROM，其文件系统存储在硬盘独立划定的区域内。

由于 EFI 采用的高级语言是现在编程人员熟悉的平台，因此，可以和许多驱动程序共用开发资源，并可以在 BIOS 上外挂许多应用程序。这样可以减少开发成本，缩短开发时间，PC 厂商也可以用外挂程序来丰富自己的产品，增加个性体现。

EFI 具有标准化的开放技术。在基础架构上，EFI 仍然与传统 BIOS 一致；但它不仅能应用于 PC 平台，还可运用到 IBM 的 PowerPC 等平台。从本质上看，EFI 是一个被简化的迷你型操作系统，它提供了标准的高分辨率彩色图形界面，支持鼠标操作，用户可以方便地管理各种硬件信息。

EFI 具有良好的扩展性：EFI 采用模块化设计。利用 Intel 提供的编程接口，主板制造商可以充分发挥自己的聪明才智，为 BIOS 添加各种功能强大的附加模块，比如集成类似 Ghost 功能的镜像恢复功能、联网进行远程诊断等。用户不用进入操作系统，就可以进行磁盘管理、启动管理、远程配置甚至引导。

EFI 将 BIOS 架构分为硬件控制与 OS 软件管理两部分，而且可以方便地自由组合。采用 EFI 固件可以获得如下益处：迅速支持新外围设备、针对某模块单独对固件升级、省略硬件检测实现高速启动、配备 GUI 界面提高操作性。

### 1.3.2 EFI 的缺点

EFI 的软件部分是基于大众化的 C 或者 C++语言编写的，程序容易遭到破译，这就对 EFI 的安全性提出了更高的要求。EFI 的文件系统是储存在硬盘上，建立有单独的分区。如果硬盘不稳定或者该分区受到攻击，很容易造成系统的崩溃。同时，EFI 在 DXE 阶段就引入了完整的网络应用，比如 TCP、UDP、MTFTP 等支持，这一方面为我们提供了调试网络设备的便利，但同时也为下一步的网络安全留下了隐患。

## 1.4 研究目标

首先详细介绍了UEFI的基本架构和TPM核心的基本原理，进一步，在UEFI环境下设计并实现虚拟TPM，利用UEFI环境下的虚拟TPM 做系统安全上的应用实例，从实用性角度进一步论证本课题的可行性。

## 1.5 本文主要工作与组织结构

本文的主要工作包括深入研究 UEFI EDKII[8]体系架构和运行机制，掌握符合标准 UEFI EDKII 驱动模型[9]的驱动程序以及 UEFI Shell 应用程序的设计方法。根据 TPM 工业标准，裁剪并修改 TPM 核心架构，并根据修改后的 TPM 核心，设计用于服务于虚拟 TPM 的自定义协议及其接口实例。研究并提出驱动程序与 Shell 应用程序数据交互解决方案，通过该方案，在 UEFI Shell[10]下成功设计并实现两种系统安全领域的应用：加密解密和文件监控系统。通过这两种应用，更有力地支持本文的设计目标，完善了整个虚拟 TPM 的设计结构。

本文组织结构如下：

第 1 章是绪论，简单介绍了课题研究背景与研究意义、国内外研究现状、研究目标和研究内容等。

第 2 章是 UEFI 总体架构及 TPM 核心概述。本章对 UEFI 的架构模块和启动阶段的各个流程进行详细介绍和分析，同时对 TPM 核心进行了简要概述，为后文虚拟 TPM 裁剪与修改奠定理论基础。

第 3 章是虚拟 TPM 驱动程序研究。本章在设计驱动程序之前，先详细讲述 UEFI 驱动模型的基本原理。然后主要研究虚拟 TPM 驱动程序的设计以及实现方法，其中包括虚拟 TPM 核心及其相关协议的设计。

第 4 章是基于虚拟 TPM 的应用设计与实现。在第 3 章驱动程序中自定义协议设计基础上实现两个简单安全应用示例，文件加密系统和文件监控系统，从实用性角度进一步论证本课题的研究意义。

最后是总结和展望，本章对本文的工作做了全面的总结并提出下一步计划。

# 第 2 章　UEFI 总体架构及 TPM 核心概述

## 2.1　UEFI 架构与组成

Intel 提出的新一代 UEFI 固件架构把固件分成三层，分别是 PAL[11]（Process Abstraction Layer）、SAL（System Abstraction Layer）和 UEFI（Unique Extensible Firmware Interface）。由 PAL 和 SAL 对系统的硬件资源进行管理和操作。PAL 是专门针对处理器层的抽象，由处理器制造商负责开发。SAL 是针对系统其他资源（如系统控制芯片组）的抽象，由主板开发商负责开发。

UEFI 并不直接对硬件进行操作，而是通过调用 SAL 和 PAL 操作硬件。UEFI 的引导服务和运行时服务都调用 SAL 和 PAL 的相应功能。UEFI 还提供一些调试系统的工具软件，属于 UEFI 的 API[12]。总的来说，UEFI 在 Intel 提出的 Firm-ware 模型里起到承上启下、屏蔽硬件层物理特性的作用，具体结构如图 2.1 所示。

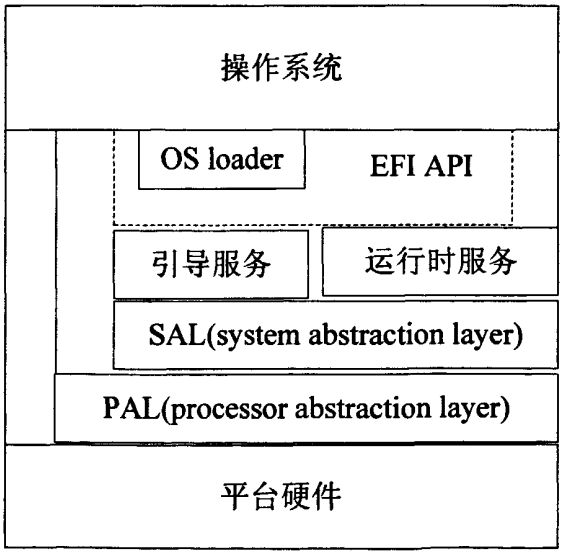


图 2.1 UEFI 结构

UEFI 主要由引导管理器、内核、协议和驱动模型等主要部分组成[13]，各部分的主要作用如下：

（1）引导管理器（Boot Manager）。引导管理器从 UEFI 文件系统的文件

或映像中引导 UEFI 应用和 UEFI 驱动程序，最终从介质上加载 OS loader，将系统控制权转到 OS loader，完成系统的引导。同时，引导管理器也提供一个引导菜单，供用户选择系统引导设备。

例如，系统中连接有 SCSI 硬盘、普通软驱以及 CD-ROM，都是引导设备。引导管理器在接收到系统控制权后会向用户显示一个引导菜单，列举出所有引导设备上能用的操作系统，让用户选择。用户选定后，引导管理器加载必要的驱动程序（使用对应 Driver Binding 协议中的 Start()函数）和相关的应用程序，在硬件的相应位置找到 OS loader，将 OS loader 加载到内存，并将系统控制权转到 OS loader[14]。

（2）UEFI 内核（UEFI Core）。UEFI 内核提供两种服务：引导服务和运行时服务。服务是 UEFI 对系统硬件的一个抽象，使得调用这些服务的程序不必关注系统硬件层的物理特性。

引导服务为已经加载的 UEFI 映像抽象一个通用的引导环境，采用全 64 位设计，对将来硬件技术的发展预留一定的扩展空间。抽象使平台和操作系统能相对独立地进行发展。引导服务所提供的接口功能只在系统引导期间使用，操作系统加载以后就不再需要。引导服务分为几种，各提供相应的功能。CreateEvent、CloseEvent 等服务用于事件；RaiseTPL、RestoreTPL 等用于改变任务优先级；AllocatePages、FreePages 等服务用于管理内存资源；InstallProtocolInterface 等服务用于操作协议句柄；LoadImage、UnloadImage 等为映像服务。

与引导服务不同，运行时服务能够在引导和操作系统运行时被调用。运行时服务能够将物理内存地址转换到虚拟内存地址，只有少数几个运行时服务始终运行在物理内存地址方式。为了不影响系统运行时的性能，运行时服务数量较少，也很简单，如 Get（Set） Variables、Get（Set）Timer、SetVirtualAddressMap 以及 Reset System 等服务都属于运行时服务。

（3）协议（Protocol）。协议是 UEFI 对于平台中各个设备的抽象，通过特定协议提供的接口对相应设备进行操作。UEFI 的协议接口有很多函数指针，用户可以通过将这些指针指向不同的功能函数，使协议驱动不同的硬件。任何一个设备要能够被 UEFI 所使用，首先必须支持相应的协议，然后在使用前在此设备的句柄上安装（注册）该协议；使用完毕以后，也应该卸载该协议。安

装和卸载的动作由 Driver Binding 协议完成。在 Start 函数中，需要在设备句柄上安装协议，设置好协议接口内的私有数据，并且将接口函数的指针指向对应的函数，从而完成驱动程序的安装。卸载的时候，则在 Stop 函数中将分配的资源释放。

（4）UEFI 驱动模型[15]（UEFI Driver Model）。UEFI 驱动模主要为简化设计和实现设备的驱动程序而设计。为尽量减少映像，一些复杂的处理被放到了对应总线的驱动程序中，其他大部分被放到固件的通用服务中。关于 UEFI 驱动模型详细论述请参考 3.1 节。

## 2.2 UEFI 的阶段分析

基于UEFI体系的固件可以提供一个平台固件的完全实现，符合UEFI规范接口实现的标准。UEFI引入许多现代计算机科学的软件设计原理到固件开发领域。不同于传统BIOS实现，UEFI按阶段来初试化平台，将OS启动的过程分为4个主要阶段：SEC，PEI，DXE和BDS[16]。

第一个阶段是安全阶段（SEC），是上电后最先的步骤。SEC支持检查系统执行时最先的操作码，确保选择的平台镜像没有破坏。SEC阶段通常介于硬件和上层固件组件的很小一层。

第二个阶段是Pre Initialization Environment（PEI）表示最小数量的代码，这些代码用来查找和初始化内存，以及用来执行切换到C语言代码其他的一些资源。PEI代码做了少量的工作来发现和初始化内存，很多芯片组和其他组件的初始化一直延迟到驱动程序执行环境（DXE）起来并运行之后，才开始进行。早先的PEI代码倾向于用适合机器的汇编代码来编写，一旦发现内存，PEI就准备状态信息来表述平台资源图，并将其初始化，然后跳转到DXE阶段。注意PEI到DXE的转化是一个单向的状态迁移过程，一旦为DXE的初始化装入程序完成后，PEI代码将不可用，并且DXE成为一个设备齐全的操作环境。因为选择了高级语言C语言，所以需要内存提供堆栈，第一个任务就是找到一块初始化过的内存，使得C代码可以在给定的系统上实际执行。PEI和DXE阶段的划分就是基于这个原则。

DXE是大多数系统初始化执行的一个阶段。在DXE阶段有几个组件，包含

DXE核（core）、DXE分派程序（Dispatcher）和一组DXE驱动程序。DXE核产生一组启动服务，运行时服务和DXE服务。DXE分派程序负责按照正确的顺序发现和运行DXE驱动程序。DXE驱动程序负责初始化处理器，芯片组和平台组件，同时为系统服务、控制台设备和启动设备提供软件抽象。这些组件一起工作来初始化平台，并且提供启动一个操作系统所需的服务[17]。

启动设备选择（Boot Device Selection，BDS）阶段和DXE一起工作，创建一个控制台，并且尝试从可用的启动设备来启动操作系统。BDS是控制权交给操作系统之前的最后一个阶段。在BDS阶段，可以向用户展示图形界面，可以被OEM修改用来定制DXE适应的系统，这个过程用来决定哪个设备根据镜像来启动，以及从哪个设备来启动。

## 2.3 TPM 核心的概述

TPM（Trusted Platform Module）全名可信平台模块，是嵌入在计算机主板上的一枚重要的安全芯片，该安全芯片内存储着非常丰富的资源，包括微处理器单元、存储器单元、密钥管理单元以及 I/O 输入输出单元，它为计算机提供加密和安全认证服务。更具体地说，TPM 安全芯片既是密钥生成器，又是密钥管理器件，还提供了统一的编程接口（其实也就是我们日常使用的 TPM 软件与 TPM 芯片进行交流的底层端口）。密钥[18]是打开加密文件的唯一钥匙，TPM 安全芯片的一个重要作用即加强了对密钥的管理，芯片以硬件来生成、存储和管理密钥，TPM 安全芯片可以将密钥存储在受 TPM 控制器保护的非易失性存储器中。TPM 嵌入在计算机主板上，通过系统总线与计算机相连，CPU 通过系统总线向 TPM 发送控制命令和运算数据.

### 2.3.1 TPM 可信计算

传统的安全保护基本上以软件为基础，附以密钥技术，侧重以防为主。事实证明这种保护并不是非常可靠而且存在着被篡改的可能性。因此，在我国目前硬件、操作系统、安全等许多关键技术还严重依赖国外的情况下，对可信计算的要求迫切地摆在用户的面前。可信计算不同于传统的安全的概念，它将加密、解密、认证等基本的安全功能写入硬件芯片，并确保芯片中的信息不能在外部通过软件随意获取。

在可信平台中，TPM 等硬件是“信任根”，信任的建立是“链式展开”，从而实现身份证明、平台完整性度量、存储保护、远程证明等[19]。这些功能的实现大多与各种密钥密切相关。比如，EK 实现平台惟一身份标识，是平台的可信汇报根；SRK 实现对数据和密钥的存储保护，是平台的可信存储根；AIK 代替 EK 对运行环境测量信息进行签名从而提供计算平台环境的证言等等。可以说可信计算的技术基础就是公开密钥技术，公钥体制中密钥管理体系的安全性直接关系到整个应用系统的安全程度。因此，作为密码系统基本要素之一的密钥管理的好坏直接决定整个可信计算平台本身的安全性，是实现终端可信的核心环节，在整个可信计算体系中占有举足轻重的地位。

可信计算平台[20]中用到的密钥分成以下几类：

（1）背书密钥 EK（Endorement Key）

对应公钥、私钥分别表示为 PUBEK、PRIVEK。其中私钥只存在于 TPM 中，且一个 TPM 对应惟一的 EK。EK 可以用来表明 TPM 属主身份和申请“证言身份证书 ”时使用，并不直接提供身份证明。

（2）存储密钥 SK（Storage Keys）

用来提供数据和其它密钥的安全存储。其根密钥为 SRK（Storage Root Key），每个可信计算平台只对应一个惟一的 SRK。

（3）签名密钥（Signing Keys）

非对称密钥，用来对普通数据和消息进行数字签名。

（4）证言身份密钥 AIK[20]（ Attestation Identity Key）

对应一组公私密钥对，专门对来源于 TPM 的数据进行签名，实现对运行环境测量信息进行签名从而提供计算平台环境的证言。每个可信计算平台没有限制 AIK 密钥的数量，但必须保证 AIK 密钥不会重复使用。

5）会话密钥：加密传输 TPM 之间的会话。

在信息处理系统中，密钥的某些信息必须放在机器中，总有一些特权用户有机会存取密钥，这对密码系统的安全是十分不利的。解决这一问题的方法之一是研制多级密钥管理体制。

SRK 作为一级密钥（也称主密钥），存储在安全区域，用它对二级密钥信息加密生成二级密钥。依次类推，父节点加密保护子节点，构成整个分层密钥树结构。在密钥分层树中，叶子节点都是各种数据加密密钥和实现数据签名密

钥。这些动作都应该是连贯的密箱操作。相比之下，纯软件的加密系统难以做到密箱操作。但如果把主密钥、加密算法等关键数据、程序固化在硬件设备 TPM 中，就能解决密箱操作的难题。

在整个密钥体系中，每个密钥在开始创建的时候都指定了固定的密钥属性。密钥按照属性不同分为：可移动密钥（Migratable Key）、不可移动密钥（Non-Migratable）。可移动存储密钥并不局限于某个特定平台，可以由平台用户在平台之间互换而不影响信息交互。不可移动密钥则永久与某个指定平台关联，任何此类密钥泄漏到其它平台都将导致平台身份被假冒。不可移动密钥能够用来加密保护可移动密钥，反之则不行。

### 2.3.2 TPM 密钥管理

密钥管理是可信计算实现技术中的重要一环，密钥管理的目的是确保密钥的真实性和有效性。一个好的密钥管理系统应该做到：

（1）密钥难以被窃取。

（2）在一定条件下窃取了密钥也没有用，密钥有使用范围和时间的限制。

（3）密钥的分配和更换过程对用户透明，用户不一定要亲自掌管密钥。

在可信计算平台中，密钥管理基于安全 PC 平台和 PKI/CA。其中 CA 由证书生成和管理两部分组成。证书生成包括用户公钥证书和私钥证书的生成模块。证书管理主要响应公钥证书请求，CA 为证书用户生成密钥对，请求作废一个证书，查看 CRL，直接从证书服务器中接收有关 CA 密钥或证书的更新、CRL 刷新和用户废弃证书通告等信息。CA 可以是平台制造商、组件生产厂商或者可信第三方。在可信计算平台中所产生的密钥对有些永久存在于 TPM 之中，有些可以存储于外部存储设备中。为了保证可信计算平台中不同类型密钥生成、管理、存储等过程中的安全性，加强对各种密钥的系统管理，提高可信计算平台自身的安全性，本文依据可信计算平台自身安全需求，针对可信计算平台中分层密钥管理体系结构，提出了一种系统化密钥管理模型。

EK 是 TPM 中最核心的密钥，它是 TPM 的惟一性密码身份标识。基于 EK 本身的重要性，在产生 EK 时基于以下几个前提：

（1）EK 在最初创建时就必须是保密的。

（2）EK 创建时，设备必须是真实的并且是没有被篡改的。

（3）密码算法的弱点不会危及该秘密信息的安全。

（4）设备的各项操作不会导致 EK 的泄露。

EK 可以通过密钥生成服务器，采用两种方法来产生：一是使用 TPM 命令，TCG 规范定义了一组背书密钥操作命令，其中创建背书密钥对的命令为：TPM_CreateEndorsement KeyPair，产生密钥长度要求至少 2048 位；另一种方法是密钥"注入"技术，在信任制造商的前提下，由 TPM 制造商产生背书密钥对，然后采用人工方式注入，注入的方法有：键盘输入、软盘输入、专用密钥枪输入等。对比这两种方法，前者必须依赖硬件中提供受保护的功能（Protected Capability）和被隔离的位置（Shielded Location），从而保证在设备内部产生密钥对，而且密钥对是在篡改保护的环境下产生，能够很好地减少密钥对泄露的风险；后者则对环境、管理和操作方法要求较高，首先密钥的装入过程应当在一个封闭的环境下进行，不存在可能被窃听装置接收的电磁泄露或其它辐射，所有接近密钥注入工作的人员应该绝对可靠。采用密钥枪或密钥软盘应与键盘输入的口令相结合，并建立一定的接口规范，只有在输入了合法的加密操作口令后，才能激活密钥枪或软盘里的密钥信息。在密钥装入后，应将使用过的存储区清零，防止一切可能导出密钥残留信息的事件发生。

在 TPM 中，可以采用篡改检测电路和篡改检测协议技术，确保当攻击者试图采用物理攻击时 TPM 内的秘密信息（包括 EK）将自动销毁。同时采用硬件锁机制，建立受保护页面来防止特权软件盗取或者修改秘密信息，保证秘密信息的隐私性和完整性。这样 EK 从开始生成之后，一直到销毁的整个生命周期内都能够安全存储在 TPM 的非易失性存储设备中，永远不会泄露给外界。

## 2.4 本章小结

本章主要讨论了基于 UEFI 开发主要技术和 TPM 核心的功能。UEFI 技术部分首先介绍 UEFI 的总体架构结构与技术特点，然后详细剖析了 UEFI 的各个运行阶段。TPM 核心部分介绍了 TPM 硬件芯片的体系结构，以及一些密钥管理机制，这为后面各章节提供理论基础。

# 第 3 章　虚拟 TPM 驱动程序研究

本章将主要研究虚拟TPM驱动程序的设计实现方法。驱动程序的设计方法与传统的在操作系统下设计方法有很大不同，UEFI下的驱动程序按照UEFI驱动模型标准，根据需要的协议编写出协议下函数接口实例即可完成驱动程序设计工作。因此，在UEFI驱动程序设计之前很有必要详细介绍UEFI驱动模型。

## 3.1　UEFI 驱动模型

统一可扩展固件接口（UEFI）为当今的工业标准总线所支持的设备提供了设备驱动模型。它的出现大大简化了系统设备驱动的设计和实现，Intel 公司为广大设备厂商提供了统一的接口标准，只要用户符合 UEFI 接口标准，即可方便快捷地编写出特定的设备驱动程序。这种设计带来一种好处，那就是系统功能的复杂部分放在了总线驱动里面。该驱动模型要求在驱动程序载入的镜像句柄上生成设 Driver Binding Protocol（驱动绑定协议），该协议查看固件中驱动程序是否与设备控制器连接。如果连接，在设备驱动程序中将在设备控制器句柄生成另一种协议，该协议用于抽象 IO 操作。总线驱动也完成设备驱动类似的任务，只是总线驱动还会生成子控制器句柄[21]。

在任何平台中，固件服务、总线驱动和设备驱动都是由各种特定设备服务提供商提供的，这些设备服务提供商相互合作生成 IO 设备协议，因此 UEFI 驱动模型实际上是把这些驱动服务紧密结合起来。

### 3.1.1　设备驱动程序初始化

驱动程序是在由存储介质加载到内存执行，加载函数为 LoadImage()，可执行文件格式为 PE/COFF。当驱动镜像加载到内存后，系统为其创建一个句柄，并在该句柄下创建 Load Image Protocol 协议，此时驱动程序尚未开始执行，而只是在内存中，直到 StartImage()函数被调用。该过程不只适用于基于 EFI 的驱动程序，也适用于基于 EFI 的应用程序。基于 UEFI 驱动模型的驱动程序必须严格遵守一定规则：

（1）驱动程序与具体硬件设备无关，在基于 UEFI 系统中，最基本的对象

是协议，驱动程序只需要在特定句柄上安装相应的协议实例。

（2）基于 UEFI 的驱动程序要求在自身的镜像句柄上必须安装驱动绑定协议实例，而驱动配置、组件名称、驱动诊断等协议是可选的，具体协议功能参见表 3.1。

表3.1.符合驱动模型的驱动程序所支持的协议

| 协议 | 功能 |
| --- | --- |
| 驱动绑定协议 | 查看驱动程序是否用于特定硬件设备；开始设备驱动程序运行；停止设备驱动程序运行。该协议是可选的。 |
| 组件名称协议 | 为控制硬件的提供命名服务。该协议是可选的。 |
| 驱动诊断协议 | 调试诊断硬件设备的接口。该协议是可选的。 |
| 驱动配置协议 | 为用户提供配置驱动所控制硬件的功能。该协议是可选的。 |

（3）当启动服务函数 ExitBootServices()调用后，如果驱动程序想实现特定的操作，用户可以通过事件通知函数创建事件服务实现。

### 3.1.2 设备驱动

根据 Intel 提出的 UEFI 标准，设备驱动程序要安装在现有已知的设备句柄上，用户不允许创建新的设备句柄。设备驱动程序将 I/O 抽象绑定到总线设备创建的设备句柄上，某些 I/O 抽象可以用于启动基于 UEFI 的 OS，某些 I/O 抽象用于文本输入输出。以 PCI 控制器的设备驱动程序为例说明基于 UEFI 驱动模型驱动程序执行过程，如图 3.1 所示：

（1）PCI 控制器句柄是 PCI 总线句柄子句柄，它下面包含两个基本协议：PCI I/O 协议和设备路径协议。PCI I/O 协议作用是使 PCI 总线支持用于进行 I/O 操作，设备路径协议由 PCI 总线驱动设置，它仅用于定位物理设备，对于与本课题介绍的类似的虚拟设备驱动程序，它是可有可无的。

（2）按上节所述，本例中与 PCI 控制器句柄挂接的设备驱动程序必须在它的镜像句柄上安装驱动绑定协议（Driver Binding Protocol）。驱动绑定协议包括三个重要函数：Supported()，Start()和 Stop()。Supported()用于测试驱动程序是否支持该 PCI 设备控制器，同时，驱动程序检查是否 PCI 设备控制器句柄安装设备路径协议和 PCI I/O 协议。如果驱动程序 Supported()成功返回，表明驱动程序与设备控制器成功连接。此时，驱动程序自动调用 Start()函数。Start()函数

作用是把 I/O 协议添加到 PCI 设备控制器句柄下。为了正常停止设备，驱动绑定协议中还要包括 Stop()函数，在该函数中，删除 Start()函数中添加的协议并释放内存资源。

（3）UEFI 驱动绑定协议 Supported()，Start()和 Stop()三个函数中，经常使用两个函数：OpenProtocol()和 CloseProtocol()。通过 OpenProtocol()或者 LocateProtocol()来获取必要的协议，通过 CloseProtocol()删除协议释放资源，这两个函数都是用来更新 UEFI 固件系统的句柄数据库。因此句柄数据库中的信息非常有用，它可用来检索驱动程序和控制器信息。

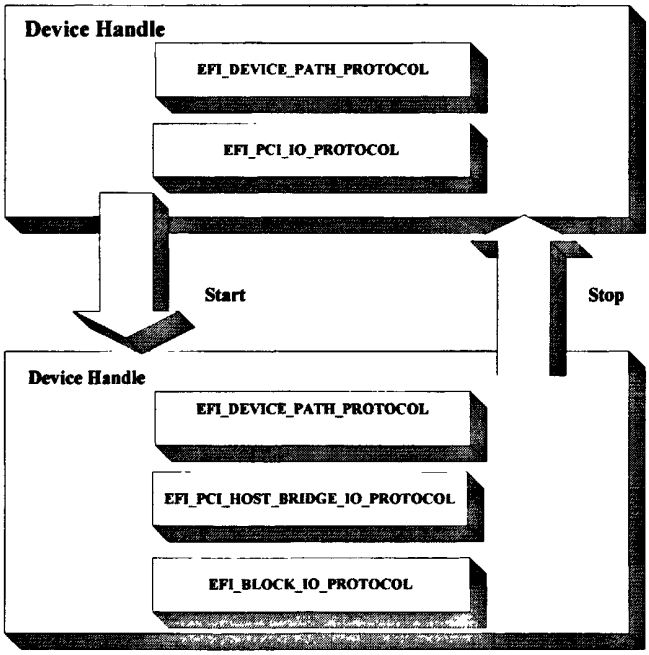


图 3.1 设备驱动程序协议

### 3.1.3 总线驱动

总线驱动和设备驱动基本相同，略有不同的是，如果总线上有设备控制器，那么总线驱动会为这个控制器创建子设备句柄。因此，总线驱动相对设备驱动要复杂许多，这就导致系统功能复杂的部分集中在总线驱动中，而设备驱动在设计和实现上相对简单许多。总线驱动可分为两种：第一种总线驱动是当驱动绑定协议内第一次调用 Start()函数时，总线驱动为总线下每一个控制器创建句柄；第二种总线驱动是通过驱动绑定协议多次调用 Start()函数来创建多个控制

器句柄，这意味着总线从所有挂载的设备控制器中有选择的创建子句柄。总线在为每个设备控制器创建句柄时候是通过枚举检测每个设备控制器实现的，因此平台启动过程中要消耗一段时间来扫描检测每个设备。总线驱动过程如图 3.2 所示：

（1）图中 A,B,C,D 代表总线控制器下的子控制器句柄，每当总线控制器驱动程序执行 Start()函数时，系统就会创建一个子控制器句柄。

（2）总线驱动要为它所创建的每个句柄安装协议接口。协议接口是可选择的，但是提供 I/O 抽象的协议接口是必须要安装的，它用于总线向它挂载的设备控制器提供 I/O 服务。

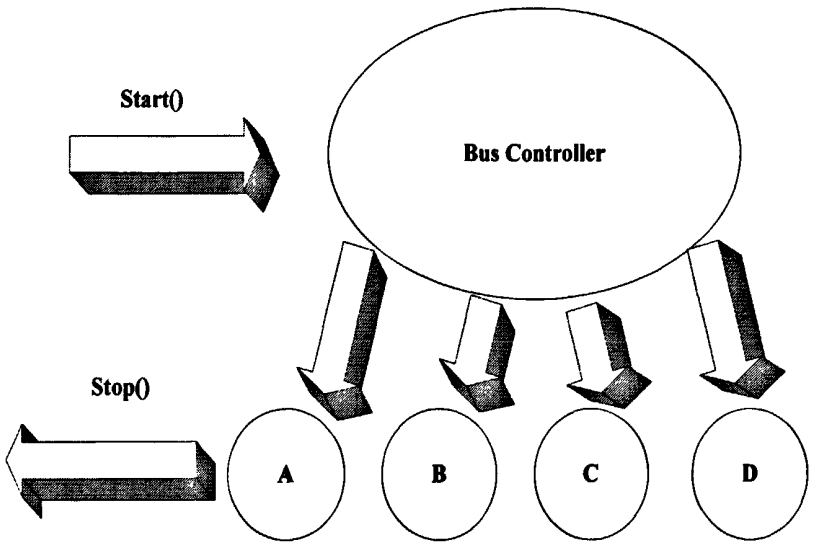


图 3.2 总线驱动过程

### 3.1.4 平台部件选择

基于 UEFI 驱动模型的固件平台提供两个控制函数：ConnetcController()和 Disconnectcontrollor()用于驱动程序连接设备控制器和从设备控制器上卸载，这两个功能通过 UEFI 驱动服务管理器实现，用户通过 UEFI 驱动服务管理器可自选挂接设备和卸载设备，因此根据用户需求方便灵活定制平台部件成为 UEFI 一项很重要的功能。

对于各个平台部件，设备路径是必不可少的，系统核心通过它准确定位系统每一个平台部件，因此每一 UEFI 固件平台的每一平台部件，都有唯一的设

备路径与之对应。除此之外，UEFI 固件平台提供平台驱动重载协议[22]（Platform Driver Override Protocol），这样平台固件会决定某一驱动程序与控制器连接的优先级。

## 3.2 虚拟 TPM 驱动程序架构设计

### 3.2.1 EDKII 开发框架简介

本课题采用EDKII（EFI Development Kit）作为UEFI平台开发框架。EDKII是一种开源框架，其中包含一系列的开发实例和基本的API函数，能够作为基于UEFI驱动程序和应用程序开发、调试的综合平台。EDKII是基于EDK发展而来，针对EDK功能的局限性，EDKII在类库和PCD机制上有了较大的扩展，平台独立性得到了很好的体现，在不久的将来EDKII会得到更广泛的应用。针对本课题，EDKII中重要的开发包如图3.3所示：

Build：框架生成的各个平台的镜像文件、可执行文件以及中间文件。

EdkCompatibilityPkg：EDK代码库，确保EDKII版本支持EDK版本，因此EDKII版本代码库能够全面兼容EDK版本和传统BIOS。

MdeModulePkg：提供通用跨平台组件，例如控制台、磁盘模块。它对系统底层函数进行有效封装，从而使用户能够方便裁剪和定制自定义固件平台。

TianoModulePkg：与MdeModulePkg类似，提供系统重要跨平台组件和用户自定义组件。本课题的虚拟驱动模块置于此目录下。

Shell：UEFI向用户提供的shell源代码目录，该目录是基于EDK的源代码，EDKII版本的shell继承于EDK版本，没有做扩展shell功能。

Nt32Pkg：运行于Windows操作系统下的UEFI模拟器源代码。该模拟器使开发者在不与底层硬件打交道的前提下方便了基于UEFI程序的开发。

MdePkg：平台最重要的开发包，提供所有通用组件底层库函数、协议。

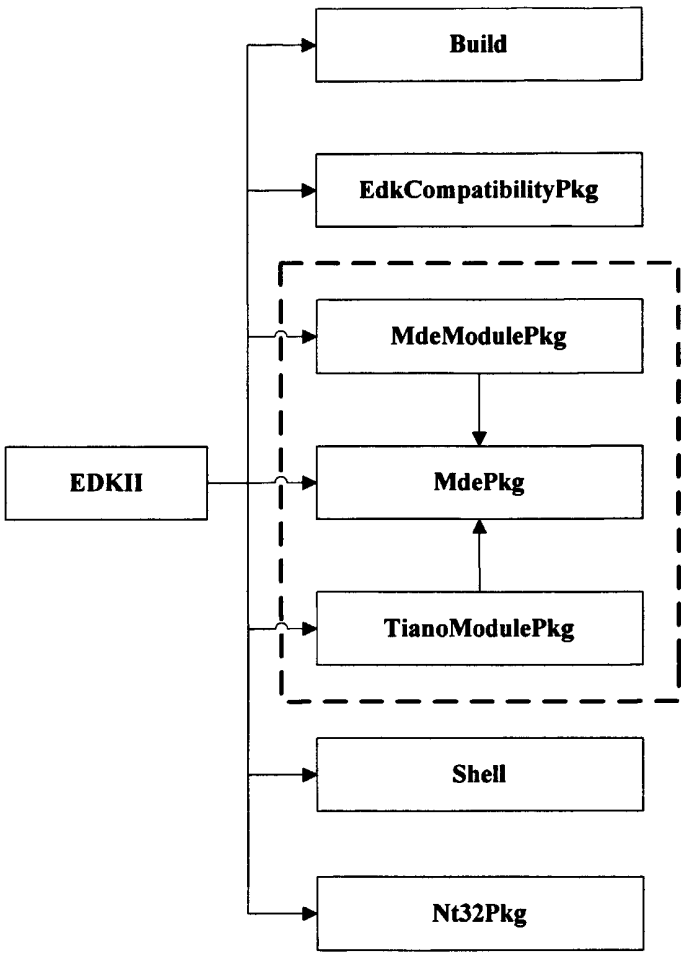


图3.3EDKII重要开发包

### 3.2.2 虚拟驱动开发流程

（1）通过SVN软件将最新EDKII的开发包下载到本地目录，比如整个代码库下载在C:\EDKII。

（2）在C:\EDKII\TianoModulePkg\Universal开发路径下新建文件夹VTpm，在此文件夹下面新建四个文件：VTpmComponentName.c、VTpmComponent.h、VTpmDriver.c、VTpmDriver.h、VTpmDriver.inf，其中VTpmComponentName.c和VTpmComponentName.h用于进行设备命名控制，VTpmDriver.c和VTpmDriver.h做为设备驱动程序主文件，VTpmDriver.inf做为驱动程序配置文件，配置驱动信息和编译选项。配置信息配置如下：

```
[Defines]
INF_VERSION          = 0x00010005
BASE_NAME            = VTpmDriver
FILE_GUID            = E8B6FCFF-2EE7-438d-8477-D9C88C14D3F8
MODULE_TYPE                    = UEFI_DRIVER
ENTRY_POINT                    = InitializeVTpmDriver
[Sources.common]
VTpmDriver.h
VTpmDriver.c
VTpmComponentName.h
VTpmComponentName.c
[Packages]
MdePkg/MdePkg.dec
TianoModulePkg/TianoModulePkg.dec
[LibraryClasses]
UefiLib
BaseLib
UefiDriverEntryPoint
DebugLib
UefiBootServicesTableLib
UefiRuntimeServicesTableLib
DevicePathLib
MemoryAllocationLib
BaseMemoryLib
PrintLib
[Protocols]
gEfiDevicePathProtocolGuid
gEfiSimpleFileSystemProtocolGuid
gEfiVisualTpmProtocolGuid
[Guids]
```

gEfiFileInfoGuid

Defines字段：定义了基本的驱动模块信息，其中INF_VERSION定义了配置文件版本号，默认即可；BASE_NAME定义生成可执行文件名；FILE_GUID由Visual Studio的GUID工具生成，做为系统模块唯一标识；MODULE_TYPE定义模块类型，这里为驱动模型；ENTRY_POINT定义驱动程序代码执行的入口点，代码从该入口点开始执行。

Sources.common字段：定义驱动模块的源代码文件。

Packages：驱动模块所在的开发包，该驱动模块注册于TianoModulePkg开发包中，而且TianoModulePkg依赖于MdePkg开发包。

LibraryClasses：驱动模块所依赖的库，例如内存分配、控制台打印、运行时服务等。

Protocols： 驱动模块所需要的协议，例如设备路径协议、文件系统协议、用户自定义协议等。

Guids：模块所需要的GUID信息。

（3）准备工作完成后，开始编译整个工程。进入VisualStudio2005 Command Prompt，进入C:\EDKII，运行edksetup.bat自动设置默认的开发配置脚本，并输入：

“build -a IA32 -p TianoModulePkg\TianoModulePkg.dsc –m TianoModulePkg\Universal\VTpm\ VTpmDriver.inf”。

（4）工程编译成功后，生成可执行文件 VTpmDriver.efi，在 EFI Shell 下执行 Load VTpmDriver.efi 命令，开始运行设备驱动程序。图 3.4 所示 Shell 下驱动程序加载执行成功的结果。


图 3.4 驱动程序执行成功结果

### 3.2.3 虚拟设备创建

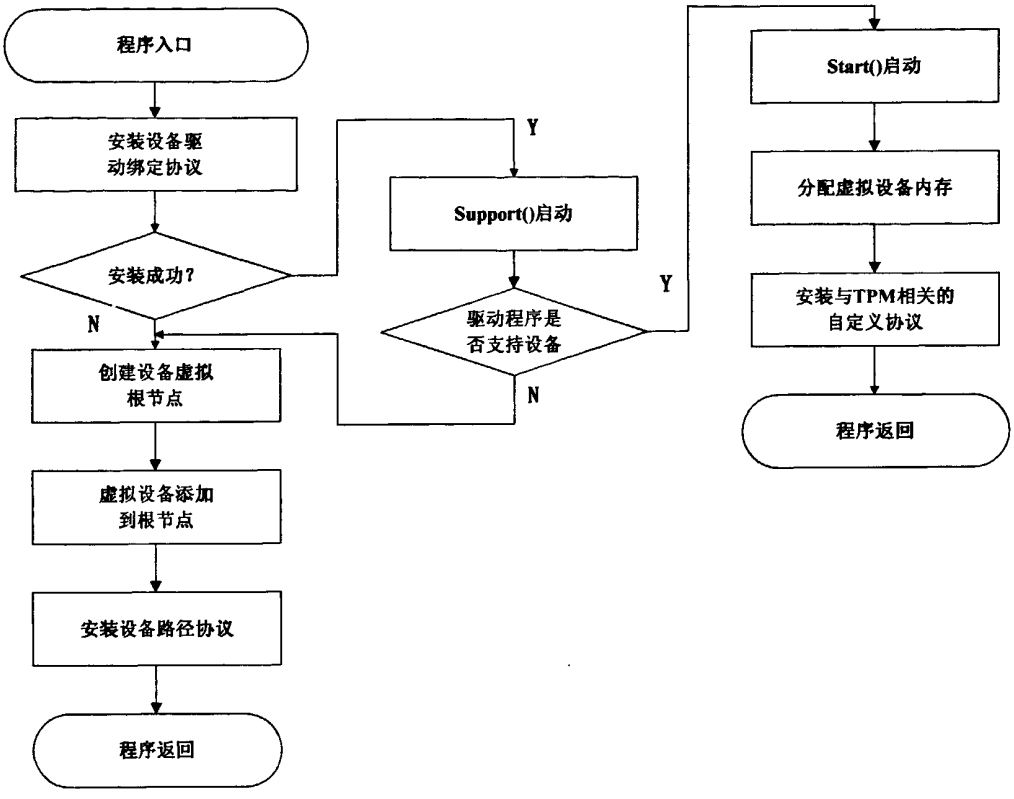


图 3.5 虚拟驱动程序结构图

本课题的虚拟设备抽象成如下结构体：

```
typedef struct {
  INT32                          Signature;
  EFI_VISUAL_TPM_PROTOCOL   VTpmDriverInterface;
} VTPMDEVICE;
```

其中，Signature 代表设备号，VTpmDriverInterface 代表设备控制器句柄下挂载的虚拟 TPM 协议，通过实现 VTpmDriverInterface 下的各个接口实例可达到设备控制目的。由 3.1 节 UEFI 设备驱动模型分析可知，虚拟设备的所有驱动代码在 Driver Binding Protocol 中实现，UEFI 的分配算法将系统中所有设备句柄作为参数传入驱动句柄的 Driver Binding Protocol 接口中。对于本课题虚拟 TPM 驱动程序，先要在驱动程序入口处安装驱动绑定协议，安装该协议的 API

如下[23]：

```
EFI_STATUS
EFIAPI
EfiLibInstallDriverBindingComponentName2 (
IN CONST EFI_HANDLE                          ImageHandle,
N CONST EFI_SYSTEM_TABLE                     *SystemTable,
IN EFI_DRIVER_BINDING_PROTOCOL               *DriverBinding,
N EFI_HANDLE                                 DriverBindingHandle,
IN CONST EFI_COMPONENT_NAME_PROTOCOL         *ComponentName,
OPTIONAL
IN CONST EFI_COMPONENT_NAME2_PROTOCOL        *ComponentName2
OPTIONAL）;
```

参数 DriverBinding 由已定义好的指向驱动绑定协议实例指针传入，接下来创建虚拟设备根节点，安装设备路径协议。注意：对于硬件设备而言，设备路径协议必须安装，用来使 UEFI 系统核心准确定位设备；对于虚拟设备而言，设备路径协议可以不必安装。

本课题中 Support()、Start()和 Stop()三个接口实例是驱动绑定协议规定必须实现的，它们用于识别和控制虚拟 TPM 设备。其中，Support()接口实例主要目的是判断系统是否检测得到该设备控制器句柄，如果检查成功则表示该设备控制器句柄是驱动入口函数中创建的虚拟设备句柄。当该 Support()接口实例调用成功返回后，Start()接口实例开始分配内存抽象虚拟设备，安装虚拟 TPM 协议。Stop()接口实例做善后处理工作，包括停止设备、卸载协议和释放内存资源等，这些处理工作要严格按照顺序进行，否则驱动程序不能正常关闭，这样导致无法下次正常开启驱动程序。UEFI Shell 提供一种验证命令：Reconnect –r，该命令使系统用于重新启动驱动程序挂接设备。本课题中的三个重要的驱动绑定协议接口实例原型如下：

Support()接口实例：

```
EFI_STATUS
EFIAPI
VTpmDriverBindingSupported (
```

```
  IN EFI_DRIVER_BINDING_PROTOCOL   *This,
  IN EFI_HANDLE                    Controller,
  IN EFI_DEVICE_PATH_PROTOCOL      *RemainingDevicePath)
```

Start()接口实例：

```
EFI_STATUS
EFIAPI
VTpmDriverBindingStart (
  IN EFI_DRIVER_BINDING_PROTOCOL   *This,
  IN EFI_HANDLE                    Controller,
  IN EFI_DEVICE_PATH_PROTOCOL      *RemainingDevicePath)
```

Stop()接口实例：

```
EFI_STATUS
EFIAPI
VTpmDriverBindingStop (
  IN   EFI_DRIVER_BINDING_PROTOCOL     *This,
  IN   EFI_HANDLE                      Controller,
  IN   UINTN                           NumberOfChildren,
  IN   EFI_HANDLE                      *ChildHandleBuffer)
```

## 3.3 虚拟 TPM 核心设计

本论文提出的虚拟 TPM 核心设计包括七大模块：I/O 单元、随机数生成器、RSA 加密算法引擎、密钥生成器、存储单元、密钥管理器以及执行单元。同硬件 TPM 芯片相比，软件虚拟的 TPM 在安全性上要打些折扣，但是在应用的灵活性和可扩展性方面却是硬件 TPM 不能比拟的。本论文提供的方案作为 TPM 的基本原型，可以根据实际需要扩展。TPM 在系统安全领域广泛应用，关于它的具体应用实例将在下一章提供。下面就几个重要模块进行详细介绍。

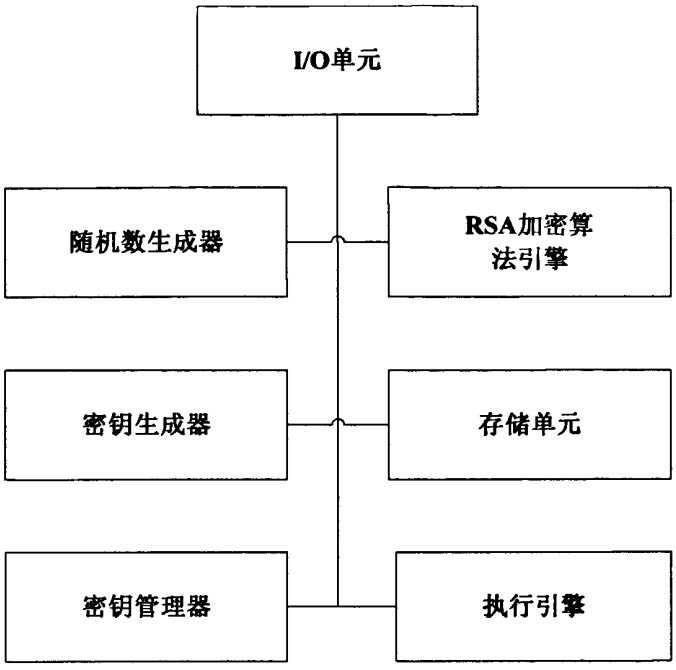


图 3.6 虚拟 TPM 核心结构图

### 3.3.1 I/O 单元模块

硬件 TPM 的 I/O 模块负责从系统总线接收指令，传递到具体的 TPM 核心模块进行处理。本论文提出的软件模拟 TPM 的 I/O 模块负责从 Shell 接口接收命令，例如在 Shell 下运行 a.efi –s，用户 Shell 向驱动程序核心发送生成随机数指令，虚拟 TPM 的 I/O 模块将 Shell 参数发送到随机数生成器产生随机数。

I/O 单元模块是通过驱动程序里用户自定义协议实现的，协议下面的特定的函数接口负责接收 Shell 参数，为了保证 Shell 参数传递到 UEFI 内核空间，UEFI 标准提供一种 Shell 到 UEFI 内核空间的通信机制，通信原理图如图 3.7 所示。该通信机制要求在 EdkCompatibilityPkg 开发包的协议库中添加 UEFI 内核空间用户自定义协议的一份拷贝，本论文将自定义的协议 TianoModulePkg\Include\Protocol\Protocol_self.h 拷贝到 EdkCompatibilityPkg \Foundation\Protocol 下。关于协议的分析与设计将在下一节给出。

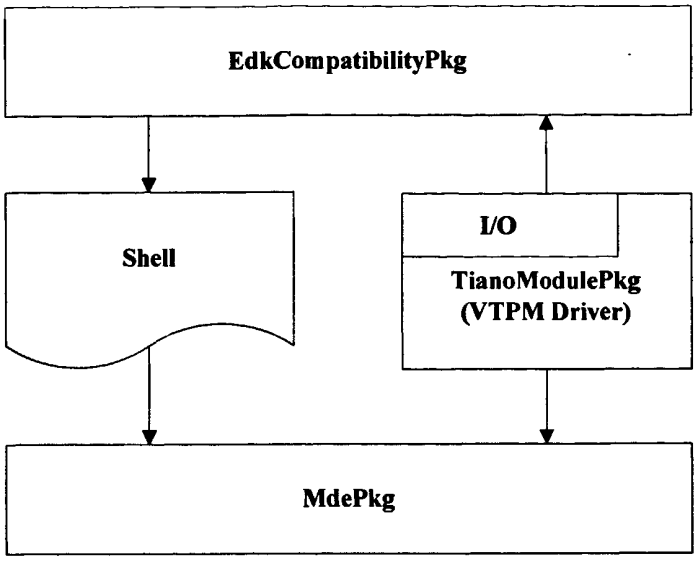


图 3.7Sehll 与 UEFI 通信原理图

### 3.3.2 随机数生成器模块

本模块的设计需要应用到 UEFI 运行时服务（Runtime Services），它为时钟计数产生随机数种子提供必要时钟服务。

UEFI 运行时服务能够为系统预启动和 OS 运行时阶段提供服务，是 UEFI 的一种非常重要的服务机制。这些服务包括：时钟服务、虚拟内存服务、变量服务和平台启动服务等。每种服务包括一些特定功能的 API，例如时钟服务提供了获取设置当前系统时间等功能，虚拟内存服务提供内存物理地址虚拟地址映射等功能，变量服务为系统提供了临时变量存取功能等，启动服务提供系统关机重启功能等。由于本模块只涉及时钟服务，因此将时钟服务功能在本节详细列出，如表 3.2 所示：

表 3.2UEFI 运行时服务时钟功能表

| 函数名称 | 类型 | 说明 |
|---|---|---|
| GetTime | Runtime | 返回系统当前时间和日期信息 |
| SetTime | Runtime | 设置系统时间和日期信息 |
| GetWakeupTime | Runtime | 返回系统当前报警时间和日期信息 |
| SetWakeupTime | Runtime | 设置系统报警时间和日期信息 |

根据时钟需要提供的服务，系统定义了如下重要数据结构（Intel corporation. 2006-07）：

```
typedef struct {
  UINT16   Year;          //  Year:       1998 - 20XX
  UINT8    Month;         //  Month:      1 - 12
  UINT8    Day;           //  Day:        1 - 31
  UINT8    Hour           //  Hour:       0 - 23
  UINT8    Minute;        //  Minute:     0 - 59
  UINT8    Second;        //  Second:     0 - 59
  UINT8    Pad1;
  UINT32   Nanosecond;    //  Nanosecond: 0 - 999,999,999
  INT16    TimeZone;      // TimeZone:-1440 to 1440 or 2047
  UINT8    Daylight;
  UINT8    Pad2;
} EFI_TIME;
```

UEFI 系统虽然采用 C 语言开发，但是没有使用标准 C 语言库函数，内部的所有函数调用完全由 UEFI 代码库提供。限于应用局限性，UEFI 并没有提供随机数相关函数，而本课题的设计需要生成随机数功能。解决方法是利用 UEFI 的运行时时钟服务，模拟 windows 下的随机数种子生成函数写出 UEFI 系统下的随机数种子生成函数。由于本方案与 windows 下随机数生成原理相似，也是通过时钟计数原理产生随机数种子，因此也是“伪随机”，这样在安全性上和硬件 TPM 真随机相比要打折扣。随机数种子发生器函数如下：

```
INT32
EFIAPI
SRand(VOID)
{  ...
  gRT->GetTime (&Time, NULL);
  Seed = (~Time.Hour << 24 | Time.Day << 16 | Time.Minute << 8 | Time.Second);
  Seed ^= Time.Nanosecond;
  Seed ^= Time.Year << 7;
```

```
    ...
}
#define    Rand(Seed)    (((Seed = Seed * 214013L + 2531011L) >> 16) & 0x7fff)
```

其中 gRT 是 UEFI 核心层的运行时服务全局指针，通过 gRT->GetTime()调用可以进行系统时钟访问，时间精确到纳秒级。这里随机数生成器定义为宏，包涵一个参数，这与 windows 下的随机数发生器稍有区别。

### 3.3.3 RSA 加密算法引擎

TPM 的主要功能是进行数据加密，因此算法引擎设计是 TPM 设计的核心。本模块的算法引擎采用目前广泛流行的 RSA 算法，RSA 是非对称密钥算法，能够同时用于加密和数字签名，它是目前最优秀的公钥方案之一。本模块作为虚拟 TPM 的最核心模块，加密算法的选择是非常重要的，但鉴于本课题作为 TPM 的软件实现，因此加密算法的选择可以非常灵活，可以裁剪与更换。本论文选择 RSA 作为加密算法引擎是出于这样几点考虑：

（1）RSA 是普遍被认为安全性较高的加密算法，广泛用于加密数字签名领域。

（2）RSA 采用大数运算，计算强度很大，硬件计算和软件计算的速度相差不大，这样可以认为软件实现在性能上损失相对较小。下面简单介绍一下 RSA 算法：

RSA 的算法涉及三个参数，n、e1、e2。其中，n 是两个大质数 p、q 的积，n 的二进制表示时所占用的位数，就是所谓的密钥长度。

e1 和 e2 是一对相关的值，e1 可以任意取，但要求 e1 与（p-1）*（q-1）互质；再选择 e2，要求（e2*e1）mod（（p-1）*（q-1））=1。（n 及 e1），（n 及 e2）就是密钥对。

RSA 加解密的算法完全相同,设 A 为明文，B 为密文，则：A=B^e1 mod n；B=A^e2 mod n；e1 和 e2 可以互换使用，即：A=B^e2 mod n；B=A^e1 mod n。

本论文的 RSA 算法实现采用大数组作为大数的存储结构，具有一定的数据格式。详细数据结构参考第 4 章。

### 3.3.4 存储单元

存储单元的设计方案是在驱动程序内采取全局变量，开辟 1M 内存空间保存数据。由于采用软件方式模拟 TPM，这样在存储单元内的数据只能是临时存储不能永久保存，系统关闭导致存储单元数据丢失。

## 3.4 协议分析与设计

### 3.4.1 Driver Binding Protocol

Driver Binding Protocol 定义如下：

```
typedef struct _EFI_DRIVER_BINDING_PROTOCOL {
  EFI_DRIVER_BINDING_PROTOCOL_SUPPORTED    Supported;
  EFI_DRIVER_BINDING_PROTOCOL_START        Start;
  EFI_DRIVER_BINDING_PROTOCOL_STOP         Stop;
  UINT32                                   Version;
  EFI_HANDLE                               ImageHandle;
  EFI_HANDLE                               DriverBindingHandle;
   }EFI_DRIVER_BINDING_PROTOCOL;
```

UEFI 系统通过调度 Driver Binding Protocol 来为设备安装驱动。安装该协议的句柄被 UEFI 视为驱动句柄，并将系统中所有设备句柄依次传入该协议的接口函数[24]。

从上述定义可以看出，该协议有三个函数 Supported()、Start()和 Stop()。UEFI 系统调用 Supported()来检测一个设备是否被这个驱动支持，如果该函数返回成功值 EFI_SUCCESS，则其 Start()函数将被调用。否则，系统继续检测其他设备是否被该驱动支持。Stop()用来停止这个驱动，并且这一动作在两种情况下发生，即用户卸载该协议或者其依赖的协议被卸载。

（1）EFI_DRIVER_BINDING_PROTOCOL.Supported()定义如下：

```
typedef
    EFI_STATUS
(EFIAPI *EFI_DRIVER_BINDING_PROTOCOL_SUPPORTED)(
```

```
IN EFI_DRIVER_BINDING_PROTOCOL        *This,
IN EFI_HANDLE                          ControllerHandle,
IN EFI_DEVICE_PATH_PROTOCOL            *RemainingDevicePath
);
```

该函数用来测试驱动是否支持某个给定的 ControllerHandle。其中的 This 参数是结构体指针，在 C 语言面向对象的程序编写中需要主动传入 This 指针。ControllerHandle 就是给定的，需要被测试的设备 HANDLE。RemainingDevicePath 参数会被设备驱动忽略，而总线驱动则一般会同时利用该参数指定的子设备和 ControllerHandle 指定的总线控制器来判断驱动是否支持。

（2）EFI_DRIVER_BINDING_PROTOCOL.Start()定义如下：

```
typedef
   EFI_STATUS
(EFIAPI *EFI_DRIVER_BINDING_PROTOCOL_START) (
IN EFI_DRIVER_BINDING_PROTOCOL      *This,
IN EFI_HANDLE                        ControllerHandle,
IN EFI_DEVICE_PATH_PROTOCOL          *RemainingDevicePath
);
```

该函数用来启动一个驱动支持的控制器。控制器由 ControllerHandle 指定，而驱动则由 This 指针指定。当一个控制器被成功启动后，函数返回 EFI_SUCCESS 值，反之返回相应的错误值。如果该驱动是设备驱动，则 RemainingDevicePath 会被忽略；如果该驱动是总线驱动，并且 RemainingDevicePath 为空，那么总线控制器上的所有设备将被创建为该控制器的子设备，并对应相应的 HANDLE。如果该驱动是总线驱动，并且 RemainingDevicePath 不为空，则驱动将试图为 RemainingDevicePath 制定的子设备创建设备 HANDLE。即便如此，总线控制器仍然可以遍历所有的设备，但是只能创建指定设备的 HANDLE。

（3）EFI_DRIVER_BINDING_PROTOCOL.Stop()定义如下：

```
typedef
    EFI_STATUS
(EFIAPI *EFI_DRIVER_BINDING_PROTOCOL_STOP) (
```

```
IN EFI_DRIVER_BINDING_PROTOCOL   *This,
IN EFI_HANDLE                    ControllerHandle,
IN UINTN                         NumberOfChildren,
IN EFI_HANDLE                    *ChildHandleBuffer OPTIONAL
);
```

该函数用来停止驱动，当驱动被用户卸载或其依赖的驱动被卸载时该函数被调用。Stop()的行为取决于驱动是设备驱动还是总线驱动。NumberOfChildren。如果该驱动是设备驱动，则传入的 NumberOfChildren 应当为零。如果该驱动是总线驱动，则传入的 NumberOfChildren 应当不为 0。对于前者，Stop()应当对应的释放 Start()中申请的所有系统资源，包括栈内存，或是动态获取的页面。对于后者，则必须根据 ChildrenHandleBuffer 释放为所有子设备创建的数据结构，包括为子设备创建的 HANDLE 以及装载在这些 HANDLE 上的所有协议。

在分析完以上数据结构后需要指出的是，在 EDK II 中每个驱动模块都有入口函数，但入口函数并不是起到驱动设备的作用。入口函数应当负责安装 Driver Binding Protocol，由这个协议来驱动设备。入口函数只在驱动二进制文件加载时被调用一次，该协议则依据驱动模型随时被系统调用。

### 3.4.2 Device Path Protocol

Device Path（设备路径）表示 UEFI 系统中实体如设备、文件等的物理路径，以及这些实体的树形关系，对于总线结构意义重大。设备路径由一组连续的节点组成，这些节点由 UEFI 定义的 Device Path Protocol 表示。

```
typedef struct{
  UINT8       Type;
  UINT8       SubType;
  UINT8       Length[2];
  }EFI_DEVICE_PATH_PROTOCOL;
```

Device Path Protocol 的定义如上。这个结构体通常作为设备路径节点的头部，紧接结构体之后，连续的存放工业标准或者用户自定义数据。该结构体 Length 域表示包括头部和数据在内的设备路径结点的总长度。数据和该结构

体一起构成 Device Path 的一个结点。

UEFI 标准规定的设备路径结点的类型有六种，见表 3.3。SCSI、ATAPI、USB 以及网络设备的 IP 地址等使用 0x03 消息设备路径。硬盘、光驱以及其中存储的文件使用 0x04 媒体设备路径。完整的路径以 0x7F 类型的结点结尾。

表 3.3 UEFI 设备路径类型

| Type | Description | Type | Description |
|---|---|---|---|
| 0x01 | Hardware Device Path | 0x04 | Media Device Path |
| 0x02 | ACPI Device Path | 0x05 | BIOS Boot Specification Device Path |
| 0x03 | Messaging Device Path | 0x7F | End of Device Path |

对于处于硬件系统根结点的总线控制器，设备路径只有 Root Bus Controller 结点，并以 Tail Node 结尾。连接在该总线上的设备则在该总线控制器设备路径后加上自己的结点（如 Level 1 Device 和 Level 1 Bus Controller）构成新的设备路径。UEFI 系统中的整个设备路径树形结构以该方式递归形成。设备路径以树形结构表示系统中设备间的父子关系如图 3.8 所示：

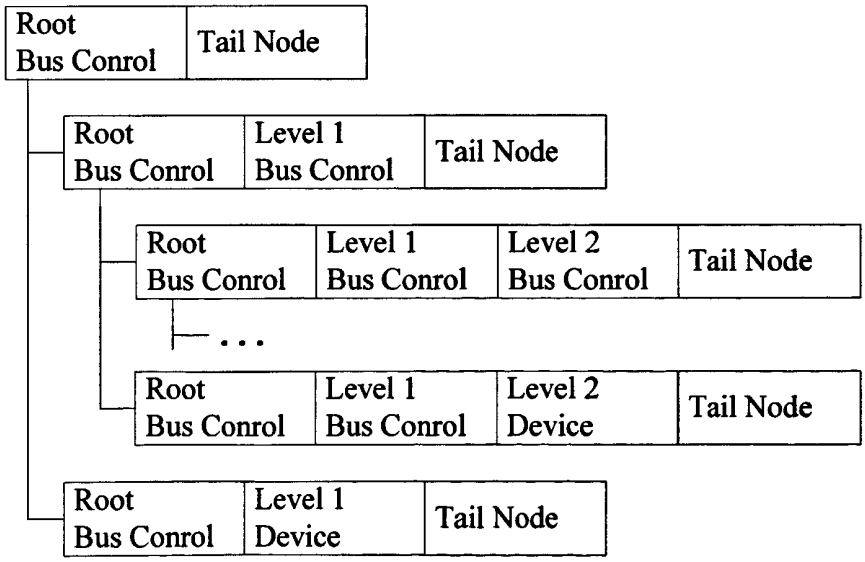


图 3.8 设备路径的树形结构

从图 3.8 可以看出，所有设备的设备路径中均包括其祖先结点的所有信息。特别的，文件的设备路径中包含了它所属硬件的路径信息。

### 3.4.3 Virtual TPM Protocol

根据上一节的虚拟 TPM 核心设计，本论文提出的自定义协议设计如下：

```
typedef struct _EFI_VIRTUAL_TPM_PROTOCOL {
EFI_VTPM_START              Start;
EFI_VTPM_READ               Read;
EFI_VTPM_WRITE              Write;
EFI_VTPM_SETTAG             SetTag;
EFI_VTPM_CLEAR              Clear;
EFI_VTPM_EXPORT             Export;
EFI_VTPM_CACULATE           Caculate;
} EFI_VIRTUAL_TPM_PROTOCOL;
```

该协议提供一种抽象的总线控制器用来提供虚拟 TPM 的操作接口，定义中的所有接口都为虚拟 TPM 的核心功能服务，包括传递参数、传递数据、传递命令和结果回馈等等，具体接口功能及实例分析如下：

（1）Start

自定义协议 Virtual TPM Protocol 中的函数接口 Start 实例如下：

```
EFI_STATUS
EFIAPI VTpmStart(
  IN EFI_VIRTUAL_TPM_PROTOCOL       *This
  )
```

该接口实例作用是作为虚拟 TPM 开关，当该接口实例被调用后，开始进行加密运算，运算结果存储在临时内存区。其中 This 参数是结构体指针，作为输入，指向协议结构体本身，在 C 语言面向对象的程序编写中需要主动传入 This 指针。如果函数调用成功，则返回 EFI_SUCCESS，否则异常退出。

（2）Read

自定义协议 Virtual TPM Protocol 中的函数接口 Read 实例如下：

```
EFI_STATUS
EFIAPI VTpmRead(
  IN EFI_ VIRTUAL _TPM_PROTOCOL       *This,
```

```
OUT EFI_FILE_INFO                    *Buffer,
OUT CHAR16                           *DerectoryPath
)
```

该接口实例作用是负责在 UEFI 驱动程序空间和 UEFI Shell 之间交换数据，数据流从 UEFI 驱动程序到 UEFI Shell，它为 Shell 接口下系统监控应用提供特定服务。其中的 This 参数是结构体指针，指向协议结构体本身；Buffer 作为输出，指向一段 EFI_FILE_INFO 类型的缓冲区，缓冲文件信息，DerectoryPath 同样作为输出，用于缓存文件所在路径信息，具体的系统监控应用请参考下一章。如果函数调用成功，则返回 EFI_SUCCESS，否则异常退出。

（3）Write

自定义协议 Virtual TPM Protocol 中的函数接口 Write 实例如下：

```
EFI_STATUS
EFIAPI VTpmWrite (
  IN EFI_ VIRTUAL _TPM_PROTOCOL      *This,
  IN EFI_FILE_INFO                   *Buffer,
  IN CHAR16                          *DerectoryPath
  )
```

该接口实例作用是负责在 UEFI 驱动程序空间和 UEFI Shell 之间交换数据，数据流从 UEFI Shell 到 UEFI 驱动程序，它为 Shell 接口下系统监控应用提供特定服务。参数意义与函数接口 Read 实例类似，如果函数调用成功，则返回 EFI_SUCCESS，否则异常退出。

（4）SetTag

自定义协议 Virtual TPM Protocol 中的函数接口 SetTag 实例如下：

```
EFI_STATUS
EFIAPI VTpmSetTag(
IN EFI_ VIRTUAL _TPM_PROTOCOL      *This,
IN BOOLEAN                         Tag
)
```

该接口实例作用是控制数据流方向。在实际应用中，常常需要驱动程序与 UEFI Shell 进行数据交换，当变量 Tag 值为 TRUE 时，数据流方向是从驱动程

序到 UEFI Shell；当遍历 Tag 值为 FAILSE 时，数据流方向是从 UEFI Shell 到驱动程序。这在实际应用中是非常重要的。如果函数调用成功，则返回 EFI_SUCCESS，否则异常退出。

（5）Clear

自定义协议 Virtual TPM Protocol 中的函数接口 Clear 实例如下：

```
EFI_STATUS
EFIAPI VTpmClear(
IN EFI_VIRTUAL_TPM_PROTOCOL          *This
)
```

该接口实例作用是清空虚拟 TPM 的存储单元数据。在 TPM 的存储单元内，经常会存留一些临时数据，由于该部分数据位于内存全局存储区，在驱动程序执行期间不会自动清除，因此保留该接口，这样在实际使用过程中用户需要手动调用该接口实例清理该存储单元。如果函数调用成功，则返回 EFI_SUCCESS，否则异常退出。

（6）Export

自定义协议 Virtual TPM Protocol 中的函数接口 Export 实例如下：

```
EFI_STATUS
EFIAPI VTpmExport(
IN EFI_ VIRTUAL _TPM_PROTOCOL          *This,
OUT INT32                               *eBuf,
OUT INT32                               *dBuf,
OUT INT32                               *nBuf
)
```

该接口实例特用于 RSA 加密算法引擎，它将加密密钥和解密密钥导出至 UEFI Shell，供 Shell 应用程序使用。如果函数调用成功，则返回 EFI_SUCCESS，否则异常退出。

（7）Caculate

自定义协议 Virtual TPM Protocol 中的函数接口 Caculate 实例如下：

```
EFI_STATUS
EFIAPI VTpmCaculate(
```

```
IN   EFI_ VIRTUAL _TPM_PROTOCOL        *This,
IN   INT32                             *mContext,
IN    INT32                             *e,
IN    INT32                             *n,
OUT   INT32                              *cContext
)
```

该接口实例特用于RSA加密算法引擎，它的作用是直接向外部应用程序如UEFI Shell提供明文密文相互转换的服务接口。这样在UEFI Shell下提供明文数据，传入驱动程序后进行加密运算。如果函数调用成功，则返回EFI_SUCCESS，否则异常退出。

## 3.5 本章小结

本章详细地介绍了UEFI环境下虚拟TPM驱动程序的设计方法。根据UEFI驱动模型，分析了虚拟 TPM 驱动程序的架构的同时，重点介绍了程序开发流程和相关协议的分析与设计。TPM驱动程序架构核心部分借鉴标准TPM核心，但受软件模拟限制，部分 TPM 硬件特性无法实现。协议部分涉及三个协议：驱动绑定协议（Driver Binding Protocol）、设备路径协议（Device Path Protocol）、虚拟TPM协议（Virtual TPM Protocol）。驱动绑定协议是遵循UEFI驱动模型驱动程序不可缺少的协议，设备路径协议在本课题虚拟驱动中可以不必添加，因此没有必要过多提及，值得一提的是本课题的自定义协议：虚拟TPM协议，是设计的关键，UEFI标准允许用户根据特定需要自定义协议，本文的TPM核心功能在自定义协议中集中体现。

# 第 4 章　基于虚拟 TPM 的应用设计与实现

上一章介绍了虚拟 TPM 的驱动程序研究和设计情况，本文针对安全应用，修改并裁剪了 TPM 的核心功能，并在此基础上，自定义协议接口实例，使 TPM 核心功能被封装到 UEFI 驱动程序中供应用程序使用。

本章在自定义设计协议基础上实现两个简单安全应用示例：文件加密解密系统和文件监控系统。两个示例均充分利用上一章自定义协议下的各接口实例，实现特定的功能。本章的应用设计全部基于 UEFI Shell，因此，UEFI Shell 与 UEFI 驱动程序的数据交互显得格外重要。在介绍这两个应用示例前首先简单介绍 UEFI Shell 程序开发的基本情况。

## 4.1　UEFI Shell 程序简介

UEFI Shell 是一个用来启动应用程序、载入 EFI 协议和设备驱动程序、执行简单脚本的控制台界面。它和 DOS 控制台相仿，只能够进入 FAT（VFAT）格式化的介质，其运行界面，如图 4.1 所示。

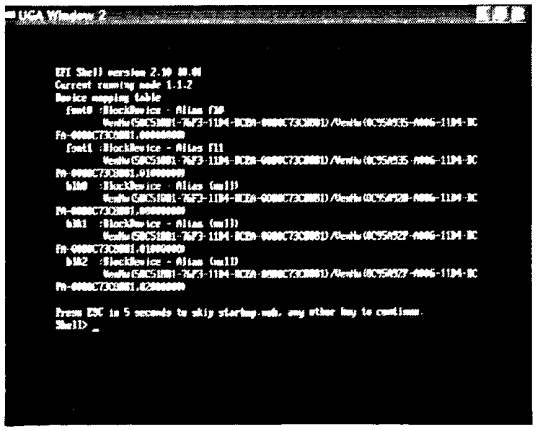


图 4.1　UEFI Shell 运行环境截图

UEFI Shell 作为 UEFI 体系下的应用程序，也有其一定结构。一个标准的 Shell 应用程序应当具有如下所示入口函数。

Main (

```
  IN EFI_HANDLE                ImageHandle,
  IN EFI_SYSTEM_TABLE          *SystemTable
  )
{
  EFI_STATUS    Status;
  EFI_SHELL_APP_INIT (ImageHandle, SystemTable); // 初始化 Shell 环境
  //
  // 程序代码
  //
  return Status;
    }
```

在 EDK II 中，Shell 程序的主函数名不必是 Main，可以在 Shell 模块的 INF 文件中指定入口函数名称。但是入口函数类型必须是上述代码所示。

入口函数与 Windows 或者 Linux 的 main 函数不同，不能从函数参数获得命令行参数。Shell 应用使用另外的方法获得命令行参数 Shell Interface Protocol。EDK II 中提供该协议的全局实例 SI，用户可以通过诸如 SI->Argv、SI->Argc 等方式获取命令行参数。

Linux 中 main 函数的第三个参数用以获得环境变量，Shell 应用则使用 Shell Environment Protocol 在 Shell 中的全局实例 SE 或者 SE2，后者是目前 EFI 中更高版本的 Shell Environment Protocol 实例。图 4.2 代码中初始化 Shell 环境的主要任务即是初始化 Shell Interface Protocol 和 Shell Environment Protocol 实例指针 SI 和 SE。程序代码中可使用 SI 和 SE 作为指针获得这两个协议提供的信息以及服务。关于 Shell Interface Protocol 和 Shell Environment Protocol 两个协议本文不做介绍。

## 4.2 虚拟 TPM 的加密与解密应用设计与实现

本文基于虚拟 TPM 的应用程序设计方案是在 UEFI Shell 下提出的，其思路是：在标准 UEFI Shell 程序代码中利用 Shell 框架已定义好的启动服务全局变量 BS 下的本地协议打开自定义协议 EFI_VIRTUAL_TPM_PROTOCOL，通

过指向该协议指针间接访问驱动程序中封装的 TPM 资源。

### 4.2.1 密钥对生成

关于虚拟 TPM 内部核心加密算法已在 3.3 节详细介绍，本节中对于 RSA 算法的密钥对生成是通过标准 UEFI Shell 程序打开 EFI_VIRTUAL_TPM_PROTOCOL 向驱动程序发送控制命令，在驱动程序实现密钥生成运算，将运算结果的密钥对返回给标准 UEFI Shell 程序，结果显示在 Shell 程序终端上。流程图如图 4.3 所示：

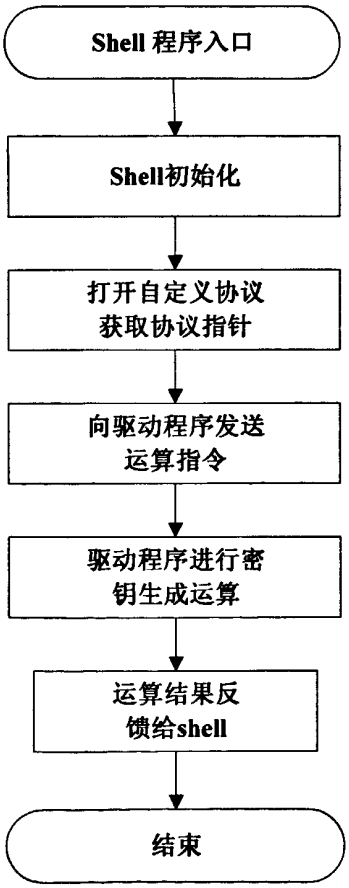


图 4.3 密钥生成流程图

### 4.2.2 密钥对保存和载入

类似于 Windows 或者 Linux 文件操作，UEFI Shell 支持文件操作。标准 UEFI

Shell 代码库中\Shell\Library\FileIO.c 文件是对文件操作的定义，包括文件的打开和读写访问。因此，可以方便的保存从驱动程序中导出的密钥，也可以方便的将保存好的密钥载入内存进行加密解密使用。下面的 Shell API 是对文件打开操作：

```
EFI_STATUS
LibOpenFileByName(
  IN   CHAR16                          *FileName,
  OUT EFI_FILE_HANDLE                  *FileHandle,
  IN UINT64                            OpenMode,
  IN UINT64                            Attributes
  )
```

其中，FileName 是要打开的文件名，16 位字符格式；FileHandle 是文件句柄，作为函数输出参数，通过该 API 的调用系统为打开的文件分配一个句柄，供后面的读写操作；OpenMode 是控制文件读写模式，常用的模式有 EFI_FILE_MODE_READ EFI_FILE_MODE_WRITE EFI_FILE_MODE_CREATE 三种，分别对文件读写和创建；Attributes 作为保留参数，一般设为 0。

文件的读写 API 分别为：

```
EFI_STATUS
LibReadFile (
    IN EFI_FILE_HANDLE      FileHandle,
    IN OUT UINTN            *ReadSize,
    OUT VOID                *Buffer
    )和 EFI_STATUS
  LibWriteFile (
    IN EFI_FILE_HANDLE    FileHandle,
    IN OUT UINTN          *BufferSize,
    OUT VOID              *Buffer
    )
```

其中，FileHandle 为文件句柄，打开文件操作时获得，BufferSize 和 Buffer 分别

为文件进行读写控制的缓冲区大小和缓冲区首指针。

### 4.2.3 文件加密

文件加密软件在现在 OS 上应用非常普遍，但是在 UEFI 下目前尚未实现。本论文通过在 UEFI 框架下的虚拟 TPM 设计能够实现这一应用，这样就会使得文件系统加密独立于 OS，能够在 Pre Boot 环境下得到文件加密目的，从而能够更加安全的对文件实施保护，该方法在安全领域具有很高的创新意义。

本论文提出的文件加密算法采用 3.3.3 节 RSA 加密算法引擎，该引擎封装在驱动程序核心中，而数据结构在 UEFI Shell 环境下，采用链表结构。具体方案如下：

依次扫描待加密文件的每个字符，并将这些字符存储在链表 A 中，链表中的每个节点定义如下：

```
struct CharNode
{
INT32   CharArray[MAX]; //每个节点用大数组存储一个字符的 ASCII 码
struct   CharNode   *Next;      //指向下一节点
};
```

其中 CharArray 是一个大数组，每一个节点用来存储待加密文件的每一个字符的 ASCII 码，存储的数据格式如图 4.4 所示。数组前 MAX-2 项用于存放字符 ASCII 码值的每一位，注意，这里存放顺序是倒序存放的，即 CharArray [0]存放的是 ASCII 码值最低位，这样做的好处是数组存储完所有的 ASCII 码值以后，数组的其他项补 0，倒数后两项除外。数组倒数第二项存储 ASCII 码符号位，0 表示 ASCII 码值为负，1 表示 ASCII 码值为正。数组最后一项存放 ASCII 码值的数据位数。为了达到安全目的，3.3 节 RSA 加密算法引擎采用大数进行加密运算，因此数据存储格式与 3.3 节的 RSA 算法数据格式一直，都是采用数组的形式表示大数。

| | | | | |
|---|---|---|---|---|
| 2 | ..... | 0 | 符号位 | 有效数据个数 |

图 4.4 数据存储格式

待文件中的每个字符都保存至链表 A 以后，Shell 应用程序中再次开辟缓冲区维护一个新的链表 B，用来保存加密后的数据。每个加密后的数据依然采

用图 4.4 的格式保存至链表 B 的节点中。通过 Shell 应用程序中的打开协议函数获取指向 EFI_VIRTUAL_TPM_PROTOCOL 的指针，来将 RSA 运算命令发送至驱动程序并将运算结果返回至 Shell。最终，遍历链表 B 将加密后数据保存至文件中。文件加密过程的流程图如图 4.5 所示：

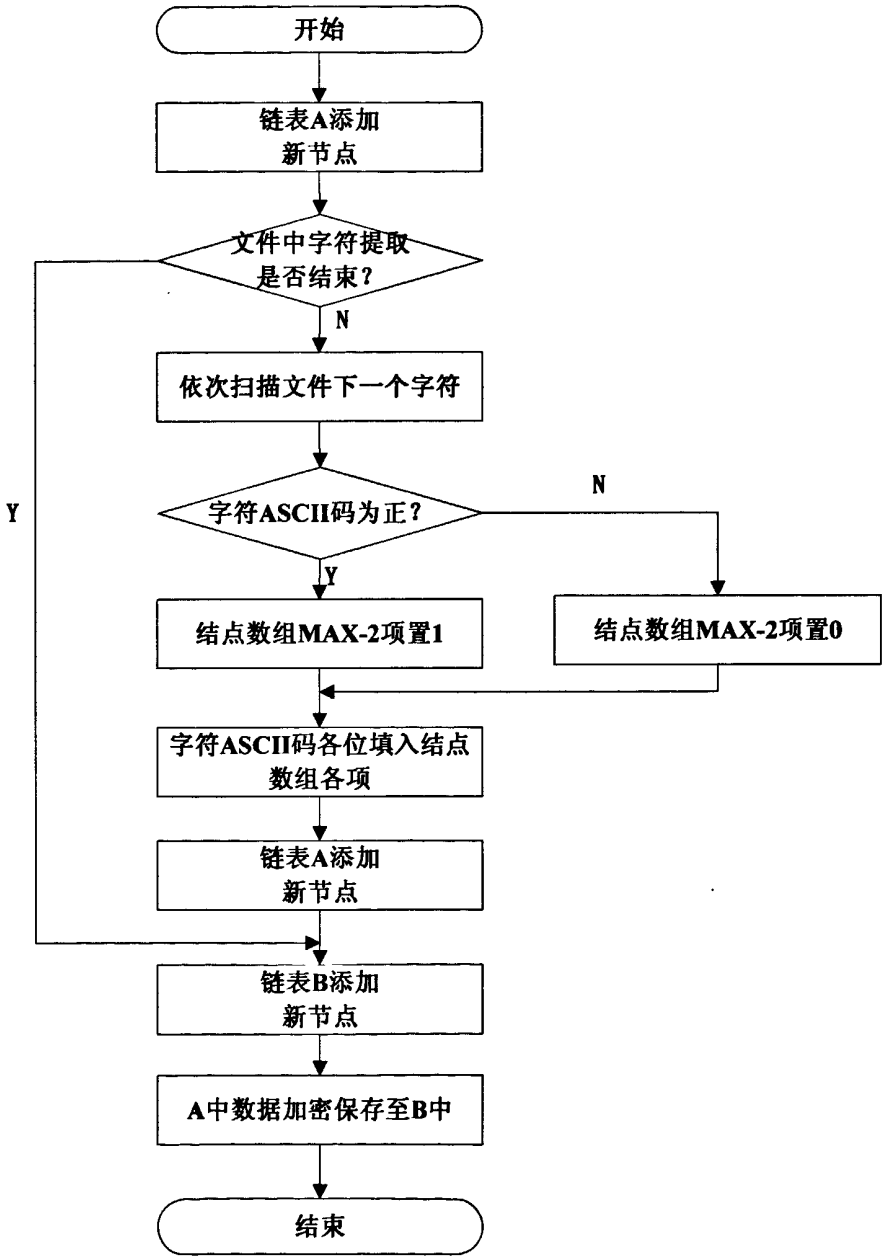


图 4.5 文件加密流程图

### 4.2.4 文件解密

由上一节文件加密过程分析可知，加密后的文件内容是从 0 到 9 的数字依次排列，每个明文字符的经过加密后，密文数据具有一定格式，也可以看作数据帧：符号位+密文数据位数+密文正文。例如，文件只有一个字符 a，经过加密后的密文是 13412467733013468134834051007848655l1，其中，第一个数字 1 是符号位表示密文是正值，第二三个数字 34 表示密文数据的位数是 34 位，其余的数字是密文数据。

文件解密过程可看作文件加密的逆向过程，具体方法与文件加密类似，都是依次扫描待处理的文件，并在内存维护两个链表，一个用于保存密文内容，另一个用于保存解密后的明文内容。区别在于密文在文件的存储中有特定的存储格式，解密过程需要对这种特定的数据格式进行解析。本文提出的方案是符号位+密文数据位数+密文正文，并且密文数据位数规定两位。因此将密文保存至链表时的方法如下：

（1）解析每一帧密文数据，每帧数据的前三位分别是符号位和数据位数，其中数据位数占两位，然后分别将此三位数据提取保存至链表结点数组的 MAX-2 和 MAX-1 项中。

（2）从每帧加密数据的第四位起扫描到该帧密文数据位数后，此部分数据是密文数据正文，然后将该部分数据保存至链表结点数组的第 0 项至第 MAX-3 项中，链表结点数组多余项补 0。

经过上述两步操作后，整个密文文件内容已经保存至链表中，与上一节文件加密过程类似，通过 Shell 应用程序中的打开协议函数获取指向 EFI_VIRTUAL_TPM_PROTOCOL 的指针，将 RSA 运算命令发送至驱动程序并将解密运算结果返回至 Shell。最终，在利用建立新链表将解密后数据保存至文件中。

## 4.3 虚拟 TPM 文件监控系统应用设计与实现

### 4.3.1 文件系统遍历算法

文件监控[25]是系统安全领域的一项重要应用，通过对文件系统的监控来捕捉文件异常，从而能够发现恶意木马或病毒。本文在虚拟 TPM 设计的基础上，

在 UEFI Shell 下设计并实现了简单的文件监控系统，能够跟踪磁盘上的任意目录，对该目录实施监控。当监控目录下的任意文件夹或者文件发生改变时，系统会生成 Log 信息供系统管理员分析。标准 UEFI 提供的文件属性如下：

```
typedef struct {
  UINT64     Size;
  UINT64     FileSize;
  UINT64     PhysicalSize;
  EFI_TIME   CreateTime;
  EFI_TIME   LastAccessTime;
  EFI_TIME   ModificationTime;
  UINT64     Attribute;
  CHAR16      FileName[1];
} EFI_FILE_INFO;
```

其中，Size 为结构体 EFI_FILE_INFO 本身大小；FileSize 为文件大小，单位是字节；PhysicalSize 为文件在文件系统卷中占用的物理空间大小；CreateTime 为文件创建日期时间，日期格式在在 3.3.2 节中定义；LastAccessTime 是上次访问文件的时间；ModificationTime 是上次修改文件的时间；Attribute 为文件属性字段；FileName 为文件非终端 Unicode 命名。因此，可以根据文件的上述属性判断文件的修改情况，同时，还需要一种有效方法遍历待监控目录下的所有文件和文件夹。本文提供一种如下遍历目录的算法，流程图如图 4.6 所示。

（1）获得当前监控目录有效句柄，该句柄通过 UEFI Shell 应用程序 API 调用获得，传入监控目录名作为参数，默认为 Shell 终端当前目录[25]。该过程与 Linux 或 Windows 下控制台应用程序开发类似。

（2）通过上个步骤获得的监控目录句柄，将该句柄下的目录搜索位置置 0，即初始化当前读入文件该过程通过句柄下的 SetPosition 函数实现的。

（3）进入死循环，利用步骤（1）获得的监控目录句柄依次不间断读取文件信息，直到该监控目录下所有文件读取完毕为止，退出死循环。

（4）在上个步骤中遇到读取的文件是文件夹时，做一次算法递归调用，此时，递归调用传入的参数是子文件夹句柄。

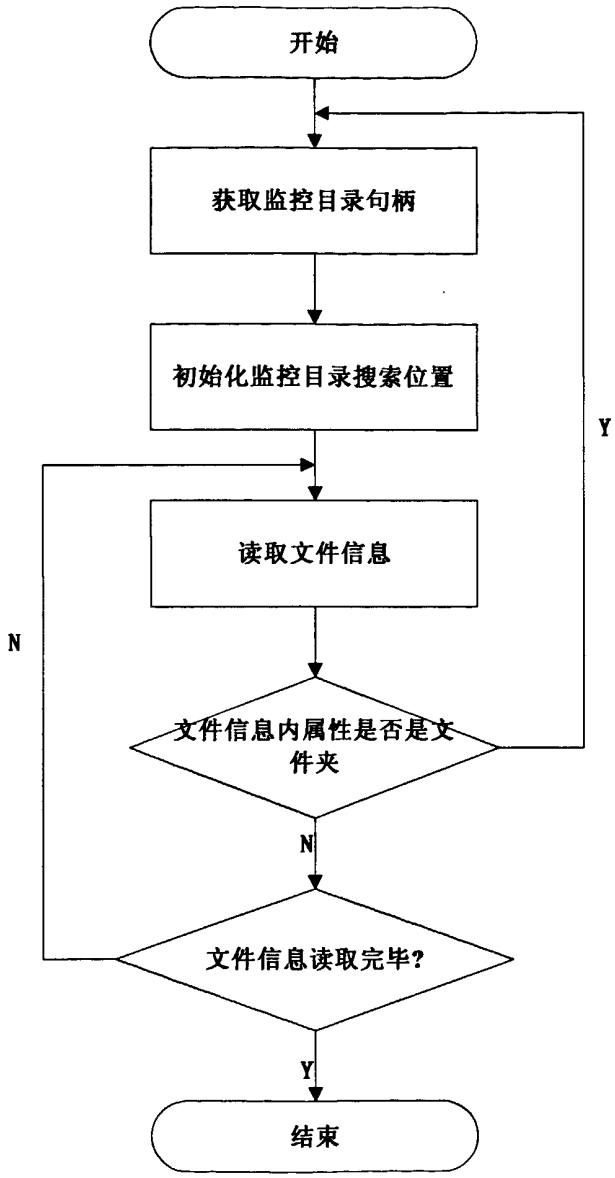


图 4.5 文件遍历算法流程图

判断文件读取是否完毕的方法如下：当每次读取文件信息时，这些文件信息将会保存到一段缓冲区内，只要判断是否该缓冲区大小为 0 即可获知监控目录下的所有文件是否读取完毕。

### 4.3.2 文件信息上传与下载

实现文件监控系统[27]，需要在某时刻开启监控时将监控目录内的所有文件

信息保存至虚拟 TPM 驱动程序的存储单元，该存储单元的接口实例以及与 Shell 通信接口在 3.3.4 节设计设计完毕。因此，本节的文件信息上传与下载功能实际是与驱动程序的信息交互过程。当由于环境改变导致监控目录内文件增加、删除或者某一文件被修改时，此时将监控目录下的所有文件信息与驱动程序的存储单元中的文件信息依次比较，比较的项目是 EFI_FILE_INFO 结构体内各个变量值，这样可比较出监控目录下是否有文件改变。因此，本设计最核心的问题在于监控目录下所有文件信息的上传与下载。

（1）监控目录信息上传：

监控目录信息上传功能是把监控目录下所有文件信息以及文件的路径信息保存至驱动程序存储单元中。为实现该功能，需要在 Shell 应用程序下打开 EFI_VIRTUAL_TPM_PROTOCOL，并使用接口实例 Write 将信息上传到驱动程序存储单元中。

（2）监控目录信息下载：

监控目录信息下载功能与上传功能是逆向过程，数据流向是从驱动程序存储单元到 Shell。同样，需要在 Shell 应用程序下打开 EFI_VIRTUAL_TPM_PROTOCOL，并使用接口实例 Read 将信息下载到 UEFI Shell 应用程序。

### 4.3.3 命令参数控制与运行结果

为应用程序配置不同的 Shell 参数达到不同控制目的，EDK II Shell 通过 EFI_SHELL_INTERFACE 类型变量向应用传递命令行参数。通过 EDK II Shell 中的一个全局实例 SI，采用 SI->argv 和 SI->argc 来获取命令行参数。本课题将参数信息存入固件卷中，参数信息不会随着系统关闭而丢失，下次应用时可以继续采用固件卷中的配置参数。关于对固件卷中的参数读写要应用 UEFI 运行时服务下的 GetVariable()和 SetVariable()函数。关于 UEFI 的运行时服务的知识可参考 3.3.2 节。

本设计采用三个参数进行信息配置：-s，-g，-e，分别实现监控器开启、生成监控报告和监控器关闭，文件监控执行命令和监控结果分别如图 4.7 和图 4.8 所示：

```
fsnt1:\> Monitor.efi -s
Starting Monitor ...
 - [ok]
```

```
fsnt1:\> Monitor.efi -g
Generating Monitor Report ...
 - [ok]
```

```
fsnt1:\> Monitor.efi -e
Stopping Monitor ...
 - [ok]
```

图 4.7 文件监控执行命令

```
Monitor Report OK

Monitor derectory: fsnt1:\

Active      CreateTime          ModifyTime          FileSize    FileName

[Add]       06/09/09  06:47p    06/09/09  06:47p          0     fsnt1:\a

[Add]       06/09/09  06:47p    06/09/09  06:47p  4,021,184     fsnt1:\a\AppForUTpm.efi

[Add]       06/09/09  06:47p    06/09/09  06:47p          0     fsnt1:\MonitorReport.txt

[Del]       06/09/09  05:32p    06/09/09  06:47p         43     fsnt1:\aaa.txt

[Del]       06/09/09  05:32p    06/09/09  06:47p      2,248     fsnt1:\mmm.txt

[Mod][Old]  06/09/09  05:32p                              0     fsnt1:\bbb.txt
[Mod][New]                      06/09/09  06:47p        332
```

图 4.8 文件监控结果截图

## 4.4 本章小结

本章在自定义协议基础上进行两个应用示例开发，文件加密系统和文件监控系统。两种设计的核心问题在于与驱动程序的数据交互，实现方案是在 Shell 应用程序中打开协议激活协议下各个接口实例。其中文件加密系统重点在于大数存储的数据结构设计，该方案用于加密文件内容不大的情况。文件监控系统

部分重点在于监控目录下文件遍历算法设计，本文的方案虽然采用递归策略，但实验表明，遍历文件的时间复杂度是线性的。

# 结　论

本文基于目前最新的UEFI EDKII体系架构，采用纯软件方式虚拟了计算机主板上一枚重要的安全芯片—TPM，在此基础上，做一些系统安全上的应用设计，并给出了具体的程序实现。

主要工作具体有以下几个方面：

（1）深入研究了UEFI EDKII体系架构和运行机制，掌握了UEFI开发环境EDK2的配置调试方法以及编写符合标准UEFI EDKII驱动模型的驱动程序和UEFI Shell应用程序方法。

（2）根据TPM工业标准，裁剪并修改了TPM核心架构。修改后的虚拟TPM架构包括七大模块：I/O单元、随机数生成器、RSA加密算法引擎、密钥生成器、存储单元、密钥管理器以及执行单元，这些已经足够满足系统安全领域的一般应用。

（3）详细分析了UEFI下驱动模型的两个重要协议：驱动绑定协议和设备路径协议，这两个协议用于本文的虚拟驱动程序框架搭建工作。同时，根据虚拟TPM核心功能，设计出用于服务TPM各项功能的协议：虚拟TPM协议，并实现了该协议下的各个接口，从而为保证驱动程序与Shell应用程序通信打下基础。

（4）研究并提出驱动程序与Shell应用程序通信的一种解决方案，通过该方案，在UEFI Shell下顺利设计并实现两种系统安全领域的应用：加密解密和文件监控系统。这两种应用的实现，更有力地支持本文的设计目标，完善了整个虚拟TPM的设计结构，并且能够在实际中得到广泛应用。

本文的内容紧紧围绕UEFI下虚拟TPM驱动程序以及配套的应用程序设计展开论述，还有以下几点可以继续展开：

（1）对于TPM核心可以继续扩展协议接口，本文只提供了原型设计，它能够在安全领域中做少量的应用。但随着计算机安全技术不断发展，系统安全强度不断增加，应用范围不断扩展，TPM接口功能的设计要求也更为完善。按照本文的方案，开发者可以灵活设计出各种扩展协议接口，做到更广泛应用。

（2）本文的设计开发工作基于 UEFI Windows 下的模拟器，没有在 Intel 高端架构 X64 和 IPF 体系架构下运行，因此结果同 Windows 模拟器相比必然会有所不同。下一步计划将本设计应用于这三种架构平台，得出进一步结论。

（3）从效率和安全性角度分析比较 UEFI 和 OS 下的加密算法，确认 UEFI 下开发类似产品的优势和不足之处。

随着 UEFI 技术的不断成熟，越来越多的 BIOS 生产商、PC 生产商开始使用 UEFI 标准。可以说 UEFI 可扩展固件接口取代传统 BIOS 成为历史必然。因此，UEFI 的各项应用也得到了 Intel 以及 UEFI 社区积极支持与鼓励。本文对虚拟 TPM 在 UEFI 系统上作了一次尝试，并成功证明了该方案的可行性，期待本文的设计原型能够发展成产品原型。

# 参考文献

[1] Trusted Computing Group. TCG Specification Architecture Overview, August, 2007: 6-17P

[2] Trusted Computing Group. TPM Main Part 2 TPM Structures Specification version 1.2 Level 2 Revision 103, October, 2006: 20-50P

[3] 刘冬，文伟平. EFI及其安全性分析. 软件学报，2009，5：3-8页

[4] 肖政，韩英. 基于可信计算平台的体系结构研究与应用. 计算机应用，2006，26（8）：7-9页

[5] 王斌，谢小权. 可信计算机BIOS系统安全模块的设计. 计算机安全，2006，9：1-3页

[6] 李新荣. BIOS与CMOS. 四川教育学院学报，2006，11：1-2页

[7] 倪光南. UEFI BIOS是软件业的蓝海. UEFI技术大会，2007，6. http://soft.chinabyte.com/90/3382590_1.shtml

[8] Intel corporation. EDK II Module Development Environment Library Test Infrastructure, Oct 12, 2006

[9] 潘登，刘光明. EFI结构分析及Driver开发. 计算机工程与科学，2006，4：1-10P

[10] Framework Open Source Community. EFI_Shell_Developer_Guide_Ver0- 91, June 27, 2005. https://efi-shell.tianocore.org/servlets/ProjectDocumentList

[11] Intel corporation. UEFI 2.1 Porting Guide, December 21, 2007: 31-35P

[12] Intel corporation. MDE Library Spec, October 9, 2006: 2-33P

[13] Framework Open Source Community. Pre_EFI Initialization Core Interface, March 2008. http://www.uefi.org/specs/

[14] 王昱. 基于可扩展固件接口（EFI）的硬盘数据保护方案设计与实现. 浙江大学硕士论文，2006

[15] 潘登，刘光明. EFI结构分析及Driver开发. 计算机工程与科学，2006，2：2-5页

[16] Dr.Gaurav Banga. EFI/UEFI将带领PC产业进入下一世代，2008，

# 攻读硕士学位期间发表的论文和取得的科研成果

5. http://www.22cc.net/news/20080506/17634_2.html

[17] UEFI Forum. UEFI Primer FINAL. http://www.uefi.org/home

[18] 郭传鹏，王宇. 可信计算中的密钥管理技术研究. 计算机应用研究，2008，5：1-13页

[19] 方炜炜，杨炳儒. 基于EFI的可信计算平台研究. 计算机应用研究，2009，8:2-4页

[20] 张海明. EFI下可信链建立关键技术的研究. 北京:北京交通大学，2007

[21] Intel corporation. EDK II Extended INF File Specification Revision 1.1. August 2008：2-4P

[22] UEFI Forum. UEFI Specification Version2.1. January 23, 2007. http://www.u-efi.org/specs/

[23] Intel corporation. EDK II C Coding Standard Revision 0.51，July 13，2006：14-16P

[24] Framework Open Source Community. Shared Architectural Elements，March2008.http://www.uefi.org/specs/

[25] 顾桃峰. 终端安全与文件保护系统研究. 南京信息工程大学硕士论文，2008：42-44页

[26] Framework Open Source Community. Edk Getting Started Guide[1] 0.41，January 14, 2005.https://efi-shell.tianocore.org/servlets/ProjectDocumentList：5-7P

[27] 李骥. 基于安全的文件监控系统的设计与实现. 吉林大学硕士论文，2004：31-37页

# 致　谢

首先我由衷的感谢我的导师顾国昌教授。感谢顾老师在我读研的两年半时间里，对我无微不至的关怀和孜孜不倦的教诲。导师以其渊博的学识，严谨求实的治学态度勤奋创新的钻研精神所树立的学者风范使我受益匪浅，并将成为我终生效仿的楷模。在此谨向顾老师致以最崇高的敬意和深深的谢意。

其次要感谢实习所在的 Intel 公司领导和同事们在该课题的研究方面提供的巨大的支持，特别感谢王岩、刘江、田峰和 Gu,Alex，感谢他们在专业技术上的培训和帮助。同时也要感谢哈尔滨工程大学计算机科学与技术学院的所有老师，感谢他们六年来对我的学习和生活上的帮助。

感谢 336 和 325 实验室全体成员，特别感谢吴艳霞师姐、王超伦、孙延腾、桂坤、吴楠等同学在学业上的帮助，在与他们共同渡过的几年中，我深深体会到了友谊的珍贵，他们的关心与鼓励支撑着我不断向上，与他们在一起的美好时光我将终身难忘。也要感谢陈阳等朋友在我生活上无私的支持和帮助。最后，尤其要感谢我的父母，我人生中的每一点进步，都有他们的无私的奉献与默默的支持，感谢他们这些年来对我的辛苦付出，让我最终顺利的完成硕士阶段的学习。