

中文图书分类号: TP391  
密 级: 公开  
UDC : 004  
学 校 代 码: 10005



# 硕 士 专 业 学 位 论 文

PROFESSIONAL MASTER DISSERTATION

论 文 题 目: 基于 UEFI 的硬盘文件安全加载系  
统设计与实现

论 文 作 者: 唐治中

专业类别/领域: 计算机技术

指 导 教 师: 张建标

论文提交日期: 2021 年 6 月



UDC: 004  
中文图书分类号: TP391

学校代码: 10005  
学 号: S201861799  
密 级: 公开

# 北京工业大学硕士专业学位论文

## (非全日制)

题 目: 基于 UEFI 的硬盘文件安全加载系统设计于实现

英文题目: DESIGN AND IMPLEMENTATION OF HARD  
DISK FILE SECURITY LOADING SYSTEM  
BASED ON UEFI

论 文 作 者: 唐治中  
学 科 类 别: 计算机技术  
研 究 方 向: 信息安全  
申 请 学 位: 工程硕士专业学位  
指 导 教 师: 张建标教授  
所 在 单 位: 信息学部  
答 辩 日 期: 2021 年 5 月  
授 予 学 位 单 位: 北京工业大学



## 独 创 性 声 明

本人声明所呈交的论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得北京工业大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

签 名：\_\_\_\_\_

日 期： 2021 年    月    日

## 关于论文使用授权的说明

本人完全了解北京工业大学有关保留、使用学位论文的规定，即：学校有权保留送交论文的复印件，允许论文被查阅和借阅；学校可以公布论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存论文。

（保密的论文在解密后应遵守此规定）

签 名：\_\_\_\_\_

日 期： 2021 年    月    日

导师签名：\_\_\_\_\_

日 期： 2021 年



## 摘 要

在一个计算机系统中，底层软件系统占有着重要的位置，他不仅为上层系统和应用提供基础接口，也从底层为整个系统的安全和可信提供了保障。《信息安全技术网络安全等级保护基本要求》国家标准中就指出，实际运行中的关键系统需要进行各阶段的可信验证工作。UEFI 规范作为近些年来替代传统 BIOS 的一种底层系统规范，从各个机构的服务器到每个人的个人计算机，都得到了广泛的应用。UEFI BIOS 在可扩展性和开发效率、开发难易程度上都得到极大的增强，但正是这种 C 语言来替代传统汇编语言的方案，使得 UEFI BIOS 同样会遭受 C 语言代码的攻击，不同处理器架构的统一规范开发方式也为攻击底层固件系统提供了更多的条件。从硬盘这样的块设备经过 UEFI BIOS 启动操作系统仍然是如今主流的系统启动方式，所涉及到的不光是硬盘设备的被攻击的可能性，也涉及到固件系统层面的攻击，如对于 UEFI 文件系统这样的操作硬盘的内部逻辑，就更易受到针对性的攻击对硬盘文件的安全性带来威胁。因此研究基于 UEFI 的硬盘文件的安全加载就具有重要意义。本文主要工作如下：

(1) 针对 UEFI BIOS 系统加载硬盘文件过程中存在的安全威胁，结合可信计算理论提出基于 UEFI 的硬盘设备文件可信加载系统的总体框架，并使用的底层可信平台 BMC 及基板管理控制器，实现 UEFI BIOS 中 BMC 驱动程序。

(2) 针对 UEFI BIOS 系统加载 UEFI BIOS 存储闪存设备中的驱动文件过程中存在的安全问题，设计提出 UEFI BIOS 启动阶段所需度量的 UEFI 文件系统协议栈驱动程序，并实现度量模块功能。

(3) 针对安全方案中的可信度量功能，设计并实现了通过 BMC 的日志存储功能，并完成确保驱动按顺序加载的依赖表达式编写。

(4) 根据本系统安全方案的设计，在申威平台中对驱动度量模块、日志生成功能和驱动加载顺序修改功能进行实现和测试，以保证功能开发过程的有效性和安全方案的可实施性。

**关键词：**UEFI；文件系统协议栈；可信计算；信任链





## Abstract

In a computer system, the underlying software system occupies an important position. It not only provides a basic interface for the upper system and applications, but also provides a guarantee for the safety and credibility of the entire system from the bottom. According to the national standard of "Information Security Technology Network Security Graded Protection Basic Requirements", critical systems in actual operation need to be trusted at various stages. As a low-level system specification that replaces traditional BIOS in recent years, UEFI specification has been widely used from servers of various organizations to personal computers of everyone. UEFI BIOS has been greatly enhanced in terms of scalability, development efficiency, and development difficulty. However, it is this C language that replaces the traditional assembly language program, which makes UEFI BIOS also subject to C language code attacks. The unified specification development method of the processor architecture also provides more conditions for attacking the underlying firmware system. Starting the operating system from a block device such as a hard disk through UEFI BIOS is still the mainstream system startup method today. It involves not only the possibility of the hard disk device being attacked, but also the firmware system level attack, such as the UEFI file system. Such internal logic of operating the hard disk makes it easier to receive targeted attacks that threaten the security of hard disk files. Therefore, it is of great significance to study the secure loading of UEFI-based hard disk files. The main work of this paper is as follows:

(1) Aiming at the security threats in the process of loading hard disk files in the UEFI BIOS system, combined with trusted computing theory, a UEFI-based overall framework for trusted loading of hard disk device files is proposed, and the underlying trusted platform BMC and baseboard management controller are used to implement UEFI BIOS In the BMC driver.

(2) Aiming at the security problems in the process of loading the driver files in the UEFI BIOS storage flash memory device in the UEFI BIOS system, the UEFI file system protocol stack driver program for the measurement required by the UEFI BIOS startup phase is designed and proposed, and the measurement module function is realized.

(3) Aiming at the credibility measurement function in the security scheme, the log storage function through BMC is designed and realized, and the dependency expression

writing to ensure that the driver is loaded in order.

(4) According to the design of the safety scheme of this system, the drive measurement module, log generation function and drive loading sequence modification function are implemented and tested in the Sunway platform to ensure the effectiveness of the function development process and the implementability of the safety scheme.

**Key words:** UEFI; File system protocol stack; trusted computing; Chain of trust

## 目 录

摘 要 .....	I
Abstract .....	III
第 1 章 绪论 .....	1
1.1 背景及研究意义.....	1
1.2 国内外研究现状.....	1
1.3 主要研究内容 .....	2
1.4 本文组织结构 .....	3
第 2 章 相关技术介绍 .....	5
2.1 UEFI 概述.....	5
2.1.1 UEFI 系统结构.....	5
2.1.2 UEFI 协议运作方式 .....	6
2.2 UEFI 固件文件系统数据存储方式介绍 .....	8
2.3 UEFI 文件系统协议栈.....	9
2.3.1 总体介绍.....	9
2.3.2 相关驱动介绍 .....	11
2.4 BMC 技术介绍 .....	11
2.4.1 BMC 与 BIOS 通讯方式 .....	12
2.5 本章小结 .....	12
第 3 章 基于 UEFI 的硬盘文件安全加载系统总体设计 .....	13
3.1 安全漏洞分析 .....	13
3.1.1 硬盘文件系统建立过程分析 .....	13
3.1.2 UEFI 环境文件加载过程分析 .....	15
3.1.3 从硬盘攻击关键文件.....	16
3.1.4 从固件芯片攻击关键驱动 .....	17
3.2 信任链的设计 .....	18
3.3 安全方案启动阶段设计 .....	18
3.3.1 SEC 阶段 .....	19
3.3.2 PEI 阶段 .....	19
3.4 总体架构设计 .....	21
3.5 本章小结 .....	23

第 4 章 基于 UEFI 的硬盘文件安全加载系统的详细设计 .....	25
4.1 可信度量驱动的实现 .....	25
4.1.1 可信度量值计算模块 .....	25
4.1.2 固件文件系统访问模块 .....	26
4.1.3 硬盘文件系统访问模块 .....	30
4.1.4 驱动文件度量模块 .....	31
4.1.5 BMC 通信模块 .....	33
4.2 UEFI 启动阶段详细设计 .....	36
4.2.1 SEC 阶段设计 .....	36
4.2.2 PEI 阶段设计 .....	37
4.2.3 DXE 阶段设计 .....	38
4.2.4 BDS 阶段设计 .....	39
4.2.5 DXE 阶段驱动依赖关系 .....	40
4.3 本章小结 .....	42
第 5 章 硬盘文件安全加载方案实现及测试 .....	43
5.1 实验环境 .....	43
5.1.1 申威 6A 服务器真机环境 .....	43
5.2 驱动度量功能实现与测试 .....	45
5.2.1 功能实现 .....	45
5.2.2 测试目的 .....	47
5.2.3 测试步骤 .....	47
5.2.4 测试过程 .....	47
5.3 BMC 驱动连通性实现与测试 .....	48
5.3.1 功能实现 .....	48
5.3.2 测试目的 .....	50
5.3.3 测试步骤 .....	50
5.3.4 测试过程 .....	50
5.4 依赖表达式验证 .....	51
5.4.1 测试目的 .....	51
5.4.2 测试步骤 .....	51
5.4.3 测试过程 .....	51
5.5 本章小结 .....	52
结 论 .....	53
参考文献 .....	55

攻读硕士学位期间所取得的学术成果 .....	59
致 谢 .....	61



## 第 1 章 绪论

### 1.1 背景及研究意义

目前，在计算机应用领域，统一可扩展固件接口 UEFI（Unified Extensible Firmware Interface）已经成为了主流基础输入输出系统 BIOS（Basic Input/Output System）实现方式并逐渐取代了传统 BIOS，作为一个越来越成熟的 BIOS 系统，UEFI 环境就包括了对硬盘文件加载和运行的功能，其中包括：在调试关键驱动程序时，可能需要把驱动文件放入硬盘并在 UEFI SHELL 环境中手动加载<sup>[1-2]</sup>；操作系统启动需要经过 UEFI 环境，并在其中加载硬盘中的操作系统引导程序完成启动，也包括 UEFI 远程安装并启动操作系统的功能<sup>[3]</sup>。在 UEFI 初始化完成后，可完整使用 UEFI 提供的系统功能，其中就包括了硬盘中的可扩展固件接口文件系统分区 ESP（Efi system partition）分区数据操作。这块 UEFI 可访问到的硬盘空间上会存储一些 UEFI 编译过程中重要的可扩展固件接口可执行文件文件，其中就包括了操作系统启动引导文件 start\_kernel.efi 这样的关键文件<sup>[4]</sup>，因此如何在 UEFI 环境中安全的加载硬盘设备文件就成了关键问题。

因为在 UEFI 环境下访问硬盘文件，所以这些文件存在通过硬盘被篡改和通过 UEFI 文件系统被篡改的可能性<sup>[5]</sup>。现有技术中有针对硬盘文件的硬件攻击方法<sup>[6]</sup>，通过硬件手段修改硬盘中关键文件的内容起到注入木马程序的效果。还有一种现有技术中针对存放 BIOS 程序的闪存 flash 芯片提出了一种攻击手段<sup>[7]</sup>，其攻击原理为通过硬件手段和 flash 中数据存储格式，修改 flash 芯片中的驱动程序内容起到注入木马程序的效果<sup>[8-9]</sup>。因为 UEFI 中访问硬盘文件需要经过 UEFI 文件系统协议栈，因此通过现有技术手段可修改协议栈相关驱动<sup>[10]</sup>，从 UEFI 固件层面达到攻击效果。因此对于 BIOS 固件层面的块设备文件系统的可信验证是极其具有必要性的。

### 1.2 国内外研究现状

房强等在“基于固件文件系统的 UEFI 安全机制研究”一文中通过对 UEFI 安全威胁研究的分析与总结<sup>[11]</sup>，提出了基于可信平台模块 TPM（下同）的静态度量固件文件系统中驱动程序的方法。该方案以 TPM 为可信锚点，此信任根可根据如基板管理控制器 BMC（下同）这样的底层硬件进行替代和改进；其次对于固件文件系统 FFS（下同）中的驱动文件度量为静态度量过程，及在 UEFI 初始化阶段结束后再进行度量，这样就无法保证初始化过程中的安全性。段晨辉等在文献[12]一文中通过对 UEFI 启动阶段中信任链的设计和底层基于可信计算组

织 TCG（下同）的可信链构建方案，提出了在 UEFI 完整启动阶段过程中通过逐个阶段度量后面阶段的内容起到可信启动的效果。但该方案在整个度量过程中需要涉及到驱动加载准备阶段 PEI 阶段（下同）核心代码、所有 PEI 模块程序、驱动加载阶段 DXE（下同）核心代码、DXE 调度器代码、所有 DXE 驱动程序代码的度量工作，缺少对于特定如硬盘设备文件的文件系统的针对性安全验证，并且整个度量过程繁琐耗时，虽然做到了动态度量效果，但具有相当大的局限性，可行性不高。

文献[13]提出了一种通过 UEFI 固件层面的文件安全存储策略，来保证硬盘文件受到攻击和恶意篡改时可通过 BIOS 固件中的文件信息进行复原。但是该专利忽略了对于固件硬件平台的攻击可能性，无法保证固件中备份文件的安全性，即文件的硬件安全防护能力并不突出。文献[14]提出了一种通过检测硬盘设备是否安全可信的机制完成硬盘文件的安全加载。但是该方案忽略了通过篡改 BIOS 固件中文件系统相关驱动达到修改硬盘文件的手段，一旦固件中的文件系统被恶意篡改，在 UEFI 环境中加载硬盘文件不存在可信而言。

在 UEFI 安全领域<sup>[15]</sup>，大部分基于 UEFI 的文件加载方案，没有对固件层面的 UEFI 文件系统驱动可信度量的过程，无法保证固件层面针对硬盘文件内容的攻击<sup>[16-17]</sup>。一些完整性度量方案中着重度量系统的全部信息，缺少一些针对性的驱动内容的度量工作<sup>[18]</sup>。

现有研究中也通过 USB Key 的方式存储操作系统引导文件<sup>[19]</sup>，避免了计算机硬盘被攻击的可能，但 USB 设备仍然属于块设备硬件，也存在被攻击的风险。UEFI 系统的可信启动也成了近年来重点研究的对象<sup>[20-21]</sup>，从 UEFI 启动的各个阶段逐一度量后一阶段然后进行控制权的传递。随着国产化的流行<sup>[22]</sup>，同时也为了适应各种国产硬件如 CPU 等设备的适配，UEFI 的国产化研究适配工作也急需进行。

### 1.3 主要研究内容

鉴于 UEFI 是 BIOS 系统的一种统一可扩展标准和方案，拥有着模块化的系统部件添加结构，使他成为 BIOS 功能开发的首选方案，目前市场上的 BIOS 实现也越来越多的统一使用 UEFI。与此同时，UEFI BIOS 作为计算机启动到加载操作系统的一个中间阶段，需要在系统启动时加载硬盘中 ESP 分区中的操作系统引导程序；并且随着 UEFI BIOS 的功能的不断发展，UEFI BIOS 环境中甚至能够播放视频音频，而硬盘设备又是计算机中数据的主要存储介质，这就增加了 BIOS 与硬盘设备之间交互频率<sup>[23]</sup>，BIOS 环境中加载硬盘设备文件的安全性就显得更加重要。因此研究基于 UEFI 的硬盘设备文件加载的安全机制就具有重要



意义。

本文主要研究内容如下：

(1) 针对 UEFI BIOS 系统加载硬盘文件过程中存在的安全威胁，结合可信计算理论提出基于 UEFI 的硬盘设备文件可信加载的总体框架，并使用的底层可信平台 BMC 及基板管理控制器，实现 UEFI BIOS 中 BMC 驱动程序。

(2) 针对 UEFI BIOS 系统加载 UEFI BIOS 存储闪存设备中的驱动文件过程中存在的安全问题，设计提出 UEFI BIOS 启动阶段所需度量的 UEFI 文件系统协议栈驱动程序，并实现度量模块功能。

(3) 针对安全方案中的可信度量功能，设计并实现了通过 BMC 的日志存储功能，并完成确保驱动按顺序加载的依赖表达式编写。

(4) 根据本系统安全方案的设计，在申威平台中对驱动度量模块、日志生成功能和驱动加载顺序修改功能进行实现和测试，以保证功能开发过程的有效性和安全方案的可实施性。

## 1.4 本文组织结构

全文结构一共分为六大部分，各部分内容如下：

- 第一章 绪论

首先对本文的研究背景和研究意义进行了介绍，然后阐述了现有的 UEFI 环境下硬盘文件保护方法的现状及本系统在这个基础上做的改进。介绍了本文的主要研究内容，最后介绍了本文的组织结构。

- 第二章 相关知识及技术介绍

首先对 UEFI BIOS 及可扩展固件接口标准的基础输入输出系统的整体设计和层次结构进行了介绍，包括启动流程、协议加载方式、数据库句柄及固件文件系统数据存储格式。其次对 UEFI 文件系统协议栈及涉及到的具体驱动程序做了进一步说明。然后介绍了 BMC 及基板控制管理器的基本功能及与 BIOS 通信方式。最后介绍了可信计算技术的现状及发展。

- 第三章 基于 UEFI 的硬盘文件安全加载系统总体设计

首先对 UEFI 环境中加载硬盘文件的安全漏洞进行了分析，其次介绍了此系统中信任链的涉及与传递，然后介绍了系统的总体架构设计，包括了 DXE 和运行时两个主要的度量阶段。最后介绍了此系统的功能模块划分。

- 第四章 基于 UEFI 的硬盘文件安全加载系统详细设计

分别对度量计算模块、硬盘文件度量模块、驱动文件度量模块、固件和硬盘访问模块、BMC 通信模块给出了具体设计和关键代码实例，并对 SEC、PEI 和 DXE 阶段做了补充说明。

- 第五章 硬盘文件可信加载方案实现及测试

根据此系统的安全方案设计，介绍了如 BMC 驱动、文件系统相关驱动的实现、实验及测试过程，以及实验相关的编译运行环境。通过实验说明了安全方案的可实施性和此系统的预防效果。

- 结论

对本文的相关研究工作进行了总结和分析，并根据功能测试结果指出本方案的不足之处。并提出了下一步的研究方向。

## 第2章 相关技术介绍

### 2.1 UEFI 概述

本章将会介绍 UEFI 系统中关系到本文关键技术的基础内容，其中具体包括了 UEFI 基本结构、UEFI 固件存储格式、UEFI 文件系统协议栈相关内容及驱动程序介绍，还包括基板管理控制器 BMC 及可信计算的基础内容。

#### 2.1.1 UEFI 系统结构

UEFI（Unified Extensible Firmware Interface，统一可扩展固件接口）定义了操作系统和平台固件直接的接口，UEFI 提供了一个统一可扩展的固件平台，并针对平台特性定义了一系列接口。该平台整体处于硬件与操作系统中间，平台最上层的可扩展固件接口包含了平台提供的 API 函数、启动时服务（EFI Boot Services）、运行时服务（EFI Runtime Services）和操作系统引导程序<sup>[24]</sup>，下层则是根据 UEFI 规范实现的平台固件。UEFI 的平台架构如图 2-1 所示：

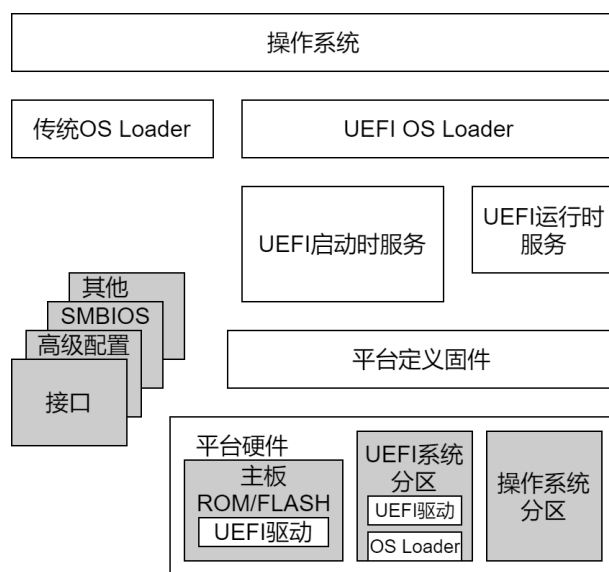


图 2-1 UEFI 系统框架图

Figure 2-1 Infrastructure of UEFI

图 2-1 描绘了 UEFI 框架中各模块的关系，UEFI 固件作为承上启下的模块，对底层硬件进行了抽象处理，又不断地对上层的操作系统提供服务，并在不同的服务层的连接中采用了标准的接口。图中 UEFI 保留了对传统 BIOS 引导操作系统的兼容性，所以在可扩展固件接口的实现中有两种操作系统引导方式，分别为 Legacy OS Loader 和 UEFI OS Loader。

UEFI 允许操作系统预处理，实现了操作系统的引导和一些系统软件执行所需要的其它应用程序，如诊断程序、UEFI Shell、系统调试软件等，这些程序统称为 UEFI 实体（UEFI Image）。根据 UEFI 规范，UEFI Image 包含三种：UEFI 应用程序、UEFI 驱动和 OS Loaders，这些实体都是在 UEFI API 调用的基础上实现的。

从图中可以看出，UEFI 的启动时服务 **Boot Service** 中包含了 UEFI 在 **PEI** 和 **DXE** 两个主要加载驱动完成系统初始化过程中所需要的驱动程序、内存管理、设备管理、协议贮存等信息。而 UEFI 的启动时服务也有他自己的生命周期，从下图就可以看出启动时服务在 UEFI 启动过程中的位置。

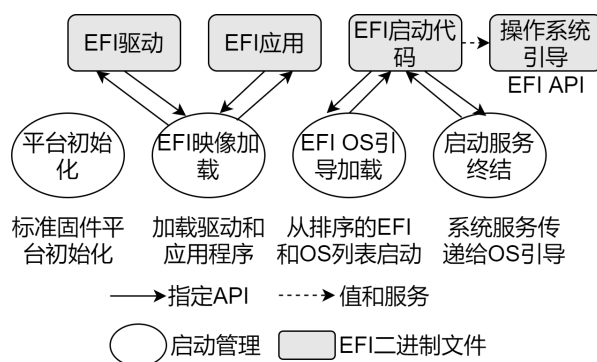


图 2-2 统一可扩展固件接口启动流程图

### Figure 2-2 Booting Sequence of UEFI

从图 2-2 中可以看出，Platform Init 平台初始化过程中建立 Boot Service 系统服务和 RunTime Service 系统服务，EFI Image Load 阶段包括了 PEI 和 DXE 两个主要驱动加载过程，负责加载 BIOS 固件中的 UEFI 驱动和 UEFI 应用程序的 EFI 可执行文件，之后在启动设备选择也就是 BDS 阶段中，通过用户的选择，UEFI BIOS 选择适当的操作系统引导程序进行加载，并同时退出 Boot Service 服务，并继续向上层操作系统提供 RunTime Service 服务。因而在操作系统运行过程中，可以继续使用底层 UEFI BIOS 提供的运行时服务。结合图 2-1 也可以发现，UEFI 的一些关键驱动程序，和 OS Loader 也会存放在如硬盘块设备的 ESP（EFI System Partition）系统分区中。由于本文主要对于 UEFI 启动过程中的 DXE、BDS 阶段进行安全方案设计，因此，启动时服务其中包含的如协议加载函数等为本文的主要研究对象。

### 2.1.2 UEFI 协议运作方式

UEFI 中协议设计的思想为，由于 UEFI 的官方提供实现的版本为 C 语言实现，而 C 语言是一种面向过程的语言，而完全使用面向过程的思想来管理和使用众多 UEFI 协议将会使程序变得非常复杂。Protocol 作为一种对象来设计管理

会更加直观<sup>[25-26]</sup>。因而 UEFI 中的 Protocol 引入了面向对象的思想，其中包括：

- 用 struct 来模拟 class。
- 用函数指针（Protocol 的成员变量）模拟成员函数，此种函数的第一参数必须是指向 Protocol 的指针，用来模拟 this 指针。

从图 2-1 中可以看出，UEFI 中的协议包含于 UEFI 启动时服务中（Boot Services），由启动时服务提供的功能进行协议的加载、保存和调用等操作。其中 UEFI 启动时服务提供的协议相关的功能函数如表 2-1 所示。

表 2-1 启动时服务协议功能表  
Table 2-1 Boot Service Protocol Interface Functions

名称	类型	描述
InstallProtocolInterface	Boot	在设备句柄上安装一个协议接口
UninstallProtocolInterface	Boot	从设备句柄上移除一个协议接口
ReinstallProtocolInterface	Boot	在设备句柄上重新安装协议接口
RegisterProtocolNotify	Boot	注册一个事件，只要接口有信号为指定的协议安装
LocateHandle	Boot	返回支持指定协议的句柄数组
HandleProtocol	Boot	查询句柄以确定它是否支持指定的协议
LocateDevicePath	Boot	找到支持指定路径的设备路径上的所有设备协议并将句柄返回到最接近的设备路径
OpenProtocol	Boot	将元素添加到使用协议的代理列表中接口
CloseProtocol	Boot	从代理列表中移除一个元素，也就是消耗一个协议接口
OpenProtocolInformation	Boot	检索当前正在使用的代理列表协议接口
ConnectController	Boot	使用一组优先规则来找到最佳的驱动程序集管理一个控制器
DisconnectController	Boot	通知一组驱动程序以停止管理控制器
ProtocolsPerHandle	Boot	检索安装在句柄上的协议列表，函数返回的缓冲区是自动分配的
LocateHandleBuffer	Boot	从句柄数据库中检索句柄列表，该列表符合搜索条件，返回缓冲区自动已分配
LocateProtocol	Boot	在句柄数据库中找到第一个支持所需协议的句柄
InstallMultipleProtocolInterfaces	Boot	将一个或多个协议接口安装到指定句柄上
UninstallMultipleProtocolInterfaces	Boot	从指定句柄中卸载一个或多个协议接口

表 2-1 中列出了 Boot Service 中包含的所有 UEFI 协议相关的功能函数，其中最为常见的如 InstallMultipleProtocolInterfaces 这样的加载协议函数，在众多 UEFI 应用以及底层驱动程序中都十分常见，因为他可以同时提供出需要加载的多个协议 GUID。

同时，由于启动时服务提供了追踪最新安装了的新协议内容以及他们的使用情况，因此对于 UEFI 的启动时服务来说，它可以安全地卸载并重新安装由 UEFI 驱动程序使用的协议接口。

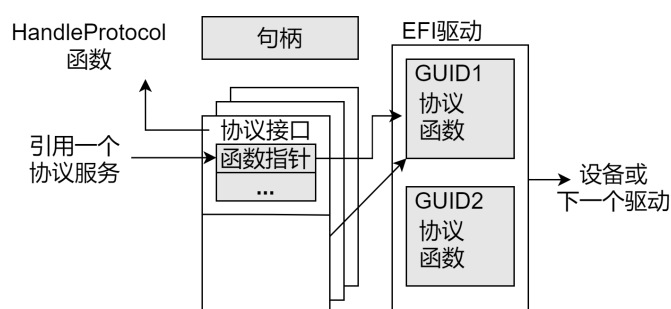


图 2-3 统一可扩展固件接口协议加载方式图

Figure 2-3 Locating Protocol of UEFI

协议的加载过程可通过图 2-3 分析得知。在图 2-3 中可以看出，协议由 **HandleProtocol** 等表 2-1 中列出的装载协议用的功能函数加载到 **Handle** 句柄上，而所有的 **Handle** 则由 UEFI 内核统一存储于句柄数据库，句柄数据库也是一个链表结构用于存储记录所有的 **Handle** 句柄，这些句柄可由任意的 UEFI Image（镜像）访问，从而达到函数调用的效果。而这些协议中包含的是指向具体函数的 C 语言中的函数指针，这些具体函数则是在 **DXE** 阶段由 UEFI 系统表提供的加载驱动镜像函数加载并驻留在内存中的。

## 2.2 UEFI 固件文件系统数据存储方式介绍

UEFI 固件文件系统指的是 BIOS 闪存芯片中的数据存储格式，它通过统一的固件文件系统标准用来统一闪存芯片中的文件内容和 UEFI 启动阶段内存中文件的内容<sup>[27]</sup>。具体的固件中 UEFI 可执行程序文件存储格式如图 2-4 所示。

在图 2-4 中，左边为通过结构图的方式说明固件文件系统的数据存储格式，右边为通过数据结构中的树形结构来阐述 FFS 中的文件存储。右图中，蓝色方框代表了一个完整的 FFS 中的文件映像也就是 UEFI 中可执行文件的二进制数据，黄色方框代表了一个父目录结构，黄色方框中包含的灰色方框代表了父目录中的子目录，也就是文件影响中的最小单位。对应到右图的树结构中，黄色节点就是树中有子节点的父节点，而灰色节点则代表了叶子节点。了解固件文件系统中的文件数据存储格式有助于理解 UEFI 内核在系统初始化过程中调用相关解析 FFS 中文件函数的运行过程，也有助于帮助分析本文安全方案中可信度量的具体数据内容。

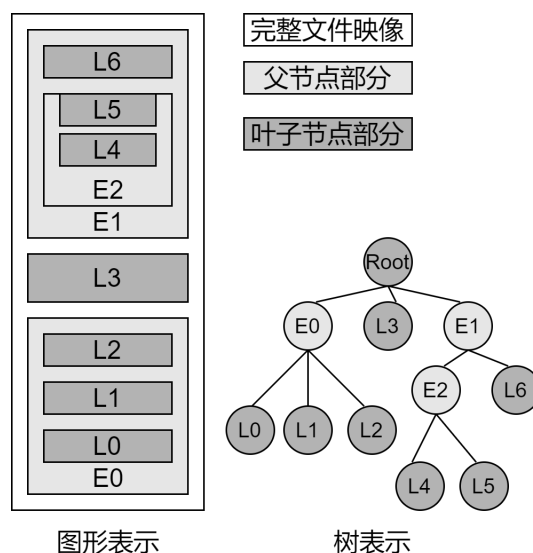


图 2-4 固件文件系统文件存储格式

Figure 2-4 Firmware file system file storage format

## 2.3 UEFI 文件系统协议栈

UEFI BIOS 中的文件系统协议栈指的是针对块设备，也就是对应一般应用于 PC 和服务器的硬盘设备，UEFI 文件系统协议栈通过把对硬盘设备的不同操作功能分层起到区分和满足应用层与内核层不同功能需要的目的，下面本节将做具体的介绍。

### 2.3.1 总体介绍

UEFI BIOS 和传统 BIOS 都支持硬盘的读写功能。UEFI 优于传统 BIOS 的地方在于，UEFI 还提供了对文件系统的支持<sup>[28]</sup>。其中传统 BIOS 由于直接由汇编语言编写，实现内容十分依赖于具体的硬件平台，因此不易在运行流程中进行像对硬盘设备这种次要设备的过多交互，像 ESP (EFI System Partition) 分区中文件的操作也就只通过直接操作硬盘驱动程序来存取具体扇区中的数据内容。然而进入了 UEFI 时代，BIOS 的实现得到了以 C 语言为具体实现工具并进行了国际统一的实现标准，提供统一固件接口，因此从需求性和实现性方面都更加需要 UEFI 系统与硬盘的交互功能，因此才出现了用于管理 UEFI 中硬盘设备文件的 UEFI 文件系统协议栈。

通常，每个 UEFI 系统至少有一个 ESP 分区，在这个分区上存放了操作系统启动文件。既然操作系统加载器以文件的形式存放在 ESP 分区内，UEFI 就需要有读写文件的功能。文件的读写与管理在一定需求量和为了统一便捷的前提下必须通过文件系统来完成，要支持读写文件，UEFI 必须首先能操作 ESP 分区上的文件系统。ESP 主要用来存放操作系统加载器相关的文件，有时在一些实际项目的调试过程中，也需要通过将事先编译好的 EFI 可运行特定硬件的驱动程序

通过操作系统放入 UEFI 能访问到的 ESP 分区内，并从 UEFI SHELL 中手动加载所述驱动程序来使调试工作更加便捷。总体来说，UEFI 对文件系统的要求比较简单，所以采用 FAT 文件系统就可以满足需求，随着 UEFI 技术的不断发展，对硬盘数据访问的要求也越来越高，因此目前已出现 ext2 等文件系统用来支持 UEFI，并已发布到官方开源库中。UEFI 文件系统协议栈图如图 2-5 所示。

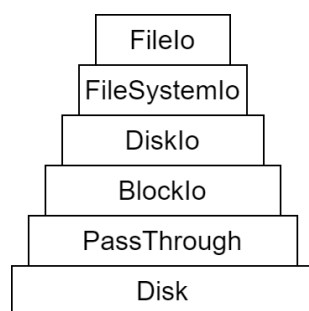


图 2-5 统一可扩展固件接口文件系统协议栈

Figure 2-5 UEFI File system protocol stack

如图 2-5 所示，最下层 Disk 层表示如硬盘设备这样的块设备硬件。PassThrough 层表示硬盘设备与总线接口的协议，EFI\_ATA\_PASS\_THRU\_PROTOCOL 提供有关 ATA 控制器以及将 ATA 命令块发送到与该 ATA 控制器相连的任何 ATA 设备的信息。而 EXT SCSI\_PASS\_THRU\_PROTOCOL 协议提供了将 ATAPI 命令块发送到连接到该 ATA 控制器或 SCSI 控制器的 ATAPI 设备的功能。此协议可直接操作硬盘具体扇区中的数据内容，为 UEFI 内核提供最底层的硬盘接口功能函数。

再往上一层是块输入输出 BlockIo（下同）层，BlockIo 是可扩展固件接口输入输出协议 EFI\_BLOCK\_IO\_PROTOCOL 的缩写，主要提供了按块访问设备的功能，包括块的读写函数和刷新 flush 函数。再往上一层的硬盘输入输出 DiskIo 层，在 BlockIo 的基础上进行再次封装，DiskIo 为 EFI\_DISK\_IO\_PROTOCOL 的缩写。此协议提供了从磁盘任意地址读写任意长度数据的功能，这也符合了硬盘作为块设备可以随机访问数据的特点。

再往上一层为文件系统输入输出 FileSystemIo（下同）层，他对应于简易文件系统协议 EFI\_SIMPLE\_FILE\_SYSTEM\_PROTOCOL 协议，这个协议是对文件系统驱动加载时，驻留在内存中的文件系统功能函数的引用和封装。再往上一层为文件输入输出 FileIo 层，对应于 EFI\_FILE\_PROTOCOL 协议，用于对以文件为单位的数据进行操作。在 UEFI 运行环境中如 Shell 环境中加载.efi 文件时，就经历了从下到上完整的协议栈的过程，最终以文件数据流的形式传递给 Shell 环境，并完成后续的执行或其他加载操作。



### 2.3.2 相关驱动介绍

UEFI 中的驱动大致分为两类：一类是符合 UEFI 驱动模型的驱动，成为“UEFI 驱动”；另一类是不遵循 UEFI 驱动模型的驱动，称为“DXE 驱动”，也称为服务型驱动。服务型驱动加载过程较为简单，在 Image 初始化的时候，即在执行模块入口函数时，将 Protocol 安装到自身 Handle 即可。如 2.1 节中 UEFI 协议运作方式的介绍可知，UEFI 协议作为一个包含了所需功能函数指针的结构体，需要具体的 UEFI 驱动程序为其实现所需的功能。UEFI 文件系统协议栈相关的驱动程序分别为：AtaatapiPassThruDxe、PartitionDxe、DiskIoDxe、FileSysDxe，从命名可以看出，他们四个驱动都属于 DXE 驱动即服务型驱动，用于在 UEFI 的 DXE 和 BDS 阶段为系统提供服务，其中的 PassThruDxe 和 FileSysDxe 根据具体的硬盘接口与具体文件系统不同而定。其中 PassThruDxe 驱动程序负责定义硬盘设备与 ATA 或 SCSI 总线进行数据交互的功能；PartitionDxe 用于解析存储在硬盘 ESP 分区中的 GPT 分区表，用于获取到硬盘中数据的分区信息，磁盘在使用前都要进行分区，也就是将硬盘划分为一个个逻辑的区域，不同的分区上可装载不同的文件系统，以方便用户使用。GPT 分区表 (GUID Partition Table)：全局唯一标识磁盘分区表<sup>[29]</sup>，是一个实体硬盘的分区表的结构布局的标准，是可扩展固件接口 (EFI) 标准，被用于替代 BIOS 系统中的一 64bits 来存储逻辑块地址和大小信息的主开机纪录 (MBR) 分区表，与普遍使用的主引导记录 (MBR) 分区方案相比，GPT 提供了更加灵活的磁盘分区机制。LBA(Logical Block Addressing) 逻辑块寻址模式将 CHS 这种三维寻址方式转变为一维的线性寻址，它把硬盘所有物理扇区的 C/H/S 编号通过一定的规则转变为一线性的编号，系统效率得到大大提高，避免了烦琐的磁头/柱面/扇区的寻址方式。如图中的 LBA0 扇区为保护 MBR，它的作用是阻止不能识别 GPT 分区的磁盘工具试图对其进行分区或格式化等操作。LBA1 扇区开始则为分区表和表项，每个表项对应一个用户区域内的磁盘分区，并标识这个分区的开始和结束位置。关于 GPT 分区表的详细分析及恶意篡改方式将在第三章中介绍。

DiskIoDxe 驱动用于定义硬盘最基础的输入输出数据功能，也就是等同于操作系统中的硬盘驱动程序；而 FileSysDxe 驱动则负责为 UEFI 提供与硬盘硬件相同的文件系统数据组织形式，用以在 UEFI 系统中通过 UNIX 风格的文件系统为其他模块提供硬盘文件的交互功能支持。

## 2.4 BMC 技术介绍

BMC(Baseboard Management Controller) 基板管理控制器是一种用于实时检测服务器硬件信息并具备独立运算能力的计算机系统组件，它通过 IPMI 协

议与服务器中的其他硬件设备进行通信。在基于 IPMI 协议的管理平台中，系统对平台各个硬件设备的信息管理策略，都是通过与 BMC 通信并统计运算而实现的<sup>[30-31]</sup>。BMC 拥有除了独立处理器外的独立固件闪存、系统电源、网卡设备，是一个具有独立性的基于服务器平台的管理子系统。在如今的服务器国产化过程中，BMC 越来越得到各个厂商的广泛使用。

### 2.4.1 BMC 与 BIOS 通讯方式

IPMI 协议作为 BMC 与服务器其他硬件设备通信的桥梁主要有两种通信方式，他们分别是 BT (Block Transfer) 接口和 KCS (Keyboard Controller Style) 接口，其中 BT 接口主要是用于传输数据的块传输，即最小传输单位设置为一块，256 字节。而 KCS 接口主要用于以最小单位为一字节来进行数据传输。而在 BIOS 与 BMC 的通信过程中，主要使用 BMC 来进行一些度量过程中基准值的存储功能，因此使用 KCS 接口来进行通讯设计，以避免不必要的传输资源浪费。

	7	6	5	4	3	2	1	0	I/O address
Status (ro)	S1	S0	OEM 2	OEM 1	C/D#	SMS_ATN	IBF	OBF	base+1
Command (wo)									base+1
Data_Out (ro)									base+0
Data_In (wo)									base+0

图 2-6 KCS 接口寄存器信息

Figure 2-6 KCS Interface Registers

如图 2-7 所示，为 KCS 接口所使用到的寄存器即相关信息，主要涉及到四个寄存器，分别是用来查看 BMC 此时状态的状态寄存器、用于指定 BMC 命令的命令寄存器、和两个分别用来进行数据输入和数据输出缓存的寄存器。有关这些寄存器具体使用方法和流程将在第四章的 BMC 驱动实现中做详细说明。

## 2.5 本章小结

本章首先介绍了 UEFI 规范中的整体系统设计结构，并着重介绍了 UEFI 的协议运作方式和对本文有重要作用的启动时服务提供的具体功能函数。然后对固件文件系统中驱动数据的存放方式进行了描述，为后面的驱动内容读取功能提供了理论基础；分析了 UEFI 文件系统协议栈在系统中的作用和具体到主要的四个驱动程序名称。最后对 UEFI BIOS 中与 BMC 系统的通信方式进行了分析，选用了 IPMI 的 KCS 访问模式，来完成数据的交互。

## 第3章 基于UEFI的硬盘文件安全加载系统总体设计

### 3.1 安全漏洞分析

UEFI 环境中，加载硬盘设备上的 ESP 分区中的系统文件是一个十分丰富的过程，它不仅包含了 UEFI 系统内部的文件加载过程，它还包含着如硬盘分区的创建、硬盘文件系统的建立、操作系统安装过程中向 ESP 分区内添加操作系统引导文件 BootLoader 等文件的过程，以及 UEFI 环境中识别硬盘分区方式、识别硬盘特定分区中的文件系统并通过 UEFI 内部的相同文件系统格式定义，使其完成 UEFI 内核与硬盘设备间的文件数据交互功能。

本节将介绍硬盘文件系统的建立过程，UEFI 环境中硬盘文件加载的过程，以及从硬盘硬件层面和从 UEFI 固件层面两方面对硬盘文件进行攻击以达到篡改 ESP 分区中关键系统文件的可能性。

#### 3.1.1 硬盘文件系统建立过程分析

在一个服务器或个人电脑上安装操作系统时，其实都经历着硬盘文件系统建立的过程<sup>[32]</sup>。如图 3-1 所示，中建立 GPT 硬盘分区格式的目的为让 UEFI 可识别出硬盘分区的信息，传统 MBR 分区格式只能由传统 BIOS 系统识别。为硬盘创建 FAT 文件系统的目的也在于统一硬件上的数据组织形式和 UEFI 内存中的硬盘数据处理格式<sup>[33]</sup>。

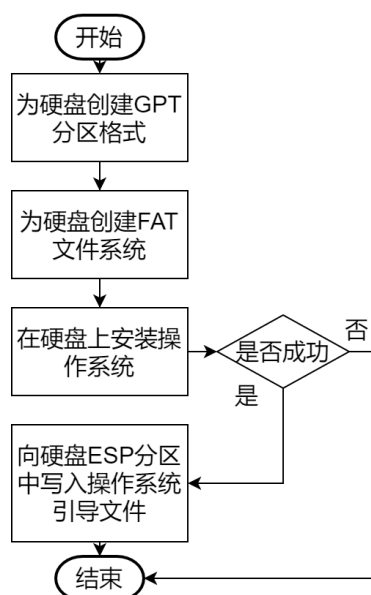


图 3-1 硬盘文件系统建立过程

Figure 3-1 Hard disk file system establishment process

而 ESP 分区之所以作为 EFI 系统分区,是因为在安装操作系统的过程中,操作系统安装程序将 GPT 分区格式硬盘的其中一个分区初始化为 ESP 分区,并将引导加载操作系统其他模块的 BootLoader 引导程序等系统关键 EFI 程序,放入到这个自己建立的 ESP 分区中。这样便有了 UEFI 环境中加载硬盘设备 ESP 分区中系统引导文件及其他关键 EFI 文件的由来<sup>[34]</sup>。

### 3.1.1.1 服务器启动流程分析

UEFI BIOS 作为一个基础输入输出系统,为操作系统提供基础的硬件支持,同时也能引导操作系统的启动,具体实施方式为:

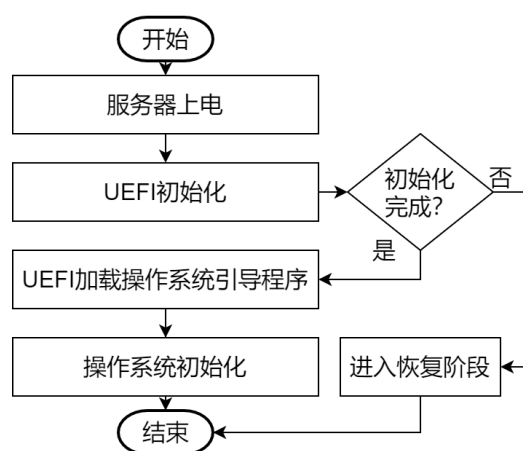


图 3-2 服务器启动流程

Figure 3-2 Server startup process

如图 3-2 所示,服务器上电,随后初始化 UEFI BIOS 系统,再由 UEFI 系统通过内部的文件系统协议栈加载硬盘 ESP 分区中的操作系统引导文件,随后控制权交给引导程序并退出 UEFI 系统初始化流程。

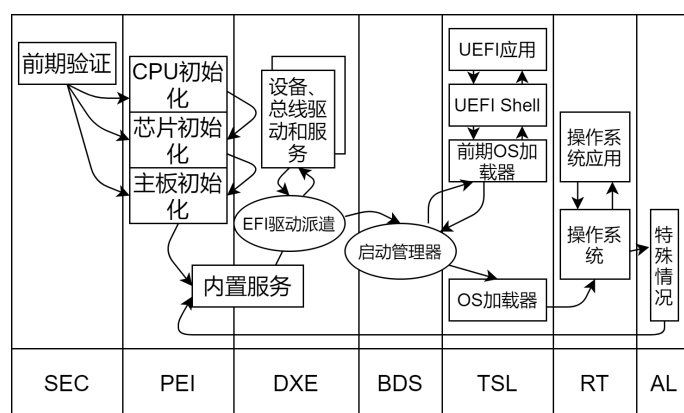


图 3-3 UEFI 启动流程

Figure 3-3 UEFI boot process

本文重点研究的过程是 UEFI 初始化过程,和 UEFI 加载硬盘 ESP 分区文件

过程。如图 3-3 所示，为 UEFI 系统的初始化流程。如图所示，UEFI 系统在遵循 UEFI 标准的 SEC 安全启动阶段后，进入 PEI 和 DXE 两个 UEFI 初始化阶段关键的驱动程序加载过程，在这里完成 UEFI 系统的内存及内存地址的建立、硬件设备驱动程序的加载、硬件设备存储器 I/O 端口映射等系统关键初始化步骤。随后通过 BDS 阶段的 BDS core 核心程序，也就是启动管理器程序，加载位于硬盘 ESP 分区中的前期 OS 加载器以及操作系统加载器。其中，前期 OS 加载器用来作为 SHELL 用户接入程序、PXE 网络启动程序等这些过程的引导器使用；而操作系统加载器则负责引导服务器上在 UEFI 启动前安装的特定操作系统。

### 3.1.2 UEFI 环境文件加载过程分析

由前面的分析可知，UEFI 环境中的硬盘文件加载过程建立在硬盘块设备已经建立好 GPT 分区格式和与 UEFI 内核中对应的文件系统格式的前提下，通过 UEFI 文件系统协议栈的层级关系，一层一层调用来实现硬盘文件数据加载到 UEFI 系统内存的过程。

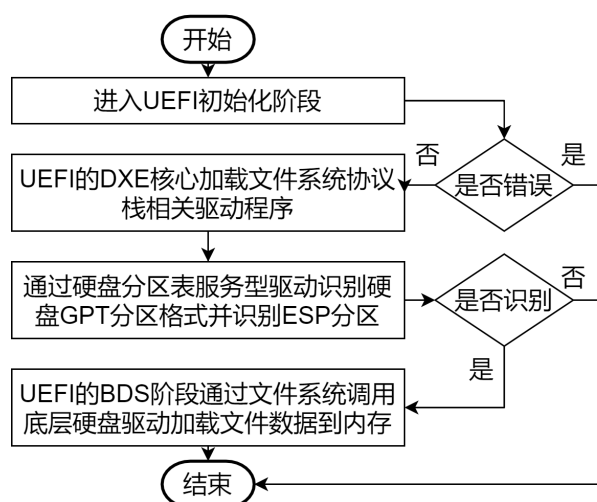


图 3-4 UEFI 文件加载过程

Figure 3-4 UEFI load file process

如图 3-4 所示，UEFI 会先通过 DXE 加载 UEFI 文件系统协议栈相关的驱动程序，具体驱动程序为：PassThruDxe、PartitionDxe、DiskIoDxe、FileSysDxe 四个驱动，其中 PassThruDxe 和 FileSysDxe 会根据具体系统的不同而确定不同的特定硬盘接口和文件系统。通过加载 UEFI 文件系统协议栈驱动程序，UEFI 内核已经完成了文件系统协议栈的建立，包括各个驱动功能函数在 UEFI 永久内存中的驻留以及对应 UEFI 协议对特定硬盘驱动的函数引用；再通过 DXE 阶段加载的 PartitionDxe 硬盘分区表服务型驱动程序，识别出硬盘的 GPT 分区格式并获取到 GPT 分区表，通过 GPT 分区表再得到硬盘的 ESP 分区的起始和结束扇区位置，从而确定 UEFI 内核在 BDS 阶段读取操作系统引导文件的分区位置；接

下来就是 UEFI 进入 BDS 阶段并通过文件系统协议栈加载硬盘 ESP 分区中的文件到内存中，并移交控制权到这个 EFI 可运行文件身上。

UEFI 文件系统协议栈加载硬盘文件数据的过程，从函数调用的角度看，就是上层 FileIo 系统调用，调用通过 FileSysDxe 加载得到的 UEFI 内核中的文件系统，并通过文件系统调用底层的由 PassThruDxe 和 DiskIoDxe 驱动提供的硬盘驱动程序。在 UEFI 环境中，驱动功能的调用通过与其对应并对其引用的协议完成。

### 3.1.3 从硬盘攻击关键文件

有了对前面的硬盘分区和文件系统，UEFI 系统内的文件系统的建立过程和 UEFI 环境中文件的加载过程的了解，可知黑客可通过硬件手段攻击位于硬盘硬件设备上的文件系统结构<sup>[35]</sup>、和攻击用来存储 BIOS 的固件芯片两方面，来达到篡改 UEFI 加载硬盘文件的文件内容。

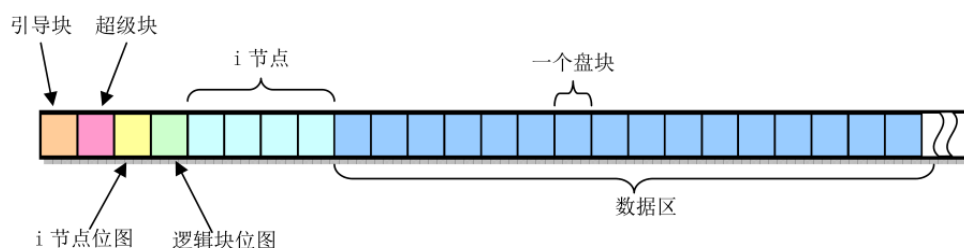


图 3-5 MINIX 文件系统硬盘布局

Figure 3-5 MINIX file system hard disk layout

首先是针对硬盘设备的攻击，如图 3-5 所示，是 MINIX 文件系统在硬盘上的数据结构布局图<sup>[36]</sup>，作为一个类 UNIX 文件系统，其与标准 UNIX 文件系统的设计结构基本相同，FAT 文件系统和 ext2 文件系统也同样属于类 UNIX 文件系统。由于各种类 UNIX 文件系统的基础设计理念和结构基本相同，不同之处仅在于更新一代的文件系统会提供更为丰富的如数据恢复、文件操作日志记录等内容，而底层基础功能结构不变，因此以 MINIX 作为说明<sup>[37]</sup>。

其中引导块对应于 UEFI 系统中硬盘的 ESP 分区，里面装载了操作系统所需的引导程序。超级块用于存放盘设备上文件系统结构的信息，并说明各个部分大小。i 节点位图用于说明 i 节点是否被使用，每个比特位代表一个 i 节点。逻辑块位图用于描述盘上的每个数据盘块的使用情况，每个比特位代表盘上数据区中的一个盘块。盘上的 i 节点部分存放着文件系统中文件或目录的索引节点，每个文件或目录都对应一个 i 节点。其中每个特定文件对应的 i 节点数据结构中，都存有一个其包含的所有文件数据磁盘块的映射关系，用于通过 i 节点索引文件数据。

由以上分析可知，黑客可通过技术手段将恶意代码文件事先存入硬盘 ESP

分区或引导块中<sup>[38]</sup>，并获取到恶意代码文件和UEFI将加载的系统文件分别对应的*i*节点，然后通过硬件手段修改*i*节点数据结构中文件数据块映射的关系，将系统文件链接到恶意代码文件的数据块链，使UEFI加载系统文件时实际运行的却是恶意代码。因此在UEFI系统中，在加载硬盘系统文件或关键文件前，对其进行完整性测量以保证加载的文件没有经过篡改，这个过程就显得十分必要。

### 3.1.4 从固件芯片攻击关键驱动

作为UEFI内核与硬盘硬件设备连接的桥梁，文件系统定义了双方共同的数据组织形式和结构，来达到数据互通的效果。因此，作为存储在BIOS固件系统上的文件系统驱动程序，同样可以作为黑客用来篡改硬盘ESP分区中关键文件信息的攻击对象。

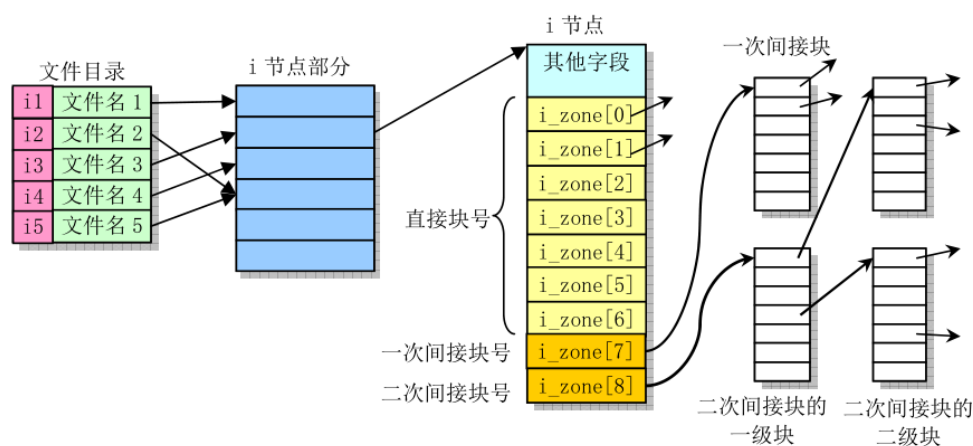


图 3-6 内存中的文件系统索引文件方式

Figure 3-6 Method of index file by file system in memory

如图 3-6 所示，当UEFI加载了文件系统驱动之后，内存中便驻留了文件系统组织文件的方式及特定的数据结构。当UEFI系统通过文件名索引硬盘中的文件数据时，会通过目录项这个数据结构中的*i*节点信息确定这个绝对路径下特定文件名对应的硬盘中的*i*节点，并以此来获取到文件数据。

在这个文件系统内部通过文件名检索文件数据的过程中，黑客可以通过改变文件目录项中文件名与*i*节点的对应关系，来篡改特定系统文件名所对应的具体硬盘数据内容<sup>[39]</sup>。同样可以在内存的文件系统中，更改*i*节点与文件数据块关系，也就是更改图 3-6 中的*i\_zone*指针所指向的位置，来完成恶意代码数据对系统文件数据的更替效果。因此，在UEFI加载文件系统模块及其相关驱动前，对他们在BIOS固件芯片中内容进行完整性度量，以确保UEFI运行的是可信的文件系统代码，可以从固件层面消除黑客对硬盘文件内容篡改的可能。

### 3.2 信任链的设计

在对 UEFI 文件系统协议栈驱动程序和硬盘 ESP 分区中系统关键 EFI 文件进行可信度量的同时，需要首先建立 UEFI 系统的可信启动。根据 TCG 平台规范，本文选用具有存储 UEFI 驱动程序度量基准值功能的 BMC 系统作为 UEFI 可信启动信任链的可信平台模块<sup>[40]</sup>，用包含基准值存取、可信度量日志记录基本功能的 BMC 来作为替代传统 TPM 的底层平台。

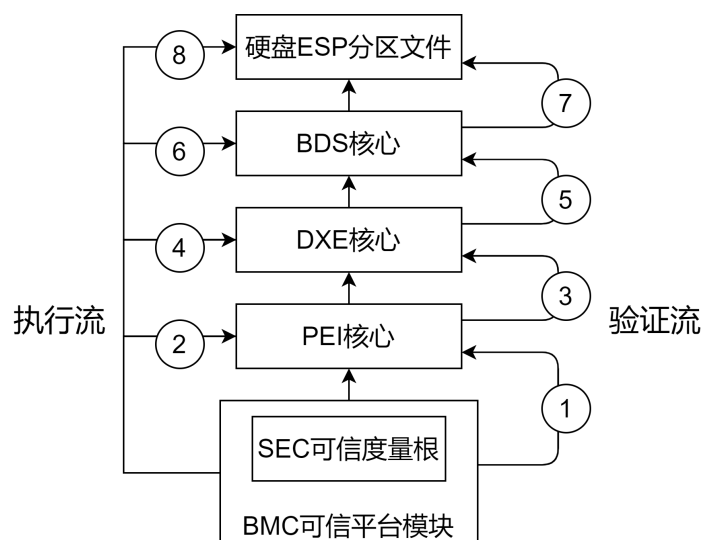


图 3-7 UEFI 信任链传递

Figure 3-7 UEFI trusted chain transfer

如图 3-7 所示，信任链以 BMC 系统作为可信平台模块<sup>[41]</sup>，并以 UEFI 的第一阶段 SEC 启动阶段作为信任链中的可信根。然后在 SEC 阶段，借助 BMC 系统中的可信平台模块对 PEI 阶段的核心代码进行可信度量，在得到可信度量的结果为 PEI 核心可信时，将系统控制权交给 PEI 核心；PEI 核心借助 BMC 中的基准值，对系统 cpu、主板和芯片组的 PEIM 组件进行可信度量，并根据度量结果最后对 DXE 核心进行度量，并将系统控制权交给 DXE 核心；DXE 核心阶段加载系统其余组件，同时对 UEFI 文件系统协议栈的驱动程序进行度量，以保证在 BDS 阶段加载硬盘文件时，数据流所经过的底层 UEFI 文件系统协议栈可信。最后对 BDS 核心进行可信度量；系统控制权交给 BDS 阶段核心代码，在加载硬盘 ESP 分区中关键文件时，对文件内容进行度量，若可信，再将控制权交给文件进行 UEFI 系统调用<sup>[42]</sup>。

### 3.3 安全方案启动阶段设计

根据 UEFI 启动阶段信任链的设计，本安全方案涉及到的 UEFI 启动阶段为 SEC、PEI、DXE、BDS，以做到在保证硬盘文件数据经过的硬盘设备、硬盘文



件系统协议栈安全可信的同时，还要保证在 BDS 阶段加载硬盘文件时，BDS 核心代码的安全可信。因此安全方案分成 SEC、PEI、DXE、BDS 四个阶段进行设计<sup>[43]</sup>，其中的 DXE 和 BDS 阶段包含了此安全方案的主要度量内容，因此单独在下一小结进行他们和总体架构的介绍。下面将对安全方案在 SEC 和 PEI 阶段的设计进行描述。

### 3.3.1 SEC 阶段

SEC (Security Phase) 阶段是 UEFI 平台初始化的第一个阶段，计算机系统加电后首先进入这个阶段。SEC 阶段首先接收并处理系统启动和重启信号，系统加电信号、系统重启信号、系统运行过程中的异常信号。还有一个重要过程为初始化临时存储区域。系统运行在 SEC 阶段时，仅 CPU 和 CPU 内部资源被初始化，而各种外部设备和内存都没有被初始化。因此系统需要一部分临时内存用于代码和数据的存储，一般称为临时 RAM，临时 RAM 只能位于 CPU 内部。最常用的临时 RAM 是 Cache，将其当成内存使用，这种技术称为 CAR (Cache As RAM)。

SEC 阶段的目的是为 PEI 阶段准备一切所需的资源，包括了通过 CAR 得到的 RAM 地址和大小、栈地址和大小以及固件卷 fv 的位置，使 PEI 可以通过这些信息加载系统资源及后续流程。本安全方案在 SEC 阶段的设计为，度量 PEI core 代码，得到度量结果，并判断是否将控制权由 SEC 传递给 PEI。

由于 SEC 阶段是此可信链中的信任根，因此 SEC 阶段的数据内容安全可信，又因为 SEC 阶段占有很少的系统资源，仅有很小的临时内存，因此不涉及 BMC 的交互功能，将 SEC 阶段度量 PEI core 的基准值信息，也就是 PEI core 的 SHA1 散列值，和临时的 SHA1 算法，集成在 SEC core 的代码中，以此来完成阶段度量的任务。SEC 阶段的 BMC 日志信息的输出由于不借助 IPMI 协议，因此不需要 BMC 驱动程序，只需调用上述 BMC 串口写入函数，即可将日志信息写入 BMC 芯片中。

### 3.3.2 PEI 阶段

系统控制权到了 PEI (Pre-EFI Initialization) 阶段，并由 PEI core 核心代码负责运行 PEI 阶段的功能。虽然 SEC 阶段对 CPU 和 CPU 内的资源进行了初始化，但是 PEI 阶段可用的资源依旧十分有限，该阶段对内存进行初始化，主要功能是为 DXE 阶段准备执行环境，将所需要传递给 DXE 的信息组成 HOB (Hand Off Block) 列表，最终将控制权转交到 DXE。

PEI 和 DXE 一个不同之处在于，DXE 拥有适量的系统永久 RAM 可供使用；而 PEI 仅仅拥有一些有限的临时 RAM，并且这些临时 RAM 在 PEI 阶段初始化

永久内存后可能会被重新配置以作其它的用途，比如缓存（Cache）。因此，PEI 没有 DXE 的资源丰富。而 PEI 阶段的一个主要目的就是以最少的系统资源初始化永久内存，也就是 UEFI 系统和后面的操作系统都可以使用的系统全部内存空间。

FV（Firmware Volume）的内容遵循 EFI 闪存文件格式的格式。PEI 基础按照 EFI 闪存文件格式的格式来发现 FV 中的 PEIM。一个平台特有的 PEIM 可以通知 PEI 基础系统中的其它固件卷所处的位置，这就允许 PEI 基础在其它固件卷体中找到 PEIM。PEI 基础和 PEIM 在 EFI 闪存文件系统中用一个唯一的 ID 来命名。

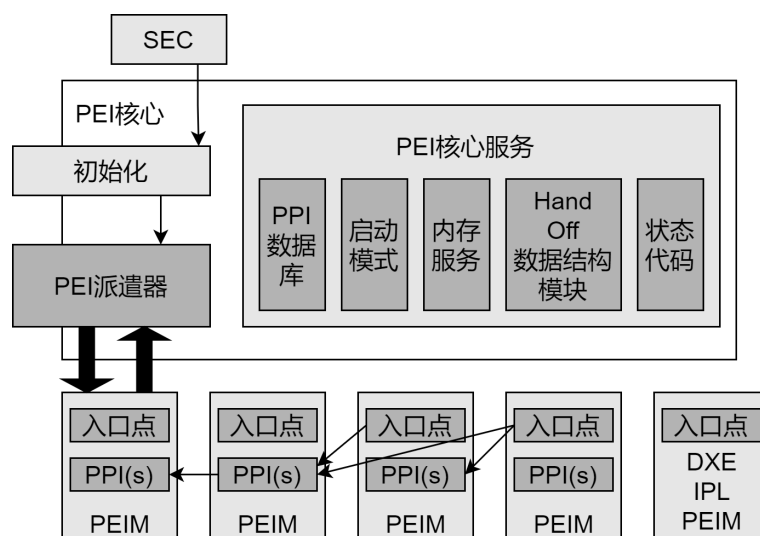


图 3-9 PEI 阶段系统结构

Figure 3-9 System structure of PEI

如图 3-9 所示，PEI 阶段的主要结构分为 PEI core 也就是 PEI 基础代码，负责 PEI 阶段的整体功能执行；PEI 阶段的系统服务，用于 PEI 阶段功能函数的调用；PEI Dispatcher 调度程序，用于调用 FV 固件卷中的 PEIM 模块，其中就包括系统最基本的 cpu、主板和芯片组的 PEIM。当一个 PEIM 调度完成之后，PEI 调度程序将继续检查固件容卷，直到出现下列两种情况之一为止：

- (1) 所有被发现的 PEIM 都已经被调度
- (2) 不再有 PEIM 可被调度

这是因为任何 PEIM 都不能满足上述列出的调度条件。一旦达到上述两个条件中的任一个，PEI 调度程序的工作就完成了，并且它调用一个用于启动下一阶段框架的架构 PPI，即 DXE 初始程序加载（Initial Program Load, IPL）PPI。

PEI 阶段的 BMC 通信功能的实现与 DXE 阶段实现过程相同，唯一不同点在于 BMC 驱动中的系统调用。其中 PEI 阶段的 BMC 通信模块需要调用名为 EFI\_PEI\_SERVICES 的 PEI 阶段核心服务结构体，也就是 PEI 阶段的系统服务，

而 DXE 阶段的 BMC 通信模块需要调用 `EFI_SYSTEM_TABLE` 也就是 DXE 阶段的系统服务。PEI 和 DXE 两个阶段不同的系统服务都是在过程开始阶段，通过前一阶段传来的 HOB 数据结构，分别是 SEC 阶段和 PEI 阶段建立的 HOB，来确定系统函数的初始化工作。由于在两个不同阶段的 BMC 通信模块中，都需要使用系统函数来进行一些 UEFI 系统缓存的建立，因此必须要分开实现。

`EFI_STATUS`

`EFI_API`

`PeiServicesLocatePpi (`

```

    IN CONST EFI_GUID          *Guid,
    IN UINTN                    Instance ,
    IN OUT EFI_PEI_PPI_DESCRIPTOR **PpiDescriptor,
    IN OUT VOID                 **Ppi

```

`)`

还有一方面必须两个阶段分开实现 BMC 通信模块的原因在于，PEI 阶段是通过 PEIM 安装的 PPI 接口来实现 PEIM 之间的通信，以及 PEI core 对 PEIM 实现的功能函数的调用，通过上面的代码把以通过加载 PEIM 安装到系统的 PPI 找到，并存放在 \*\*PPI 指针中。而 DXE 阶段则是通过 Protocol 来进行驱动函数的安装及调用。PPI 的安装和查找调用也是通过 `EFI_PEI_SERVICES` 来进行系统调用实现的。

### 3.4 总体架构设计

此安全方案中对四种 UEFI 文件系统协议栈驱动程序的度量过程和硬盘 ESP 分区中关键文件的度量过程是在 DXE 和 BDS 阶段完成的，因此这两个阶段包含了此安全方案的总体架构设计内容，具体内容如下所示。图 3-13 展示了此安全方案在 DXE 阶段的总体功能设计。

如图 3-13 所示，此系统涉及到用来存储各个阶段核心程序和文件系统协议驱动程序的基准值的 BMC 固件系统，而此安全方案中的可信度量模块以 DXE 驱动程序的形式，存储于 UEFI BIOS 固件芯片中，并通过 UEFI 系统原有的 DXE core 和 DXE 调度程序加载可信度量驱动至内存中。

在度量过程中，由 DXE 阶段的调度程序给可信度量驱动中的驱动度量模块发送度量信号，由驱动度量模块通过文件系统协议栈驱动的全局唯一标识信息匹配出四个驱动程序，并通过 UEFI 中的固件文件系统 FFS 获取到存储于 FV 固件芯片中的驱动程序数据到 UEFI 内存，并调用可信度量值计算模块计算出 FV 中驱动的 hash 值；再调用 BMC 通信模块，通过 GUID 获取到存储于 BMC 芯片

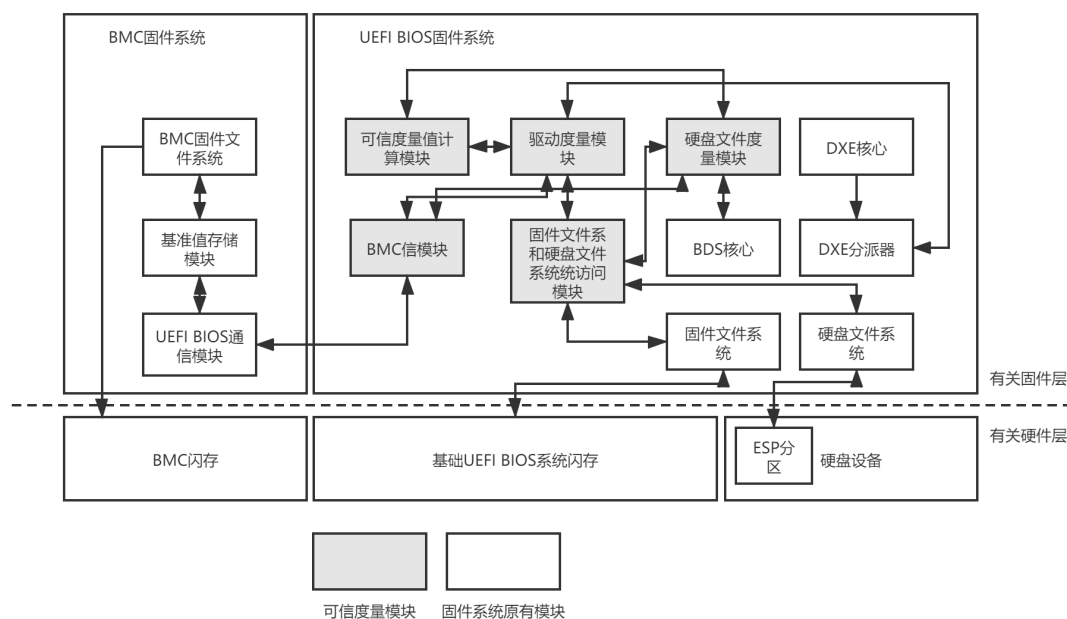


图 3-13 系统结构图

Figure 3-13 System framework

中的度量基准值，通过驱动度量模块对基准值和度量值的比较得到度量结果。

到了 BDS 阶段，在 BDS core 加载硬盘文件前，通过调用硬盘文件度量模块负责调度其他可信度量模块，具体过程和驱动度量模块相同，不同之处在于，所获得的硬盘数据文件需要通过 UEFI 文件系统协议栈加载存储于硬盘 ESP 分区中的关键文件，并进行度量获得度量结果。下面将对每个功能模块的设计和完成的内容进行介绍。

#### (1) 驱动度量模块

此模块负责通过固件文件系统 FFS 从 UEFI BIOS flash 芯片中加载 AtaAtapi-PassThruDxe、PartitionDxe、DiskIoDxe、FileSystemDxe 驱动代码得到各个驱动在内存中对应的内存映像 Image（下同）的基地址和 Image 大小，从而得到各个驱动的数据内容。其中获取 flash 芯片中数据过程通过调用固件文件系统访问模块完成。并通过 BMC 通信模块获取基准值，存储于基准值缓存模块中。此模块中通过调用动态可信度量值计算模块计算驱动的 SHA1 值，并在驱动文件可信度量顺序映射模块中进行对比得到验证结果。

#### (2) 硬盘文件度量模块

此模块通过 UEFI 环境中的文件系统协议栈调用关系，调用硬盘文件系统访问模块提取硬盘位于 ESP 分区中的关键文件数据 start\_kernel.efi 文件，并通过调用动态可信度量计算模块计算此文件的 SHA1 值，由动态可信度量计算模块将此 start\_kernel.efi 文件基准值返回给驱动文件可信度量顺序映射模块并存储在这个模块中。然后通过 BMC 通信模块获取此文件基准值，返回的结果存储于此模块中。

### （3）可信度量值计算模块

动态可信度量值计算模块，此模块通过在固件编译时加载进去的 opensslpkg 包中的安全相关 SHA1 功能函数，对驱动文件可信度量顺序映射模块中的四个 UEFI 文件系统协议栈驱动程序 AtaAtapiPassThruDxe、PartitionDxe、DiskIoDxe、FileSystemDxe 的内容进行 SHA1 值计算，以获取运行过程中的驱动度量值，用以和基础性 BMC 固件系统中的基准值存储模块中存入的基准值比对。

### （4）固件文件系统和硬盘文件系统访问模块

此模块通过判断传入的访问请求，确定需要访问固件文件系统中数据或硬盘数据。其中访问固件中文件数据过程中，此模块封装了 UEFI 服务提供的访问固件文件系统 FFS 的功能函数，将文件系统协议栈相关驱动程序加载进 UEFI 系统内存中。访问硬盘数据过程中，此模块通过使用 UEFI 文件系统协议栈内容，并调用最上层 FileIo 协议，读取硬盘设备中的关键文件数据到 UEFI 系统内存中。

### （5）BMC 通信模块

此模块负责实现 BMC 的 UEFI 驱动程序，其中通信协议为 IPMI 协议，访问模式遵循 KCS 方式，实现向 BMC 发送 IPMI 格式的命令信息，并实现获取 BMC 返回基准值的函数。从而实现 UEFI 环境中与 BMC 设备通信的功能，完成可信度量模块中获取基准值的工作。

## 3.5 本章小结

本章首先对 UEFI 系统加载硬盘 ESP 分区中文件数据的过程进行了分析，通过对文件系统数据组织结构的分析找出了文件加载过程中的安全漏洞，并针对通过 UEFI 文件系统协议栈加载硬盘数据的过程，划分了 UEFI 启动不同阶段中的安全方案，并提出了 DXE 阶段和 BDS 阶段所涉及到的可信度量驱动和相关模块的调用关系及 BMC 系统存储基准值的作用，为后文安全方案的具体实施奠定了基础。



## 第 4 章 基于 UEFI 的硬盘文件安全加载系统的详细设计

UEFI 环境中硬盘文件的安全加载依赖于 UEFI 固件系统中对 UEFI 各个启动阶段核心代码的可信验证，和对特定文件系统协议栈驱动程序和硬盘 ESP 分区中文件的可信测量，本文在第三章中已经描述了此安全方案在基于原有 UEFI BIOS 的基础上进行的阶段设计和可信度量模块的设计。在本章将会针对第三章中的设计方案提出对应的具体实施过程，其中包括可信度量模块各个模块的详细设计和根据各个启动阶段的特点进行的启动阶段安全方案详细设计。

### 4.1 可信度量驱动的实现

此安全方案设计的 DXE 阶段可信度量驱动程序属于 DXE 服务型驱动，其中包含了几个主要的功能模块，他们分别是第三章中系统结构图提出的可信度量值计算模块、固件文件系统和硬盘文件系统访问模块、BMC 通信模块、驱动程序度量模块和硬盘文件度量模块，本节将对这些模块做更细致的介绍和实现细节。

#### 4.1.1 可信度量值计算模块

可信度量值计算模块是对自定义 SHA1 散列函数的封装，用于对 DXE 阶段的四个 UEFI 文件系统协议栈驱动程序进行完整性度量，并对 BDS core 进行度量；也负责在 BDS 阶段对硬盘文件数据进行可信测量。

此度量过程采用 SHA1 散列值计算方法，SHA1 是由 NISTNSA 设计为同 DSA 一起使用的，它对长度小于 2 的 64 次方的输入，产生长度为 160bit 的散列值，因此抗穷举 (brute-force) 性更好。SHA-1 设计时基于和 MD4 相同原理，并且模仿了该算法。SHA-1 是由美国标准技术局 (NIST) 颁布的国家标准，是一种应用最为广泛的 hash 函数算法，也是目前最先进的加密技术，被政府部门和私营业主用来处理敏感的信息。而 SHA-1 基于 MD5，MD5 又基于 MD4。

```
typedef struct {
    EFI_SHA1_INIT SHA_Init;
    EFI_SHA1_UPDATE SHA_Update;
    EFI_SHA1_FINAL SHA_Final;
    EFI_SHA1_CLEAN SHA_Clean;
} EFI_SHA1_PROTOCOL;
```

代码列出的是自定义的 EFI\_SHA1\_PROTOCOL 协议,用于在加载 DXE 阶段的可信度量驱动时通过 Openprotocol 的启动时服务的系统调用加载到 UEFI 系统的句柄数据库中。其中 EFI\_SHA1\_INIT,EFI\_SHA1\_UPDATE,EFI\_SHA1\_FINAL,EFI\_SHA1\_CLEAN 为四个函数指针,用于指向位于驱动中的函数实现。SHA\_Init 函数指针所指向的函数用于初始化一个用于 SHA1 算法加密过程的数据结构 SHA\_CTX,该结构存放了生成 SHA1 散列值的一些参数。SHA\_Update 函数用于处理大文件,将其分散成等份的较小值,并对每一块分别调用 SHA\_Update 生成对应的散列值。SHA\_Final 函数用于将 SHA\_Update 函数生成的分块的散列值通过运算形成一个最终的 160bits 的散列值。SHA\_Clean 函数用于清除 SHA\_CTX 数据结构中针对 SHA1 算法初始化的数据。

```
typedef struct SHAsate_st {
    SHA_LONG h0,h1,h2,h3,h4;
    SHA_LONG Nl,Nh;
    SHA_LONG data[SHA_LBLOCK];
    unsigned int num;
} SHA_CTX;
```

在结构体 SHA\_CTX 中,SHA\_LONG 定义为 unsigned int 类型,SHA-1 采用 160 位的信息摘要,也以 32 位为计算长度,就需要 5 个链接变量,因此 h0-h4 用来在 SHA\_Init 过程中初始化并存储这 5 个链接变量用于度量过程中的计算。其中的 SHA\_LBLOCK 变量的扩展值为 64,意味着 SHA1 在进行分组运算时,每一组的长度为 512bits 及 64Bytes。

#### 4.1.2 固件文件系统访问模块

固件文件系统访问模块用于给可信度量值计算模块提供各个阶段的核心代码和 DXE 阶段的文件系统协议栈驱动程序在 FV 固件卷中的数据信息。需要根据 FV 固件卷中 FFS 固件文件系统的数据格式提取出对应的核心或驱动文件的 Image 信息<sup>[44]</sup>。但 FV 固件卷中存储的 FFS 格式的文件中并不只包含着可运行的 EFI 类型二进制文件,还有着许多标识着文件类型、文件修改信息等众多段信息,若要对 UEFI 各个阶段核心代码和特定驱动文件进行度量,必须确定需要度量的 EFI 可执行文件的 data 部分内容,以确保基准值的计算与 UEFI 启动过程中度量值的计算所计算的 EFI 数据内容统一。

本节将详细说明 FFS 文件的存储格式以及此安全方案的固件文件系统访问模块如何进行的设计与实现。



4.1.2.1 FFS 文件存储格式

根据第二章中所述的 FV 固件卷数据存储方式的介绍可知，FV 固件中的文件以 section 的形式分段存储，每个数据段存储特定的文件规格信息。但要在 UEFI 内存中精确的获取到核心和驱动程序文件的 EFI 数据内容，就需要更详细的 FFS 文件存储格式研究。FV 固件卷中文件的存储方式表示在图 4-1 中。

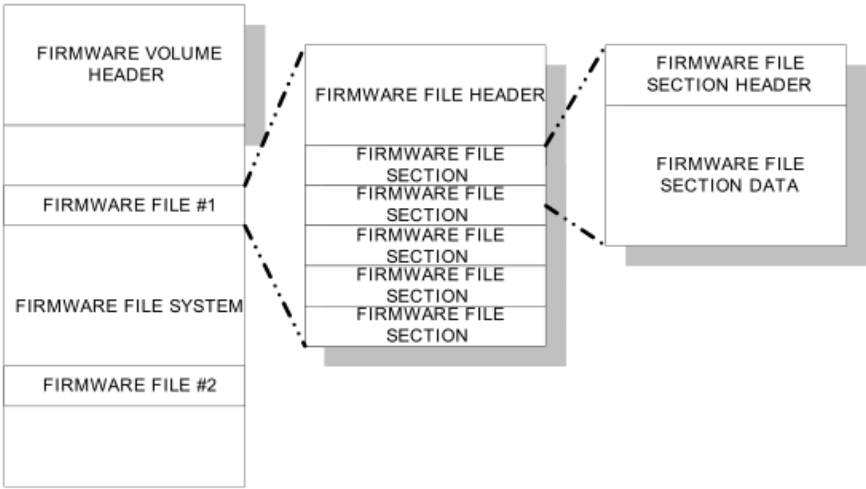


图 4-1 固件卷数据存储格式

Figure 4-1 The Firmware Volume Format

如图 4-1 所示，符合 PI 规范的 FV 固件卷文件包含主要两部分，其中一部分是如图所示的 firmware volume header 固件卷头部，另一部分是 firmware volume data 固件卷数据部分。其中固件卷头部用于描述此 FV 文件的所有属性信息，同时也包含了一个用来标识组织固件中数据存储格式的固件文件系统的 UEFI 全局标识符 GUID。固件卷头部不仅可以支持 UEFI 特定的 FFS 固件文件系统，还支持一切符合 PI 规范的固件文件系统。

当 UEFI 系统加载 FV 中的文件信息时，首先识别固件卷头部，并用名为 `EFI_FIRMWARE_VOLUME_HEADER` 的数据结构用来在 UEFI 内存中存储头部信息。固件卷数据部分也用 GUID 的形式在内存中进行唯一标识，其中 UEFI 规范认可的固件文件系统 GUID 用 `EFI_FIRMWARE_FILE_SYSTEM2_GUID` 和 `EFI_FIRMWARE_FILE_SYSTEM3_GUID` 写死在 BIOS 的固件芯片中。当单个文件内容小于 16MB 时，系统将采用 `EFI_FIRMWARE_FILE_SYSTEM2_GUID` 来描述固件文件系统，若单个文件内容大于 16MB，并且向后兼容 `SYSTEM2` 类型的 FFS 时，将采用 `EFI_FIRMWARE_FILE_SYSTEM3_GUID` 来描述。

如图 4-1 所示，固件卷中每一个文件以单独的结构进行存储，文件名称为 FFS file。一个文件中包含了文件头部，用于记录文件中 section 内容及文件属性的信息，如图 4-2 中所示的文件状态、文件大小、文件名称、文件属性、文件类

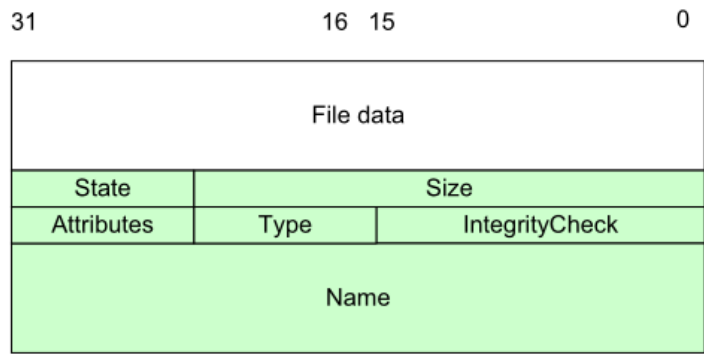


图 4-2 固件文件系统文件布局

Figure 4-2 Typical FFS File Layout

型等信息，图 4-2 展示的是一个 ffs 文件的文件结构组成，其中所有的头部和数据信息都是 32 位对齐的。同时包含了一组 section。每一个 section 中也分别包含一个 section 头部信息，还有 section 中的数据内容。UEFI 规范所规定的固件文件系统 section 分类及信息如表 4-1。

表 4-1 固件文件系统分段信息

Table 4-1 The Firmware File System Section Imformation

名称	数值	描述
EFI_SECTION_COMPRESSION	0x01	封装了其他被压缩的分段
EFI_SECTION_GUID_DEFINED	0x02	封装部分，其中其他部分的格式由 GUID 定义
EFI_SECTION_DISPOSABLE	0x03	在构建过程中使用的封装部分，但执行时不需要
EFI_SECTION_PE32	0x10	PE32 格式信息和可执行映像
EFI_SECTION_PIC	0x11	位置无关的代码段
EFI_SECTION_DXE_DEPEX	0x13	DXE 阶段依赖表达式
EFI_SECTION_VERSION	0x14	版本号，文字和数字信息
EFI_SECTION_RAW	0x19	raw 数据信息
EFI_SECTION_PEI_DEPEX	0x1b	PEI 阶段依赖表达式

如表 4-1 所示，在 FV 固件卷中的每个 section 中，都对应着不同的 section 分类信息，其中可以看到，EFI\_SECTION\_PE32 这个 section 中存放着 DXE 驱动程序相关的关键信息，其中包含了 PE32 格式的说明信息，还有就是对应的 EFI 可执行文件，也就是驱动程序真正的可执行代码部分内容。在此安全方案的可信度量模块中，度量的就是 UEFI 启动各个阶段核心代码和驱动程序代码在 FV 固件卷中对应这个 section 里的相关信息。从表 4-1 中还可以看出，如 EFI\_SECTION\_GUID\_DEFINED 段的信息内容可以在 FV 固件卷中标识驱动程序的 GUID 信息，这也为在 UEFI 启动阶段在加载驱动程序时，通过内存中的驱

动 GUID 和 FV 固件中的 GUID 匹配做好了实现基础。在众多 section 中，同样包括了如 EFI\_SECTION\_VERSION 这样的驱动文件版本的文字和数字信息标识。

表 4-1 中的 EFI\_SECTION\_DXE\_DEPEX 和 EFI\_SECTION\_PEI\_DEPEX 段的信息内容同样是两个关键的段信息，他们分别用于表达 DXE 阶段和 PEI 阶段对应的各个驱动程序和 PEIM 的加载顺序，此安全方案的实现需要借助这两个驱动依赖关系的字段来确定关键驱动加载的顺序问题，具体的依赖表达式关系及实施将在后面的小结中详细说明。

#### 4.1.2.2 固件文件访问模块详细设计

在确定了 FFS 固件文件系统数据存储的具体格式之后，将在此可信度量驱动中的固件文件系统访问模块中实现 UEFI 内存中的驱动文件加载方法和内存中的存储方式。此模块将调用 Boot Service 启动服务中的 LoadImage () 系统函数，在 UEFI 中，所有系统服务中存储的都是具体函数的函数指针，因此真正的映像加载功能函数由 CoreLoadImage () 函数负责实现，并且其他系统函数也遵循这样的命名规则。具体函数的调用过程可通过代码分析得知，为 CoreLoadImage 调用 CoreLoadImageCommon 再最终通过 CoreLoadPeImage 函数进行 PE32 格式的文件数据加载。而这个格式也就对应上面提到的 FV 中的 EFI\_SECTION\_PE32 段里的内容。

通过 CoreLoadPeImage 函数来完成驱动程序在固件芯片中的加载。其中几个重要的参数包括，Pe32Handle 输入参数指定了加载到 UEFI 内存中的 PE32 格式的映像句柄。Image 参数为 LOADED\_IMAGE\_PRIVATE\_DATA 格式的数据结构，用于在系统内存中表示此 DXE 驱动程序的详细信息，其中也包含了 PE32 文件格式的头部信息和 data 数据信息。DstBuffer 参数为可选输入参数，用于指定将此驱动文件映像存储至调用者指定的内存 buffer 缓存中。EntryPoint 参数同样为可选参数，用于指定 PE32 section 中 EFI 可执行文件的入口点。Attribute 为 32 位的 image 属性信息参数，每一 bit 位分别用于表示 image 信息。

其中 IN 和 OUT 为通过 C 语言 #define 宏定义定义的空类型名称，用于给编码人员确定函数的输入和输出参数的设定，UEFI 实现中函数统一采用这种方式，符合 UEFI 规范。

```
typedef struct {
    .....
    PE_COFF_LOADER_IMAGE_CONTEXT ImageContext;
    EFI_STATUS                      LoadImageStatus;
} LOADED_IMAGE_PRIVATE_DATA;
```

如上面代码所示, LOADED\_IMAGE\_PRIVATE\_DATA 数据结构用于在 UEFI 内存中表示 DXE 驱动程序的完整信息, 其中 Handle 和 EntryPoint 和 CoreLoad-PeImage 函数的参数保持一致。ImageBasePage 和 NumberOfPages 字段用于描述此驱动文件在内存中所占以页为单位的起始页位置和所占页数。也包括了用 32 位存储的驱动类型信息, 其中 UEFI 中支持的驱动映像类型包括:

```
#define EFI_IMAGE_SUBSYSTEM_EFI_APPLICATION 10
#define EFI_IMAGE_SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER 11
#define EFI_IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER 12
```

他们分别表示 UEFI 上层应用型映像文件, 启动时服务的驱动映像文件和运行时驱动映像文件, 并且 PE32 格式的映像类型属于 EFI\_IMAGE\_SUBSYSTEM\_EFI\_RUNTIME\_DRIVER 类型, 可在加载驱动映像时进行类型的判别。

LOADED\_IMAGE\_PRIVATE\_DATA 数据结构中的ImageContext 关键字段为可信度量过程的主要依据, ImageContext 是一个 PE\_COFF\_LOADER\_IMAGE\_CONTEXT 类型结构体, 用来记录驱动文件加载信息。

```
typedef struct {
    .....
    VOID          *ImageBase;
    UINT64        ImageSize;
    .....
} PE_COFF_LOADER_IMAGE_CONTEXT;
```

其中 ImageBase 就是记录内存中 PE32 格式文件的起始地址, ImageSize 提供了文件的大小, 通过把 PE32 格式驱动映像文件加载到指定地址, 并将其记录在这两个变量中, 就可以在度量时获取到驱动文件内容。

#### 4.1.3 硬盘文件系统访问模块

硬盘文件系统访问模块提供 FAT 文件系统中 ESP 分区内系统文件数据的加载过程, 此模块由 BDS core 代码调用, 用来将指定硬盘文件数据加载到 UEFI 内存中, 过程如下所示。

```
Status = VolumeInterface->OpenVolume(VolumeInterface, &RootIo);
Status = RootIo->Open(RootIo, &FileIo, imgdir, EFI_FILE_MODE_READ,
    EFI_FILE_READ_ONLY);
Status = FileIo->Read(FileIo, &Size, (VOID *)DesAddr);
```

如代码清单所示, 展示了硬盘文件访问过程中几个重要的过程, 包括了硬盘卷的打开、根索引阶段的获取、文件的搜索和文件数据的读取。

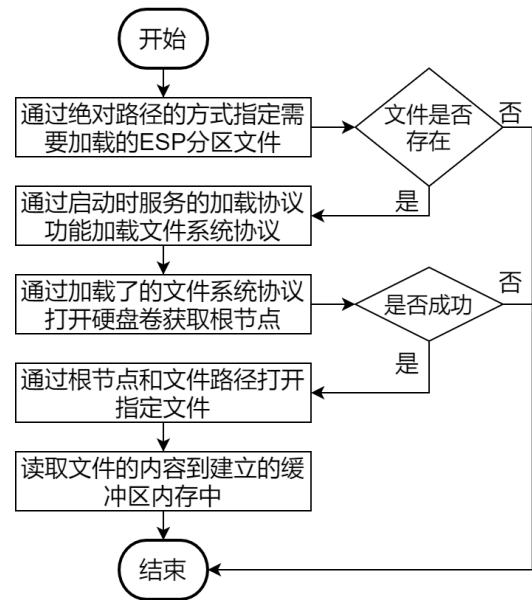


图 4-3 加载硬盘文件流程

Figure 4-3 Process of loading hard disk files

首先需要通过 Boot Service 的 HandleProtocol 服务来加载系统提供的 EFI\_SIMPLE\_FILE\_SYSTEM\_PROTOCOL 文件系统协议，将其安装在提供的 Handle 句柄上。然后调用协议中的 OpenVolume 打开硬盘卷文件，将信息挂载到 RootIo 这个 EFI\_FILE 结构中，这个结构对应的是 UEFI 文件系统协议栈里的 FileIo 协议。然后就是通过提供的具体文件相对路径，通过 EFI\_FILE\_READ\_ONLY 文件只读模式将其打开到 FileIo，并通过文件协议中的 Read 方法读取文件内容到指定地址。

#### 4.1.4 驱动文件度量模块

驱动文件度量模块是此安全方案的核心内容，因为此模块需要负责对 UEFI 文件系统协议栈的四个驱动程序进行可信度量，以确保他们的内容不被非法篡改。在有了固件文件系统访问模块的基础上，已经可以获取到 FV 固件中的驱动文件内容，而度量模块，需要对加载到 UEFI 内存中的驱动文件内容进行信息匹配，以确保进行特定驱动程序的度量工作。因此需要根据 UEFI 规范，确定四个驱动程序的 GUID 内容，以进行匹配。

[Defines]

```
BASE_NAME    = DiskIoDxe
FILE_GUID    = 6B38F7B4-AD98-40e9-9093-ACA2B5A253C4
...
```

[Defines]

```
BASE_NAME    = PartitionDxe
```

```

FILE_GUID    = 1FA1F39E-FEFF-4aae-BD7B-38A070A3B609
...
[Defines]
BASE_NAME    = AtaAtapiPassThruDxe
FILE_GUID    = 5E523CB4-D397-4986-87BD-A6DD8B22F455
...
[Defines]
BASE_NAME    = FatFileSys
FILE_GUID    = a6a0274b-78da-4b77-8fb2-9c355a2e7f6a
...

```

如上面代码清单所示，四个 UEFI 文件系统协议栈驱动的 GUID 分别标注在了四个驱动 INF 模块文件的 [Defines] 字段中。在确定了 GUID 后，就需要在度量功能的代码中进行 GUID 的匹配，再进行驱动数据内容的度量。需要说明的是，在四个驱动程序中，由于 UEFI 使用的 GPT 分区格式和各个硬件设计为统一格式，因此 PartitionDxe 和 DiskIoDxe 驱动在各个真机平台上是统一的；而 PassThruDxe 和 FileSys 驱动需要根据不同的物理平台确定不同的驱动和特定 GUID。这里采用的是申威 6A 型号服务器的具体驱动名称和 GUID 信息<sup>[45-46]</sup>。

在驱动度量模块中，另一个重要的功能就是生成度量日志的功能。此功能的目的是向 BMC 系统发送 UEFI 启动过程中度量 UEFI 文件系统协议栈的四个驱动和特定硬盘 ESP 分区文件时，将度量过程中产生的 hash 值和 BMC 中取出的基准值，以及度量结果通过日志写入的方式，存储于 BMC 系统固件中。由于此功能不需要 UEFI 系统和 BMC 系统尽心数据互送，只需要 UEFI 系统单向得将 ASCII 码信息写入 BMC 系统，因此通信方式没有采用 IPMI 协议，而是通过 IO 端口映射的方式，将数据流写入目的地址。

如图 4-4 所示，描述的是度量日志写入功能中向串口写入信息的函数。其中输入的日志信息是使用系统调用 AsciiVSPrint 函数后，通过 DEBUG 信息生成 ASCII 码格式的字符串。PRINTCONFIG 结构用来在内存中表示打印信息参数的数据结构，用于记录每次 Buffer 需要写入的目的地址，其中 Start 字段是 PRINTCONFIG 在内存中的起始地址，Index 字段用于在每次写入以此 Buffer 数据后，自加一以达到下次调用串口写入函数时，写入到上次数据内容的后面。IO 映射地址的位置是在系统 DXE 阶段初始化过程中确定的。值得注意的是，这个串口写入函数是处理器体系结构相关的，因为 PRINTCONFIG 结构在计算地址时用到了 cpuid 值，并且在系统初始化时 PRINTK\_BUF\_CONFIG 值的内容也是根据具体物理真机系统的自定义而决定的<sup>[47]</sup>。

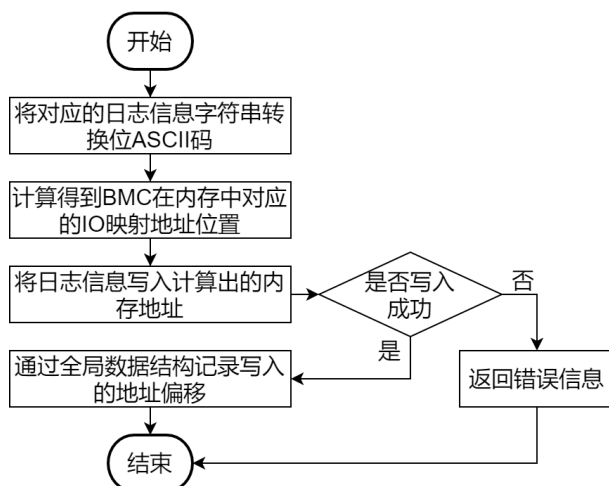


图 4-4 日志信息写入流程

Figure 4-4 Process of writing log information

硬盘文件度量模块的逻辑与驱动度量模块相似，需要通过文件名来调用 BMC 通信模块，获取到 BMC 芯片中存储的文件基准值信息，并在内存中通过 SHA1 算出加载前的度量值，进行比较得到度量结果。

#### 4.1.5 BMC 通信模块

BMC 通信模块是图 3-13 中所示的，在此安全方案中，属于 DXE 阶段和 BDS 阶段所使用的与 BMC 系统通信的功能模块。由于 DXE 和 BDS 阶段共同使用的是 DXE 阶段初始化的 Boot Service 和 Runtime Service，因此保证了 BMC 通信模块中系统调用的统一性。由于需要通过 BMC 通信模块获取基准值，因此不光需要向 BMC 发送数据，还需要 BMC 处理并返回对应的结果，因此借助于 IPMI 协议来实现<sup>[48]</sup>。

```

struct _EFI_IPMI_INTERFACE_PROTOCOL {
    EFI_IPMI_INTERFACE_SEND_COMMAND SendCommand;
    EFI_IPMI_INTERFACE_RECEIVE_PACKET ReceivePacket;
    EFI_IPMI_INTERFACE_GET_STATUS GetStatus;
};
  
```

如上面的代码清单所描述，DXE 阶段的 BMC 通信模块定义了一个名为 EFI\_IPMI\_INTERFACE\_PROTOCOL 的 IPMI 通信协议结构体，其中包含了三个主要的函数指针。SendCommand 函数用于通过 UEFI 系统向 BMC 系统发送 IPMI 格式的命令代码，ReceivePacket 函数用于在 UEFI 系统中读取 BMC 系统写入特定硬件缓存并通过 UEFI 的 IO 端口映射到内存中的地址，GetStatus 函数则用于读取特定的 BMC 寄存器，并按位解析 BMC 状态信息<sup>[49]</sup>。

BMC 通信模块以 DxeInitIpmiKcs 作为入口函数，加载对应的 IPMI 协议

中三个主要功能函数到 Protocol 数据结构中, 并通过 gBS->InstallMultipleProtocolInterfaces, Boot Service 中的安装协议函数通过 gEfiIpmiInterfaceProtocolGuid 协议的全局标识信息, 安装于 EFI\_IPMI\_INSTANCE 结构的 Handle 字段中。

#### 4.1.5.1 发送数据过程

发送函数接收主要的 8 位标识的命令信息, 和命令数据大小, 然后根据 IPMI 协议提供的 KCS 数据发送状态流, 依次设置 KCS 命令和查询相关寄存器, 配合数据接收寄存器的数据写入过程<sup>[50]</sup>。代码的写入流程如图 4-5。

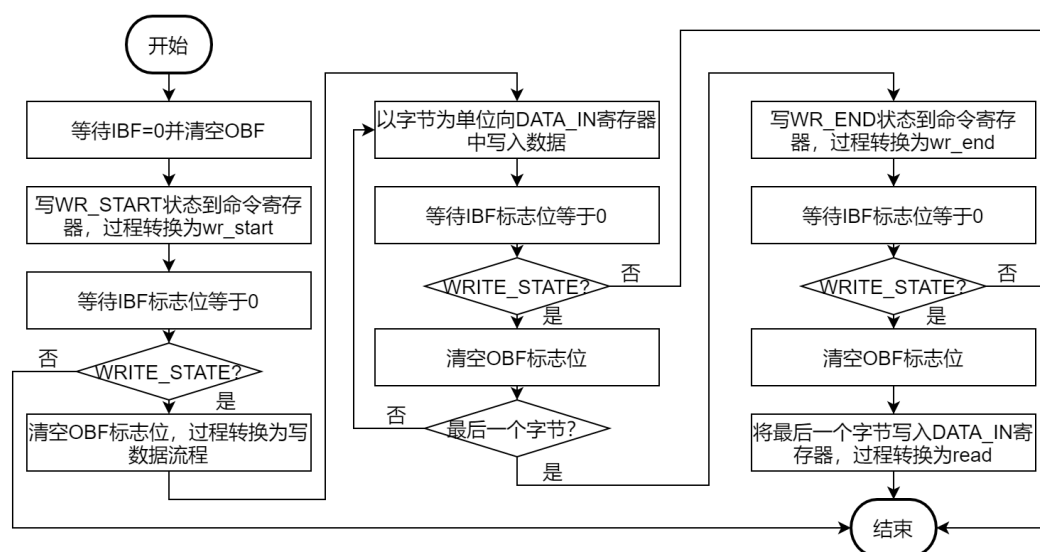


图 4-5 kcs 模式数据写入流程

Figure 4-5 KCS Write Transfer Flow

如图 4-5 所示, 是 IPMI 协议的 KCS 访问模式中数据写入 BMC 的执行流程, 其中一共涉及到 4 种 BMC 寄存器的访问与操作, 他们包括 Data\_In\_R、Data\_Out\_R、Status\_R 和 Command\_R, 其中 Status\_R 是查询 BMC 状态的寄存器, Data\_Out\_R 和 Data\_In\_R 负责给 BMC 和外界系统如 BIOS, 提供数据的读出和写入缓存, 用寄存器的方式来实现。Command\_R 寄存器用来表示输入 BMC 系统的命令。

在图 4-3 中, 首先等待 Status\_R 寄存器中的 IBF 标志位为 0, 表示此时没有数据输入到 Data\_In\_R 中。然后通过将 WRITE\_START 命令写入 Command\_R 寄存器, 将 BMC 状态改变为 wr\_start 写入开始状态。随后通过查询 Status\_R 中的值, 等待 IBF 标志位为 0, 表示 BMC 收到了 WRITE\_START 命令并判别有效; 同时通过 8 位的状态寄存器获取到的值确认 BMC 此时的状态确实改变为了 WRITE\_STATE。后面就进入了数据写入的循环中, 由于 KCS 模式是按字节传输数据, 因此为 Command\_R 寄存器设置两个命令分别为 WRITE\_START 和 WRITE\_END 来确定传入缓冲寄存器中的第一个和最后一个字节为单位的数据



包。每写入一个字节的的数据之后，都要通过 Status\_R 检查 IBF 位，确定数据输入是否有效，同时检查 BMC 是否还保持 WRITE\_STATE 状态。当根据数据 size 判断下一个字节是最后一个传输字节时，退出写入循环，并向 Command\_R 写入 WRITE\_END 命令，来通知 BMC 下一个字节是最后一个。写入最后一个字节的流程和前面循环中的写入流程一致。最后 BMC 的状态转换成了 read，表示等待外部系统读取 Data\_Out\_R 中的返回值。

表 4-2 BMC 状态寄存器标志位说明  
Table 4-2 KCS Interface Status Register Bits

比特位	名称	描述
0	OBF	输出数据有效标志位
1	IBF	输入数据有效标志位
2	SMS_ATN	表明 BMC 有事务需要软件系统处理
3	C_D_n	表明最后写入的是 Command_R 还是 Data_In_R，“1”表示 Command_R
5: 4	OEM	保留标志位
7: 6	S1:S0	KCS 读写状态机状态

如表 4-2 所示，其中 7: 6 也就是 bit7 和 bit6 形成一个四种情况的状态组合，有限状态机组合情况如下：

00: IDLE\_STATE，表示 KCS 接口是空闲的，外部系统不应预期收到或向其发送任何数据。

01: READ\_STATE，读状态表示 BMC 正在向外部系统传输一个一字节大小的数据包，此时外部系统也应处于对应的读取信息状态。

10: WRITE\_STATE，写状态表示 BMC 正在从外部系统接收一个数据包，此时外部系统需要向 BMC 写入一个命令。

11: ERROR\_STATE，表示 BMC 在接口级别检测到协议违规，或者传输已中止。外部系统可以使用 Get\_Status 控制代码来请求错误的性质，也可以仅重新执行该命令。下面将介绍 UEFI BIOS 从 BMC 获取返回值的流程。

4.1.5.2 接收数据过程

KCS 访问模式下外部系统读取 BMC 数据过程，所使用到的寄存器信息与发送过程一致。如图 4-6 所示，由于在外部系统写入数据到 BMC 后，BMC 状态机转换位等待读取的状态，因此首先检查 IBF 标志位保证上一次输入 BMC 的数据有效后，检查状态是否为 READ\_STATE，此状态为读取 BMC 数据的循环判断状态。当状态有效时，在 OBF 为 1，保证输出有效的情况下，读取 DATA\_OUT 中的一字节数据，随后再向 DATA\_IN 中写入 READ，通过外部系统将 BMC 设

置为 READ\_STATE，然后进入读取循环。直到读取完最后一个字节后，并由外部系统写 READ 指令后，BMC 改变自身状态为 IDLE\_STATE，以此来退出读取循环，并写入输出缓存一个 dummy 字节。外部系统读取 dummy 字节后，完成整个读取操作。

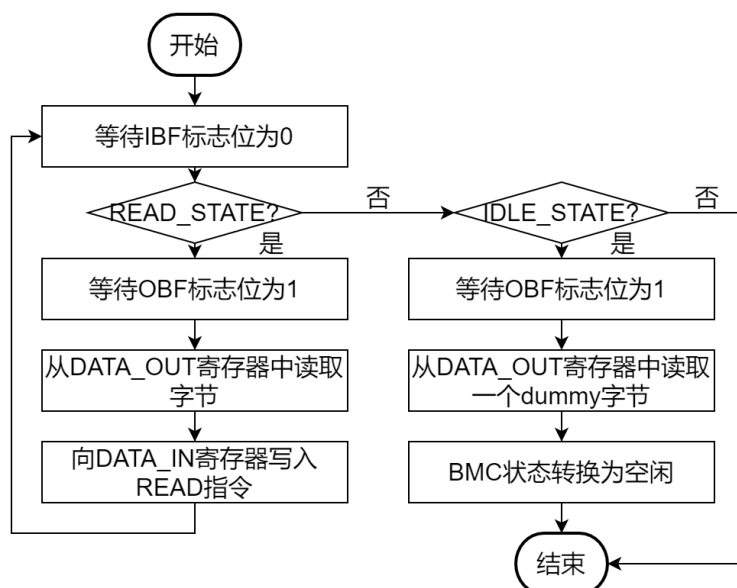


图 4-6 kcs 模式数据读取流程

Figure 4-6 KCS Read Transfer Flow

## 4.2 UEFI 启动阶段详细设计

### 4.2.1 SEC 阶段设计

由于 SEC 阶段是此可信链中的信任根，因此 SEC 阶段的数据内容安全可信，又因为 SEC 阶段占有很少的系统资源，仅有很小的临时内存，因此不涉及 BMC 的交互功能，将 SEC 阶段度量 PEI core 的基准值信息，也就是 PEI core 的 SHA1 散列值，和临时的 SHA1 算法，集成在 SEC core 的代码中，以此来完成阶段度量的任务。SEC 阶段的 BMC 日志信息的输出由于不借助 IPMI 协议，因此不需要 BMC 驱动程序，只需调用上述 BMC 串口写入函数，即可将日志信息写入 BMC 芯片中。

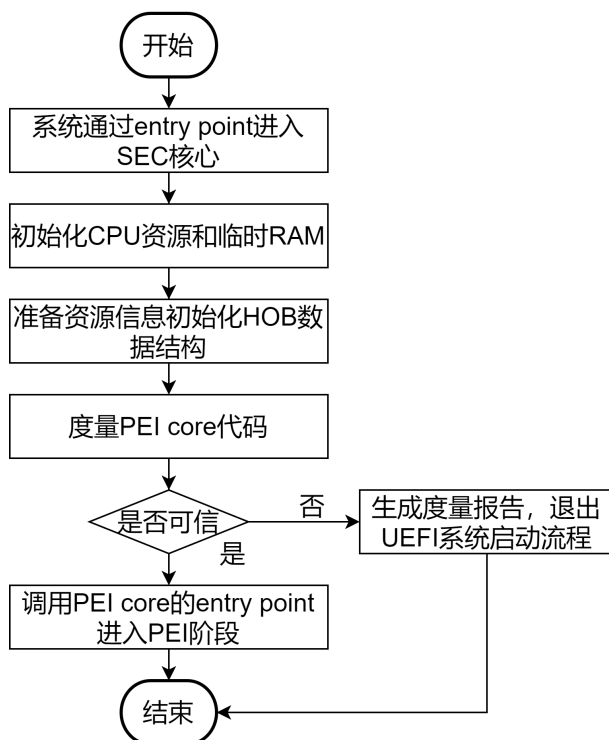


图 4-7 SEC 阶段执行流程

Figure 4-7 Execution process of SEC

如图 4-7 所示，在系统正常执行 SEC 阶段代码中加入可信度量 PEI core 代码的过程，并根据度量结果，若 PEI core 内容可信，则控制权交给 PEI 的 entry point 入口函数，否则停止 UEFI 启动流程，并生成度量日志。

#### 4.2.2 PEI 阶段设计

如图 4-8 所示，为本安全方案设计的 PEI 阶段流程。UEFI 系统进入 PEI 阶段后，首先根据从 SEC 阶段传来的 HOB 数据结构初始化 PEI 系统服务，用于给 PEIM 的加载过程提供支持；然后通过 PEI 阶段的调度程序加载位于 FV 固件卷中的 PEIM 程序，除了 cpu、主板和芯片组的 PEIM 外还包括如提供 PPI 通信协议的 PEIM 等内容；随后 PEI core 通过这些最小的功能集合初始化 UEFI 系统的永久内存；在系统获得永久内存后，PEI core 再次调用 PEIM 调度程序，这次的目的是在完整的永久内存中加载这些 PEIM，并在永久内存中执行后面流程；随后加载 FV 中的 PEI 阶段的可信度量模块 PEIM，用这个 PEIM 的功能度量 FV 中的 DXE core 代码，并根据度量结果继续加载 DXE core 或是停止 UEFI 启动流程。

由于 PEI 阶段设计为只度量 DXE core 核心代码，并且 PEI 阶段存在着两次驱动加载过程，分别是临时内存的关键 PEIM 加载，这个目的是用最少的系统资源初始化 post memory 永久内存；永久内存建立后，再次调用 PEIM 的调度程序，将 PEIM 加载到永久内存中。而 DXE core 入口函数需要在第二次永久内存中调用，因此，只需在第二次加载 PEI 阶段的可信度量 PEIM 即可，这样可以减少系

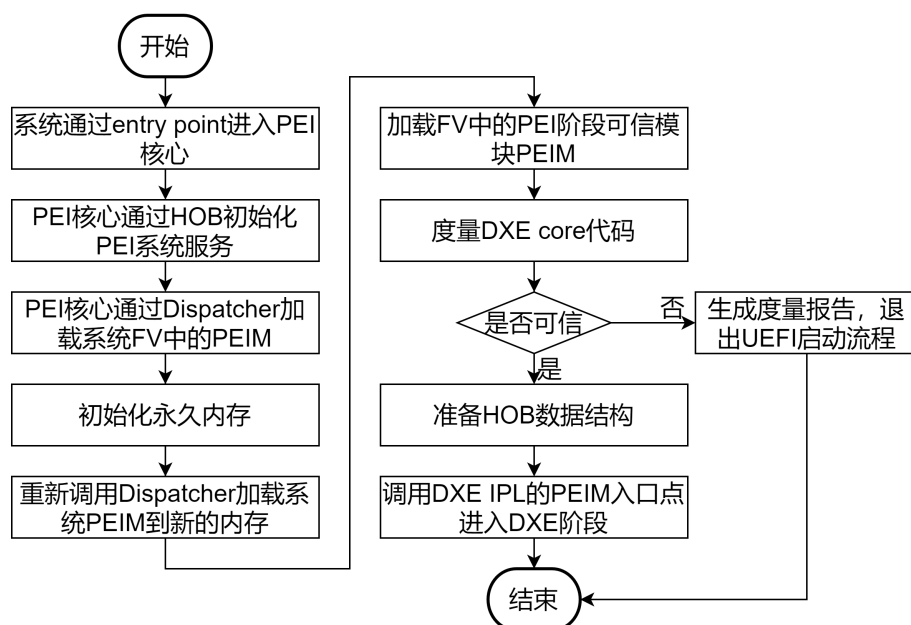


图 4-8 PEI 阶段流程图

Figure 4-8 Execution process of PEI

统不必要的加载 PEIM 带来的开销，具体流程如图 3-10 所示。

在 PEI 阶段的 PEIM 调度程序中，会通过 PEI core 句柄中的 FvPpi 调用其中的 GetFileInfo 函数，将 PEIM 的信息存放在 EFI\_FV\_FILE\_INFO 结构体中，因此可以通过 EFI\_FV\_FILE\_INFO 中的 EFI\_GUID 字段，来判断此 PEIM 是否为 PEI 阶段的可信度量模块，若是，则通过 EFI\_PEI\_SERVICES 中的 PeiMemoryInstalled 布尔类型的变量来确定此次调度器的调用是否在永久内存中，若是，再进行此可信度量模块的加载，否则跳过。由于 PEI core 和 DXE core 不是平台相关的，因此大部分平台都会调用相同的 EDKII 实现的 PEI 和 DXE 核心程序和调度程序，此处提到的变量名称与 EDKII 开源项目中的名称统一。

### 4.2.3 DXE 阶段设计

DXE 阶段为本安全方案度量 UEFI 文件系统协议栈驱动程序的主要系统启动阶段，目的是为后面 BDS 阶段通过文件系统协议栈加载硬盘文件时，保证内核中文件系统服务程序的安全可靠。由于此安全方案的主要目的为保证 UEFI 环境加载硬盘文件的安全可信，因此将 DXE 驱动度量范围缩小至 UEFI 文件系统协议栈相关驱动，达到减小系统开销的目的。

如图 4-9 所示，系统进入 DXE 阶段后，首先根据 PEI 阶段传来的 HOB 数据结构初始化 DXE 阶段的 Boot Service 和 Runtime Service，用以给 DXE 阶段的驱动程序及其加载提供系统服务；随后控制权交给 DXE 阶段的调度程序调度 DXE 驱动的加载，根据 DXE 驱动程序的依赖表达式，调度程序会首先加载那些系统相关度较低的驱动程序，根据安全方案的设计，需要在加载 UEFI 文件系统协议

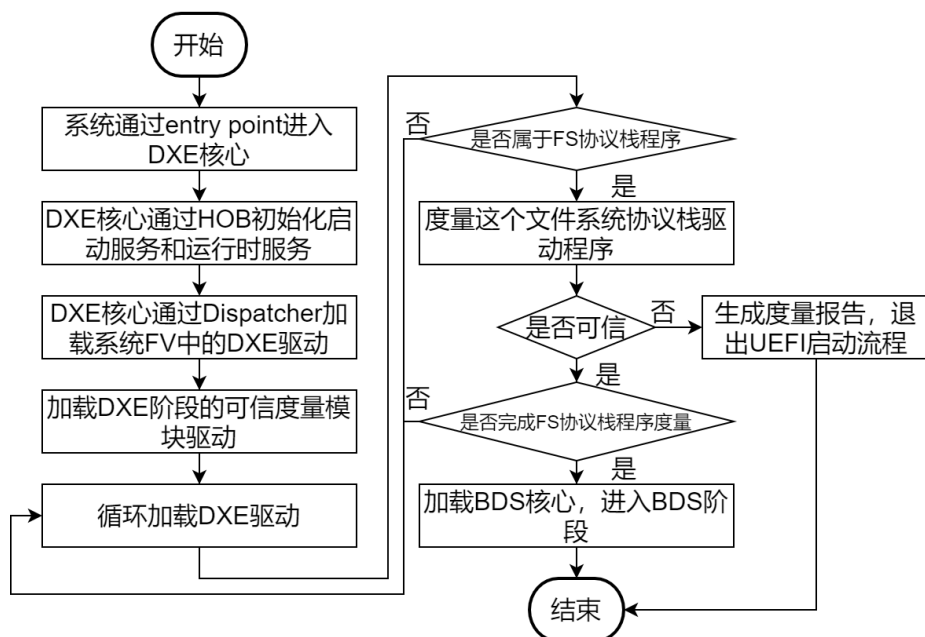


图 4-9 DXE 阶段流程图

Figure 4-9 Execution process of DXE

栈的四个相关的驱动之前加载 DXE 阶段的可信度量驱动。由于 PEI 和 DXE 两个阶段使用不同的系统服务，而可信度量模块中需要分别使用到 PEI 系统服务和 DXE 系统服务，因此为 PEI 和 DXE 阶段分别设计两个可信度量功能的相关模块或驱动；之后在加载四个文件系统协议栈的驱动程序前分别对其进行可信度量；根据四个驱动的可信度量结果，若都可信，则继续加载后续驱动并将系统控制权最终交给 BDS core，否则，停止 UEFI 系统启动。

#### 4.2.4 BDS 阶段设计

BDS core 通过 BDS 架构协议定位和加载在启动准备服务环境中执行的各种各样的应用。这些应用可能表示一个传统的 OS 启动装载程序，或者表示可代替最终的 OS 运行或在加载最终的 OS 之前运行的扩展服务。这样的扩展启动准备服务可能包括安装配置、扩展诊断、闪存更新支持、OEM 服务，或者 OS 启动代码。

如图 4-10 所示，系统进入 BDS 阶段执行 BDS core 代码，并根据 BDS 加载的硬盘 ESP 分区中的文件调用 DXE 阶段加载的可信度量服务型驱动对该文件进行可信度量，若结果可信，则通过 UEFI 文件系统协议栈加载此文件到 UEFI 系统的永久内存中，并交给此文件系统的控制权；否则停止该文件的加载，退出 UEFI 启动。

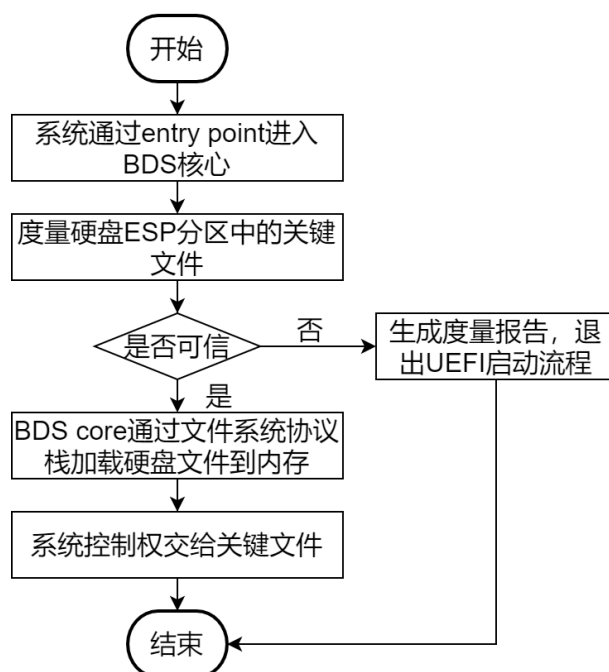


图 4-10 BDS 阶段流程图

Figure 4-10 Execution process of BDS

#### 4.2.5 DXE 阶段驱动依赖关系

依赖关系是 UEFI 中一个调度程序相关的事项，存在于 PEI 调度程序和 DXE 调度程序中。由于 PEI 阶段的安全方案设计是通过手动控制修改调度程序代码，来实现最后加载可信度量的 PEIM，因此不存在依赖关系的建立。而 DXE 阶段驱动众多，需要度量的文件系统协议栈驱动程序和 BDS core 代码项目也较多，因此需要借助 UEFI 提供的依赖关系表达式来实现可信度量模块驱动在四个文件系统驱动前进行加载，这样才能在加载四个文件系统驱动时调用可信度量驱动中的 IPMI 协议来实现系统加载驱动前进行度量。

依赖关系表达式存储于图 4-1 中对应的 dependency section 中，存储在依赖项部分中的依赖项表达式设计得较小，以节省空间。另外，它们的设计理念为简单、快速，以减少执行开销。通过设计一个小的基于堆栈的指令集来对依赖项表达式进行编码，可以实现这两个目标。DXE 分派器必须为此指令集实现一个解释器，以便评估依赖项表达式。

表 4-3 依赖表达式命令摘要  
Table 4-3 Dependency Expression Opcode Summary

执行代码	命令	描述
0x00	BEFORE <File Name GUID>	在提供的 GUID 表示的驱动前加载此驱动
0x01	AFTER <File Name GUID>	在提供的 GUID 表示的驱动后加载此驱动
0x02	PUSH <Protocol GUID>	向栈中添加一个布尔类型的变量，如果这个 GUID 表示的协议在句柄数据库中，则添加 TRUE，否则添加 FALSE
0x03	AND	与运算，用于判断栈中判断结果
0x04	OR	或运算，用于判断栈中判断结果
0x05	NOT	出栈一个布尔值，通过非运算计算后，将结果添加进栈
0x06	TRUE	表示总判断结果为真
0x07	FALSE	表示总判断结果为假
0x08	END	依赖表达式的结束标志符
0x09	SOR	表示驱动还未加载，等同于 NOP 效果

如表 4-3 所示，是 DXE 阶段的依赖表达式命令集，PEI 阶段的依赖表达式除了没有 BEFORE 和 AFTER 之外，其他命令与 DXE 阶段相同。UEFI 系统中，通过 INF 文件的 [Protocols] 字段下添加协议名称，可以通过 EDKII 的 BaseTool 目录下的依赖表达式相关的词法语法分析器，形成对应的依赖表达式，并通过编译存储与驱动文件的 dependency section 中。除了通过这种方式，还可以通过手写依赖文件的方式确定驱动的加载优先级。通过编写后缀名为 dxs 的驱动依赖文件，也可以达到将此信息写入 FV 中驱动的 dependency section 的目的。

在 DXE 调度器解析 dependency section 时，会根据每一个 GUID 向依赖栈中压入句柄数据库中此协议或驱动的查询结果，然后发现 END 结束符时，在通过 AND 这样的各个结果判断关系，依次将布尔值出栈，最后通过最终的依赖表达式确定此驱动是否在此轮进行调度加载。FV 固件卷中的依赖表达式事例如图 4-11。

如图 4-11 所示，FV 中有一个 Priori File 用来描述系统规定的强类型优先级关系，即这个优先级文件中规定的加载顺序必须严格按照规定执行。而依赖表达式则属于弱类型优先关系，原因是 UEFI 通过一个队列来控制每一轮通过调度程序加载驱动，依赖表达式只能保证若此轮此驱动所依赖的协议或其他驱动已加载完成，则保证此驱动在此轮加载，而不能保证的是此驱动的精确加载顺序。如图中所示，Security、Runtime 和 Variable 驱动严格按照顺序加载，而依赖于 CPUIO 协议的 Metronome 和 Reset 驱动则只要系统完成 CPUIO 协议的加载，则加载这两个驱动，但不能保证这两个驱动加载的先后顺序，因此名为弱类型。

固件卷	
优先级文件	
Security Driver	
Runtime Driver	
Variable Driver	
Runtime Driver	Depex=TRUE END 生产出:EFI_RUNTIME_ARCH_PROTOCOL
CPU Driver	Depex=TRUE END 生产出:EFI_CPU_ARCH_PROTOCOL,EFI_CPU_IO_PROTOCOL
Timer Driver	Depex=EFI_CPU_IO_PROTOCOL AND EFI_CPU_ARCH_PROTOCOL END 生产出:EFI_TIMER_ARCH_PROTOCOL
Variable Driver	Depex=TRUE END 生产出:EFI_VARIABLE_ARCH_PROTOCOL
Metronome Driver	Depex=EFI_CPU_IO_PROTOCOL END 生产出:EFI_METRONOME_ARCH_PROTOCOL
Reset Driver	Depex=EFI_CPU_IO_PROTOCOL END 生产出:EFI_RESET_ARCH_PROTOCOL
DXE核心	
BDS Driver	Depex=TRUE END 生产出:EFI_BDS_ARCH_PROTOCOL
Security Driver	Depex=TRUE END 生产出:EFI_SECURITY_ARCH_PROTOCOL

图 4-11 固件卷依赖表达式存储

Figure 4-11 Dependency Expression In Firmware Volume

此安全方案中驱动顺序的具体调整，就是通过在四个文件系统协议栈驱动程序的文件中的 [Depex] 字段下添加可信度量驱动所生成的 IPMI 协议和 SHA1 计算协议即可。也可通过编写 dxs 依赖文件来对可信度量驱动和四个文件系统协议栈驱动程序的加载顺序进行自定义，实现方法如下。

DEPENDENCY\_START

EFI\_SHA1\_PROTOCOL\_GUID AND

EFI\_IPMI\_INTERFACE\_PROTOCOL\_GUID

DEPENDENCY\_END

最后在四个 UEFI 文件系统协议栈驱动程序的 INF 文件的 [nmake.common] 字段下，引用依赖文件 DPX\_SOURCE= FileSysStack.dxs，从而将 dxs 文件添加进驱动文件的编译过程。

### 4.3 本章小结

本章节介绍了此安全方案的各个模块的详细设计即关键数据结构和协议的代码实现，其主要包括 DXE 和 BDS 阶段起作用的可信度量驱动，也包括其中的驱动文件度量模块、硬盘文件度量模块、hash 算法模块和 BMC 通信模块，也有负责缓存硬盘文件和 FV 中驱动文件的中间层模块。然后介绍了 PEI 和 DXE 阶段分成两个可信度量驱动实现的理由，以及 SEC 阶段、PEI 阶段的详细设计思路，最后是 DXE 阶段依赖表达式的利用方法，以确保安全方案的实施。



## 第5章 硬盘文件安全加载方案实现及测试

有了三四章节的理论和代码基础，本章将在此基础上，逐步进行实验验证分析。验证内容包括，DXE 阶段驱动程序度量功能的验证、BDS 阶段硬盘文件度量功能的验证、UEFI 系统中 BMC 驱动程序的连通性验证、通过 EDKII 编译生成的 fd 固件文件格式的验证以及 DXE 阶段修改驱动程序加载顺序的验证。下面将先对实验环境进行介绍，以及选用的原因。

### 5.1 实验环境

#### 5.1.1 申威 6A 服务器真机环境

实验的真机环境是可信固件实验室提供的申威 6A 型服务器，实验环境包括了固件的编译环境以及服务器中 BMC 提供的 fd 固件文件烧写环境，具体信息如下：

- (1) 固件编译系统环境：Ubuntu 10.04 版本，并配置 uuid-dev-2.17 包
- (2) 固件编译工具：原版 gcc4.4.3，和申威定制的 swgcc-4.5.3-交叉编译器
- (3) EDKII 信息：带有申威定制包 SenweiPkg 的 EDKIIR13995 版本
- (4) BMC 烧写环境：Ubuntu 16.04 版本和 Firefox 浏览器

Ubuntu 10.04 版本的选择主要原因在于它提供的 C 语言库函数版本问题，由于 swgcc-4.5.3-交叉编译器的特殊性，需要特定库版本的支持，在以往实验中这一点经过验证。由于真机环境上运行的固件是经过申威 ALPHA64 架构处理器指令集定制的，因此根据编译环境的特殊性做一些编译方法的介绍。交叉编译器的设置是通过 linux 系统软链接的方式实现的：

```
ln -s sw_64sw2-unknown-linux-gnu-ar swar
ln -s sw_64sw2-unknown-linux-gnu-as swas
ln -s sw_64sw2-unknown-linux-gnu-gcc swgcc
ln -s sw_64sw2-unknown-linux-gnu-ld swld
ln -s sw_64sw2-unknown-linux-gnu-objcopy swobjcopy
```

通过以上命令分别将申威定制版本的 gcc 编译器链接到官方版本 gcc 的调用名称上面，来实现编译命令的调用。

图 5-1 所示的是查看软链接的修改效果。接下来是交叉编译申威固件程序的过程，首先需要使用原版 gcc4.4.3 编译器进行 edksetup 程序的初始化工作，这个程序的目的在于为正在打开着的终端窗口设置好通过 build 编译时一切所需的环

```
lrwxrwxrwx 1 root root 29 2018-08-17 04:55 swar -> sw_64sw2-unknown-linux-gn
u-ar*
lrwxrwxrwx 1 root root 29 2018-08-17 04:55 swas -> sw_64sw2-unknown-linux-gn
u-as*
lrwxrwxrwx 1 root root 30 2018-08-17 04:55 swgcc -> sw_64sw2-unknown-linux-g
nu-gcc*
lrwxrwxrwx 1 root root 29 2018-08-17 04:55 swld -> sw_64sw2-unknown-linux-gn
u-lld*
lrwxrwxrwx 1 root root 34 2018-08-17 04:55 swobjcopy -> sw_64sw2-unknown-lin
ux-gnu-objcopy*
root@ubuntu1004:/ctools/cross-tools/bin#
```

图 5-1 编译器软连接

Figure 5-1 Compiler Soft Link

境变量及一些通过 C 语言根据 UEFI 规范自动生成的代码内容。

```
root@ubuntu1004:/ctools/cross-tools/bin# gcc -v
Using built-in specs.
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 4.4.3-4ubuntu5.1' --wi
th-bugurl=file:///usr/share/doc/gcc-4.4/README.Bugs --enable-languages=c,c++,fortran,
objc,obj-c++ --prefix=/usr --enable-shared --enable-multiarch --enable-linker-build-i
d --with-system-zlib --libexecdir=/usr/lib --without-included-gettext --enable-threads
s=posix --with-gxx-include-dir=/usr/include/c++/4.4 --program-suffix=-4.4 --enable-nl
s --enable-clocale=gnu --enable-libstdc++-debug --enable-plugin --enable-objc-gc --di
sable-werror --with-arch=32=i486 --with-tune=generic --enable-checking=release --buil
d=x86_64-linux-gnu --host=x86_64-linux-gnu --target=x86_64-linux-gnu
Thread model: posix
gcc version 4.4.3 (Ubuntu 4.4.3-4ubuntu5.1)
root@ubuntu1004:/ctools/cross-tools/bin# export PATH=/ctools/cross-tools/bin/:$PATH
root@ubuntu1004:/ctools/cross-tools/bin# export LD_LIBRARY_PATH=/ctools/cross-tools/l
ib:$LD_LIBRARY_PATH
root@ubuntu1004:/ctools/cross-tools/bin# gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/ctools/cross-tools/bin/../libexec/gcc/sw_64sw2-unknown-linux-gnu
/4.5.3/lto-wrapper
Target: sw_64sw2-unknown-linux-gnu
Thread model: posix
gcc version 4.5.3 (GCC)
root@ubuntu1004:/ctools/cross-tools/bin#
```

图 5-2 GCC 版本切换过程

Figure 5-2 GCC version switching process

如图 5-2 所示，是通过 export 改变环境变量的命令，将我们设置好的 swgcc 交叉编译器软链接目录添加到两个系统环境变量中的过程，通过这样的方式，就可以实现用 gcc 4.4.3 版本进行 edksetup 程序的运行，并用 swgcc 交叉编译器进行平台 dsc 文件的构建编译过程。具体编译过程，采用 linux SHELL 脚本的方式进行使用，如图下代码所示。

```
cd dev
cd uefibasic
. edksetup.sh
export PATH=/ctools/cross-tools/bin/: $PATH
export LD_LIBRARY_PATH=/ctools/cross-tools/lib:$LD_LIBRARY_PATH
cd SenweiPkg
. build.sh
```

其中的实现为开发的脚本语言编译程序，图 5-4 为最终的固件编译结果。

```

Generate Region at Offset 0x1F8000
  Region Size = 0x8000
  Region Name = None

GUID cross reference file can be found at /home/j/dev/uefibasic/Build/SenweiPkg
/RELEASE_MIPSELFGCC/FV/Guid.xref

FV Space Information
FVMAIN_COMPACT [69%Full] 2031616 total, 1404632 used, 626984 free
FVMAIN [99%Full] 9350656 total, 9350240 used, 416 free

- Done -
06:50:28, Feb.08 2021 [00:39]

Te Section Found
Offset of entry point: 0x240
BranchInstruction : 0x1000008f
Offset : 0x0000008f

return status : 0

[INFORMATION]
Based on commit:
Version: -

stat: No such file or directory
root@ubuntu1004:/home/j#

```

图 5-3 交叉编译结果

Figure 5-3 Cross compile result

## 5.2 驱动度量功能实现与测试

本文中在 DXE 阶段加载的可信度量驱动程序，主要用来度量四个 UEFI 文件系统协议栈驱动程序，并向 BMC 发送度量日志。基于以上功能对可信度量驱动程序进行功能开发与测试。

### 5.2.1 功能实现

驱动度量函数首先根据传进来的内存中的驱动数据结构表示，判别驱动类型，由于四个文件系统协议栈驱动都属于 `EFI_IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER` 类型<sup>[5]</sup>，因此可以通过这个条件进行初步判断。在确定驱动类型正确之后，进行 GUID 的判别，其中的四个特定驱动程序 GUID 通过 INF 文件获取的方式写死在代码中。若确定此正在加载的驱动是四个驱动之一，则通过可信度量值计算模块中的 SHA1 计算协议进行度量，并通过 GUID 在 BMC 中获取到相应的基准值，这些功能都通过 `BmcCheck` 函数实现。

`EFI_STATUS`

`DriverMeasurement(`

`IN LOADED_IMAGE_PRIVATE_DATA *Image`

`) {`

`switch (Image->ImageContext.ImageType) {`

`...`

```

case EFI_IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER:
    if (CompareGuid(Image->ImageContext->Name, DiskIoDxeName) ||
        CompareGuid(Image->ImageContext->Name, PartitionDxeName)
        ||
        CompareGuid(Image->ImageContext->Name,
            AtaAtapiPassThruDxeName) ||
        CompareGuid(Image->ImageContext->Name, FatFileSysName))
        BmcCheck(Image->ImageContext);
    ...
}

```

GUID 匹配具体通过编写的 CompareGuid 函数比较 128 位的 GUID 值，通过把 128 位的 GUID 数据结构通过强制类型转换 (CONST UINT64\*) Guid 转换为 UEFI 系统中可以直接通过 == 符号进行比较的 UINT64 数据类型的指针，指针中包含了两个元素，分别是 GUID 的高 64 位和底 64 位，然后通过表达式 (LowPartOfGuid1 == LowPartOfGuid2 && HighPartOfGuid1 == HighPartOfGuid2) 来返回比较结果的 BOOLEAN 类型值。

UINTN

EFIAPI

```

SerialPortWrite (
    IN UINT8    *Buffer,
    IN UINT32    Size
)
{
    PRINTCONFIG* Pconfig;
    UINT8 cpuid;
    cpuid = hard_smp_processor_id ();
    cpuid = cpuid & (CORENUM_PER_CG - 1);
    Pconfig = PRINTK_BUF_CONFIG;
    Pconfig += cpuid;
    ...
    CopyMem (Pconfig->Start + Pconfig->Index, Buffer, Size);
    Pconfig->Index += Size;
    ...
}

```

上述代码描述的是度量日志写入功能中向串口写入信息的函数。其中输入参数 `Buffer` 是使用系统调用 `AsciiVSPrint` 函数后，通过 `DEBUG` 信息生成的 ASCII 码格式的字符串，用 8 位的无符号整数的指针表示，其中每 8 位正好对应一个 ASCII 码，表示一个英文字符。`PRINTCONFIG` 是一个用来在内存中表示打印信息参数的数据结构，用于记录每次 `Buffer` 需要写入的目的地址，其中 `Start` 字段是 `PRINTCONFIG` 在内存中的起始地址，`Index` 字段用于在每次写入以此 `Buffer` 数据后，自加一以达到下次调用串口写入函数时，写入到上次数据内容的后面。`PRINTCONFIG` 的地址是通过计算得到的，`PRINTK_BUF_CONFIG` 是一个系统写死的数值，内容为 `0xffffc0000870000UL`，用于记录 `PRINTCONFIG` 结构在 IO 映射地址的位置，此内容是在系统 `DXE` 阶段初始化过程中确定的。

### 5.2.2 测试目的

该可信度量功能对 `DXE` 阶段特定驱动的度量方法存在普遍性，给出了根据现有 `UEFI` 驱动加载的设计理念，自定义度量驱动内容的方法。该过程的正确性直接影响了 `BDS` 阶段加载硬盘文件的安全可信性。

### 5.2.3 测试步骤

(1) 首先编写具有上述五个模块的 `DXE` 阶段可信度量驱动程序，并将其模块 `INF` 文件通过引用的方式添加到将要编译的申威平台描述文件 `dsc` 文件中。

(2) 然后通过本章第一节中的方法，对申威 `UEFI` 组件进行交叉编译。

(3) 通过 `BMC` 芯片提供的网卡端口服务程序，用单独的客户端主机访问服务器上的网卡端口，此过程设置客户端静态 IP，设置成与 `BMC` 提供的服务 IP 同一网段内，并烧写 `UEFI BIOS` 固件文件到服务器的闪存芯片中。

(4) 服务器上电并按电源按钮，进入 `UEFI BIOS` 启动流程，过程中将根据 `DXE` 阶段依赖，加载并度量四个特定驱动程序。

(5) 通过与步骤 (3) 同样的方法，通过网卡用客户端访问 `BMC`，此时客户端需要安装 `default-jdk` 环境，并安装了 `Maintance` 维护工具。通过 `lazyrrk 30 0` 命令获取 `BMC` 日志信息，其中包括驱动的度量日志。

### 5.2.4 测试过程

首先将经过交叉编译生成的 `SENWEI.fd` 文件通过 `BMC` 功能烧写如 `BIOS` 闪存中，如图 5-6 所示。将通过通过 2400 的控制器访问并烧写提供的 `BIOS` 固件文件，烧写过程需要开机烧写，否则烧写失败<sup>[32]</sup>。



图 5-4 BMC 烧写固件界面  
Figure 5-4 BMC firmware interface

然后就是驱动度量日志的查看过程，通过安装的维护工具访问 BMC 特定端口，并通过 linux 系统中的重定向输出功能将读取到的日志信息，写入指定的文件中。图 5-7 所示为四个特定驱动的度量日志。

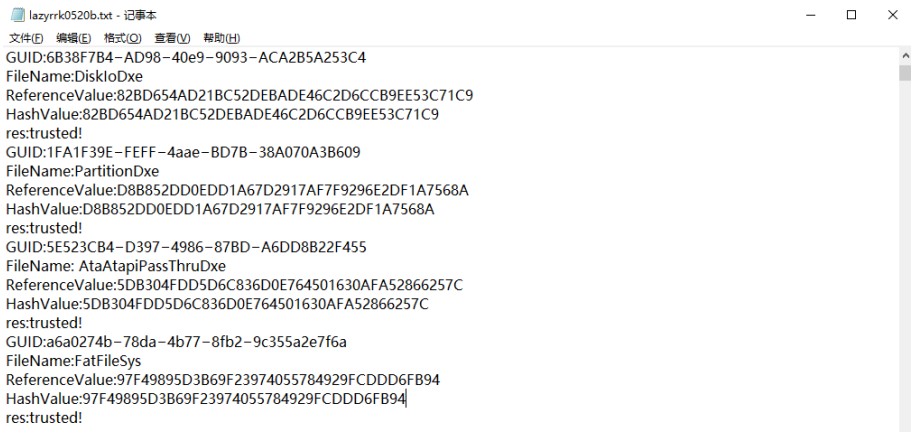


图 5-5 驱动度量日志  
Figure 5-5 Driver measurement log

## 5.3 BMC 驱动连通性实现与测试

### 5.3.1 功能实现

EFI\_STATUS

EFI\_API

IpmiSendCommand (

```
IN  EFI_IPMI_INTERFACE_PROTOCOL *This,
IN  UINT8                        Command,
IN  UINT8                        *CommandData,
IN  UINT8                        CommandDataSize
```

```

) {
    ...
    Status = KcsWaitforIbfClear( Instance );
    WriteKcsCommand(Instance,KCS_WRITE_START);
    Status = KcsCheckStatus(Instance,&KcsStatus);
    if (KcsStatus.Bits.State !=KcsWriteState){
        return KcsErrorExit( Instance );
    }
    KcsClearObf(Instance);
    for(i=0; i<DataSize-1; i++) {
        WriteKcsData(Instance,Data[i]);
        Status = KcsCheckStatus(Instance,&KcsStatus);
        if (EFI_ERROR(Status)) {
            return EFI_DEVICE_ERROR;
        }
        if (KcsStatus.Bits.State !=KcsWriteState){
            return KcsErrorExit( Instance );
        }
        KcsClearObf(Instance);
    }
    WriteKcsCommand(Instance,KCS_WRITE_END);
    Status = KcsCheckStatus(Instance,&KcsStatus);
    if (KcsStatus.Bits.State !=KcsWriteState){
        return KcsErrorExit( Instance );
    }
    KcsClearObf(Instance);
    WriteKcsCommand(Instance,Data[i]);
    ...
}

```

如上代码所示，通过解析发送命令函数来说明与 BMC 系统的通信方式。发送函数接收主要的 8 位标识的命令信息，和命令数据大小，然后根据 IPMI 协议提供的 KCS 数据发送状态流，依次设置 KCS 命令和查询相关寄存器，配合数据接收寄存器的数据写入过程<sup>[50]</sup>。实现的具体流程参照图 4-5。

### 5.3.2 测试目的

在可信度量驱动程序的开发过程中，需要一个从 BMC 系统取出基准值的过程，通过单独测试 UEFI BIOS 中 BMC 驱动程序的连通性有助于单独功能地调试可信度量驱动，保证 BMC 驱动的可用性。

### 5.3.3 测试步骤

(1) 编写 DXE 阶段的 BMC 驱动程序，编写 BMC 的 BIOS SHELL 命令，可输入 IPMI 指令和内容，并获取到 BMC 返回的结果。将 BMC 驱动的 INF 文件添加到申威平台描述文件 dsc 文件中。

(2) 然后通过本章第一节中的方法，对申威 UEFI 组件进行交叉编译。

(3) 通过 BMC 提供的烧写功能烧写 BIOS 固件。

(4) 服务器上电并按下开机按钮，按 F12 进入到 UEFI 界面并通过 SHELL 启动方式进入到 UEFI SHELL 环境。

(5) 输入开发好的 BMC 命令，并从 SHELL 界面查看结果。

### 5.3.4 测试过程

虽然 SHELL 作为 UEFI BIOS 系统中的一个上层用户程序，但作为 BIOS 来说，并不区分内核和用户的内存访问区域，因此 SHELL 程序依然可以访问到 UEFI 初始化的一切系统资源，为驱动的测试提供了基础。

```

UEFI Interactive Shell v2.0. UEFI v2.31 (EDK II - Sunway, 0x00010000). Revision 1.02
Mapping table
FS0: Alias(s):HD29a0a1::BLK1:
PciRoot(0x1)/Pci(0x0,0x0)/Pci(0x1,0x0)/Pci(0x0,0x0)/Pci(0x4,0x0)/Pci(0x0,0x0)/Pci(0x0,0x0)/Pci(0xB,0x0)/Sata(0x0,0x0,0x0)/HD(1,GPT,DCC8CE6B-8A57-40E0-8420-F85E7623073C,0x800,0x79800)
BLK0: Alias(s):
PciRoot(0x1)/Pci(0x0,0x0)/Pci(0x1,0x0)/Pci(0x0,0x0)/Pci(0x4,0x0)/Pci(0x0,0x0)/Pci(0x0,0x0)/Pci(0xB,0x0)/Sata(0x0,0x0,0x0)
BLK2: Alias(s):
PciRoot(0x1)/Pci(0x0,0x0)/Pci(0x1,0x0)/Pci(0x0,0x0)/Pci(0x4,0x0)/Pci(0x0,0x0)/Pci(0x0,0x0)/Pci(0xB,0x0)/Sata(0x0,0x0,0x0)/HD(2,GPT,05E8EDB4-5C8B-4764-BEFB-F75FE0104E0D,0x7A000,0xD98AC000)
BLK3: Alias(s):
PciRoot(0x1)/Pci(0x0,0x0)/Pci(0x1,0x0)/Pci(0x0,0x0)/Pci(0x4,0x0)/Pci(0x0,0x0)/Pci(0x0,0x0)/Pci(0xB,0x0)/Sata(0x0,0x0,0x0)/HD(3,GPT,F4406444-B87E-4503-BE59-D1E699F6A730,0xD9926000,0xF4E2800)
Press ESC in 0 seconds to skip startup.nsh or any other key to continue.
Shell> bmc_kcs 0x3e 0x01 0x12 0x34
enter shellCommandBmc function.
send data: 0x12 0x34.
recv data: 0x47 0x61.
bmc command finish.
Shell> _

```

图 5-6 BMC 命令运行结果

Figure 5-6 BMC command results

如图 5-6 所示，是通过名为 bmc\_kcs 的 SHELL 命令发送并接收数据的过程。



## 5.4 依赖表达式验证

### 5.4.1 测试目的

依赖表达式是 PEI 和 DXE 驱动加载阶段的加载顺序的主要依据，他与优先级文件对应，属于弱类型。通过验证依赖表达式在固件文件中的存储以及在驱动加载过程中如何控制加载顺序，可确保可信度量驱动在需被度量的特定驱动前加载，保证度量功能的可用性。

### 5.4.2 测试步骤

- (1) 在 DXE core 代码的调度器程序中添加向 SHELL 输出的 DEBUG 函数。
- (2) 通过配置 VS2008 来对 UEFI BIOS 进行编译，得到 NT32.fd 文件。
- (3) 通过 UEFITool 这个 fd 文件解析程序，对 NT32.fd 进行解析。
- (4) 运行模拟程序的入口点 exe 可执行文件，根据 SHELL 输出的 DEBUG 信息得到依赖表达式解析过程。

### 5.4.3 测试过程

在经过 VS2008 编译器编译后生成 Windows 平台的 UEFI 模拟环境固件文件，通过 UEFITool 工具查看到的文件信息如下。

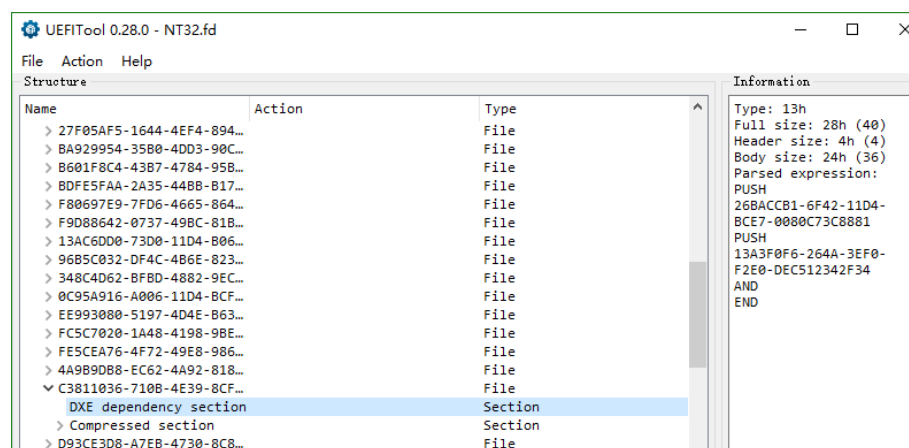
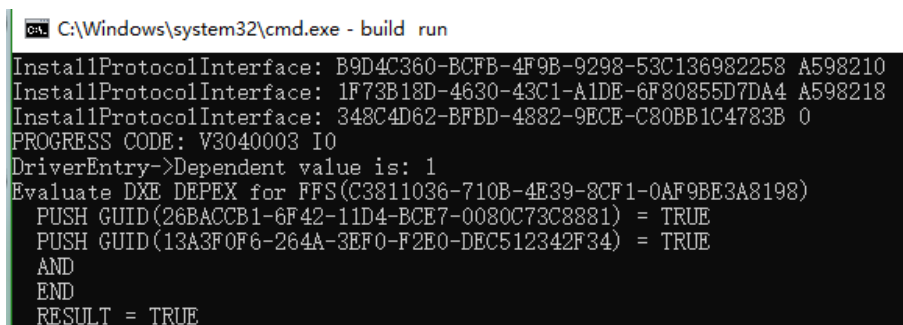


图 5-7 固件文件依赖区域内容

Figure 5-7 FD file dependency section content

图 5-8 所示的是 DXE 阶段的 TimerDxe 驱动在 FV 格式的固件卷中的 dependency section 中的内容，两个 GUID 分别代表两个协议，他们对应的名称是 gEfiCpuArchProtocolGuid 和 gEfiPcdProtocolGuid。下面是运行过程中 TimerDxe 驱动对应的加载过程日志信息显示。



```
C:\Windows\system32\cmd.exe - build run
InstallProtocolInterface: B9D4C360-BCFB-4F9B-9298-53C136982258 A598210
InstallProtocolInterface: 1F73B18D-4630-43C1-A1DE-6F80855D7DA4 A598218
InstallProtocolInterface: 348C4D62-BFBD-4882-9ECE-C80BB1C4783B 0
PROGRESS CODE: V3040003 I0
DriverEntry->Dependent value is: 1
Evaluate DXE DEPEX for FFS(C3811036-710B-4E39-8CF1-0AF9BE3A8198)
  PUSH GUID(26BACCB1-6F42-11D4-BCE7-0080C73C8881) = TRUE
  PUSH GUID(13A3F0F6-264A-3EF0-F2E0-DEC512342F34) = TRUE
  AND
  END
  RESULT = TRUE
```

图 5-8 驱动加载过程中的日志信息

Figure 5-8 Log information during driver loading

图 5-9 显示的是在通过 SecMain.exe 入口点或命令 build run 执行后在 windows 终端里打印显示的 BIOS 运行过程中的日志信息，其中可以看到入栈两个协议 GUID 的过程，并可以看出判断的最终结果为 TRUE，在这段 log 的下方可看到显示了驱动加载的日志信息。

## 5.5 本章小结

通过三个实验过程的结果可以看出，可信度量驱动具备着从 BMC 取基准值、生成度量日志和根据依赖表达式定制驱动加载顺序的功能，这些基本的功能也保证了安全方案的可实现性。

## 结 论

UEFI 文件系统协议栈作为 UEFI 系统环境中与硬盘设备硬件之间的桥梁，是一个保证硬盘 ESP 分区中数据信息不被从固件层面攻击的关键环节。UEFI 文件系统协议栈在负责 BIOS 系统与硬盘设备间进行数据交互的同时，也作为一个 UEFI 系统启动中 DXE 阶段的一些驱动程序的形式进行向系统中的集成与加载。由于硬盘 ESP 分区中数据文件的重要性，以及近年来越来越多的硬件攻击手段的出现，不仅有针对硬盘设备的攻击也存在针对固件芯片的攻击，基于这种情况，本为设计了基于 UEFI 的硬盘文件安全加载的策略，在一定程度上保证了计算机通过 UEFI BIOS 和硬盘设备启动过程中的安全性。

本文的主要工作内容总结归纳如下：

1. 针对 UEFI BIOS 系统加载硬盘文件过程中存在的安全威胁，分析目前基于 UEFI 系统的服务器在通过硬盘 ESP 分区中操作系统引导文件启动的加载过程中的安全漏洞，即硬盘设备和固件芯片设备两个可被攻击的对象结合可信计算理论提出基于 UEFI 的硬盘设备文件可信加载系统的总体框架，并使用的底层可信平台 BMC 及基板管理控制器，实现 UEFI BIOS 中 BMC 驱动程序。

2. 针对 UEFI BIOS 系统加载 UEFI BIOS 存储闪存设备中的驱动文件过程中存在的安全问题，设计提出 UEFI BIOS 启动阶段所需度量的 UEFI 文件系统协议栈驱动程序，并实现度量模块功能。

3. 针对安全方案中的可信度量功能，设计并实现了通过 BMC 的日志存储功能，并完成确保驱动按顺序加载的依赖表达式编写。由于需要在被度量驱动前先加载可信度量驱动，因此存在 DXE 阶段调度程序加载驱动顺序的问题，通过 UEFI 中的依赖表达式来完成这一要求，保证度量过程的可实施性。

4. 根据本系统安全方案的设计，在申威平台中对驱动度量模块、日志生成功能和驱动加载顺序修改功能进行实现和测试，以保证功能开发过程的有效性和安全方案的可实施性。

综上所述，本方案解决了 UEFI 系统在加载硬盘文件时容易受到硬盘设备和固件芯片两方面攻击的问题，并给出了 UEFI 启动阶段中度量的具体内容和具体方式，保证了 UEFI 环境中加载硬盘文件的安全可信。但本方案也存在着很多不足之处和需要进一步改进的地方，在于：

1. 目前可信度量的过程在 UEFI BIOS 环境中完成，但根据可信平台模块的基本功能，度量的过程应由平台模块来进行，也就是可以改进为通过 BIOS 发送在固件和硬盘中加载的特定驱动和文件的数据内容，通过 BMC 驱动发送给

BMC 中的可信平台模块，由 BMC 系统进行度量值的计算和日志生成过程，并将度量结果返回给 BIOS，BIOS 通过结果来判断下一步的执行安排。

2. 目前的度量日志的内容只能通过操作系统中的 BMC 维护程序来获取，可进一步在 UEFI BIOS 环境中编写获取 BMC 系统中存放的日志内容的功能，可从 UEFI SHELL 环境中获取日志信息，使功能更加完善。

## 参考文献

- [1] ROTHMAN M, ZIMMER V, LEWIS T. Harnessing the uefi shell: Moving the platform beyond dos[M]. [S.l.]: Walter de Gruyter GmbH & Co KG, 2017.
- [2] ZIMMER V, ROTHMAN M, MARISSETTY S. Beyond bios: developing with the unified extensible firmware interface[M]. [S.l.]: Walter de Gruyter GmbH & Co KG, 2017.
- [3] 王凯. 基于 UEFI 的计算机远程可信启动研究与实现 [D]. 北京: 北京工业大学, 2020.
- [4] 安会. 基于 UEFI 的操作系统内核完整性保护方法的研究与实现 [J]. 北京工业大学, 2017: 1-z.
- [5] 孙亮, 陈小春. 基于 UEFI 的固件级硬盘安全保护机制 [J]. 武汉大学学报 (理学版), 2019, 65(2): 223-228.
- [6] WOJTCZUK R, KALLENBERG C. Attacks on uefi security[C]//Proc. 15th Annu. CanSecWest Conf.(CanSecWest). [S.l.: s.n.], 2015.
- [7] PAUL G, IRVINE J. Take control of your pc with uefi secure boot[J]. Linux Journal, 2015 (257): 58-72.
- [8] NICHOLAS M O. System and method for managing and diagnosing a computing device equipped with unified extensible firmware interface (uefi)-compliant firmware[P]. 2016.
- [9] PUTHILLATHE C, VIDYADHARA S. Out-of-band (oob) real-time inventory and configuration of original equipment manufacturer (oem) devices using advanced configuration and power interface (acpi) and unified extensible firmware interface (uefi) services[P]. 2017.
- [10] WOJTCZUK R, KALLENBERG C. Attacking uefi boot script[C]//31st Chaos Communication Congress (31C3). [S.l.: s.n.], 2014.
- [11] 房强. 基于固件文件系统的 UEFI 安全机制研究 [D]. 成都: 电子科技大学, 2016.
- [12] 段晨辉. UEFI BIOS 安全增强机制及完整性度量的研究 [D]. 北京: 北京工业大学, 2014.
- [13] 陈小春, 孙亮, 张超, 朱立森. 一种基于 UEFI 的可执行程序文件保护系统和方法 [P]. 2014.
- [14] 陈小春, 孙亮, 张超, 朱立森. 基于 UEFI 的硬盘全加密方法及装置 [P]. 2019.
- [15] LEWIS T A. System and method for verifying changes to uefi authenticated variables[P]. 2017.
- [16] KHATRI M P, LIU W, ROSE C E, et al. System and method for managing uefi secure boot certificates[P]. 2015.
- [17] MING W. Analysis and a case study of transparent computing implementation with uefi[J]. International Journal of Cloud Computing, 2012, 1(4): 312-328.
- [18] ZIMMER V, KUMAR M, NATU M, et al. System and method to secure boot both uefi and legacy option rom's with common policy engine[P]. 2014.
- [19] HU Y, LV H. Design of trusted bios in uefi base on usbkey[C]//2011 International Conference on Intelligence Science and Information Engineering. [S.l.]: IEEE, 2011: 164-166.

- [20] BOBZIN J J. Secure boot administration in a unified extensible firmware interface (uefi)-compliant computing device[P]. 2015.
- [21] BRICKER G. Unified extensible firmware interface (uefi) and secure boot: Promise and pitfalls[J]. Journal of Computing Sciences in Colleges, 2013, 29(1): 60-63.
- [22] ZHOU J, XIE Z Y, YU H, et al. Research on domestic computer bios based on uefi[J]. Computer Engineering, 2011: S1.
- [23] 李梓实. 基于 UEFI BIOS 的 EXT2 文件系统驱动的设计与实现 [D]. 北京: 北京工业大学, 2015.
- [24] Platform initialization specification[S]. 1.6 ed. [S.l.]: UEFI Forum, 2017.
- [25] BERLIN K, TESSER G A A, POLLO L F, et al. Unified extensible firmware interface (uefi) driver and protocol[P]. 2017.
- [26] CHAIKEN C L, DOWNUM S A, MARTINEZ R L. Method for processing uefi protocols and system therefor[P]. 2017.
- [27] 吴伟民, 吴辉, 苏庆. UEFI 固件存储系统分析 [J]. 计算机工程与设计, 2017, 4.
- [28] WU W M, LI S Y, HUANG H K, et al. Research of disk operating technique in uefi shell environment[C]//Applied Mechanics and Materials: volume 263. [S.l.]: Trans Tech Publ, 2013: 1888-1891.
- [29] 金步国. GPT 分区详解 [EB/OL]. <http://www.jinbuguo.com/storage/gpt.html>.
- [30] 韦荣. 可信计算平台可信度量机制的应用与研究 [D]. 西安: 西安电子科技大学.
- [31] 付思源. 基于统一可扩展固件接口的恶意代码防范系统研究 [D]. 上海: 上海交通大学.
- [32] 韩民. UEFI 调试工具及调试信息获取的设计与实现 [D]. 北京: 北京工业大学, 2014.
- [33] CHUQUAN B. 计算机那些事 (2)——开机启动过程 [EB/OL]. <http://chuquan.me/2016/12/14/computer-boot-process>.
- [34] LARVOIRE J F. System and method to enable a legacy bios system to boot from a disk that includes efi gpt partitions[P]. 2005.
- [35] J.BACH M. UNIX 操作系统设计 [M]. 陈葆钰, 王旭, 柳纯录, 冯雪山, 译. 北京: 人民邮电出版社, 2019: 48-116.
- [36] 赵炯. Linux 内核完全注释 [M]. 北京: 机械工业出版社, 2004: 76-89.
- [37] 田宇. 一个 64 位操作系统的设计与实现 [M]. 北京: 人民邮电出版社, 2018: 516-554.
- [38] LOVE R. Linux 内核设计与实现 [M]. 陈莉君, 康华, 译. 北京: 机械工业出版社, 2011: 210-233.
- [39] 周艺华, 刘阳, 王冠, 等. 基于 UEFI 固件的攻击检测系统的设计与实现 [J]. Computer Science and Application, 2017, 7: 320.
- [40] 刘东丽. 基于 UEFI 的信任链设计及 TPM 驱动程序实现 [D]. 武汉: 华中科技大学, 2011.
- [41] 王超. 基于国产 BMC 的服务器安全启动技术研究与实现 [D]. 北京: 北京工业大学, 2019.
- [42] 朱东亮. 基于 UEFI 体系的虚拟 TPM 研究 [D]. 哈尔滨: 哈尔滨工程大学, 2010.
- [43] WILKINS R, RICHARDSON B. Uefi secure boot in modern computer security solutions[C]//UEFI Forum. [S.l.: s.n.], 2013.

- [44] JEONG D, LEE S. Forensic signature for tracking storage devices: Analysis of uefi firmware image, disk signature and windows artifacts[J]. Digital Investigation, 2019, 29: 21-27.
- [45] 周洁, 谢智勇, 余涵, 等. 基于 UEFI 的国产计算机平台 BIOS 研究 [J]. 计算机工程, 2011, 1.
- [46] 胡向东, 杨剑新, 朱英. 高性能多核处理器申威 1600[J]. 2015.
- [47] 马骏. 基于不同处理器架构平台的 UEFI 系统的研究与移植 [D]. 北京: 北京工业大学, 2014.
- [48] 王栩浩. 基于 IPMI 的服务器管理系统的实现 [D]. 上海: 东华大学, 2016.
- [49] 何毅平. 基于 ARM 的 BMC 设计与实现 [D]. 武汉: 华中科技大学.
- [50] Intelligent platform management interface specification second generation[S]. v20 ed. [S.l.]: Intel,Hewlett-Packard,NEC,Dell, 2013.
- [51] 韩德强, 马骏, 张强. UEFI 驱动程序的研究与开发 [J]. 电子技术应用, 2014(5): 10-13.





## 攻读硕士学位期间所取得的学术成果

1. 张建标，唐治中，张恒等. 一种基于基础性的 UEFI BIOS 固件系统中驱动程序的保护方法. 发明专利. 专利号/申请号：202110121999.5.



## 致 谢

时光荏苒，三年的研究生生活即将结束，研究生以及全部的在学校的学习生活也要告一段落，十分幸运有机会能够在北工大信息学部进行三年的学习生活，这里的校园环境、同学氛围都是那么美好和值得怀念。在这里我不光学习到了书本上的知识，也认识了在学习和生活中向我树立榜样的导师，也有共同学习奋斗的同学们，让我的研究生生活更加充实，充满活力。在临近毕业之际，我要向曾不断给予我鼓励和帮助的导师、家人和同学致以诚挚的感谢！

首先要向我的研究生导师张建标教授表示诚挚的感谢，在研究生阶段，有幸参加了张老师的底层系统项目，有机会接触到了以前充满未知的底层系统，这也成为了我学习和研究的兴趣所在，通过项目的进行，从一开始的漫然无知，到熟练操作项目中的各个环节，到阅读和学习底层系统的源码并加以更改添加可信功能，我很幸运的学习到了自己所感兴趣的计算机领域，也丰富了自己的知识和实际解决问题的能力。除了学习上对我的帮助外，张老师在项目中也给了我很多的实践学习机会，让我感受到了在社会中学习工作的不易，增加了我很多方面的经验。再次感谢张老师对我的指导和栽培，希望在以后的工作生活中取得更多的进步。感谢信息安全的所有老师们，很荣幸能受到您们的指导，给我的学习道路指引了方向。

同时也要感谢实验室里的所有师兄们，你们在帮助我适应研究生生活的同时，也给予了我学习和生活上的帮助，帮助我分析项目中的问题，找到解决问题的思路和技术方案，帮助我解决了一个又一个项目和学习中的困难。感谢我的同届的实验室同学们，十分珍惜和你们的朝夕共处，一起的努力学习让我们有了更深厚的感情，让研究生生活变得更加趣味横生、多姿多彩。感谢我的师弟师妹们，能够成为我学习道路上一同进取的伙伴，你们的陪伴让我更加珍惜学生时代的美好时光。

感谢我的室友们对我的照顾和陪伴，在学习和项目遇到迷茫之时，你们总能给我带来很多的慰藉，让我总能够重整心态，继续启航。希望我们的友谊能够继续长存，希望在今后的工作中能有更多的交流沟通机会。感谢所有和我一起学习过生活过的挚友们，你们的陪伴让我在学习的路上不再孤单。

感谢我的父母和家人，在我遇到困难和困苦时给我无限的鼓励，让我知道只有通过改变自己的方式，才能在不光是工作和学习中、也在生活的方方面面取得突破，不断成长。

最后，由衷的感谢参与评阅论文的所有老师们！