

申请上海交通大学工程硕士学位论文

基于 EDK2 的 UEFI 变量检查的研究和实现

论文作者：曾令静

学 号：1130332037

交大导师：黄林鹏

企业导师：蔺杰

院 系：计算机科学与工程系

工程领域：计算机技术

上海交通大学电子信息与电气工程学院

2016 年 05 月

Thesis Submitted to Shanghai Jiao Tong University
for the Degree of Engineering Master

**THE RESEARCH AND IMPLEMENTATION OF
UEFI VARIABLE CHECK BASED ON EDK2**

M.D. Candidate : Lingjing Zeng
Student ID : 1130332037
Supervisor(I) : Linpeng Huang
Supervisor(II) : Jie Lin
Department : Computer Science & Engineering
Speciality : Computer Technology

School of Electronic Information and Electric Engineering
Shanghai Jiao Tong University
Shanghai,P.R.China
May, 2016

上海交通大学

学位论文原创性声明

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

学位论文作者签名：曾令静

日期：2016 年 5 月 13 日

上海交通大学

学位论文版权使用授权书

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

保密 ☐ 在 ____ 年解密后适用本授权书。

本学位论文属于

不保密 ☒。

(请在以上方框内打“√”)

学位论文作者签名：

曾令静

指导教师签名：

苏小明

日期：2016年5月13日

日期：2016年5月13日

上海交通大学硕士学位论文答辩决议书



1130332037

姓 名	曾令静	学号	1130332037	所在学科	计算机技术
指导教师	黄林鹏	答辩日期	2016-05-14	答辩地点	徐汇校区新建楼2027

论文题目 基于EDK2的UEFI变量检查的研究及实现

投票表决结果: 5/5/5 (同意票数/实到委员数/应到委员数) 答辩结论: ☒ 通过 ☐ 未通过
评语和决议:

论文取得了下列研究成果: 1). 分析了现有的保护和恢复机制-Variable Lock 和平台配置 Variable 默认值恢复, 以及 Variable 接口的安全性。针对各种可能的恶意攻击, 提出了新的保护和恢复机制-基于 EDK2 的变量检查。2). 研究和实现了新的保护和恢复机制-基于 EDK2 的 UEFI 变量检查, 借鉴现代操作系统的磁盘空间配额管理方法, 通过属性检查、数据大小检查、语法检查、语义检查、配额管理、错误记录和恢复, 来保证从 UEFI Variable 运行时服务的 SetVariable 来的 Variable 属性、数据和数据大小的合法性, 从而更好地对 NV Variable 及 NV Variable 区域进行保护。

论文框架合理、条理清晰; 叙述清楚, 所提出的方法技术可行、有效。表明作者已具有扎实的基础理论和深入的专门知识, 具备解决工程项目实际问题的能力。答辩时叙述清楚, 条理清晰, 并能正确回答问题。

年 月 日

答 辩 委 员 会 成 员 签 名	职务	姓名	职称	单位	签名
	主席	何援军	教授	上海可计算机技术有限公司	何援军
	委员	张忠能	高级工程师	上海交通大学电子信息与电气工程学院(计算机系)	张忠能
	委员	姚天昉	副教授	上海交通大学电子信息与电气工程学院(计算机系)	姚天昉
	委员	陈玉泉	副教授	上海交通大学电子信息与电气工程学院(计算机系)	陈玉泉
	委员	张同珍	副教授	上海交通大学电子信息与电气工程学院(计算机系)	张同珍
	秘书	杨安 (04300)	工程师	上海交通大学电子信息与电气工程学院(计算机系)	杨安

基于 EDK2 的 UEFI 变量检查的研究和实现

摘 要

UEFI（统一可扩展固件接口）规范定义了操作系统与系统硬件平台固件之间的开放接口，PI（平台初始化）规范建立了固件内部接口架构以及固件和平台硬件间的接口。基于 UEFI 和 PI 规范实现的 UEFI BIOS 一般会存储在一个 NOR 非易失性块存储设备（如 SPI FLASH）中。EDK2（EFI 开发者套件 2）作为一个现代、功能丰富且跨平台的 UEFI BIOS 固件开发环境已经被业界广泛使用。

NV Variable 区域会占用 NOR FLASH 的几个区块，其存储容量有限，但其存储的数据却非常重要，这些数据包括平台配置信息、启动选项设定、用于安全启动的认证变量等等。UEFI Variable 运行时服务定义了各部件访问 Variable 的接口。NV Variable 的非易失性和 UEFI Variable 服务的运行时特性使得存储重要信息的 NV Variable 及 NV Variable 区域易受恶意程序的攻击，因此对 NV Variable 的保护就显得尤为重要。

本文分析了现有的保护和恢复机制-Variable Lock 和平台配置 Variable 默认值恢复，以及 Variable 接口的安全性。针对各种可能的恶意攻击，很多情况下现有的保护和恢复机制并不能被使用，因此我们提出了新的保护和恢复机制-基于 EDK2 的变量检查来对 NV Variable 及 NV Variable 区域进行更好地保护。

本文研究和实现了新的保护和恢复机制-基于 EDK2 的 UEFI 变量检查，它利用 UEFI 和 PI 规范定义的访问 Variable 的各种服务及接口和 Variable 区域及 Variable 本身的特性，借鉴现代操作系统的磁盘空间配额管理方法，实现变量检查库和变量检查协议，通过属性检查、数据大小检查、语法检查、语义检查、配额管理、错误记录和恢复，来保证从 UEFI Variable 运行时服务的 SetVariable 接口传入的 Variable 属性、数据和数据大小的合法性，从而更好地对 NV Variable 及 NV Variable 区域进行保护。利用 NT32 模拟开发平台的验证测试结果反映出新的保护和恢复机制-基于 EDK2 的 UEFI 变量检查能很好地抵御恶意程序的攻击。

关键词 UEFI, EDK2, Variable, 变量检查

THE RESEARCH AND IMPLEMENTATION OF UEFI VARIABLE CHECK BASED ON EDK2

ABSTRACT

The UEFI specification defines the interface between the operating system and the platform firmware, and the PI specification establishes the internal architecture in firmware and interfaces between the hardware platform and firmware. Generally, UEFI BIOS is implemented based on the UEFI and PI specification and stored in a NOR Non-Volatile block storage device (for example, SPI FLASH). As a modern, feature-rich, cross-platform UEFI firmware development environment, EDK2 has been widely used in the industry.

NV Variable region will occupy some blocks of NOR FLASH, it has limited storage size, but stores very important data that includes platform setup information, boot option setting, authenticated variable for secure boot, etc. UEFI Variable runtime services define the interface between components to access variable. Because NV Variable has non-volatility and UEFI Variable services are runtime available, NV Variable and NV Variable region with important information are easy to be attached by malicious programs, so the protection to NV Variable is very important.

This thesis analyzes the existing Variable protection and recovery mechanisms - Variable Lock and the platform setup Variable default value recovery, and the security of Variable interfaces. The existing mechanisms could not be used in many cases of the various malicious attacks, so we need to introduce a new protection and recovery mechanism - UEFI Variable Check based on EDK2 to better protect NV Variable and NV Variable region.

This thesis researches and implements a new protection and recovery mechanism - UEFI Variable Check. The mechanism makes use of UEFI and PI specification defined various Variable services and interfaces and the property of Variable region and Variable itself, learns the disk space quota management of modern operating system, and implements Variable

Check library and Variable Check protocol. Through the property check, data size check, grammar check, semantic check, quota management, error logging and recovery, the mechanism can ensure the legality of variable attributes, data and data size from SetVariable interface in UEFI variable runtime services, thus better protect NV Variable and NV Variable region. The verification test result through NT32 reflects the new protection and recovery mechanism - UEFI Variable Check could well defense the attacks from malicious programs.

Keywords: UEFI, EDK2, Variable, Variable Check

术语表

BIOS: Basic Input/Output System

EFI: Extensible Firmware Interface

UEFI: Unified Extensible Firmware Interface

PI: Platform Initialization

GUID: Global Unique Identifier

HOB: Hand-Off Block

PCD: Platform Configuration Database

FV: Firmware Volume

FVB: Firmware Volume Block

PEI: Pre-EFI Initialization

PPI: PEIM-to-PEIM Interface

DXE: Driver Execution Environment

BDS: Boot Device Selection

HII: Human Interface Infrastructure

SMM: System Management Mode

SMI: System Management Interrupt

SMRAM: System Management Random Access Memory

EDK: EFI Developer Kit

EDK2: EFI Developer Kit 2

NV: Non-Volatile

RT: Runtime

FTW: Fault Tolerant Write

DoS: Denial of Service

目 录

摘 要	I
ABSTRACT	II
术语表	IV
目 录	V
第一章 绪论	1
1.1 论文背景	1
1.2 研究内容和目标	2
1.3 创新点和难点	3
1.4 论文结构	3
第二章 相关规范及开发环境	5
2.1 UEFI 及 PI 规范	5
2.1.1 UEFI 规范	6
2.1.2 PI 规范	8
2.2 UEFI BIOS 的启动流程	10
2.2.1 安全检测 (SEC) 阶段	10
2.2.2 EFI 初始化准备 (PEI) 阶段	11
2.2.3 驱动程序执行环境 (DXE) 阶段	13
2.2.4 启动设备选择 (BDS) 阶段	15
2.2.5 瞬时系统加载 (TSL) 阶段	15
2.2.6 运行时 (RT) 阶段	15
2.2.7 生命周期后 (AL) 阶段	16
2.3 EDK2	16
2.3.1 包 (PACKAGE)	16
2.3.2 库类/库实例 (LIBRARY CLASS/LIBRARY INSTANCE)	18
2.3.3 平台配置数据库 (PCD)	19
2.4 本章小结	20
第三章 VARIABLE 接口及实现的分析	21
3.1 VARIABLE 接口	21
3.1.1 DXE 阶段	22
3.1.2 PEI 阶段	26
3.2 VARIABLE 存储区格式	28
3.3 VARIABLE 模块	31
3.3.1 DXE 阶段	32
3.3.2 PEI 阶段	37
3.4 本章小结	39

第四章 保护和恢复机制-UEFI 变量检查的提出	40
4.1 VARIABLE LOCK	40
4.2 平台配置 VARIABLE 默认值恢复	43
4.3 VARIABLE 接口安全性分析	46
4.4 UEFI 变量检查的提出	47
4.4.1 属性和数据大小检查的提出	48
4.4.2 数据检查的提出	49
4.4.3 配额管理的提出	50
4.5 本章小结	50
第五章 保护和恢复机制-UEFI 变量检查	51
5.1 变量检查模型的建立	51
5.1.1 针对 VARIABLE 的属性和数据大小的篡改	52
5.1.2 针对 VARIABLE 的数据的篡改	54
5.1.3 针对 DOS 攻击	54
5.2 基于 UEFI 的变量检查	55
5.3 基于 HII 的变量检查	56
5.4 基于 PCD 的变量检查	58
5.5 配额管理	60
5.6 验证测试	64
5.7 平台应用	68
5.8 本章小结	69
第六章 总结与展望	70
6.1 全文总结	70
6.2 展望	71
参考文献	72
附录	75
致谢	76
攻读学位期间发表的学术论文	77

第一章 绪论

本章首先会介绍论文研究背景，接着会说明研究内容和目标，然后会说明论文的创新点和难点，最后会介绍论文结构。

1.1 论文背景

UEFI 规范定义了操作系统与系统硬件平台固件之间的开放接口^[1]，PI 规范建立了固件内部接口架构以及固件和平台硬件间的接口^[2]。UEFI BIOS 一般会存储在一个 NOR 非易失性块存储设备（如 SPI FLASH）中。NOR FLASH^[3]是很常见的一种非易失性存储芯片，数据掉电不会丢失。NOR FLASH 支持 Execute On Chip，即程序可以直接在 FLASH 片内执行(这就意味着存储在 NOR FLASH 上的程序不需要被复制到 RAM 就可以直接运行)，这点是 NOR FLASH 和 NAND FLASH 最大的区别，因此 NOR FLASH 很适合作为启动程序的存储介质。NOR FLASH 的读取和 RAM 很类似（只要能够提供数据的地址，数据总线就能够正确的给出数据），但不可以直接进行写或者擦除操作，对 NOR FLASH 的写或者擦除操作需要遵循特定的命令序列，最终由芯片内部的控制单元完成写操作。

FLASH 一般都分为很多个扇区，每个扇区包括一定数量的存储单元，对有些大容量的 FLASH，还分为多个块，每个块包括一定数目的扇区，FLASH 的擦除操作一般都是以扇区、块或是整片 FLASH 为单位的。在对 FLASH 进行写操作的时候，每个 BIT 可以通过编程由 1 变为 0，但不可以由 0 修改为 1。为了保证写操作的正确性，如果其中有需要把某个 BIT 由 0 修改为 1，在执行写操作前都要先执行擦除操作，擦除操作会把 FLASH 的一个扇区、一个块或是整片 FLASH 的值全修改为 0xFF，这样写操作就可以正确完成了。

基于 EDK2 实现的 UEFI BIOS 一般会选择 NOR FLASH 中的一块区域作为 NV Variable 区域，NV Variable 区域会占用 NOR FLASH 的几个区块，其存储容量有限，但其存储的数据却非常重要，这些数据包括平台配置信息、启动选项设定、用于安全启动的认证变量等等。UEFI Variable 运行时服务定义了各

部件（驱动、应用程序以及操作系统等）访问 Variable 的接口。NV Variable 的非易失性和 UEFI Variable 服务的运行时特性使得存储重要信息的 NV Variable 及 NV Variable 区域易受恶意程序的攻击（包括对 NV Variable 的篡改和对 NV Variable 区域的 DoS 攻击^[4]等），结果可能影响整个系统的正常启动、初始化及各种功能的使用等等，因此对 NV Variable 的保护就显得尤为重要。

1.2 研究内容和目标

UEFI 和 PI 规范定义了 Variable 的属性和相关服务及接口，但并没有定义 Variable 的具体存储介质及格式，在大多数情况下，Variable 需要在系统重启之后仍能够保持（及 NV Variable），这就决定了 UEFI BIOS 的实现需要分配一块非易失性存储区域来存储这些 Variable，基于 EDK2 实现的 UEFI BIOS 一般会选择 NOR FLASH 中的一块区域作为 NV Variable 区域。

基于 EDK2 开发设计实现的 DXE Variable 驱动会遵循 UEFI 规范提供 UEFI Variable 运行时服务。为了抵御针对 NV Variable 及 NV Variable 区域的恶意攻击，我们须首先充分了解 UEFI 和 PI 规范定义的访问 Variable 的各种服务及接口和 Variable 区域及 Variable 本身的特性，收集各种可能的恶意攻击手段，最后研究和提出可行的保护 NV Variable 及 NV Variable 区域的方法，进而完善 EDK2 DXE Variable 驱动的实现，以提高系统的可靠性和安全性。

本文会分析现有的保护和恢复机制-Variable Lock 和平台配置 Variable 默认值恢复，以及 Variable 接口的安全性。针对各种可能的恶意攻击，很多情况下现有的保护和恢复机制并不能被使用，因此我们需要提出新的保护和恢复机制-基于 EDK2 的变量检查来对 NV Variable 及 NV Variable 区域进行更好地保护。

本文会研究和实现新的保护和恢复机制-UEFI 变量检查，它利用 UEFI 和 PI 规范定义的访问 Variable 的各种服务及接口和 Variable 区域及 Variable 本身的特性，借鉴现代操作系统的磁盘空间配额管理方法，实现变量检查库和变量检查协议，通过属性检查、数据大小检查、语法检查、语义检查、配额管理、错误记录和恢复，来保证从 UEFI Variable 运行时服务的 SetVariable 接口传入的 Variable 属性、数据和数据大小的合法性。NT32 模拟开发平台可以被利用来对新的保护和恢复机制-UEFI 变量检查进行验证测试。

1.3 创新点和难点

本文的创新点主要在：

- 1) 分析现有的保护和恢复机制-Variable Lock 和平台配置 Variable 默认值恢复，以及 Variable 接口的安全性。针对各种可能的恶意攻击，很多情况下现有的保护和恢复机制并不能被使用，我们研究和实现了新的保护和恢复机制-UEFI 变量检查来保证从 UEFI Variable 运行时服务的 SetVariable 接口传入的 Variable 属性、数据和数据大小的合法性，从而更好地对 NV Variable 及 NV Variable 区域进行保护。
- 2) UEFI 变量检查具有很好的适用性和可扩展性。UEFI 变量检查不会操作具体的 Variable 区域，而只对从 UEFI Variable 运行时服务的 SetVariable 接口传入的 Variable 属性、数据和数据大小进行合法性检查，是纯软的实现，具有很好的适用性。UEFI 变量检查的接口定义采用库和协议，具有很好的可扩展性。

本文的难点主要有：

- 1) 收集各种可能的对 NV Variable 及 NV Variable 区域的恶意攻击手段，为此我们需要充分了解 UEFI 和 PI 规范定义的访问 Variable 的各种服务及接口和 Variable 区域及 Variable 本身的特性。
- 2) UEFI 变量检查模型的建立需要综合考虑各模块及库的接口需求，从而保证各模块及库之前的互操作性和 UEFI 变量检查的可扩展性。

1.4 论文结构

本文共分为六个章节，结构如下：

第一章：绪论，首先介绍了论文研究背景，接着说明了研究内容和目标，然后说明了论文的创新点和难点，最后介绍了论文结构。

第二章：相关规范及开发环境，本章作为接下来章节进行分析和研究的铺垫，首先会阐述 UEFI 和 PI 规范及 UEFI BIOS 的启动流程，接着会阐述现代、功能丰富且跨平台的 UEFI BIOS 开源固件开发环境 EDK2。

第三章：Variable 接口及实现的分析，首先会分析 Variable 的相关接口，接着会分析基于 EDK2 实现的 Variable 存储区格式和 Variable 模块，DXE Variable 驱动会遵循 UEFI 和 PI 规范提供 UEFI Variable 运行时服务及产生 Variable 架构

协议和 Variable Write 架构协议， SMM Variable 驱动会产生 SMM Variable 协议， PEI Variable 模块会遵循 PI 规范产生 Variable ReadOnly PPI。 本章对我们理解 UEFI Variable 的相关原理非常重要， 是接下来章节研究 UEFI Variable 保护和恢复机制的前提。

第四章： 保护和恢复机制-UEFI 变量检查的提出， 会详细地分析现有的保护和恢复机制-Variable Lock 和平台配置 Variable 默认值恢复， 以及 Variable 接口的安全性。 针对各种可能的恶意攻击， 很多情况下现有的保护和恢复机制并不能被使用， 我们会提出新的保护和恢复机制-基于 EDK2 的 UEFI 变量检查及各种检查方法来保证从 UEFI Variable 运行时服务的 SetVariable 接口传入的 Variable 属性、 数据和数据大小的合法性， 从而更好地对 NV Variable 及 NV Variable 区域进行保护。

第五章： 保护和恢复机制-UEFI 变量检查， 首先会研究变量检查模型的建立， 接着会分别研究基于 UEFI、 基于 HII 和基于 PCD 的变量检查以及配额管理， 然后利用 NT32 模拟开发平台对新的保护和恢复机制-UEFI 变量检查进行验证测试， 最后会简要地说明一个平台如何应用 UEFI 变量检查。

第六章： 总结与展望， 会进行全文总结及展望。

在实际项目中， 本人作为提出和研究 UEFI 变量检查的主要参与者， 实现了 UEFI 变量检查。 本文最最主要的分析和研究工作在第四章和第五章， 第二章阐述了相关规范及开发环境， 是接下来章节进行分析和研究的铺垫， 第三章则对我们理解 UEFI Variable 的相关原理非常重要， 是接下来章节研究 UEFI Variable 保护和恢复机制的前提。

第二章 相关规范及开发环境

本章首先会阐述 UEFI 和 PI 规范及 UEFI BIOS 的启动流程，接着会阐述现代、功能丰富且跨平台的 UEFI BIOS 开源固件开发环境 EDK2。

2.1 UEFI 及 PI 规范

BIOS (Basic Input/Output System, 基本输入/输出系统) 最早起源于 20 世纪 80 年代, 基于 Intel 8086 架构处理器, 是计算机中的底层固件, 存储在 EEPROM (Electrically Erasable Programmable Read-Only Memory, 电可擦可编程只读存储器) 上被用来完成启动前的硬件初始化和操作系统的引导。随着计算机硬件的快速发展, 传统 BIOS 逐渐成为瓶颈, 已经很难适应现代计算机硬件的发展需求。传统 BIOS 采用汇编语言编写^[5], 虽然汇编语言具有目标代码简短、占用内存少以及执行速度快等优点, 但由于编写难度大, 因此很难对 BIOS 的功能进行扩展。传统 BIOS 以 16 位汇编代码、寄存器参数调用方式, 运行于实模式(Real Mode)下, 寻址能力为 1MB。虽然理论上和技术上可以在实模式下拥有更大的寻址空间, 但这些技术都需要采用特殊的技巧, 这使得系统与硬件的耦合度大幅增加。Option ROM 的地址空间也局限于 1MB, 这很大的限制了 Option ROM 中驱动程序的扩展能力。1985 年 Intel 发布了 32 位 CPU, 而目前的 CPU 总线已到达 64 位, 传统 BIOS 的寻址能力对此造成了严重的浪费。传统 BIOS 是操作系统与硬件的接口, 然而这个接口并没有统一的标准, 各个 BIOS 厂商的产品实现可能各不相同^{[6][7][8][9][10][11][12]}。为此, 2000 年, Intel 向业界展示了 BIOS 的新一代接口程序 EFI (Extensible Firmware Interface, 可扩展固件接口^[13]), 并将此技术应用于其安腾服务器平台上。EFI 是由 Intel 推出的一种在未来的电脑系统中用来替代传统 BIOS 的升级方案, 它采用模块化和 C 语言^[14]风格的参数堆栈传递方式的形式构建系统, 不同于传统 BIOS 的固定的、缺乏文档的、完全基于经验和晦涩约定的一个事实标准, Intel 将 EFI 定义为一个可扩展的、标准化的固件接口规范。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（Unified Extensible Firmware Interface，统一可扩展固件接口）^[15]。UEFI 联盟的工作小组包括 UEFI 规范工作组(USWG)、UEFI 测试工作组(UTWG)、平台初始化工作组(PIWG)、业界联络工作组(ICWG)和 ACPI 规范工作组(ASWG)。其中 UEFI 规范工作组负责开发、管理和推广 UEFI 规范，平台初始化工作组则负责开发、管理和推广 PI (Platform Implementation，平台初始化) 规范^{[6][9][16]}。下面分别对 UEFI 规范和 PI 规范进行阐述。

2.1.1 UEFI 规范

UEFI 规范定义了操作系统与系统硬件平台固件之间的开放接口，如图 2-1 所示 UEFI 功能示意。



图 2-1 UEFI 功能示意图^[2]

Fig.2-1 UEFI functional diagram^[2]

UEFI 规范是一个公开的纯接口定义，它并不针对某些特定的处理器架构，也不依赖于某个特定的 BIOS 制造商或者某个特定的 BIOS 的实现，它仅仅定义了平台固件必须实现的接口，以及操作系统引导装载器与操作系统可能使用的一系列接口和数据结构，其实现的方式与细节均取决于该规范的实现者。UEFI 规范还定义了 UEFI 驱动模型，使得所有遵循此模型开发的固件驱动程序能够互相协作^[15]。

EFI 系统表是 UEFI 规范定义的最重要的数据结构，如图 2-2 所示，它的指针会被当作入口参数传递给每个驱动和应用程序，进而驱动和应用程序就可以从这个数据结构得到系统的配置信息和丰富的系统服务^[7]。

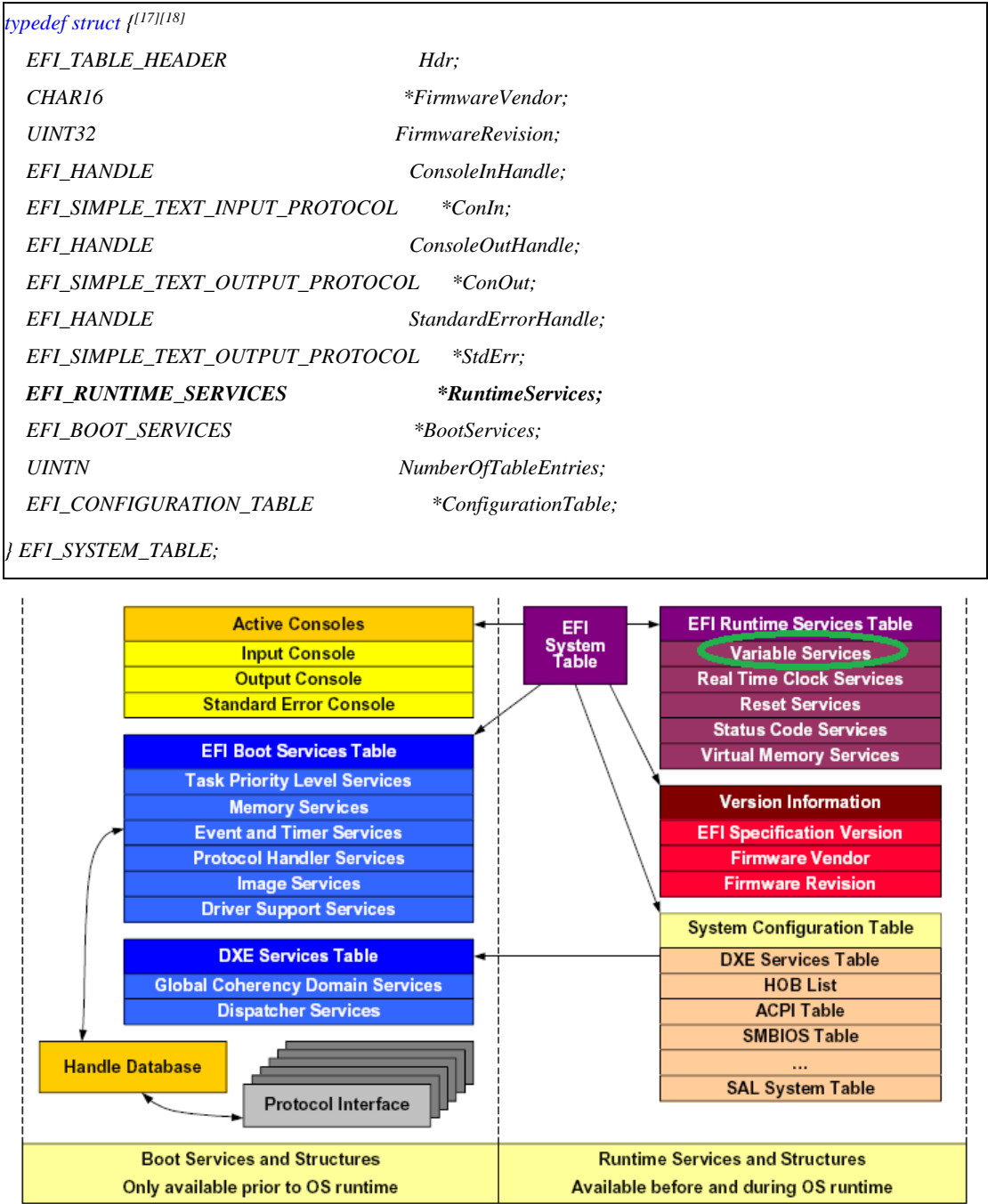


图 2-2 EFI 系统表^[7]

Fig.2-2 EFI System Table^[7]

EFI 系统表中的启动时服务和结构只在操作系统运行时之前是可用的；而运行时服务和结构则在操作系统运行时之前和操作系统运行时都是可用的，接下来的第 3.1.1 节会对其中的 UEFI Variable 运行时服务做进一步的分析。

另外 UEFI 规范定义的 Handle 数据库是 UEFI 需要维护的最重要的对象库，由 Handle 和协议(Protocol)组成。这个 Handle 数据库是所有的 UEFI Handle 的列表，每个 Handle 上可以挂载一个或者多个协议。协议是用 GUID（Globally

Unique Identifier, 全局唯一标识符)来命名并唯一标识的结构体, 可能包含一些函数指针和数据结构体。最小的协议可以只定义 GUID (如第 3.1.1 节中将会分析的 Variable 架构协议和 Variable Write 架构协议), 而不包含任何函数指针或者数据结构体。换句话说, GUID 是一个协议必须要定义的。协议通过挂载在 Handle 上来进行使用, 跟 Handle 一起存放在 Handle 数据库中。系统中的其他函数通过 GUID 在 Handle 数据库中查找对应的协议^{[7][19]}。UEFI 的可扩展特性很大程度上就体现在协议上。

2.1.2 PI 规范

UEFI 规范定义了操作系统与系统硬件平台固件之间的开放接口, 但对这些接口的具体实现并没有做过多的限制。早前 Intel 对 EFI 的实现定义了 Intel Platform Innovation Framework for EFI, 后来 UEFI 联盟基于这个 Framework 定义了 PI 规范。如图 2-3 所示 UEFI BIOS 层次结构, 从中可以很清楚地看到平台硬件、PI、UEFI 和操作系统的相互联系。

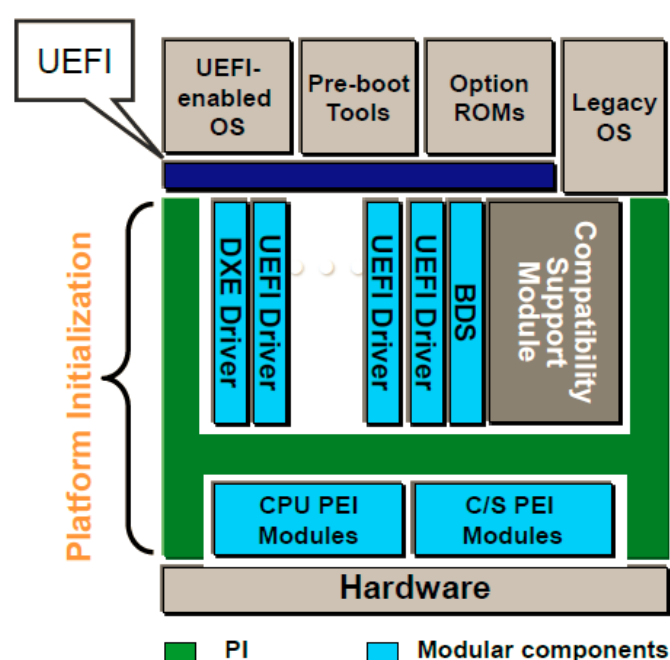


图 2-3 UEFI BIOS 层次结构^[21]

Fig.2-3 UEFI BIOS hierarhacy structure^[21]

PI 规范建立了固件内部接口架构以及固件和平台硬件间的接口, 从而使得平台硬件驱动程序具有模块化和互操作性。作为一个开放的业界标准接口, 在平台固件和操作系统之间, UEFI 规范定义了一个抽象编程接口。标准的 UEFI 接口可以有多种不同架构的具体实现方式, 但是这些接口对外的表现形式是

相同的。基于UEFI的各种协议和接口，支持UEFI的操作系统和其它应用能被加载启动；或者基于兼容性支持模块(CSM, Compatibility Support Module)，传统操作系统也能被加载启动。总体来看，PI和UEFI组成了UEFI BIOS层次结构的主体，它们往下连接了平台硬件，往上连接了操作系统和其它应用^[20]。

PI规范总共有五卷，它们各自定义的接口会有不同的侧重点。

卷1 Pre-EFI Initialization Core Interface^[22]包括了SEC和PEI阶段的各种接口定义，如PEI Services Table(EFI_PEI_SERVICES)和架构PPI（如第3.1.2节中将会分析的Variable ReadOnly PPI）等。PPI(PEIM-to-PEIM Interface)，顾名思义就是PEIM与PEIM之间通信的桥梁，它的定义类似于第2.1.1节中阐述的协议，也是用GUID来命名并唯一标识的结构体，可能包含一些函数指针和数据结构体，不同的是PEI阶段没有Handle和Handle数据库的概念，PPI不用也不会挂载在Handle上，PEI阶段的PPI通过PPI数据库来进行维护。

卷2 Driver Execution Environment Core Interface^[23]包括了DXE和BDS阶段的各种接口定义，如DXE Services Table(DXE_SERVICES)和架构协议（Architectural Protocol，如第3.1.1节中将会分析的Variable架构协议和Variable Write架构协议）等。架构协议基本上与EFI系统表中的启动时服务和运行时服务相对应。

卷3 Shared Architectural Elements^[24]主要包括Firmware Storage(FD, FV, FILE 和 SECTION)，HOB(Hand-Off Block)和PCD(Platform Configuration Database)等相关的定义。第3.2节中将会分析的NV Variable区域就是一个特定的FV，NV Variable区域在NOR FLASH中的起始位置和大小以及每个独立的Variable的最大大小就是由PCD指定的，另外NV Variable HOB就是一个GUIDED类型的HOB。

卷4 System Management Mode Core Interface^[25]包括SMM环境下工作的各种接口的定义。第3.1.1节中将会分析的SMM Variable协议就工作在SMM环境下，但SMM Variable协议并没定义在PI规范中，而是基于EDK2实现的与UEFI Variable运行时服务及Variable架构协议和Variable Write架构协议相对应的在SMM环境下工作的Variable协议。

卷5 Standards^[26]包括其它各种的标准接口的定义。

2.2 UEFI BIOS 的启动流程

基于UEFI和PI规范实现的UEFI BIOS启动流程如图2-4所示。

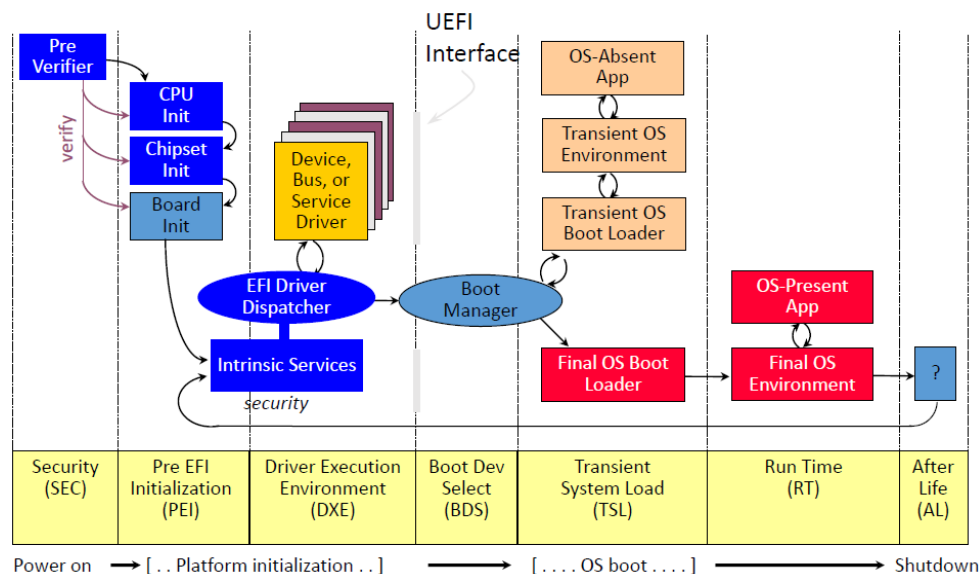


图 2-4 UEFI BIOS 启动流程^[21]

Fig.2-4 UEFI BIOS boot flow^[21]

按照启动先后顺序，该流程分为以下7个阶段：

- 1) 安全检测(SEC, Security)阶段；
- 2) EFI初始化准备(PEI, Pre-EFI Initialization)阶段；
- 3) 驱动程序执行环境(DXE, Driver Execution Environment)阶段；
- 4) 启动设备选择(BDS, Boot Device Selection)阶段；
- 5) 瞬时系统加载(TSL, Transient System Load)阶段；
- 6) 运行时(RT, Run Time)阶段；
- 7) 生命周期后(AL, After Life)阶段。

下面分阶段进行阐述。

2.2.1 安全检测(SEC)阶段

在X86的计算机系统上，系统开机，CPU从4GB的最高端地址0FFFFFFF0h及重置向量处(ResetVector)开始执行指令。ResetVector组件会被打包成一个固件文件系统(FFS, Firmware File System)高文件卷文件(VTF, Volume Top File)^[7]，它负责找到和执行SecCore(SEcurity CORE)模块。接着SecCore模块会进行基本的系统初始化、为CPU打微码补丁、初始化CAR(Cache As RAM)、切换到保护模式、获取启动固件卷(BFV, Boot Firmware Volume)的地址和找到

PeiCore(PEI_CORE)及其入口地址以准备进入PEI阶段。在CAR被初始化好之前, 系统没有内存来作为堆栈使用, ResetVector组件和SecCore模块的代码采用汇编实现; 在CAR被初始化好之后, 一小块CPU cache空间将被当作临时内存(temporary memory)来使用, SecCore模块会进入C语言执行环境。接着SecCore模块利用获取的启动固件卷地址和初始化了的CAR来给EFI_SEC_PEI_HAND_OFF中的启动固件卷地址和大小、临时内存地址和大小、PEI临时内存地址和大小以及栈地址和大小域赋值, 最后把EFI_SEC_PEI_HAND_OFF与SEC阶段提供的PpiList一起传递给PeiCore, 从而进入PEI阶段。SEC阶段提供的PpiList可能会包含安全或者验证服务, 如TCG(Trusted Computing Group)访问服务, 因为在一个遵循TCG的系统中, SEC就是可信度量根的核心(CRTM, Core Root-of-Trust Measurement)^[22]。

2.2.2 EFI 初始化准备(PEI)阶段

PEI阶段的主要任务是决定系统的启动路径(正常启动、S3唤醒或者Recovery等)^[7]; 初始化各种平台硬件, 包括CPU和芯片组等; 检测和初始化系统内存; 报告和处理DXE固件卷, 最后找到DxeCore(DXE_CORE)及其入口地址以进入DXE阶段。

PEI阶段的调度过程如图2-5所示。

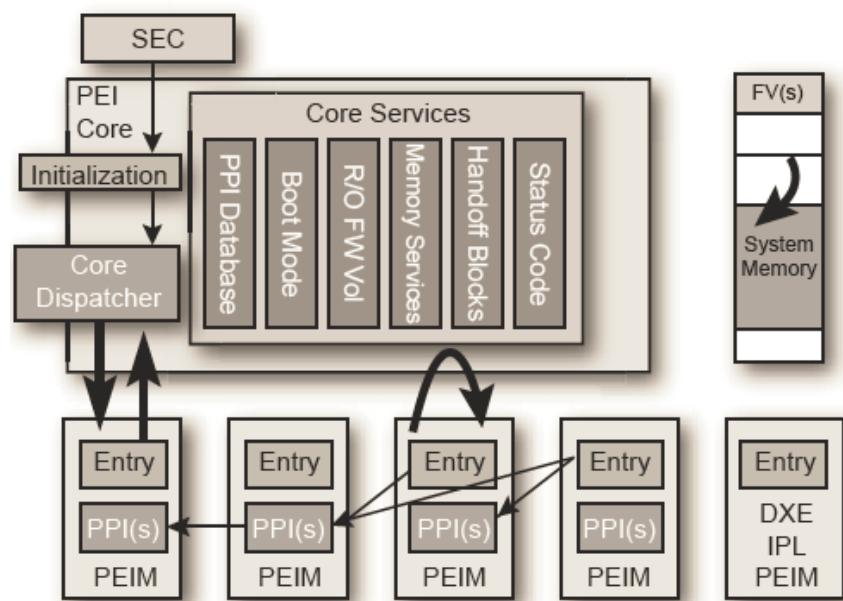


图 2-5 PEI 阶段的调度过程^[2]

Fig.2-5 The dispatch process of PEI phase^[2]

PeiCore(PEI_CORE)首先会对PEI的基础服务(如内存、HOB、PPI和FV

服务等)进行初始化,接着PEI调度器开始利用FV服务根据固件文件格式从启动固件卷查找PEI模块(PEIM,如第3.3.2节中将会分析的PEI Variable模块),检查它们的依赖关系(dependency expression)是否已经满足,进而执行PEI模块,PEI模块可能会产生PPI为其它PEI模块提供服务(如第3.3.2节中将会分析的PEI Variable模块就会产生Variable ReadOnly PPI)。DxeIpl PEI模块会产生DxeIpl PPI。内存初始化模块在检测和初始化系统内存之后会通过PEI的基础服务InstallPeiMemory来向PeiCore报告PEI阶段可用的常驻内存(permanent memory)和建立Resource HOB来报告系统内存。接着PeiCore将会迁移PEI正在使用的临时内存(包括栈(stack)和堆(heap),HOB列表就在堆中)里面的所有数据到常驻内存中去,以及产生MemoryDiscovered PPI来通知其它所有依赖于系统内存的PEI模块,让它们知道系统内存可以使用了,然后其它PEI模块通过PEI基础服务NotifyPpi注册的MemoryDiscovered PPI回调函数(callback)就会被执行,如平台PEI模块注册的MemoryDiscovered PPI回调函数可能会设置系统内存的缓存属性和通过产生FvInfo PPI来报告DXE固件卷。PEI阶段的内存映射和使用如图2-6所示。

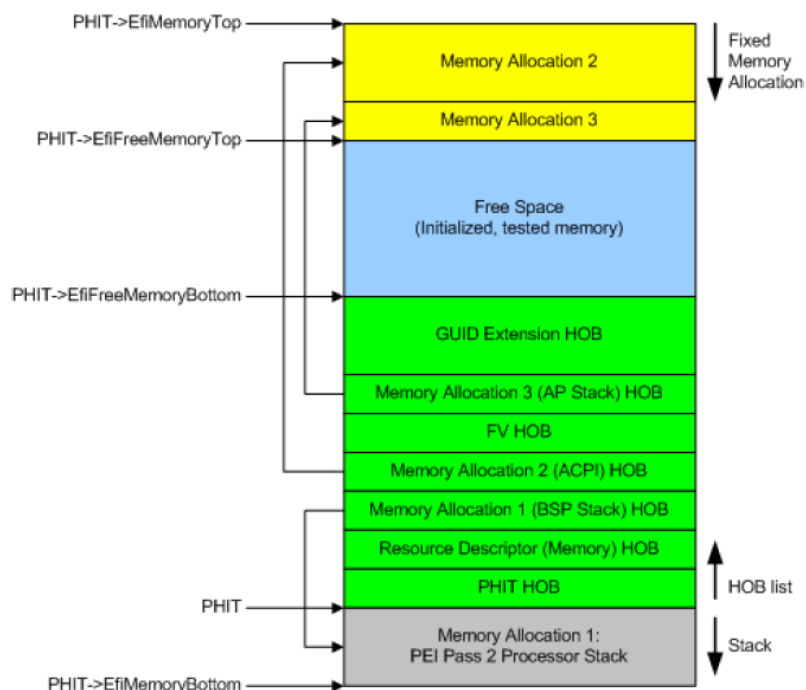


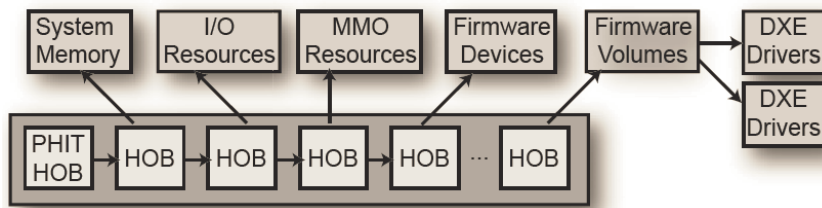
图 2-6 PEI 阶段的内存映射和使用^[24]

Fig.2-6 The memory mapping and usage at PEI phase^[24]

PEI调度器在执行完所有的PEI模块之后会调用DxeIpl PPI,接着DxeIpl PPI就会从DXE固件卷中找到DxeCore及其入口地址,最后把PEI阶段建立的HOB列

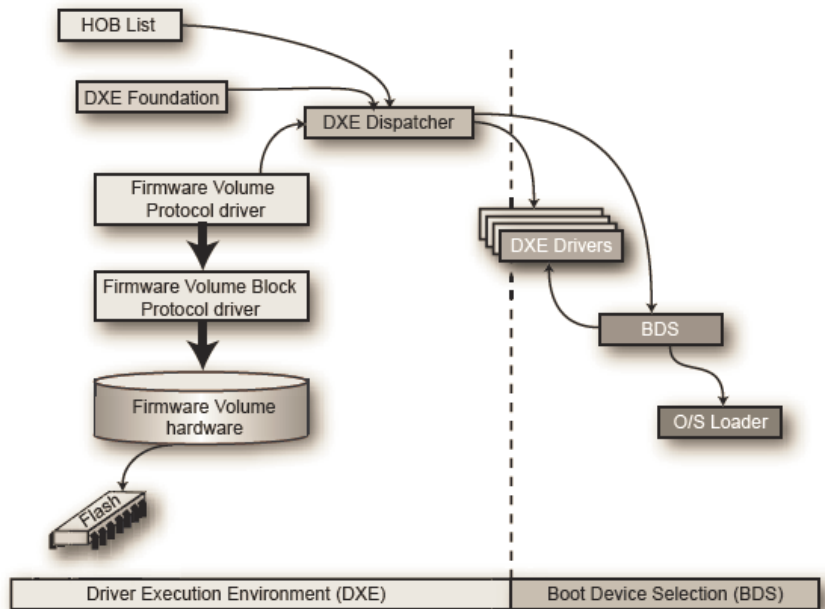
表传递给DxeCore，从而进入DXE阶段。在X86的计算机系统上，如果DXE阶段为64位，DxeIpl PPI在执行DxeCore之前还需要建立页表(page table)和切换CPU到IA32e模式(long mode)。

PEI阶段建立的HOB列表如图2-7所示，开始的一个HOB必须是PHIT(Phase Handoff Information Table) HOB，最后一个HOB必须是end of list HOB。第3.2节中将会分析的NV Variable HOB就是一个GUIDED类型的HOB。

图 2-7 HOB 列表^[2]Fig.2-7 HOB list^[2]

2.2.3 驱动程序执行环境(DXE)阶段

DXE阶段包含的组件：DXE Foundation、DXE调度器和DXE驱动程序^[7]，如图2-8所示。

图 2-8 DXE 阶段的组件^[2]Fig.2-8 The components of DXE phase^[2]

DXE阶段的主要任务是生成一套完整的UEFI接口，和后续的BDS阶段一起工作，建立控制台并尝试启动操作系统。DXE阶段是UEFI BIOS最重要的阶段^[7]。

DxeCore(DXE_CORE)首先会初始化启动时服务表、运行时服务表和DXE服务表，利用PEI阶段传递过来的HOB列表中的Resource HOB初始化DXE服务表中的GCD(Global Coherency Domain)服务以及启动时服务表中的内存服务，基于HOB列表中的FV HOB产生FVB协议及FV协议。DxeCore还会把DXE服务表和HOB列表安装到EFI系统表的配置表(ConfigurationTable)中。

接着DXE调度器会开始利用FV协议根据固件文件系统的格式从DXE固件卷查找DXE驱动(DRIVER，如第3.3.1节中将会分析的DXE Variable驱动)，检查它们的依赖关系是否已经满足，进而执行DXE驱动。

DXE驱动有两类：较早执行的DXE驱动和UEFI驱动(符合UEFI驱动模型的DXE驱动)。较早执行的DXE驱动程序会初始化平台硬件，包括CPU和芯片组等，安装配置表(如ACPI^[27]表和SMBIOS^[28]表就会被相应的DXE驱动安装到配置表中)和产生架构协议(如第3.3.1节中将会分析的DXE Variable驱动就会产生Variable架构协议和Variable Write架构协议，BDS驱动会产生BDS架构协议)等。如图2-9所示DXE阶段产生的架构协议，架构协议基本上与EFI系统表中的启动时服务和运行时服务相对应。UEFI驱动在所有的架构协议产生之后才会被执行，它们会产生Driver Binding协议，这些Driver Binding协议在BDS阶段连接控制器(利用启动时服务表中的ConnectController接口)的时候会被连接，进而为控制台和启动设备提供软件抽象。

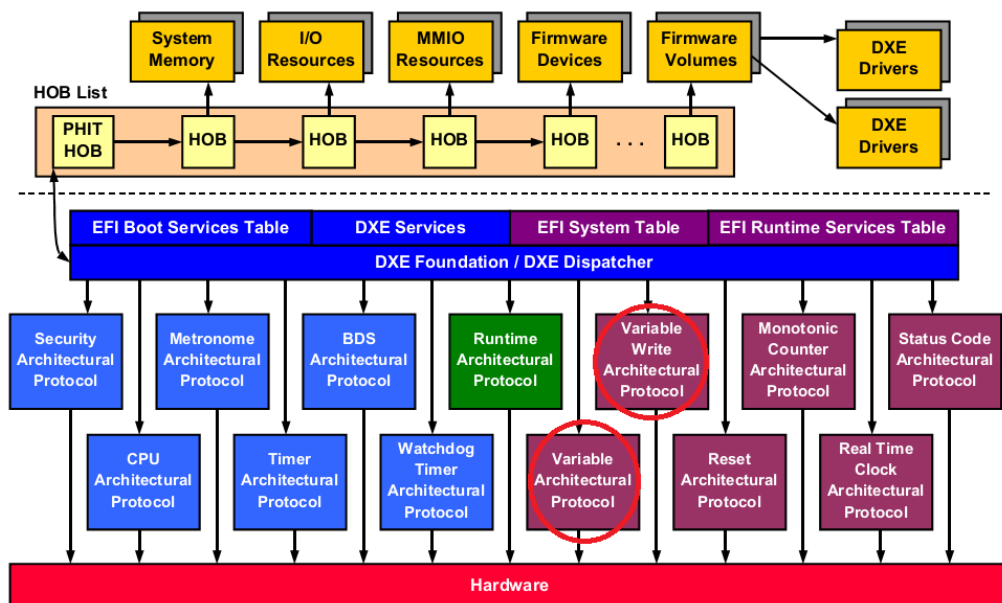


图 2-9 DXE 阶段产生的架构协议^[7]

Fig.2-9 The architectural protocols produced at DXE phase^[7]

DXE阶段会生成一套完整的UEFI接口，DXE调度器在执行完所有的DXE驱动之后会调用BDS架构协议，从而进入BDS阶段，DXE阶段和BDS阶段一起工作来建立可供操作系统启动的平台。注意，只有EFI系统表中的运行时服务和由运行时DXE驱动程序提供的服务在操作系统运行时阶段还继续存在。

2.2.4 启动设备选择(BDS)阶段

BDS阶段的主要功能就是为平台选择一个合适的启动设备，并尝试加载它^[7]。BDS架构协议是DXE阶段的BDS驱动产生的，DXE调度器在执行完所有的DXE驱动之后会调用BDS架构协议，从而进入BDS阶段。在BDS阶段，BDS架构协议会在连接控制器（利用启动时服务表中的ConnectController接口）的时候连接DXE阶段UEFI驱动产生的Driver Binding协议，进而为控制台和启动设备提供软件抽象。BDS阶段是一个独特的启动管理阶段，根据平台的启动策略，UEFI规范定义的全局启动选项Variable L"Boot####"和启动选项顺序Variable L"BootOrder"会被建立。一个启动选项对应于相应启动设备中的一个UEFI应用程序(APPLICATION)，如UEFI BIOS的SETUP、启动菜单、扩展的诊断工具和操作系统加载器等等。UEFI启动管理器会按照启动顺序来引导启动选项，启动选项对应的UEFI应用程序会被加载和执行。如果是操作系统加载器被加载和执行，操作系统加载器就会获得控制权，然后进入接下来的TSL阶段来加载操作系统。

2.2.5 瞬时系统加载(TSL)阶段

操作系统加载器在BDS阶段被加载和执行，EFI系统表的指针会被当作入口参数传递给操作系统加载器。在TSL阶段，操作系统加载器会使用UEFI接口加载操作系统。TSL阶段结束及操作系统运行时阶段开始的标志是操作系统加载器调用启动时服务表中的ExitBootServices接口，ExitBootServices接口会触发ExitBootServices事件来通知其它模块启动时服务将要被终止了。

2.2.6 运行时(RT)阶段

操作系统加载器在TSL阶段调用启动时服务表中的ExitBootServices接口而进入操作系统运行时阶段。启动时服务表中的ExitBootServices接口一旦被调用，EFI系统表中所有的启动时服务都会被终止，结果就只有EFI系统表中的

配置表和运行时服务在操作系统运行时阶段可用。操作系统可以从操作系统加载器得到 EFI 系统表的指针，通过这个指针，操作系统可以获得各种平台配置信息（如从配置表得到 ACPI 表和 SMBIOS 表等）和访问运行时服务（如第 3.1.1 节中将会分析的 UEFI Variable 运行时服务）。

2.2.7 生命周期后(AL)阶段

生命周期后阶段代表的是平台的控制权从操作系统端重新回到了 UEFI BIOS 端。这个阶段可以说是操作系统运行时阶段的延续，可能是重启系统（如操作系统调用运行时服务表中 ResetSystem 接口）、进入到 ACPI 睡眠状态（如进入到 S3，S5）或者进入 SMM（在 X86 CPU 检查到一种特殊的系统中断 SMI 的时候，处理器会进入一种特殊的操作模式，这种操作模式叫做 SMM，它对于 OS 来说是完全透明的^[29]）等^[7]。

2.3 EDK2

为了解决用户所反馈的在使用 EDK（EFI Developer Kit，EFI 开发者套件）^[30]时暴露出的问题，英特尔开始了一个现在被称为是 EDK2（EFI Developer Kit 2，EFI 开发者套件 2）的重构计划，旨在使用户能够更容易的为一个平台设计、开发和定制模块^[31]。现如今，EDK2 作为一个现代、功能丰富且跨平台的 UEFI BIOS 开源固件开发平台已经被业界广泛使用，不同于传统的 BIOS 实现，EDK2 应用现代软件体系设计的思想，基于 UEFI 和 PI 规范，采用模块化设计。EDK2 项目的源代码以开源的形式发行在 <https://github.com/tianocore/edk2>，支持各种处理器架构（如 IA32、X64、IA64、ARM 和 AArch64）和编译器（如微软 VC，GNU GCC 和英特尔 ICC）。

EDK2 引入了三个核心的概念：包(Package)、库类/库实例(Library Class/Library Instance)和平台配置数据库(PCD，Platform Configuration Database)^[31]，下面分别对它们进行阐述。

2.3.1 包(Package)

在没有整套源代码的情况下，EDK 不能完成编译工作，而且 EDK 要求以整套源代码作为发行单位。为了解决这个问题，EDK2 引入了“包”的概念，包是 EDK2 最基本的源代码发行单位。这样相对于 EDK，EDK2 的用户就可

以包为单位进行开发及代码发行，而不需要以整套源代码作为发行单位^[31]。

一个包是最小的发行单位，而且在一个多用途的大项目中提供了一个自然分割。例如，从硬件的角度来看，开发人员可以把处理器/芯片组/平台相关的定义和驱动程序分割到三个单独的包，这样就能使用户更容易的发布和重用这些处理器/芯片组/平台相关的定义和驱动程序。开发人员还可以把所有平台无关的模块放到一个单独的包中，这样开发人员在开发一个新的平台的时候就只需要专注于平台相关的代码就可以了^[31]。

EDK2 提供了一个包的集合，开发人员可以基于 EDK2 源码库创建和发布他或她自己的包。因为有些包对于编译一个给定的模块或固件不是必须的，固件开发人员可以只选择相关的包来完成编译和发行^[31]。EDK2 提供的主要包的描述如下：

- **BaseTools:** 提供了基于 Python^[32]和 C 语言实现的编译工具源代码、编译工具和编译所需要的 3 个关键配置文件的模板。
- **Conf:** 只包含了一个 readme.txt 文件。这个目录将会被用来包含编译所需要的 3 个关键配置文件：target.txt、build_rule.txt 和 tools_def.txt，这些配置文件定义了编译目标、编译规则和工具链等^[20]。
- **MdePkg:** 声明了 UEFI 和 PI 规范、工业标准定义的协议、PPIs、GUIDs 和相关数据结构（如第 3.1 节中将会分析的 UEFI 和 PI 规范定义的 Variable 的属性和相关服务及接口，MdePkg\Include\Uefi\UefiMultiPhase.h、MdePkg\Include\Uefi\UefiSpec.h、MdePkg\Include\Protocol\Variable.h、MdePkg\Include\Protocol\VariableWrite.h、MdePkg\Include\Ppi\ReadOnlyVariable2.h）。另外，这个包还包含了很多在开发基于不同平台架构的固件模块时需要的基础库类和库实例。
- **MdeModulePkg:** 提供了基于 UEFI 和 PI 规范实现的跨平台模块（如第 3.3 节中将会分析的基于 EDK2 实现的 Variable 模块），以及定义了为这些模块服务的协议、PPIs、GUIDs、库类和库实例（如第 3.1.1 节中将会分析的基于 EDK2 实现的 SMM Variable 协议，MdeModulePkg\Include\Protocol\SmmVariable.h，和第 3.2 节中将会分析的基于 EDK2 实现的 Variable 存储区格式，MdeModulePkg\Include\Guid\VariableFormat.h）。

- **Nt32Pkg:** 运行在微软 Windows 操作系统中的 UEFI 模拟开发平台。第 5.6 节将会利用 NT32 模拟开发平台对新的保护和恢复机制-UEFI 变量检查进行验证测试。
- **UefiCpuPkg:** 提供了兼容 UEFI 和 PI 规范的 X86 CPU 相关的定义、库和模块。

EDK2 的源代码包还包含四种类型的元数据文件：模块信息文件(INF)^[33]、包声明文件(DEC)^[34]、平台描述文件(DSC)^[35]和 FLASH 声明文件(FDF)^[36]。

2.3.2 库类/库实例(Library Class/Library Instance)

在固件的开发过程中，相同功能的不同实现可能是需要的，如使用 C 语言来提供跨平台能力，而非用汇编语言来得到更好的性能；使用 I/O 端口访问 PCI 配置空间而非用 MMIO(Memory Mapped I/O)^[31]。

库类是一组用于提供某些功能的标准接口定义，模块编写者可以直接使用它们来编程。库实例则提供了这些库类接口的实现。库类与库实例的关系是一对多的关系，也就是说一个库类可能有多种实现及多个库实例与之对应^[31]。库实例的实现可能包含构造函数(Constructor)，这些构造函数类似于 C++中的构造函数，在进入模块入口函数之前被执行。有一类特殊的库实例，它们只有构造函数，而没有相应的库类及接口定义，这类库实例被称为 NULL 库实例。

模块的编写依赖库类接口而不是具体的库实例，一个模块可以很容易地通过在平台描述文件中进行配置来与不同的库实例链接。通过库类和库实例，平台开发人员可以有针对性的为相同的功能选择不同的实现^[31]。

EDK2 提供的 MdePkg 包含了很多在开发基于不同平台架构的固件模块时需要的基础库类和库实例。例如，MdePkg 提供了一个名为 BaseMemoryLib 的库类，这个库类包含了很多与内存操作相关的接口，有多个库实例与之对应：一个库实例为了跨平台采用 C 语言实现；一个库实例为了提高内存操作的效率采用汇编语言实现^[31]。

库类的声明在包声明文件中，平台开发人员在模块信息文件中声明要链接的库类，然后模块源代码调用库类接口，最后在平台描述文件中配置模块要链接的库实例。

2.3.3 平台配置数据库(PCD)

PCD 为整个平台的模块、驱动和函数库提供了有效的信息共享机制及统一的访问接口，存储了平台各组件的配置信息。用户可以通过统一的 PCD 接口对这些被提取出来的平台配置信息进行访问，可以很容易地通过修改 PCD 的值来改变平台组件的行为，而不需要修改相应的源代码，这使开发和定制化一个平台更容易。

PCD 的类型分两大类：静态 PCD 和动态 PCD。静态 PCD 的值在编译过程中生成，包括 FeatureFlag PCD、FixedAtBuild PCD 和 PatchableInModule PCD 三种。FeatureFlag PCD 相当于一个宏开关，被用来启用或者禁用一个功能，其值为布尔(BOOLEAN)类型，为 TRUE 或者 FALSE。FixedAtBuild PCD 被用来替代一个可定制化的宏，其值存储在代码段中。PatchableInModule PCD 与 FixedAtBuild PCD 类似，但其值存储在数据段中，而非代码段。动态 PCD 的默认值会在编译过程中生成，然后在系统执行的过程中，其值还可能被一个模块设置，最后被另一个模块访问，包括 Dynamic PCD 和 DynamicEx PCD。Dynamic PCD 又分三个小类 DynamicDefault PCD、DynamicHii PCD 和 DynamicVpd PCD 应用于源代码组件发布。相应的，DynamicEx PCD 也分为三个小类 DynamicExDefault PCD、DynamicExHii PCD 和 DynamicExVpd PCD 专门应用于编译好的二进制组件发布。Default PCD 的值存储在 PEI PcdPeim 模块和 DXE PcdDxe 驱动维护的动态 PCD 数据库中，在启动过程中可以被修改，在系统断电或者重启后恢复到默认值。HII PCD 会映射到一个 UEFI Variable 或者 Variable 的一个域，默认值存储在 PEI PcdPeim 模块和 DXE PcdDxe 驱动维护的动态 PCD 数据库中，但修改的值会保存到对应 UEFI Variable 所在的 NV Variable 区域，这样下次系统启动的时候，HII PCD 的值就会从对应的 UEFI Variable 来。VPD PCD 的值是只读的，保存在 UEFI BIOS 的一段二进制数据空间中，这样在固件编译生成后，可以在不依赖编译器重新编译的情况下，利用工具对 VPD PCD 的值进行修改^{[15][20][37]}。

PCD 的定义包括 GUID 值、C 名字、支持的 PCD 类型、数据类型和默认值等^[20]，声明在包声明文件中，平台开发人员在平台描述文件中配置 PCD 的类型和值，模块信息文件中声明要使用的 PCD，模块编写者不需要知道 PCD

的类型，模块源代码调用统一的 PCD 接口就可以对 PCD 进行访问。例如，PCD "PcdDebugPrintErrorLevel"被用来控制调试打印级别，然后模块编写者只需要调用 PCD 接口 PcdGet32(PcdDebugPrintErrorLevel)就可以得到它的值^[31]。

第 3.2 节中将会分析的指定 NV Variable 区域在 NOR FLASH 中的起始位置的 PCDs(gEfiMdeModulePkgTokenSpaceGuid.PcdFlashNvStorageVariableBase 和 gEfiMdeModulePkgTokenSpaceGuid.PcdFlashNvStorageVariableSize)定义在 MdeModulePkg 中，支持 PcdsFixedAtBuild、PcdsPatchableInModule、PcdsDynamic 和 PcdsDynamicEx 类型，指定单个 Variable 的最大大小的 PCD(gEfiMdeModulePkgTokenSpaceGuid.PcdMaxVariableSize)也定义在 MdeModule Pkg 中，支持 PcdsFixedAtBuild 和 PcdsPatchableInModule 类型。

2.4 本章小结

本章作为接下来章节进行分析和研究的铺垫，首先阐述了UEFI和PI规范及UEFI BIOS启动流程，UEFI规范定义了操作系统与系统硬件平台固件之间的开放接口，PI规范建立了固件内部接口架构以及固件和平台硬件间的接口，UEFI BIOS启动流程分为安全检测、EFI初始化准备、驱动程序执行环境、启动设备选择、瞬时系统加载阶段、运行时和生命周期后七个阶段，接着阐述了现代、功能丰富且跨平台的UEFI BIOS开源固件开发环境EDK2，特别是EDK2引入的三个核心的概念：包、库类/库实例和平台配置数据库。

第三章 Variable 接口及实现的分析

UEFI 和 PI 规范定义了 Variable 的属性和相关服务及接口，但并没有定义 Variable 存储到 NOR 非易失性存储器上(NV Variable)或内存中(Volatile Variable)的具体格式以及 Variable 模块的具体实现来遵循 UEFI 和 PI 规范提供访问 Variable 的各种服务及接口。下面首先会分析 Variable 的相关接口，接着会分析基于 EDK2 实现的 Variable 存储区格式和 Variable 模块。

3.1 Variable 接口

UEFI Variable 能在各模块以及 UEFI BIOS 和操作系统之间传递数据，由一对 VendorGuid 和 VariableName 来唯一标识，Variable 数据的格式和大小由它的生产者和消费者决定。同时每个 Variable 都有相应的属性，UEFI 规范定义了如下的 Variable 属性：

```
MdePkg\Include\Uefi\UefiMultiPhase.h:[17][18]
///
/// Attributes of variable. UEFI规范定义的Variable属性
///
#define EFI_VARIABLE_NON_VOLATILE                0x00000001
#define EFI_VARIABLE_BOOTSERVICE_ACCESS        0x00000002
#define EFI_VARIABLE_RUNTIME_ACCESS              0x00000004
#define EFI_VARIABLE_HARDWARE_ERROR_RECORD      0x00000008
///
/// Attributes of Authenticated Variable
///
#define EFI_VARIABLE_AUTHENTICATED_WRITE_ACCESS 0x00000010
#define EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS 0x00000020
#define EFI_VARIABLE_APPEND_WRITE               0x00000040
```

EFI_VARIABLE_NON_VOLATILE：简写为 NV，及非易失性，系统重启之后 Variable 仍能够保持，相对的 Volatile Variable 在系统重启之后会丢失。

EFI_VARIABLE_BOOTSERVICE_ACCESS：简写为 BS，启动时可访问。

EFI_VARIABLE_RUNTIME_ACCESS：简写为 RT，运行时可访问。

EFI_VARIABLE_HARDWARE_ERROR_RECORD：简写为 HR，及硬件错记

录。

EFI_VARIABLE_AUTHENTICATED_WRITE_ACCESS: 基于计数认证的，已经被不建议使用， 本文将其简写为 AW。

EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS: 简写为 AT， 基于时间认证的， 被推荐的认证 Variable 属性。

EFI_VARIABLE_APPEND_WRITE: 附加写。

UEFI 和 PI 规范还定义了访问 Variable 的各种服务及接口， 下面分阶段对它们进行分析。

3.1.1 DXE 阶段

UEFI 规范定义的 Variable 运行时服务如下， 它们在操作系统运行时之前和操作系统运行时都是可用的。

```
MdePkg\Include\Uefi\UefiSpec.h:[17][18]
///
/// EFI Runtime Services Table. 运行时服务表
///
typedef struct {
    ///
    /// The table header for the EFI Runtime Services Table.
    ///
    EFI_TABLE_HEADER          Hdr;
    ...
    ///
    /// Variable Services Variable运行时服务
    ///
    EFI_GET_VARIABLE          GetVariable;
    EFI_GET_NEXT_VARIABLE_NAME  GetNextVariableName;
    EFI_SET_VARIABLE          SetVariable;
    ...
    EFI_QUERY_VARIABLE_INFO   QueryVariableInfo;
} EFI_RUNTIME_SERVICES;
```

GetVariable 通过 VendorGuid 和 VariableName 来获得一个 Variable 的属性、数据和数据大小。 如果相应的 Variable 在 Variable 存储区域没有找到， EFI_NOT_FOUND 就会被返回。

GetNextVariableName 会被调用多次来检索系统中所有的 Variable 的 VendorGuid 和 VariableName。 VariableName 会指向一个空字符串以作为输入来

开始整个查找过程，系统中的第一个 Variable 的 VendorGuid 和 VariableName 就会被返回，然后前一次调用的结果可以被用来作为输入以得到下一个 Variable 的 VendorGuid 和 VariableName，直到所有的 Variable 都被找到了，错误状态 EFI_NOT_FOUND 就会被返回。

SetVariable 根据输入的 VendorGuid、VariableName、属性、数据和数据大小来删除、添加或者更新一个 Variable。如果输入的 Variable 属性为 0，或者输入的 Variable 数据大小为 0 且 Variable 属性不包含附加写、AW 或者 AT，相应的 Variable 就会被删除。

QueryVariableInfo 根据属性来查询系统中相应属性的最大 Variable 存储空间大小，剩余的 Variable 存储空间大小和最大 Variable 大小。

相应的 PI 规范卷 2 定义了 Variable 架构协议和 Variable Write 架构协议。DXE Variable 驱动必须是一个运行时驱动，它负责初始化 UEFI 运行时服务表中的 GetVariable、GetNextVariableName、SetVariable 和 QueryVariableInfo 域，产生 Variable 架构协议来通知其它 DXE 模块读取 NV Variable 和读写 Volatile Variable 的 UEFI Variable 运行时服务可用了，以及产生 Variable Write 架构协议来通知其它 DXE 模块删除、添加或者更新 NV Variable 的 UEFI Variable 运行时服务可用了。如果另一个 DXE 模块需要读取 NV Variable 或者读写 Volatile Variable，那它必须有 EFI_VARIABLE_ARCH_PROTOCOL_GUID 在它的依赖关系表达式及模块信息文件里的[Depex]中；如果另一个 DXE 模块需要删除、添加或者更新 NV Variable，那它必须有 EFI_VARIABLE_WRITE_ARCH_PROTOCOL_GUID 在它的依赖关系表达式及模块信息文件里的[Depex]中。这样 DXE 调度器就可以保证这个 DXE 模块在 Variable 架构协议或者 Variable Write 架构协议被产生之后才会被执行。

```
MdePkg\Include\Protocol\Variable.h:[18][23] Variable 架构协议
/// Global ID for the Variable Architectural Protocol
#define EFI_VARIABLE_ARCH_PROTOCOL_GUID \
    { 0x1e5668e2, 0x8481, 0x11d4, {0xbc, 0xf1, 0x0, 0x80, 0xc7, 0x3c, 0x88, 0x81} }
MdePkg\Include\Protocol\VariableWrite.h:[18][23] Variable Write 架构协议
/// Global ID for the Variable Write Architectural Protocol
#define EFI_VARIABLE_WRITE_ARCH_PROTOCOL_GUID \
    { 0x6441f818, 0x6362, 0x4e44, {0xb5, 0x70, 0x7d, 0xba, 0x31, 0xdd, 0x24, 0x53} }
```

下面是如何使用 UEFI Variable 运行时服务 GetVariable 和 SetVariable 的代码实例^[18]。

```
UINTN          DataSize;
UINT32          Attributes;
UINT16          BootTimeOut;

DataSize = sizeof (UINT16);

//
// 获取UEFI全局超时Variable L"Timeout"的数据
//
gRT->GetVariable (
    EFI_TIME_OUT_VARIABLE_NAME,
    &gEfiGlobalVariableGuid,
    &Attributes,
    &DataSize,
    &BootTimeOut
);

...

DataSize = sizeof (UINT16);

//
// 更新UEFI全局超时Variable L"Timeout"的数据
//
gRT->SetVariable (
    EFI_TIME_OUT_VARIABLE_NAME,
    &gEfiGlobalVariableGuid,
    Attributes,
    DataSize,
    &BootTimeOut
);
```

与 UEFI Variable 运行时服务及 Variable 架构协议和 Variable Write 架构协议相对应的还有 SMM Variable 协议，SMM Variable 协议并没有定义在 UEFI 或者 PI 规范中，它是基于 EDK2 实现的在 SMM 环境下工作的 Variable 协议。SMM Variable 协议的 SmmGetVariable、SmmGetNextVariableName、SmmSetVariable、SmmQueryVariableInfo 接口和 UEFI Variable 运行时服务 GetVariable、GetNextVariableName、SetVariable、QueryVariableInfo 的类型定义及接口含义

基本完全相同，唯一的区别是 SMM Variable 协议工作在 SMM 环境下。在 X86 CPU 检查到一种特殊的系统中断 SMI (System Management Interrupt, 系统管理中中断) 的时候，处理器会进入一种特殊的操作模式，这种操作模式就叫做 SMM (System Management Mode, 系统管理模式)。SMI 是所有系统中断中优先级最高的，它能打断像软中断、溢出中断、调试中断甚至 NMI (Non Maskable Interrupt, 不可屏蔽中断) 等一般中断。一块专用存储空间 SMRAM (System Management Random Access Memory, 系统管理内存) 会被留出来给在 SMM 环境下运行的程序使用，通常 SMRAM 在初始化后会被锁起来以保证安全，这样非 SMM 环境下运行的程序就访问不到 SMRAM 中的内容了^[29]。SMM Variable 驱动就工作在 SMM 下，它负责产生 SMM Variable 协议。如果另一个 SMM 模块需要读写 Variable，那它必须有 EFI_SMM_VARIABLE_PROTOCOL_GUID 在它的依赖关系表达式及模块信息文件里的[Depex]中，这样 SMM 调度器就可以保证这个 SMM 模块在 SMM Variable 协议被产生之后才会被执行。

```
MdeModulePkg\Include\Protocol\SmmVariable.h:[18] SMM Variable 协议

#define EFI_SMM_VARIABLE_PROTOCOL_GUID \
{ \
    0xed32d533, 0x99e6, 0x4209, { 0x9c, 0xc0, 0x2d, 0x72, 0xcd, 0xd9, 0x98, 0xa7 } \
}

typedef struct _EFI_SMM_VARIABLE_PROTOCOL EFI_SMM_VARIABLE_PROTOCOL;

///
/// EFI SMM Variable Protocol is intended for use as a means
/// to store data in the EFI SMM environment.
///
struct _EFI_SMM_VARIABLE_PROTOCOL {
    EFI_GET_VARIABLE           SmmGetVariable;
    EFI_GET_NEXT_VARIABLE_NAME SmmGetNextVariableName;
    EFI_SET_VARIABLE           SmmSetVariable;
    EFI_QUERY_VARIABLE_INFO    SmmQueryVariableInfo;
};
```

下面是在 SMM 环境下如何使用 SMM Variable 协议中的 SmmGetVariable 和 SmmSetVariable 的代码实例^[18]。


```

EFI_SMM_VARIABLE_PROTOCOL *mSmmVariable;

//查找SMM Variable协议
gSmst->SmmLocateProtocol (&gEfiSmmVariableProtocolGuid, NULL, (VOID**) &mSmmVariable);

EFI_STATUS
EFIAPI
MemoryClearCallback (
    IN EFI_HANDLE                DispatchHandle,
    IN CONST VOID                *Context,
    IN OUT VOID                  *CommBuffer,
    IN OUT UINTN                 *CommBufferSize
)
{
    EFI_STATUS                Status;
    UINTN                     DataSize;
    UINT8                     MorControl;
    ...
    DataSize = sizeof (UINT8);
    // 获取Variable L"MemoryOverwriteRequestControl"的数据
    Status = mSmmVariable->SmmGetVariable (
        MEMORY_OVERWRITE_REQUEST_VARIABLE_NAME,
        &gEfiMemoryOverwriteControlDataGuid,
        NULL,
        &DataSize,
        &MorControl
    );
    ...
    DataSize = sizeof (UINT8);
    // 更新Variable L"MemoryOverwriteRequestControl"的数据
    Status = mSmmVariable->SmmSetVariable (
        MEMORY_OVERWRITE_REQUEST_VARIABLE_NAME,
        &gEfiMemoryOverwriteControlDataGuid,
        EFI_VARIABLE_NON_VOLATILE |
EFI_VARIABLE_BOOTSERVICE_ACCESS | EFI_VARIABLE_RUNTIME_ACCESS,
        DataSize,
        &MorControl
    );
    ...
}

```

3.1.2 PEI 阶段

PI 规范卷 1 定义了 Variable ReadOnly PPI。 Variable ReadOnly PPI 相当于

UEFI Variable 运行时服务在 PEI 阶段的轻量级版本，只提供了读取 Variable 的功能，其 GetVariable、NextVariableName 接口和 UEFI Variable 运行时服务 GetVariable、GetNextVariableName 的接口含义完全相同。PEI Variable 模块负责产生 Variable ReadOnly PPI，如果另一个 PEI 模块需要读取 Variable，那它必须有 EFI_PEI_READ_ONLY_VARIABLE2_PPI_GUID 在它的依赖关系表达式及模块信息文件里的[Depex]中，这样 PEI 调度器就可以保证这个 PEI 模块在 Variable ReadOnly PPI 被产生之后才会被执行。

```
MdePkg\Include\Ppi\ReadOnlyVariable2.h: [18][22]
#define EFI_PEI_READ_ONLY_VARIABLE2_PPI_GUID \
    { 0x2ab86ef5, 0xecb5, 0x4134, { 0xb5, 0x56, 0x38, 0x54, 0xca, 0x1f, 0xe1, 0xb4 } }
// This PPI provides a lightweight, read-only variant of the full EFI
// variable services.
struct _EFI_PEI_READ_ONLY_VARIABLE2_PPI {
    EFI_PEI_GET_VARIABLE2      GetVariable;
    EFI_PEI_GET_NEXT_VARIABLE_NAME2 NextVariableName;
};
```

下面是在 PEI 阶段如何使用 Variable ReadOnly PPI 中的 GetVariable 的代码实例^[18]。

```
EFI_STATUS      Status;
EFI_PEI_READ_ONLY_VARIABLE2_PPI *Variable;
UINTN          DataSize;
EFI_MEMORY_TYPE_INFORMATION      MemoryData[EfiMaxMemoryType + 1];
...
// 查找Variable ReadOnly PPI
Status = PeiServicesLocatePpi (&gEfiPeiReadOnlyVariable2PpiGuid, 0, NULL, (VOID **) &Variable );
if (!EFI_ERROR (Status)) {
    DataSize = sizeof (MemoryData);
    // 读取Variable L"MemoryTypeInformation"的数据
    Status = Variable->GetVariable (
        Variable,
        EFI_MEMORY_TYPE_INFORMATION_VARIABLE_NAME,
        &gEfiMemoryTypeInformationGuid,
        NULL,
        &DataSize,
        &MemoryData
    );
}
...
```

3.2 Variable 存储区格式

一个典型的 UEFI BIOS 的 FLASH 布局如图 3-1 所示。

Fv Recovery: SEC 和 PEI 阶段代码所在的 FV。

Ftw spare space 和 working space: FTW 模块工作所需要的空间。

Event Log: 事件记录存储区。

Microcode: CPU 微码补丁所在的区域。

Variable Region: NV Variable 区域。

Fv Main: DXE 阶段驱动所在的 FV。

基于 EDK2 实现的 UEFI BIOS 一般会选择 NOR FLASH 中的一块区域作为 NV Variable 区域, NV Variable 区域会占用 NOR FLASH 中的几个区块, EDK2 用定义在 MdeModulePkg 中的 PCDs(gEfiMdeModulePkgTokenSpaceGuid.PcdFlashNvStorageVariableBase 和 gEfiMdeModulePkgTokenSpaceGuid.PcdFlashNvStorageVariableSize)来指定 NV Variable 区域在 NOR FLASH 中的起始位置和大小, 如图 3-1 中箭头所指的区域。

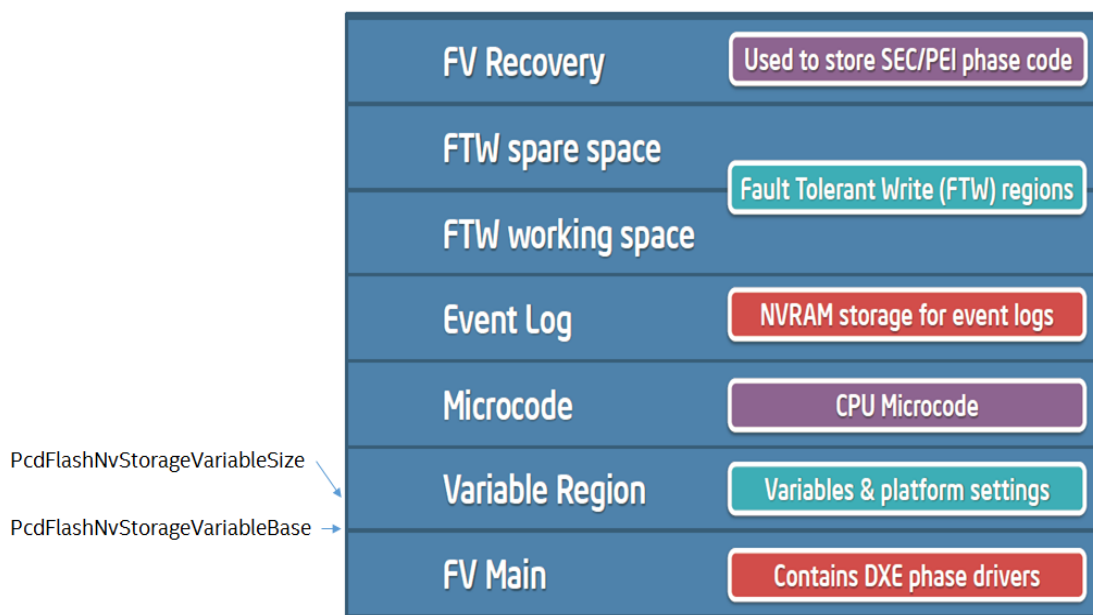


图 3-1 典型的 UEFI BIOS 的 FLASH 布局^[38]

Fig.3-1 The FLASH layout of typical UEFI BIOS^[38]

基于 EDK2 实现的 UEFI BIOS 一般会选择如图 3-2 所示格式来存储 NV Variable。首先 NV Variable 是存储在一个特定的 FV 中, 这个 FV 可以通过 FV Header 中的 FileSystemGuid 域 (gEfiSystemNvDataFvGuid) 来识别。紧接着是 Variable Store Header, Authenticated 或非 Authenticated Variable 区域则可以通过

Variable Store Header 中的 Signature 域 (gEfiVariableGuid 或者 gEfiAuthenticatedVariableGuid)来识别。Variable Store Header 中的 Format 域为 0x5A 表示 Variable 区域是格式化了的, State 域为 0xFE 表示 Variable 区域是健康的。然后是每个独立的 Variable 一个接一个地紧接在 Variable Store Header 后面, Variable Header 中的 StartId 域为 0x55AA 表示一个 Variable 的开始, State 域为 0x3F(VAR_ADDED)表示 Variable 是有效的、为 0x3D(VAR_DELETED)表示 Variable 已经被废弃了、为 0x7F(VAR_HEADER_VALID_ONLY)表示 Variable 只有 Header 是有效的、为 0x3E(VAR_IN_DELETED_TRANSITION)表示 Variable 处于向被废弃过渡的状态, 其中 0x7F 或 0x3E 是一种中间状态, 在系统突然断电或者重启之后, 下次启动的时候 Variable 区域中的 Variable 有可能处于这种中间状态。EDK2 还通过定义在 MdeModulePkg 中的 PCD(gEfiMdeModulePkg TokenSpaceGuid.PcdMaxVariableSize)来指定单个 Variable 的最大大小, 也就是说图 3-2 中所示的每个独立的 Variable 所占用的空间都不能超过这个指定的最大大小。

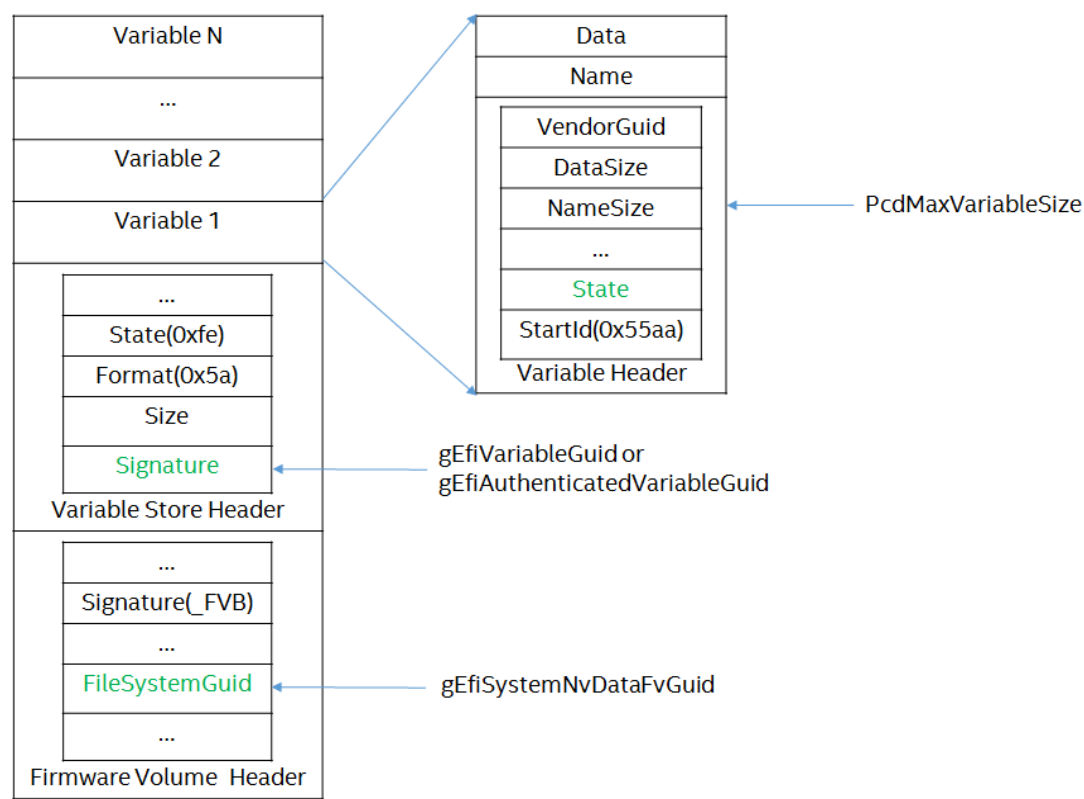


图 3-2 EDK2 NV Variable 存储区格式

Fig.3-2 The storage format of EDK2 NV Variable

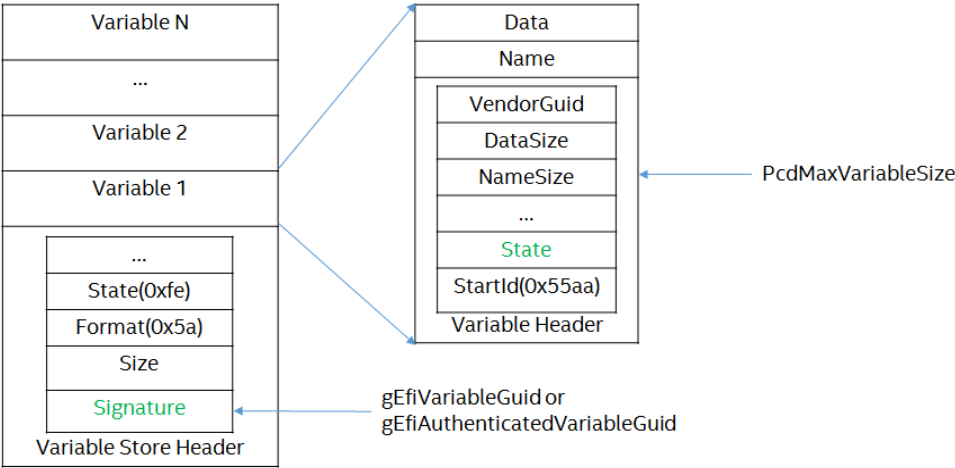


图 3-3 EDK2 Volatile Variable 存储区格式
Fig.3-3 The storage format of EDK2 Volatile Variable

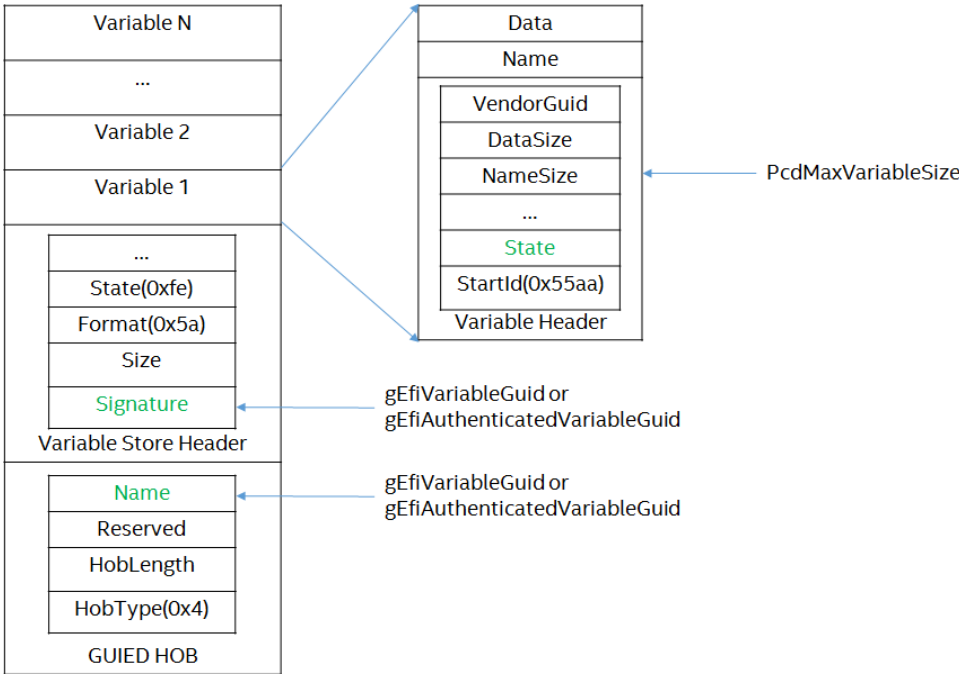


图 3-4 EDK2 NV Variable HOB 格式
Fig.3-4 The format of EDK2 NV Variable HOB

与图 3-2 NV Variable 存储区格式不同的是， 基于 EDK2 实现的 Volatile Variable 存储区（如图 3-3 所示）没有 Firmware Volume Header， DXE Variable 驱动会分配一块 gEfiMdeModulePkgTokenSpaceGuid.PcdVariableStoreSize 大小的 RAM 或者 SMRAM 来存取 Volatile Variable； NV Variable HOB（如图 3-4 所示）也没有 Firmware Volume Header， 相应的有 GUIDED HOB 头， 其中的 HobType 域为 0x4 表示 NV Variable HOB 为 GUIDED HOB 类型， Name 域与 Variable Store Header 中的 Signature 域相对应为 gEfiVariableGuid 或者 gEfiAuthenticatedVariable

Guid, NV Variable HOB 可能会在系统第一次启动或者需要恢复平台配置信息到默认值的时候被平台早期 PEI 模块建立。

```
MdePkg/Include/Pi/PiHob.h:[18][24]

#define EFI_HOB_TYPE_GUID_EXTENSION      0x0004

///
/// Describes the format and size of the data inside the HOB.
/// All HOBs must contain this generic HOB header.
///
typedef struct {
    ///
    /// Identifies the HOB data structure type.
    ///
    UINT16      HobType;
    ///
    /// The length in bytes of the HOB.
    ///
    UINT16      HobLength;
    ///
    /// This field must always be set to zero.
    ///
    UINT32      Reserved;
} EFI_HOB_GENERIC_HEADER;

typedef struct {
    ///
    /// The HOB generic header. Header.HobType = EFI_HOB_TYPE_GUID_EXTENSION.
    ///
    EFI_HOB_GENERIC_HEADER      Header;
    ///
    /// A GUID that defines the contents of this HOB.
    ///
    EFI_GUID                    Name;
    ///
    /// Guid specific data goes here
    ///
} EFI_HOB_GUID_TYPE;
```

3.3 Variable 模块

下面分阶段对基于 EDK2 实现的 Variable 模块进行分析。

3.3.1 DXE 阶段

基于 EDK2 实现的 DXE Variable 驱动会遵循 UEFI 和 PI 规范提供 UEFI Variable 运行时服务及产生 Variable 架构协议和 Variable Write 架构协议。如图 3-5 所示 EDK2 DXE Variable 驱动层次结构，有纯 DXE Variable 和 SMM Variable 两种驱动方案。

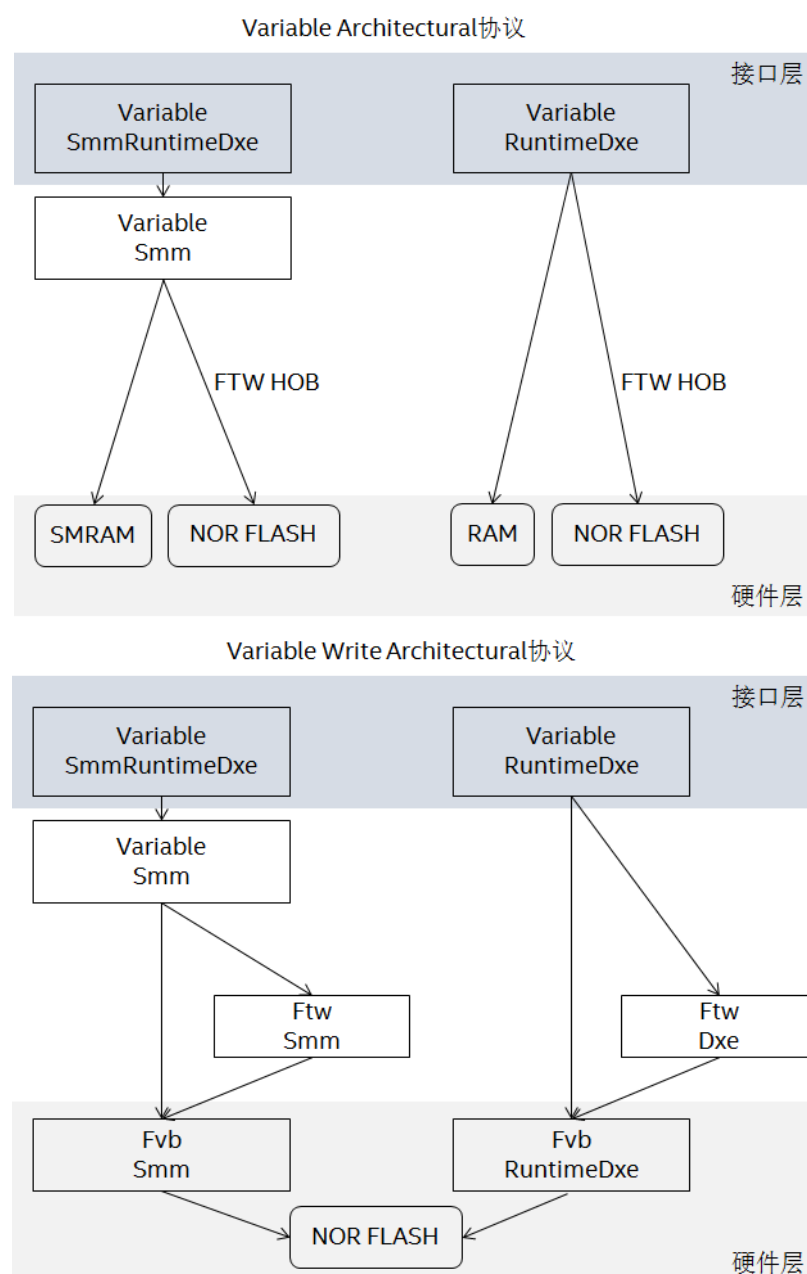


图 3-5 DXE Variable 驱动层次结构

Fig.3-5 DXE Variable driver hierarhacy structure

- 纯 DXE Variable 驱动方案：

VariableRuntimeDxe 驱动首先会初始化 UEFI 运行时服务表中的 GetVariable、

GetNextVariableName、SetVariable 和 QueryVariableInfo 域，产生 Variable 架构协议来通知其它 DXE 模块读取 NV Variable 和读写 Volatile Variable 的 UEFI Variable 运行时服务可用了。如果上次启动中在 NV Variable 空间回收的同时有错误的产生（如系统突然断电或者重启等），FTW HOB 可能会被用来访问 NV Variable 区域的备份以读取 NV Variable。

FvbRuntimeDxe 驱动会基于硬件 NOR FLASH 产生 FVB 协议，然后 FtwDxe 驱动会基于 FVB 协议产生 FTW 协议，最后 VariableRuntimeDxe 驱动会基于 FVB 协议和 FTW 协议产生 Variable Write 架构协议来通知其它 DXE 模块删除、添加或者更新 NV Variable 的 UEFI Variable 运行时服务可用了，VariableRuntimeDxe 驱动主要利用 FVB 协议来进行 NV Variable 写操作，利用 FTW 协议来帮助完成 NV Variable 空间回收。

- SMM Variable 驱动方案：

VariableSmm 驱动首先会产生 SMM Variable 协议和提供 SmmVariable Handler，如果上次启动中在 NV Variable 空间回收的同时有错误的产生（如系统突然断电或者重启等），FTW HOB 可能会被用来访问 NV Variable 区域的备份以读取 NV Variable。接着通过 SMM Communication 机制利用 VariableSmm 驱动提供的 SmmVariableHandler，VariableSmmRuntimeDxe 驱动会初始化 UEFI 运行时服务表中的 GetVariable、GetNextVariableName、SetVariable 和 QueryVariableInfo 域，产生 Variable 架构协议来通知其它 DXE 模块读取 NV Variable 和读写 Volatile Variable 的 UEFI Variable 运行时服务可用了。

FvbSmm 驱动会基于硬件 NOR FLASH 产生 SMM FVB 协议，然后 FtwSmm 驱动会基于 SMM FVB 协议产生 SMM FTW 协议，同样 VariableSmm 驱动主要利用 SMM FVB 协议来进行 Variable 写操作，利用 SMM FTW 协议来帮助完成 Variable 空间回收，最后 VariableSmmRuntimeDxe 驱动会通过 SMM Communication 机制利用 VariableSmm 驱动提供的 SmmVariableHandler 产生 Variable Write 架构协议来通知其它 DXE 模块删除、添加或者更新 NV Variable 的 UEFI Variable 运行时服务可用了。

DXE Variable 驱动会根据 Variable 的属性是 NV 或者 Volatile 从不同的 Variable 存储区域读写 Variable，如图 3-6 所示 DXE 阶段 Variable 存储区域。

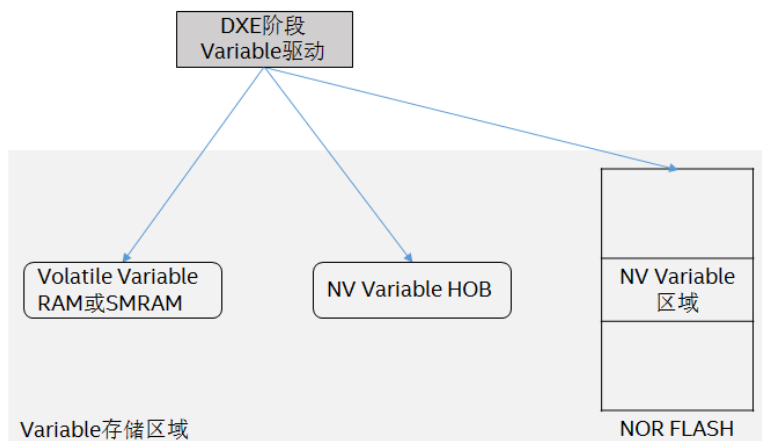


图 3-6 DXE 阶段 Variable 存储区域

Fig.3-6 DXE phase Variable regions

NV Variable HOB: 系统可能在第一次启动或者需要恢复平台配置信息到默认值的时候会建立 NV Variable HOB (第 4.2 节将会详细地分析 NV Variable HOB 的建立), 它相对 NV Variable 区域有更高的优先级。

NV Variable 区域: 如果 NV Variable HOB 不存在或者一个 NV Variable 从 NV Variable HOB 找不到, NV Variable 区域就会被用来查找及读写 NV Variable。

Volatile Variable RAM 或 SMRAM: 根据纯 DXE Variable 和 SMM Variable 两种驱动方案的不同, 一块 gEfiMdeModulePkgTokenSpaceGuid.PcdVariableStoreSize 大小的 RAM 或者 SMRAM 会被分配来存取 Volatile Variable。

另外由于从 NOR FLASH 读取数据的速度比从内存慢, DXE Variable 驱动会分配一块 RAM 或者 SMRAM 来缓存 NOR FLASH 中的 NV Variable 区域, 这块内存被称为 NV Variable Cache。当有 GetVariable 调用者想要读取某个 NV Variable 的时候, DXE Variable 驱动就可以从 NV Variable Cache 读取相应的 Variable, 这样单个 NV Variable 的读取速度就可以被加快, 而一般系统启动过程中可能会有很多次的 NV Variable (如 UEFI 规范定义的全局平台语言 Variable L"PlatformLang") 读取, 这样 NV Variable Cache 的使用就可以很好地减少一次系统启动过程中读取 NV Variable 所消耗的时间。

DXE Variable 驱动会根据调用者输入的 VendorGuid 和 VariableName 来在 Volatile Variable 区域/NV Variable HOB/NV Variable 区域找到相应 Variable 所在的位置, 进而返回 Variable 的属性和数据及数据大小给 GetVariable 调用者; 为 SetVariable 调用者, 根据输入的 Variable 属性和数据大小的不同, 删除、添加或者更新相应的 Variable。

根据第 1.1 节中说明的 NOR FLASH 的特性（每个 BIT 可以通过编程由 1 变为 0，但不可以由 0 修改为 1，如果需要把某个 BIT 由 0 修改为 1，在执行写操作前都要先执行擦除操作），Variable Header 中的 State 域被设计用来维持单个 Variable 更新过程中的原子性。单个 Variable 的更新流程如下：

- Step 1: 旧的 Variable 被标记成 VAR_IN_DELETED_TRANSITION State;
- Step 2: 紧接着之前最后一个 Variable 的后面写入新的 Variable Header，State 为 0xFF;
- Step 3: 新的 Variable 被标记成 VAR_HEADER_VALID_ONLY State;
- Step 4: 紧接着新的 Variable Header 后面写入新的 Variable 数据;
- Step 5: 新的 Variable 被标记成 VAR_ADDED State;
- Step 6: 旧的 Variable 被标记成 VAR_DELETED State。

根据以上说明的 Variable 更新流程来看，新的 Variable 并不能也不是写在原来旧的 Variable 所在的位置，而是写在之前最后一个 Variable 的后面，然后再把旧的 Variable 废弃掉。这样在系统发生很多次的 Variable 删除或者更新操作之后，Variable 区域中会存在着很多已经被废弃但还占用着 Variable 空间的旧的 Variable。接着如果又有新的更新 Variable 或者添加新的 Variable 操作，且 Variable 区域已经没有足够的剩余空间的时候，基于 EDK2 实现的 DXE Variable 驱动就会先对 Variable 空间进行重新整理（如图 3-7 所示），把已经被废弃了 Variable 剔除，然后则需要把新的重新整理过的 Variable 区域数据写回 Variable 区域以完成 Variable 空间回收。

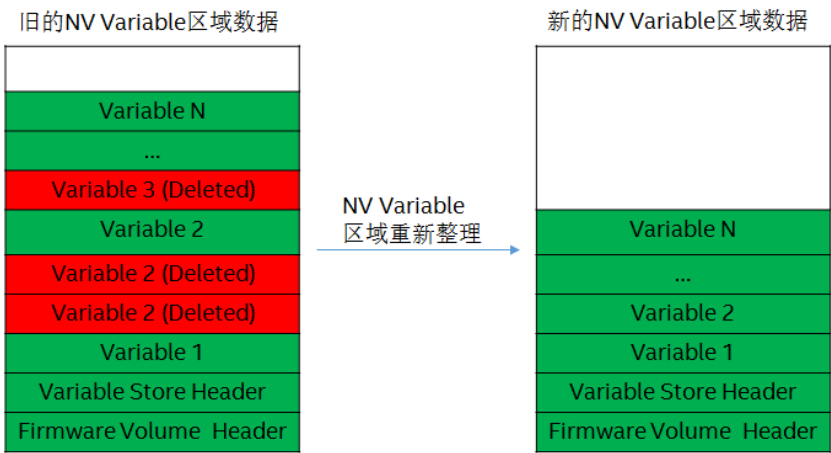


图 3-7 NV Variable 区域重新整理
Fig.3-7 NV Variable region rearrangement

另外在 ReadyToBoot 事件被触发的时候， DXE Variable 驱动注册在 ReadyToBoot 事件上的回调函数就会被调用， 这个回调函数会检查剩余的 NV Variable 空间， 如果剩余的 NV Variable 空间已经不足 gEfiMdeModulePkgTokenSpaceGuid.PcdMaxVariableSize 及一个最大 Variable 的大小了， 基于 EDK2 实现的 DXE Variable 驱动也会主动触发 NV Variable 区域重新整理以及进行 NV Variable 空间回收。

FtwDxe(Smm)驱动产生的(SMM) FTW 协议（特别是其中的 Write 接口）会被用来帮助完成 NV Variable 空间回收。 FTW 为 Fault Tolerant Write^[39]的简写，译为容错写， FtwDxe(Smm)驱动采用的容错写机制容许在写操作的同时有错误的产生（如系统突然断电或者重启等）， 一个空闲块被用来保证要么旧的要么新的目标块数据是有效的。 当然， 这个空闲块须足够大来备份要写入的整个目标块的数据。 同时为了实现容错写机制， 容错写模块需要记录容错写过程中的各种状态机， 具体的状态信息会记录在如图 3-8 所示灰色的容错写工作区间中。 通过记录在容错写工作区间中的状态信息， 在错误产生后下次系统启动的时候， 容错写模块就可以知道上次的容错写操作是否成功， 从而在 PEI 阶段和 DXE 早期阶段提供备份在空闲块中的目标块数据的具体信息给其他模块， 以及在 DXE 阶段把备份在空闲块中的目标块数据恢复到目标块中。 具体如图 3-8 所示抽象的 5 个步骤就是在以容错写的方式对 NOR FLASH 中的目标块(NV Variable 区域)进行写入数据操作。 如果错误发生在擦除目标块之前， 那很明显目标块里面的原始数据还是有效的； 如果错误发生在擦除目标块之后， 则新的目标块数据已经备份到空闲块中了。

Step 1: 读取目标块中的原始数据到内存 buffer 中；

Step 2: 用写入 buffer 中的数据替换内存 buffer 中要更新的部分；

Step 3: 写入更新过的内存 buffer 到空闲块中；

Step 4: 擦除目标块；

Step 5: 最后把空闲块中的数据写入到目标块中。

要注意一点， 基于 EDK2 实现的 DXE Variable 驱动不支持在操作系统运行时的 NV Variable 空间回收， 因为 NV Variable 空间回收需要对相应的 NOR FLASH 块进行擦除操作， 而 NOR FLASH 块擦除操作一般会很费时， 这将可

能会违反操作系统对自身被打断（特别是在 SMM 环境下）的最大时间限制，最终可能导致操作系统认为系统有故障而产生异常。

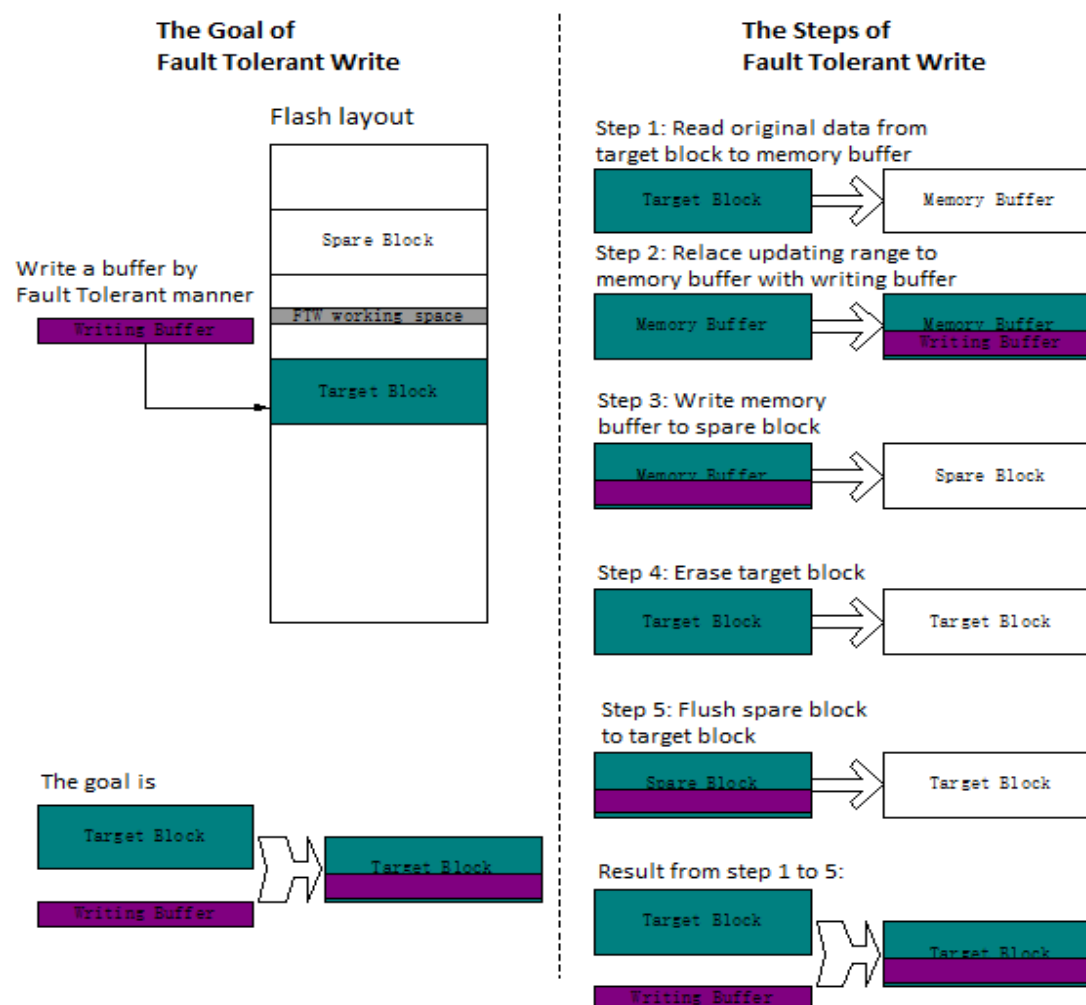


图 3-8 容错写基本原理图

Fig.3-8 The basic schematic diagram of fault tolerant write

3.3.2 PEI 阶段

基于 EDK2 实现的 PEI Variable 模块会遵循 PI 规范产生 Variable ReadOnly PPI。因为 Volatile Variable 在系统重启之后就会丢失，PEI Variable 模块及 Variable ReadOnly PPI 只提供对 NV Variable 的访问。如图 3-9 所示 EDK2 PEI Variable 模块层次结构，VariablePei 模块会产生 Variable ReadOnly PPI 来通知其它 PEI 模块读取 NV Variable 的 PEI Variable 服务可用了，如果上次启动中在 NV Variable 空间回收的同时有错误的产生（如系统突然断电或者重启等），FTW HOB 可能会被用来访问 NV Variable 区域的备份以读取 NV Variable。

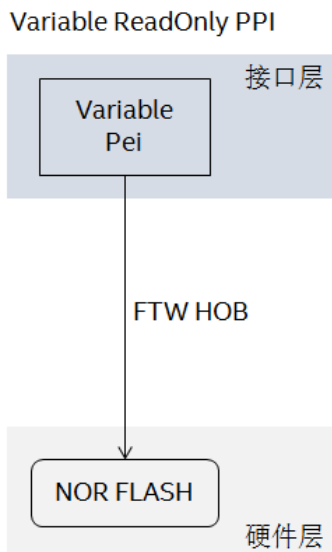


图 3-9 PEI Variable 模块层次结构

Fig.3-9 PEI Variable module hierarhacy structure

与 DXE 阶段 Variable 存储区域类似， PEI 阶段 Variable 存储区域如图 3-10 所示， 唯一的区别是 PEI 阶段没有 Volatile Variable 区域。

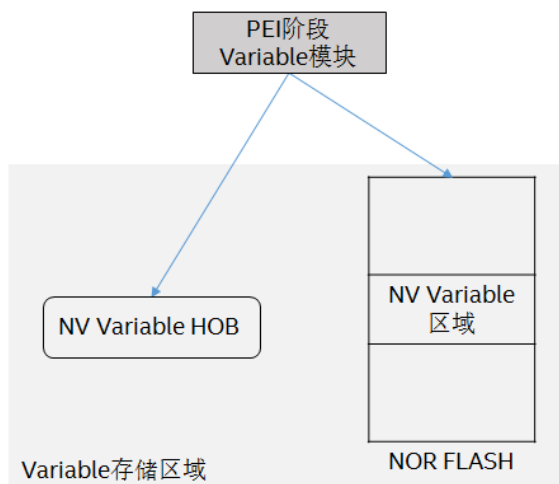


图 3-10 PEI 阶段 Variable 存储区域

Fig.3-10 PEI phase Variable regions

NV Variable HOB: 系统可能在第一次启动或者需要恢复平台配置信息到默认值的时候会建立 NV Variable HOB (第 4.2 节将会详细地分析 NV Variable HOB 的建立)， 它相对 NV Variable 区域有更高的优先级。

NV Variable 区域: 如果 NV Variable HOB 不存在或者一个 NV Variable 从 NV Variable HOB 找不到， NV Variable 区域就会被用来查找及读取 NV Variable。

PEI Variable 模块提供的 GetVariable 服务与 DXE Variable 驱动提供的 GetVariable 服务的功能也基本一样， 不同的是 PEI Variable 模块提供的

GetVariable 服务只提供对 NV Variable 的访问，另外因为 PEI 阶段没有足够多的内存来缓存 NOR FLASH 中的 NV Variable 区域，PEI Variable 模块会在查找 NV Variable 的时候为 NOR FLASH 中的 NV Variable 建立索引表(VARIABLE_INDEX_TABLE)来加快读取 NV Variable 的速度。

PEI Variable 模块会根据调用者输入的 VendorGuid 和 VariableName 来在 NV Variable HOB/NV Variable 区域找到相应 Variable 所在的位置，进而返回 Variable 的属性和数据及数据大小给 GetVariable 调用者。

3.4 本章小结

本章首先分析了 Variable 的相关接口，包括 UEFI Variable 运行时服务、Variable 架构协议和 Variable Write 架构协议、SMM Variable 协议和 Variable ReadOnly PPI，接着分析了基于 EDK2 实现的 Variable 存储区格式和 Variable 模块，DXE Variable 驱动会遵循 UEFI 和 PI 规范提供 UEFI Variable 运行时服务及产生 Variable 架构协议和 Variable Write 架构协议，SMM Variable 驱动会产生 SMM Variable 协议，PEI Variable 模块会遵循 PI 规范产生 Variable ReadOnly PPI。本章对我们理解 UEFI Variable 的相关原理非常重要，是接下来章节研究 UEFI Variable 保护和恢复机制的前提。

第四章 保护和恢复机制-UEFI 变量检查的提出

本章会详细地分析现有的保护和恢复机制-Variable Lock 和平台配置 Variable 默认值恢复，以及 Variable 接口的安全性。针对各种可能的恶意攻击，很多情况下现有的保护和恢复机制并不能被使用，因此我们需要提出新的保护和恢复机制-基于 EDK2 的 UEFI 变量检查来对 NV Variable 及 NV Variable 区域进行更好地保护。

4.1 Variable Lock

在分析 Variable Lock 之前，有必要先对 PI 规范定义的 EndOfDxe 事件做说明。

```
MdePkg\Include\Guid\EventGroup.h:[18]
#define EFI_END_OF_DXE_EVENT_GROUP_GUID \
    { 0x2ce967a, 0xdd7e, 0x4ffc, { 0x9e, 0xe7, 0x81, 0xc, 0xf0, 0x47, 0x8, 0x80 } }
```

根据 PI 规范卷 2 中的定义，从 SEC 到 EndOfDxe 事件被触发之前，系统中所有的组件都被认为是安全可信的。在第三方可扩展模块（如 UEFI 应用程序或者可扩展卡上的 UEFI 驱动）产生的 Driver Binding 被连接之前，PI 模块需要在 EndOfDxe 事件上注册回调函数，然后在 EndOfDxe 事件被触发的时候，回调函数就会被执行来锁定或者保护关键的系统资源（如有些重要的寄存器需要被锁定，然后这些寄存器将不能再被更改，除非系统重启）^[23]。EndOfDxe 事件的触发与其它事件（如 UEFI 规范定义的 ReadyToBoot 事件和 ExitBootServices 事件）类似，只是被触发的时间点不一样。EndOfDxe 事件被触发的时间点由平台决定，一般会在平台 BDS 通过连接控制器连接第三方可扩展模块产生的 Driver Binding 之前。

与那些关键的系统资源一样，一些关键的 Variable 也需要被锁定或者保护，如 TCG2 物理存在标志 Variable L"TCg2PhysicalPresenceFlags"，UEFI 规范定义的全局启动选项支持 Variable L"BootOptionSupport"、操作系统指示支持 Variable L"OsIndicationsSupported"、设置模式 Variable L"SetupMode"和安全启动 Variable L"SecureBoot"等。UEFI 规范定义了 Variable 的属性可以是 non-volatile (NV)、

boot services access (BS)、runtime access (RT)、hardware error record (HR)、count based authenticated write access、time based authenticated write access (AT)和 append write, 但没有只读(RO, Read Only)属性来标记一个 Variable 被锁定了。基于此, EDK2 实现了 Variable Lock 协议来支持把一些 Variable 锁定及标记为只读。

```
MdeModulePkg\Include\Protocol\VariableLock.h:[18] Variable Lock 协议
/// Variable Lock Protocol is related to EDK II-specific implementation of variables
/// and intended for use as a means to mark a variable read-only after the event
/// EFI_END_OF_DXE_EVENT_GUID is signaled.
struct _EDKII_VARIABLE_LOCK_PROTOCOL {
    EDKII_VARIABLE_LOCK_PROTOCOL_REQUEST_TO_LOCK RequestToLock;
};
```

Variable Lock 的工作流程如图 4-1 所示。

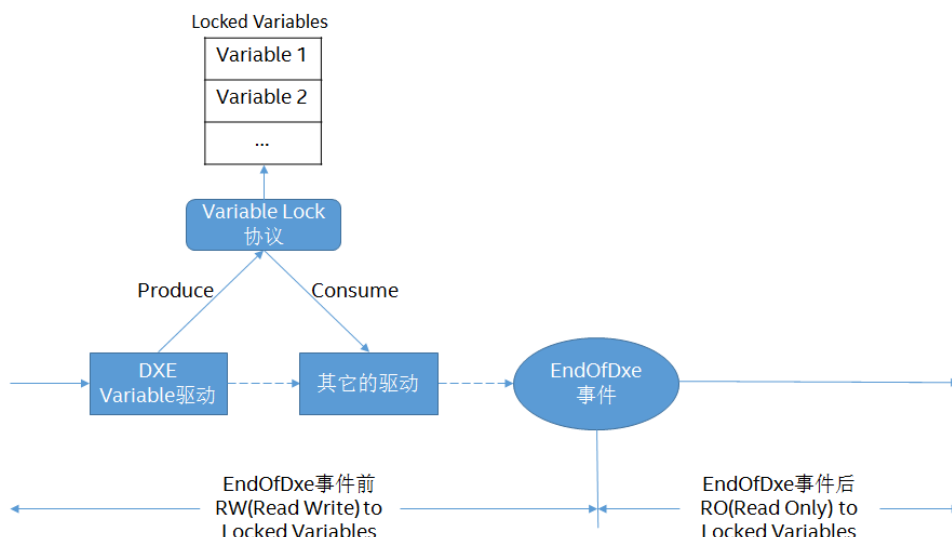


图 4-1 Variable Lock 的工作流程

Fig.4-1 The working flow of Variable Lock

首先 DXE Variable 驱动会产生 Variable Lock 协议和在 EndOfDxe 事件上注册一个回调函数。接着其它的驱动在 EndOfDxe 事件被触发之前就可以利用 Variable Lock 协议提供的 RequestToLock 接口来提交锁 Variable 请求, Variable 驱动及 Variable Lock 协议会接受请求和管理那些需要被锁住的 Variable, 并返回成功状态(EFI_SUCCESS)。然后在 EndOfDxe 事件被触发的时候, DXE Variable 驱动在 EndOfDxe 事件上注册的回调函数会被执行以知道 EndOfDxe 事件被触发了, 这样如果在 EndOfDxe 事件被触发之后还有其它的驱动使用

Variable Lock 协议提供的 RequestToLock 接口来提交锁 Variable 请求，DXE Variable 驱动及 Variable Lock 协议就会直接返回拒绝访问状态 (EFI_ACCESS_DENIED)。

DXE Variable 驱动及 Variable Lock 协议对那些需要被锁住的 Variable 采取的保护策略是：在 EndOfDxe 事件被触发之前，Variable Lock 协议从 RequestToLock 接口接受锁 Variable 请求，但锁 Variable 请求还不会起作用，也就是说那些需要被锁住的 Variable 在 EndOfDxe 事件被触发之前是可读可写(RW, Read Write)的。在 EndOfDxe 事件被触发之后，Variable Lock 协议将不再从 RequestToLock 接口接受锁 Variable 请求，之前接受的锁 Variable 请求会起作用，也就是说那些需要被锁住的 Variable 在 EndOfDxe 事件被触发之后就是只读的了，任何在 EndOfDxe 事件被触发之后通过 UEFI Variable 运行时服务的 SetVariable 接口对被锁住的 Variable 进行的删除、添加或者更新操作都会被拒绝并返回写保护状态(EFI_WRITE_PROTECTED)。

Variable Lock 在 SMM 环境下不起作用，因为 SMM 环境与从 SEC 到 EndOfDxe 事件被触发之前一样，被认为是安全可信的，也就是说通过 SMM Variable 协议的 SmmSetVariable 接口仍然可以在 EndOfDxe 事件被触发之后对被锁住的 Variable 进行删除、添加或者更新操作。

下面是如何利用 Variable Lock 协议提供的 RequestToLock 接口来提交锁 Variable 请求的代码实例^[18]。

```
EFI_STATUS                                Status;
EDKII_VARIABLE_LOCK_PROTOCOL              *VariableLockProtocol;
// This flags variable controls whether physical presence is required for TPM command.
// It should be protected from malicious software. We set it as read-only variable here.
// 查找Variable Lock协议
Status = gBS->LocateProtocol (&gEdkiiVariableLockProtocolGuid, NULL, (VOID
**)&VariableLockProtocol);
if (!EFI_ERROR (Status)) {
    Status = VariableLockProtocol->RequestToLock ( // 提交锁Variable请求
                                                    VariableLockProtocol,
                                                    TCG2_PHYSICAL_PRESENCE_FLAGS_VARIABLE,
                                                    &gEfiTcg2PhysicalPresenceGuid
                                                    );
    if (EFI_ERROR (Status)) {
        ...
    }
}
```

4.2 平台配置 Variable 默认值恢复

第 3.2 节中提到过 NV Variable HOB 可能会在系统第一次启动或者需要恢复平台配置信息到默认值的时候被建立，它相对 NV Variable 区域有更高的优先级。在系统第一次启动的时候，平台配置 Variable 如果不存在，NV Variable HOB 可能会被建立，然后 PEI Variable 模块和 DXE Variable 驱动就可以利用 NV Variable HOB 给其它模块提供平台配置信息的默认值。但 NV Variable HOB 是如何被建立的呢？在第二次及接下来启动的时候，平台固件如何判断什么时候需要恢复平台配置信息到默认值呢？下面进行详细的分析。

在分析平台配置 Variable 默认值的恢复流程之前，需要先分析平台配置 Variable 的默认值是如何生成的。

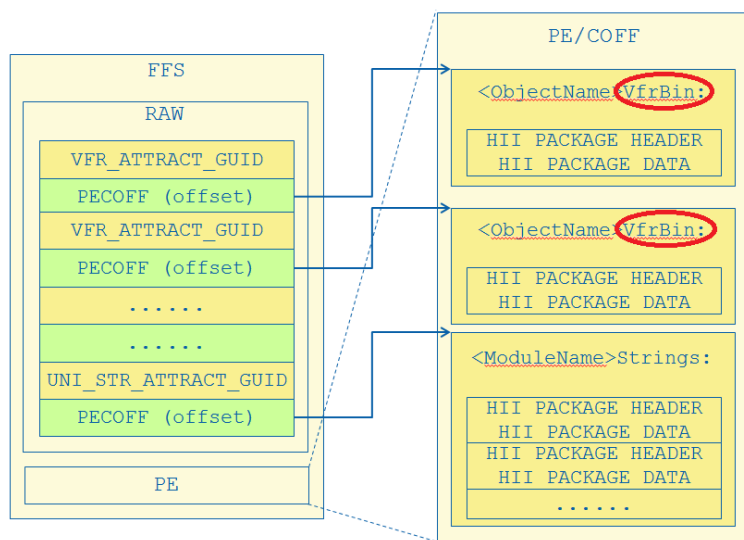


图 4-2 FFS 中的 VFR 二进制数据

Fig.4-2 The VFR binary data in FFS

如图 4-2 所示 EDK2 生成的 FFS 中的 VFR 二进制数据，EDK2 提供的 VfrCompile^[18]工具会把 VFR(Visual Forms Representation)编译生成 VFR 二进制数据(IFR, Internal Forms Representation)，VFR 二进制数据会被包含在 PE^[40] SECTION 中，同时 EDK2 的编译工具会为每个包含 VFR 的驱动生成一个 RAW SECTION，在这个 RAW SECTION 中通过 VFR_ATTRACT_GUID({ 0xd0bc7cb4, 0x6a47, 0x495f, { 0xaa, 0x11, 0x71, 0x7, 0x46, 0xda, 0x6, 0xa2 } })可以定位到 VFR 二进制数据在 PE SECTION 中的偏移量。英特尔提供的固件配置编辑器 (FCE, Firmware Configuration Editor, <http://firmware.intel.com/develop>)^[41]可以

根据 UEFI 规范 HII(Human Interface Infrastructure)部分定义的 Forms 包(package)和操作码(opcode)及相应数据格式解析 FFS 中的 VFR 二进制数据来生成 Variable 默认值。

UEFI 规范 HII 部分定义的 Forms 包(package)和操作码(opcode)如下:

```
MdePkg\Include\Uefi\UefiInternalFormRepresentation.h:[17][18]
#define EFI_HII_PACKAGE_FORMS                0x02
#define EFI_IFR_VARSTORE_EFI_OP              0x26
#define EFI_IFR_ONE_OF_OP                     0x05
#define EFI_IFR_CHECKBOX_OP                   0x06
#define EFI_IFR_NUMERIC_OP                     0x07
#define EFI_IFR_ORDERED_LIST_OP               0x23
```

HII Forms 包中的一个 EfiVarStore 操作码会定义一个 Variable 及相应的 Guid、Name、属性和数据大小，引用这个 EfiVarStore 操作码(通过 EFI_VARSTORE_ID VarStoreId)的 OneOf 操作码、Checkbox 操作码、Numeric 操作码和 OrderedList 操作码则会定义相应 Variable 偏移量的有效值及默认值和提供对应于 UEFI BIOS 的 SETUP 交互界面中的一个配置选项供用户修改。如下例子中的 EfiVarStore 操作码定义了一个 Guid 为 DRIVER_SAMPLE_FORMSET_GUID、Name 为 MyEfiVar、属性为 BS+NV 和数据大小为 sizeof(MY_EFI_VARSTORE_DATA)的 Variable，Numeric 操作码则引用了 MyEfiVar Variable 的 SubmittedCallback 域，定义其为数值型，最小值为 0、最大值为 255 和默认值为 18。

```
MdeModulePkg\Universal\DriverSampleDxe\Vfr.vfr:[18]
// Define a EFI variable Storage (EFI_IFR_VARSTORE_EFI)
efivarstore MY_EFI_VARSTORE_DATA,
    attribute = EFI_VARIABLE_BOOTSERVICE_ACCESS | EFI_VARIABLE_NON_VOLATILE, // EFI
variable attributes
    name = MyEfiVar,
    guid = DRIVER_SAMPLE_FORMSET_GUID;
    numeric varid = MyEfiVar.SubmittedCallback,
        questionid = 0x1250,
        prompt = STRING_TOKEN(STR_SUBMITTED_CALLBACK_TEST_PROMPT),
        help = STRING_TOKEN(STR_SUBMITTED_CALLBACK_TEST_HELP),
        flags = INTERACTIVE,
        minimum = 0,
        maximum = 255,
        default = 18,
endnumeric;
```

一个平台的平台配置驱动会包含一些 VFR 来为 CPU 和芯片组等平台硬件、ACPI、安全和启动等提供平台配置选项，固件配置编辑器就可以解析这个驱动相应 FFS 中的 VFR 二进制数据生成平台配置 Variable 的默认值。

固件配置编辑器有两种运行模式：普通模式和多平台模式^[41]。

普通模式：解析 FFS 中的 VFR 二进制数据生成的 Variable 默认值会被直接放到 NV Variable 区域中。这种模式并不能被用于平台配置 Variable 默认值恢复机制，而只能在系统第一次启动的时候提供平台配置信息的默认值，因为被直接放到 NV Variable 区域中的 Variable 默认值可能会在启动的时候被更改。

多平台模式：根据 VFR 中描述的 PlatformId 和 DefaultId 来解析 FFS 中的 VFR 二进制数据生成的多平台 Variable 默认值会被放在一个新的 FREEFORM 类型的 FFS 中。这种模式可以被用于平台配置 Variable 默认值恢复机制，因为被放在一个新的 FREEFORM 类型的 FFS 中的多平台 Variable 默认值可以在需要的时候被用于恢复平台配置信息到默认值。

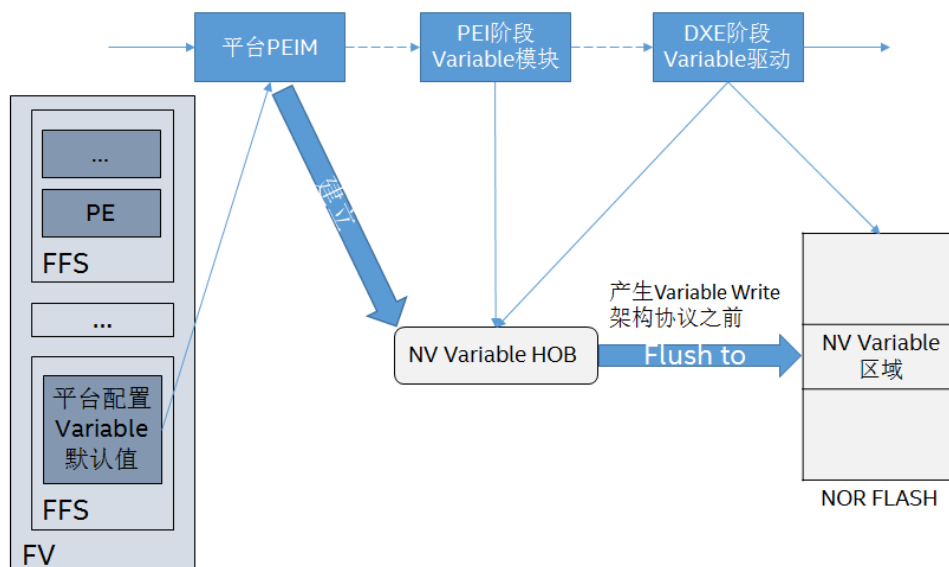


图 4-3 平台配置 Variable 默认值的恢复流程

Fig.4-3 The recovery flow of platform Setup Variable default value

平台配置 Variable 默认值的恢复流程如图 4-3 所示，首先平台早期 PEI 模块会从 FV 中找到存放多平台 Variable 默认值的 FREEFORM 类型的 FFS，解析多平台 Variable 默认值，找到匹配当前 PlatformId 和 DefaultId 的平台配置 Variable 默认值并建立 NV Variable HOB，接着 PEI Variable 模块和 DXE Variable 驱动早期阶段就可以从 NV Variable HOB 读取平台配置 Variable 默认值，到 DXE Variable 驱动准备产生 Variable Write 架构协议之前，DXE Variable 驱动会把 NV

Variable HOB 中的平台配置 Variable 默认值 Flush 到 NV Variable 区域，至此平台配置 Variable 默认值恢复完成，NV Variable HOB 就可以被废弃了。

平台早期 PEI 模块如何判断什么时候需要恢复平台配置信息到默认值呢？以下几种方法可以被使用：

- 1) 在系统第一次启动的时候，平台配置 Variable 不存在。
- 2) 平台配置 Variable 虽然存在，但其属性和数据大小却与预期不一致。
- 3) CMOS 掉电了或者被清掉了。
- 4) 特殊的通用输入输出(GPIO, General Purpose Input/Output)信号。

4.3 Variable 接口安全性分析

因为 PI 规范卷 1 定义及 PEI Variable 模块产生的 Variable ReadOnly PPI 只提供了读取 Variable 的功能，所以不用担心其它的 PEI 模块会利用 Variable ReadOnly PPI 对 Variable 及 Variable 区域进行破坏。另外 SMM Variable 驱动会在 SMM 环境下产生 SMM Variable 协议，因为 SMM 环境被认为是安全可信的，所以也不用担心其它的 SMM 驱动会利用 SMM Variable 协议对 Variable 及 Variable 区域进行破坏。比较需要担心的是 UEFI 规范定义的 Variable 运行时服务，DXE Variable 驱动会遵循 UEFI 和 PI 规范提供 UEFI Variable 运行时服务及产生 Variable 架构协议和 Variable Write 架构协议，UEFI Variable 运行时服务在操作系统运行时之前和操作系统运行时都是可用的。

根据 PI 规范卷 2 的定义，从 SEC 到 EndOfDxe 事件被触发之前，系统中所有的组件都被认为是安全可信的，也就是说 DXE 阶段在 EndOfDxe 事件被触发之前通过 UEFI Variable 运行时服务的 SetVariable 接口对 Variable 进行的删除、添加或者更新操作都是可信的。

Volatile Variable 在系统重启之后就会丢失，系统每次启动的时候都会重新创建 Volatile Variable，因此恶意程序对 Volatile Variable 的攻击不会对系统产生多大的影响。而 NV Variable 的非易失性和 UEFI Variable 服务的运行时特性则使得存储重要信息的 NV Variable 及 NV Variable 区域易受恶意程序的攻击。

通过以上的分析，我们可以知道恶意程序可能会在 EndOfDxe 事件被触发之后，特别是在进入操作系统运行时阶段之后，利用 UEFI Variable 运行时服务的 SetVariable 接口对 NV Variable 及 NV Variable 区域进行攻击。

4.4 UEFI 变量检查的提出

第 4.3 节分析出恶意程序可能会在 EndOfDxe 事件被触发之后，特别是在进入操作系统运行时阶段之后，利用 UEFI Variable 运行时服务的 SetVariable 接口对 NV Variable 及 NV Variable 区域进行攻击。



图 4-4 Windows 启动选项
Fig.4-4 Windows boot option

有些关键的 Variable 可以利用 Variable Lock 进行锁定，但很多 Variable 还需要在 EndOfDxe 事件被触发之后被创建和更新，如 UEFI 规范定义的全局启动选项 Variable L"Boot####"（操作系统会创建自己的启动选项 Variable，如图 4-4 所示 Windows 启动选项）和平台配置 Variable（用户需要通过 UEFI BIOS 的 SETUP 交互界面对平台配置驱动提供的配置选项进行修改）等，Variable Lock 就不能应用于这些 Variable。平台配置 Variable 默认值恢复机制可以被用于恢复平台配置信息到默认值，但很多其他的 Variable 并没有对应的默认值，另外默认值的恢复也就意味之前设定的丢失，所以它并不能被随意使用，而只能在系统启动出现异常的时候被用来使系统恢复正常启动。

那如何在 EndOfDxe 事件被触发之后抵御恶意程序对 NV Variable 及 NV Variable 区域的攻击呢？从 SetVariable 接口的定义，我们有了如图 4-5 所示的 UEFI 变量检查的基本思路及 UEFI 变量检查的提出，如果从 UEFI Variable 运行

时服务的 SetVariable 接口传入的 Variable 属性、数据和数据大小是非法的，这些 SetVariable 操作应该被拒绝并返回错误状态，为此我们可以在 SetVariable 把 Variable 内容写入 NV Variable 空间之前对 Variable 属性、数据和数据大小进行合法性检查，这里我们统称这些合法性检查为 UEFI 变量检查。

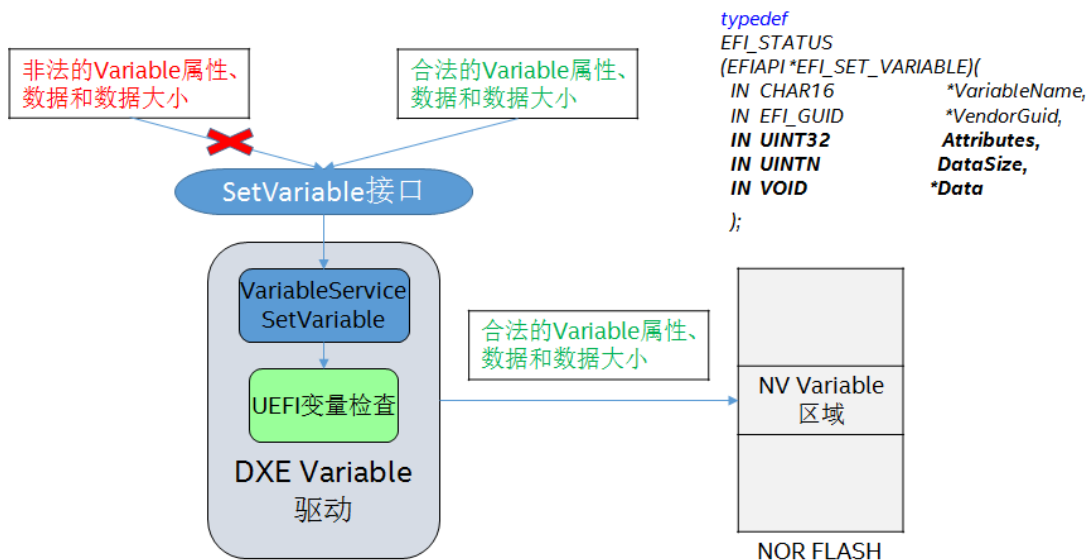


图 4-5 UEFI 变量检查的基本思路

Fig.4-5 The basic idea of UEFI Variable Check

那如何在 SetVariable 把 Variable 内容写入 NV Variable 空间之前对 Variable 的属性、数据和数据大小进行合法性检查呢？下面是各种具体的检查方法。

4.4.1 属性和数据大小检查的提出

根据 UEFI 规范的定义，一旦 ExitBootServices 接口被调用及进入操作系统运行时阶段之后，没有 RT 属性的 Variable 将不可见，只有包括 NV+RT 属性的 Variable 可以被 UEFI Variable 运行时服务的 SetVariable 接口删除、添加或者更新。如果一个已经存在的 Variable 被用不同的 Variable 属性通过 UEFI Variable 运行时服务的 SetVariable 接口更新，更新会被拒绝并返回无效的参数状态 (EFI_INVALID_PARAMETER)。

虽然通过 SetVariable 接口直接对已经存在的 Variable 的属性更新会被拒绝并返回无效的参数状态，但如果没有 Variable Lock 的保护，恶意程序仍然可以通过先删除 Variable 再添加 Variable 来达到篡改 Variable 属性的目的，这会导致接下来用正确的 Variable 属性进行正常的 Variable 更新会被拒绝并返回无效的参数状态。如果没有 Variable Lock 的保护，恶意程序还可以篡改 Variable 的数据

大小，这会使得 Variable 的消费者得不到 Variable 的一部分的数据或者得到一些错误的数据。

一些已知的 Variable 的属性是确定的，数据大小或者大小范围也是确定的，如 L"MemoryTypeInfoInformation" Variable 的属性必须是 NV+BS，大小必须在 sizeof (EFI_MEMORY_TYPE_INFORMATION)和 sizeof (EFI_MEMORY_TYPE_INFORMATION) * (EfiMaxMemoryType + 1)之间。UEFI 规范定义的全局 Variable 和 Image 安全数据库 Variable 的属性是确定的，一些全局 Variable 的数据大小或者大小范围也是确定的，如超时 Variable L"Timeout"的属性必须是 NV+BS+RT，大小必须是 sizeof (UINT16)。据此，我们有了 Variable 属性和数据大小检查的提出，如果有恶意程序想要通过 SetVariable 接口对这些 Variable 的属性或者数据大小进行篡改，这些 SetVariable 操作应该被拒绝并返回错误状态。

4.4.2 数据检查的提出

如果没有 Variable Lock 的保护，恶意程序可以随意地篡改 Variable 的数据，这会使得 Variable 的消费者得到完全错误的数据。

有些 Variable 的数据格式是确定的，另外有些 Variable 的有效值或者域的有效值是确定的。据此，我们有了 Variable 数据检查的提出，如果有恶意程序想要通过 SetVariable 接口对这些 Variable 的数据进行篡改，这些 SetVariable 操作应该被拒绝并返回错误状态。根据检查方法的不同，Variable 数据检查分为语法检查和语义检查。

4.4.2.1 语法检查

这里说的语法是指 Variable 的数据格式，UEFI 规范定义了一些全局 Variable 的数据格式，如控制台输出 Variable 的数据格式必须是 EFI_DEVICE_PATH_PROTOCOL，DevicePathLib 库类的 IsDevicePathValid 接口可以被利用来对控制台输出 Variable 的数据格式进行检查，我们称这种针对 Variable 数据格式进行的检查为语法检查。

4.4.2.2 语义检查

这里说的语义是指 Variable 的值，一个平台的平台配置驱动会包含一些 VFR 来为 CPU 和芯片组等平台硬件、ACPI、安全和启动等提供平台配置选项，如 UsbXhciSupport 配置选项，这个选项会对应于一个 CheckBox 操作码，这个

CheckBox 操作码会引用 EfiVarStore 操作码定义的平台配置 Variable 的 UsbXhciSupport 域，由于 CheckBox 操作码的值必须是 0 或者 1，那大于 1 的值对这个选项及平台配置 Variable 的 UsbXhciSupport 域来说就都是无效的，这些有效值信息可以被收集并被利用来对这些 Variable 的值进行检查，我们称这种针对 Variable 值进行的检查为语义检查。

4.4.3 配额管理的提出

系统的资源是有限的，包括 NV Variable 空间。恶意程序可以通过 DoS 攻击（DoS 是 Denial of Service 的简称，即拒绝服务，造成 DoS 的攻击行为被称为 DoS 攻击，其目的是使计算机或网络无法提供正常的服务，最常见的 DoS 攻击有计算机文件系统空间容量、网络带宽攻击和连通性攻击）^[4]，创建一个特别大的或者多个 NV Variable 来耗尽 NV Variable 空间，这会导致其它需要使用 NV Variable 的消费者没有剩余的 NV Variable 空间可用，DXE Variable 驱动提供的 UEFI Variable 运行时服务的 SetVariable 接口因为 NV Variable 空间不足就只能拒绝服务而返回资源不足状态(EFI_OUT_OF_RESOURCES)。

如果抵御这种针对 NV Variable 区域的 DoS 攻击呢？我们想到现代操作系统的磁盘空间配额管理方法，如 Windows 操作系统的磁盘空间配额管理可以为管理员组 and 用户组设置不同的磁盘空间限制，据此我们有了 NV Variable 空间配额管理的提出，把 NV Variable 空间划分为系统 Variable 空间和用户 Variable 空间，系统 Variable 空间包含所有系统启动比较关键的 Variable，用户 Variable 空间的大小可以被限制来保证关键的系统 Variable 有足够的空间可用。

4.5 本章小结

本章详细地分析了现有的保护和恢复机制-Variable Lock 和平台配置 Variable 默认值恢复，以及 Variable 接口的安全性。针对各种可能的恶意攻击，很多情况下现有的保护和恢复机制并不能被使用，我们提出了新的保护和恢复机制-基于 EDK2 的 UEFI 变量检查及各种变量检查方法来保证从 UEFI Variable 运行时服务的 SetVariable 接口传入的 Variable 属性、数据和数据大小的合法性，从而更好地对 NV Variable 及 NV Variable 区域进行保护。下一章将会具体地研究和实现新的保护和恢复机制-UEFI 变量检查。

第五章 保护和恢复机制-UEFI 变量检查

在第四章中，为了更好地对 NV Variable 及 NV Variable 区域进行保护，我们分析提出了新的保护和恢复机制-基于 EDK2 的 UEFI 变量检查，它可以利用 UEFI 和 PI 规范定义的访问 Variable 的各种服务及接口和 Variable 区域及 Variable 本身的特性，借鉴现代操作系统的磁盘空间配额管理方法，实现变量检查库和变量检查协议，通过属性检查、数据大小检查、语法检查、语义检查、配额管理、错误记录和恢复，来保证从 UEFI Variable 运行时服务的 SetVariable 接口传入的 Variable 属性、数据和数据大小的合法性。下面首先会研究变量检查模型的建立，接着会分别研究基于 UEFI、基于 HII 和基于 PCD 的变量检查以及配额管理，然后利用 NT32 模拟开发平台对新的保护和恢复机制-UEFI 变量检查进行验证测试，最后会简要地说明一个平台如何应用 UEFI 变量检查。

5.1 变量检查模型的建立

综合考虑各模块及库的接口需求以及保证各模块及库之前的互操作性和 UEFI 变量检查的可扩展性，我们可以建立如图 5-1 所示的 UEFI 变量检查模型。

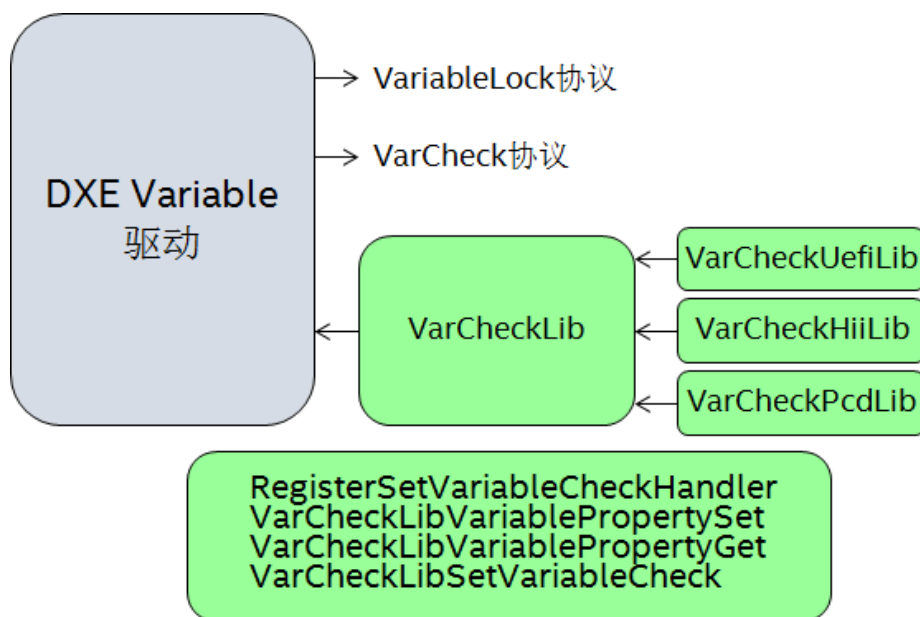


图 5-1 UEFI 变量检查模型

Fig.5-1 The model of UEFI Variable Check

变量检查库作为变量检查的管理者与 DXE Variable 驱动链接，DXE Variable

驱动基于变量检查库产生变量检查协议来给其它模块使用。 NULL 库实例（如 VarCheckUefiLib、VarCheckHiiLib 和 VarCheckPcdLib）可以直接与变量检查库链接使用变量检查接口，其它模块则可以利用变量检查协议来使用变量检查接口。

```
struct _EDKII_VAR_CHECK_PROTOCOL { // Variable Check 协议
    EDKII_VAR_CHECK_REGISTER_SET_VARIABLE_CHECK_HANDLER
    RegisterSetVariableCheckHandler;
    EDKII_VAR_CHECK_VARIABLE_PROPERTY_SET          VariablePropertySet;
    EDKII_VAR_CHECK_VARIABLE_PROPERTY_GET          VariablePropertyGet;
};
```

变量检查库需要定义 SetVariable 检查接口被 DXE Variable 驱动利用在 SetVariable 把 Variable 内容写入 NV Variable 空间之前进行变量检查，变量检查会基于从变量特性设置接口来的变量特性和注册 SetVariable 检查处理程序接口来的检查处理程序进行，以保证从 SetVariable 接口传入的 Variable 属性、数据和数据大小的合法性。

```
EFI_STATUS
EFIAPI
VarCheckLibSetVariableCheck ( // SetVariable 检查接口
    IN CHAR16                *VariableName,
    IN EFI_GUID               *VendorGuid,
    IN UINT32                 Attributes,
    IN UINTN                  DataSize,
    IN VOID                   *Data,
    IN VAR_CHECK_REQUEST_SOURCE RequestSource
);
```

变量检查的策略与 Variable Lock 一致，变量特性设置接口和注册 SetVariable 检查处理程序接口在 EndOfDxe 事件被触发之前接受设置变量特性和注册 SetVariable 检查处理程序请求，在 EndOfDxe 事件被触发之后，SetVariable 检查接口会基于会从变量特性设置接口来的变量特性和注册 SetVariable 检查处理程序接口来的检查处理程序进行变量检查，变量特性获得接口会被利用来对 NV Variable 空间进行配额管理及错误记录和恢复。

5.1.1 针对 Variable 的属性和数据大小的篡改

我们可以对特定的 Variable 进行属性检查和数据大小检查，为此需要定义变量特性结构体和变量特性设置接口。

```

#define VAR_CHECK_VARIABLE_PROPERTY_REVISION    0x0001
#define VAR_CHECK_VARIABLE_PROPERTY_READ_ONLY    BIT0
typedef struct {
    UINT16                Revision;
    UINT16                Property;
    UINT32                Attributes;
    UINTN                 MinSize;
    UINTN                 MaxSize;
} VAR_CHECK_VARIABLE_PROPERTY; // 变量特性结构体

EFI_STATUS
EFIAPI
VarCheckLibVariablePropertySet ( // 变量特性设置接口
    IN CHAR16                *Name,
    IN EFI_GUID               *Guid,
    IN VAR_CHECK_VARIABLE_PROPERTY *VariableProperty
);

```

变量特性结构体有以下域：

Revision: 变量特性结构体的版本号，便于结构体以后的兼容扩展。

Attributes: UEFI 规范定义的 Variable 属性(BS/RT/NV/...)。

Property: 非 UEFI 规范定义的 Variable 属性(RO)，标记从变量特性设置接口或 Variable Lock 协议来的锁 Variable 请求。

MinSize: 允许的 Variable 数据的最小大小。

MaxSize: 允许的 Variable 数据的最大大小。

一个平台可以利用变量特性设置接口来为一些已知的 Variable 设置变量特性，如表 1 所示，BDS 驱动产生的 L"MemoryTypeInformation" Variable 的属性必须是 NV+BS，大小必须在 sizeof (EFI_MEMORY_TYPE_INFORMATION)和 sizeof (EFI_MEMORY_TYPE_INFORMATION) * (EfiMaxMemoryType + 1)之间；FirmwarePerformanceDxe 驱动产生的 L"FirmwarePerformance" Variable 的属性必须是 NV+BS，大小必须是 sizeof (FIRMWARE_PERFORMANCE_VARIABLE)；MonotonicCounterRuntimeDxe 驱动产生的 L"MTC" Variable 的属性必须是 NV+BS+RT，大小必须是 sizeof (UINT32)；PcatRealTimeClockRuntimeDxe 驱动产生的 L"RTC" Variable 的属性必须是 NV+BS+RT，大小必须是 sizeof (UINT32)；PcatRealTimeClockRuntimeDxe 驱动产生的 L"RTCALARM" Variable

的属性必须是 NV+BS+RT，大小必须是 sizeof (EFI_TIME)^[18]。

表 5-1 一些已知的 Variable 的变量特性
Table5-1 The variable property of some known variables

Name	属性	最小大小	最大大小
L"MemoryTypeInformation"	NV+BS	sizeof (EFI_MEMORY_TYPE_INFORMATION)	sizeof (EFI_MEMORY_TYPE_INFORMATION) * (EfiMaxMemoryType + 1)
L"FirmwarePerformance"	NV+BS	sizeof (FIRMWARE_PERFORMANCE_VARIABLE)	sizeof (FIRMWARE_PERFORMANCE_VARIABLE)
L"MTC"	NV+BS+RT	sizeof (UINT32)	sizeof (UINT32)
L"RTC"	NV+BS+RT	sizeof (UINT32)	sizeof (UINT32)
L"RTCALARM"	NV+BS+RT	sizeof (EFI_TIME)	sizeof (EFI_TIME)

UEFI 规范定义的全局 Variable 和 Image 安全数据库 Variable 的属性是确定的，一些全局 Variable 的数据大小或者大小范围也是确定的，变量特性设置接口可以被利用来为这些 Variable 设置变量特性。

变量特性结构体的 Property 域提供了只读属性，因此 Variable Lock 协议也可以基于变量检查库产生，这样原来的 Variable Lock 协议实现就可以被简化了。

5.1.2 针对 Variable 的数据的篡改

我们可以对 UEFI 规范定义的一些有特定数据格式的全局 Variable 进行语法检查，我们还可以利用 HII 和 PCD 信息对平台配置 Variable 和 DynamicHii 或者 DynamicExHii 类型 PCD 的相应 Variable 进行语义检查，为此需要定义注册 SetVariable 检查处理程序接口。

```
typedef EFI_SET_VARIABLE VAR_CHECK_SET_VARIABLE_CHECK_HANDLER;

EFI_STATUS
EFIAPI
VarCheckLibRegisterSetVariableCheckHandler ( // 注册SetVariable检查处理程序接口
    IN VAR_CHECK_SET_VARIABLE_CHECK_HANDLER    Handler
);
```

5.1.3 针对 DoS 攻击

借鉴现代操作系统的磁盘空间配额管理方法，我们可以对 NV Variable 空间进行配额管理及错误记录和恢复，为此需要定义变量特性获得接口，DXE Variable 驱动利用这个接口对 NV Variable 空间进行配额管理及错误记录，平台

变量清理库(PlatformVarCleanupLib)利用这个接口对 NV Variable 空间进行清理及恢复。

```
EFI_STATUS
EFI_API
VarCheckLibVariablePropertyGet ( // 变量特性获得接口
    IN CHAR16                      *Name,
    IN EFI_GUID                    *Guid,
    OUT VAR_CHECK_VARIABLE_PROPERTY *VariableProperty
);
```

5.2 基于 UEFI 的变量检查

为了抵御针对 UEFI 规范定义的全局 Variable 和 Image 安全数据库 Variable 等的属性、 数据大小或者数据的篡改， NULL 库实例 VarCheckUefiLib 可以实现基于 UEFI 的变量检查， 如图 5.2 所示。

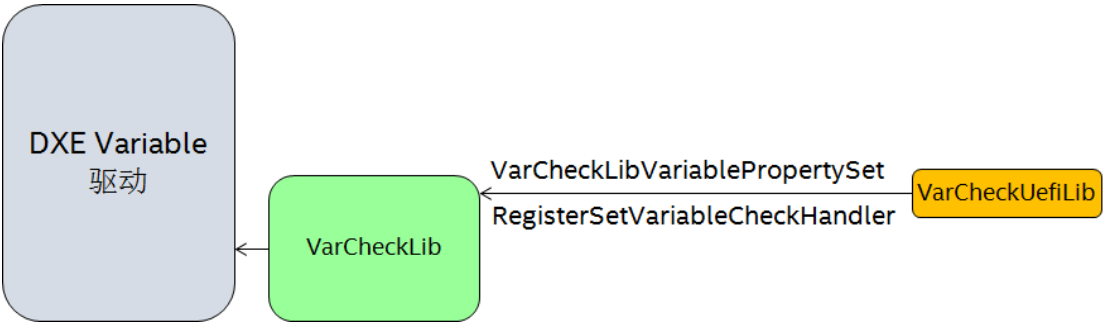


图 5-2 VarCheckUefiLib
Fig.5-2 VarCheckUefiLib

UEFI 规范定义的全局 Variable 和 Image 安全数据库 Variable 等的属性是确定的， 一些全局 Variable 的数据大小或者大小范围也是确定的， 如启动选项 Variable L"Boot#####"、 驱动选项 Variable L"Driver#####"、 键选项 Variable L"Key#####"、 启动选项顺序 Variable L"BootOrder"、 驱动选项顺序 Variable L"DriverOrder"、启动选项下次 Variable L"BootNext"、超时 Variable L"Timeout"、控制台输入 Variable L"ConIn"、 控制台输出 Variable L"ConOut"、 错误输出 Variable L"ErrOut"、 操作系统指示 Variable L"OsIndications"和平台语言 Variable L"PlatformLang"等的属性都必须是 NV+BS+RT， Image 安全数据库 Variable 的属性必须是 NV+BS+RT+AT， 启动选项下次 Variable 和超时 Variable 的大小必须是 sizeof(UINT16)， 操作系统指示 Variable 的大小必须是 sizeof(UINT64)^[17]， NULL 库实例 VarCheckUefiLib 可以利用变量特性设置接口设置这些 Variable 的

变量特性。

UEFI 规范还定义了一些全局 Variable 的数据格式，如启动选项 Variable 和驱动选项 Variable 的数据格式必须是 EFI_LOAD_OPTION，键选项 Variable 的数据格式必须是 EFI_KEY_OPTION，启动选项顺序 Variable 和驱动选项顺序 Variable 的大小必须是 sizeof(UINT16)的倍数，控制台输入 Variable、控制台输出 Variable 和错误输出 Variable 的数据格式必须是 EFI_DEVICE_PATH_PROTOCOL(可以利用 DevicePathLib 库类的 IsDevicePathValid 接口来对 Variable 的数据格式进行检查)，平台语言 Variable 的数据格式必须是 ASCII 字符串(必须以空字符结束)^[17]，NULL 库实例 VarCheckUefiLib 可以利用注册 SetVariable 检查处理程序接口注册检查处理程序对这些 Variable 的数据格式进行语法检查。

最后 SetVariable 检查接口就可以基于从上面来的变量特性和检查处理程序为 UEFI 规范定义的全局 Variable 和 Image 安全数据库 Variable 进行 Variable 属性、数据和数据大小的合法性检查，如果有恶意程序想要通过 SetVariable 接口对相应的 Variable 属性、数据或者数据大小进行篡改，这些 SetVariable 操作就会被拒绝并返回无效的参数状态。

5.3 基于 HII 的变量检查

为了抵御针对平台配置 Variable 的属性、数据大小或者数据的篡改，NULL 库实例 VarCheckHiiLib 可以实现基于 HII 的变量检查，如图 5-3 所示。

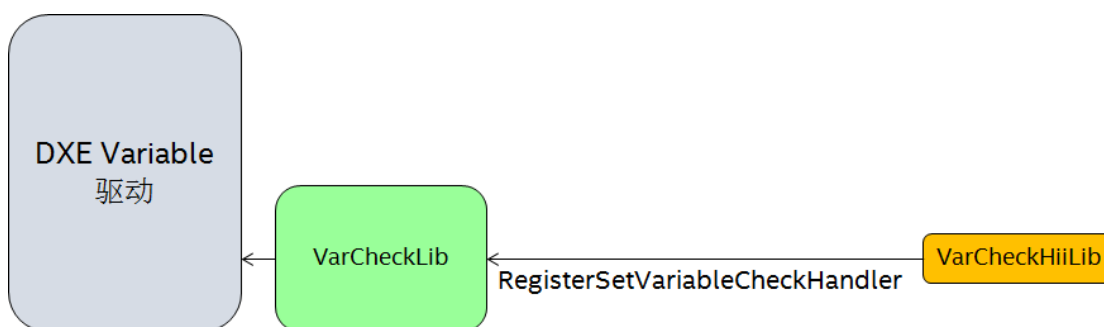


图 5-3 VarCheckHiiLib

Fig.5-3 VarCheckHiiLib

第 4.2 节中分析过英特尔提供的固件配置编辑器(FCE，Firmware Configuration Editor，<http://firmware.intel.com/develop>)可以根据 UEFI 规范 HII 部分定义的 Forms 包(package)和操作码(opcode)及相应数据格式解析 FFS 中的 VFR 二进制数据来生成 Variable 默认值。HII Forms 包中的一个 EfiVarStore 操

作码会定义一个 Variable 及相应的 Guid、Name、属性和数据大小，引用这个 EfiVarStore 操作码(通过 EFI_VARSTORE_ID VarStoreId)的 OneOf 操作码、Checkbox 操作码、Numeric 操作码和 OrderedList 操作码其实不仅定义了相应 Variable 偏移量的默认值，而且定义了相应 Variable 偏移量的有效值。如第 4.2 节中的例子就定义了 MyEfiVar Variable 的 SubmittedCallback 域为数值型且有效值范围为 0 到 255，那其它大于 255 的值对 MyEfiVar Variable 的 SubmittedCallback 域来说就是无效的。

EFI_IFR_ONE_OF_OP: 相应 Variable 偏移量的值必须是选项列表中的一个。

EFI_IFR_CHECKBOX_OP: 相应 Variable 偏移量的值必须是 0 或者 1。

EFI_IFR_NUMERIC_OP: 相应 Variable 偏移量的值必须在最小值和最大值之间。

EFI_IFR_ORDERED_LIST_OP: 相应 Variable 偏移量的数据必须是一个有序的列表。

其实 UEFI 规范 HII 部分定义的 OneOf 操作码、Checkbox 操作码、Numeric 操作码和 OrderedList 操作码通过 VFR 都会对应于 UEFI BIOS 的 SETUP 交互界面中的一个配置选项供用户修改。一个平台的平台配置驱动会包含一些 VFR 来为 CPU 和芯片组等平台硬件、ACPI、安全和启动等提供平台配置选项，如 UsbXhciSupport 配置选项，这个选项会对应于一个 CheckBox 操作码，这个 CheckBox 操作码会引用 EfiVarStore 操作码定义的平台配置 Variable 的 UsbXhciSupport 域，由于 CheckBox 操作码的值必须是 0 或者 1，那大于 1 的值对这个选项及平台配置 Variable 的 UsbXhciSupport 域来说就都是无效的。

NULL 库实例 VarCheckHiiLib 可以基于平台配置选项的有效值生成相应 Variable 的有效值数据库，这里称之为基于 HII 的变量检查数据库。基于 HII 的变量检查数据库的生成可以有两个来源：静态 HII 数据和动态 HII 数据，如图 5-4 所示。

静态 HII 数据就是包含 VFR 的驱动相应 FFS 中的 VFR 二进制数据，这些数据是静态的是因为它们它们在固件卷的 FFS 中。

动态 HII 数据是从 HiiDatabase 协议导出的 HII Forms 包数据，这些数据由包含 VFR 的驱动在启动过程中通过 HiiDatabase 协议添加到 HII 数据库，其中一部分可能会与上面的静态 HII 数据重复。

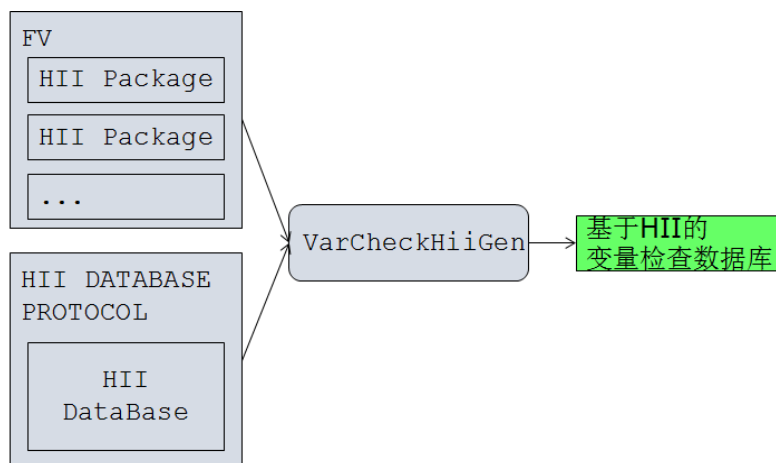


图 5-4 基于 HII 的变量检查数据库的生成

Fig.5-4 The generation of Variable Check database based on HII

NULL 库实例 VarCheckHiiLib 可以根据 UEFI 规范 HII 部分定义的 Forms 包和操作码及相应数据格式解析从固件卷来的静态 HII 数据和从 HiiDatabase 协议来的动态 HII 数据生成最后的基于 HII 的变量检查数据库，利用注册 SetVariable 检查处理程序接口注册检查处理程序来利用基于 HII 的变量检查数据库进行语义检查。

最后 SetVariable 检查接口就可以基于从上面来的检查处理程序为平台配置 Variable 进行 Variable 属性、数据和数据大小的合法性检查，如果有恶意程序想要对相应的 Variable 属性、数据或者数据大小进行篡改，这些操作就会被拒绝并返回违反安全状态(EFI_SECURITY_VIOLATION)。

5.4 基于 PCD 的变量检查

为了抵御针对 DynamicHii 或者 DynamicExHii 类型 PCD 所对应 Variable 的属性或者数据的篡改，NULL 库实例 VarCheckPcdLib 可以实现基于 PCD 的变量检查，如图 5-5 所示。

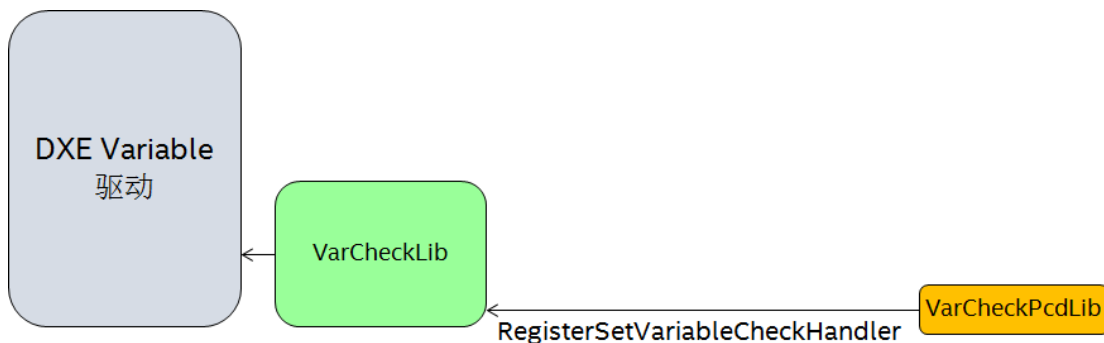


图 5-5 VarCheckPcdLib

Fig.5-5 VarCheckPcdLib

第2.3.3节阐述过EDK2中PCD的定义包括GUID值、C名字、支持的PCD类型、数据类型和默认值等，声明在包声明文件中，平台开发人员在平台描述文件中配置PCD的类型和值。DynamicHii和DynamicExHii PCD会映射到一个UEFI Variable或者Variable的一个域，默认值存储在PEI PcdPeim模块和DXE PcdDxe驱动维护的动态PCD数据库中，但修改的值会保存到对应UEFI Variable所在的NV Variable区域，这样下次系统启动的时候，HII PCD的值就会从对应的UEFI Variable来。

EDK2的包声明文件支持对PCD的有效值进行描述，如：

@ValidList: 有效值列表，PCD的值必须是列表中的一个。如，# @ValidList 0x80000001 | 1, 3, 5, 7, 9代表PCD的值必须是1/3/5/7/9中的一个。

@ValidRange: 有效值范围，PCD的值必须在有效值范围中。如，# @ValidRange 0x80000001 | 0 - 10代表PCD的值必须在0到10之间。

例如，MdePkg的包描述文件MdePkg.dec中声明了gEfiMdePkgTokenSpace Guid.PcdPlatformBootTimeOut支持PcdsFixedAtBuild、PcdsPatchableInModule、PcdsDynamic和PcdsDynamicEx类型，如果为其添加一行如下的@ValidRange描述，就能指定PcdPlatformBootTimeOut的值必须在0到10之间。

```
# @ValidRange 0x80000001 | 0 - 10  
gEfiMdePkgTokenSpaceGuid.PcdPlatformBootTimeOut/0xffff/UINT16/0x0000002c
```

一个平台的平台描述文件中会把这个PCD配置成DynamicHii或者DynamicExHii类型，映射成UEFI规范定义的全局超时Variable L"Timeout"。

```
gEfiMdePkgTokenSpaceGuid.PcdPlatformBootTimeOut/L"Timeout"/gEfiGlobalVariableGuid/0x0/5
```

通过上面包描述文件 MdePkg.dec 中的声明加上平台描述文件中的配置，PcdPlatformBootTimeOut对应的超时 Variable 的有效值范围就会被指定为0到10，那大于10的值对超时 Variable 来说就都是无效的。

EDK2 的编译工具可以在解析从包描述文件和平台描述文件中获得的 PCD 中间数据的时候，为DynamicHii 和 DynamicExHii 类型的PCD生成相应 Variable 的有效值数据库，这里称之为基于 PCD 的变量检查数据库。EDK2 的编译工具通过平台描述文件中的宏 PCD_VAR_CHECK_GENERATION 来控制是否生成

基于 PCD 的变量检查数据库。基于 PCD 的变量检查数据库需要被包含在 NULL 库实例 VarCheckPcdLib 及变量检查库链接的 DXE Variable 驱动相应 FFS 的一个 RAW SECTION 中，然后 NULL 库实例 VarCheckPcdLib 就可以在运行的时候找到这个 RAW SECTION 来获得基于 PCD 的变量检查数据库，如图 5-6 所示。

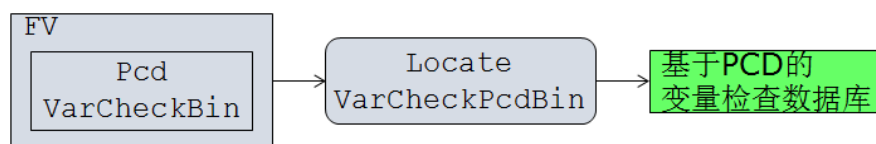


图 5-6 基于 PCD 的变量检查数据库的获得

Fig.5-6 Locate Variable Check database based on PCD

NULL 库实例 VarCheckPcdLib 可以利用注册 SetVariable 检查处理程序接口注册检查处理程序来利用基于 PCD 的变量检查数据库进行语义检查。

最后 SetVariable 检查接口就可以基于从上面来的检查处理程序为 Dynamic Hii 或者 DynamicExHii 类型 PCD 的相应 Variable 进行 Variable 属性和数据的合法性检查，如果有恶意程序想要对相应的 Variable 属性或者数据进行篡改，这些操作就会被拒绝并返回违反安全状态(EFI_SECURITY_VIOLATION)。

5.5 配额管理

为了抵御针对 NV Variable 区域的 DoS 攻击，我们可以借鉴现代操作系统的磁盘空间配额管理方法对 NV Variable 空间进行配额管理及错误记录和恢复。

EDK2 实现的 NV Variable 区域会占用 NOR FLASH 的几个区块，其存储容量有限。一个典型平台的 NV Variable 空间大小大概只有 128K~256K，而且根据微软的 Windows 操作系统硬件兼容规范，总共至少 64K 的 NV Variable 空间还需要保留给 UEFI 安全启动和 Windows 操作系统使用^[42]。恶意程序可能通过创建一个特别大的或者多个 NV Variable 来耗尽 NV Variable 空间，为了抵御这种 DoS 攻击，我们可以借鉴现代操作系统的磁盘空间配额管理方法对 NV Variable 空间进行配额管理。

如图 5-7 所示的 Windows 操作系统的磁盘空间配额管理，它可以为管理员组 and 用户组设置不同的磁盘空间限制，管理员可以使用所有的磁盘空间，而用户可能只有有限的磁盘空间可用，如图中所示的 50MB 磁盘空间。当使用量达到所设置的磁盘空间限制的时候，事件记录还可以被触发来通知用户进行磁盘清理^[43]。

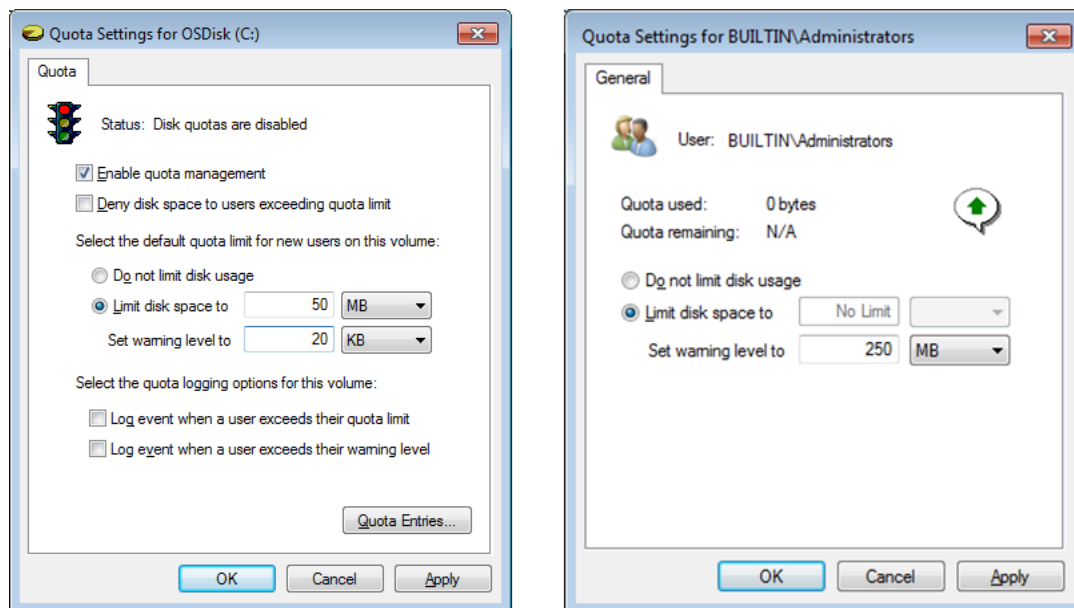


图 5-7 Windows 操作系统的磁盘空间配额管理

Fig.5-7 The disk space quota management of Windows operating system

借鉴 Windows 操作系统中的管理员组 and 用户组的概念，我们可以把 NV Variable 区分为系统 Variable 和用户 Variable。在 EndOfDxe 事件被触发之后，以下 Variable 可以被归为系统 Variable。

- 1) UEFI 规范定义的全局 Variable(gEfiGlobalVariableGuid)和 Image 安全数据库 Variable(gEfiImageSecurityDatabaseGuid)等。如启动选项 Variable、键选项 Variable、启动选项顺序 Variable、超时 Variable、控制台输入 Variable 和平台语言 Variable 等。
- 2) DXE Variable 驱动内部管理的 Variable。如安全启动启用 Variable L"Secure BootEnable"和证书数据库 Variable L"certdb"等。
- 3) 需要通过 Variable Lock 协议锁住的重要的 Variable。如 TCG2 物理存在标志 Variable 等。
- 4) 其它平台启动过程中重要的 Variable。如平台配置 Variable 等。

系统 Variable 基本上是系统启动比较关键的 Variable。如果一个 Variable 不是系统 Variable，那它就是用户 Variable。系统 Variable 都需要通过变量检查库或者变量检查协议的变量特性设置接口设置变量特性，然后下面的 IsUserVariable 函数就可以通过变量特性获得接口知道一个 Variable 是不是系统 Variable，平台变量清理库就可以利用 IsUserVariable 函数来枚举出所有的用户 Variable。

```

BOOLEAN
IsUserVariable (
    IN CHAR16                *Name,
    IN EFI_GUID              *Guid
)
{
    EFI_STATUS                Status;
    VAR_CHECK_VARIABLE_PROPERTY Property;
    if (mVarCheck == NULL) {
        gBS->LocateProtocol ( //查找Variable Check协议
            &gEdkiiVarCheckProtocolGuid,
            NULL,
            (VOID **) &mVarCheck
        );
    }
    ASSERT (mVarCheck != NULL);

    ZeroMem (&Property, sizeof (Property));
    Status = mVarCheck->VariablePropertyGet (
        Name,
        Guid,
        &Property
    );
    if (EFI_ERROR (Status)) {
        // No property, it is user variable. 不能获得变量特性, 这个Variable是用户Variable。
        DEBUG ((EFI_D_INFO, "PlatformVarCleanup - User variable: %g:%s\n", Guid, Name));
        return TRUE;
    }

    return FALSE;
}

```

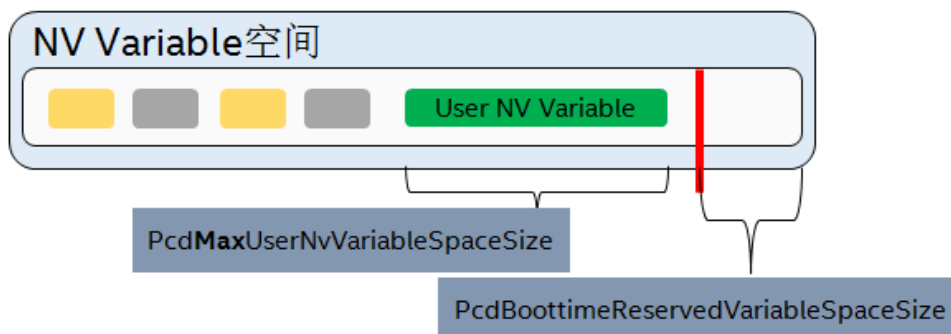


图 5-8 NV Variable 空间的配额管理

Fig.5-8 The quota management of NV Variable space

在 NV Variable 被区分为系统 Variable 和用户 Variable 之后， 我们还需要限

制用户 NV Variable 空间的大小， gEfiMdeModulePkgTokenSpaceGuid.PcdMaxUserNvVariableSpaceSize 可以被定义来指定用户 NV Variable 空间的最大大小。如图 5-8 所示 NV Variable 空间的配额管理， 系统 Variable 可以占用所有的 NV Variable 空间， 但用户 Variable 最多只能占用 PcdMaxUserNvVariableSpaceSize 大小的 NV Variable 空间， PcdMaxUserNvVariableSpaceSize 的引入可以保证关键的系统 Variable 有足够的空间可用。

为了保证系统的正常启动， 我们还可以为启动时保留一部分 NV Variable 空间， gEfiMdeModulePkgTokenSpaceGuid.PcdBoottimeReservedNvVariableSpaceSize 可以被定义来指定为启动时保留的 NV Variable 空间的大小， 这样即使恶意程序在操作系统运行时耗尽了其它所有的 NV Variable 空间， 在下次系统启动的时候， 这部分保留的启动时 NV Variable 空间则还可以被用来保证系统的正常启动。

需要注意的是是一些系统 Variable 是同类型的 Variable 以及数据是可变长的， 如启动选项 Variable L"Boot####"可能有 L"Boot0000"、 Variable L"Boot 0001"一直到 L"BootFFFF" 0x10000 个 Variable， 而且每个启动选项 Variable 的数据都是可变长的， 为了抵御恶意程序可能通过创建一个特别大的或者多个启动选项 Variable 来耗尽 NV Variable 空间， 平台开发人员可以限制特定平台的启动选项 Variable 的数量以及单个启动选项 Variable 的数据大小。

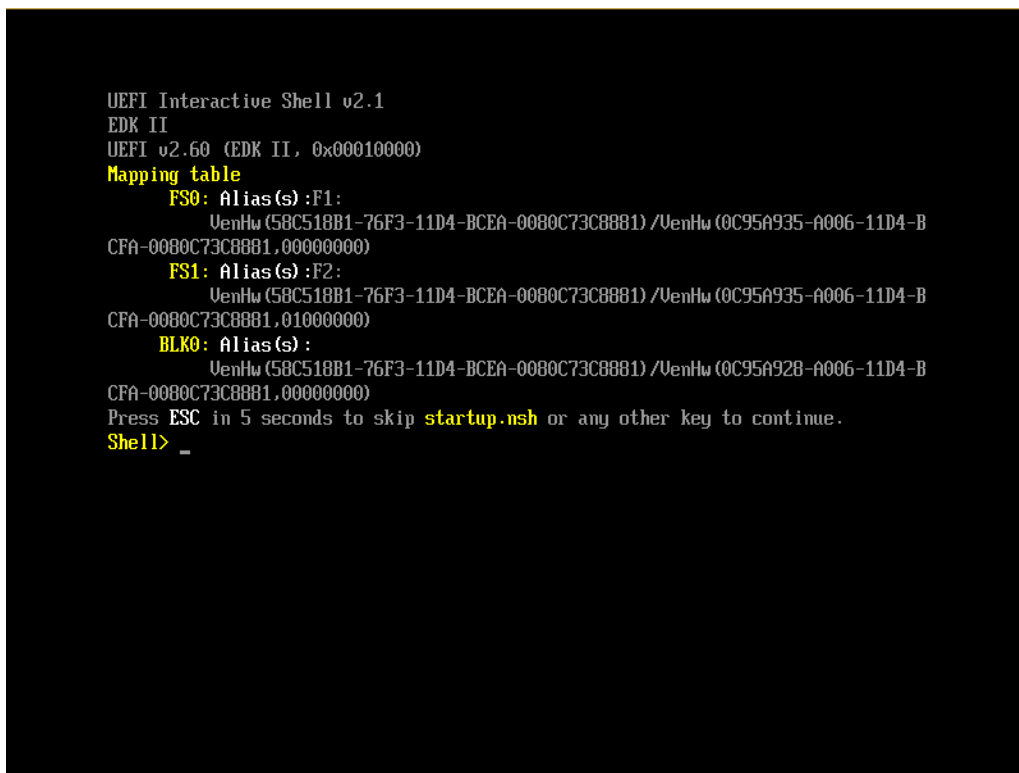
当系统 NV Variable 空间或者用户 NV Variable 空间的使用到达限制的时候， DXE Variable 驱动还可以通过一个内部管理的 Variable L"VarErrorFlag"来记录错误， 这个 Variable 使用一个字节： 0xFF 表示没有错误； 0xEF 表示系统 NV Variable 空间的使用到达了限制； 0xFE 表示用户 NV Variable 空间的使用到达了限制； 0xEE 表示系统 NV Variable 空间和用户 NV Variable 空间的使用都到达了限制。

在下次系统启动的时候， 平台 BDS 可以读取 Variable L"VarErrorFlag"的数据， 如果发现是系统 NV Variable 空间的使用到达了限制， 平台需要有机制来清理 NV Variable 空间和恢复 NV Variable 到默认值； 如果发现是用户 NV Variable 空间的使用到达了限制， 平台可以利用平台变量清理库来直接删除所有的用户 NV Variable 或者提供配置界面来列出所有的用户 NV Variable 让用户选择删除哪

些用户 NV Variable。

5.6 验证测试

通过新的保护和恢复机制-UEFI 变量检查，加上现有的保护和恢复机制-Variable Lock 和平台配置 Variable 默认值恢复，我们希望能对 NV Variable 及 NV Variable 区域进行很好地保护。NT32 模拟开发平台可以被用来对新的保护和恢复机制-UEFI 变量检查进行验证测试，UEFI SHELL^[44]（基于 UEFI SHELL 规范实现的命令行处理程序，提供了类似于 Windows Command Prompt 和 Linux SHELL 的用户命令行交互环境）的 setvar 命令可以被用来模拟在 EndOfDxe 事件被触发之后及操作系统运行时可能的恶意程序对 NV Variable 及 NV Variable 区域的攻击。



```
UEFI Interactive Shell v2.1
EDK II
UEFI v2.60 (EDK II, 0x00010000)
Mapping table
FS0: Alias(s):F1:
    VenHw (58C518B1-76F3-11D4-BCEA-0080C73C8881) /VenHw (0C95A935-A006-11D4-B
CFA-0080C73C8881,00000000)
FS1: Alias(s):F2:
    VenHw (58C518B1-76F3-11D4-BCEA-0080C73C8881) /VenHw (0C95A935-A006-11D4-B
CFA-0080C73C8881,01000000)
BLK0: Alias(s):
    VenHw (58C518B1-76F3-11D4-BCEA-0080C73C8881) /VenHw (0C95A928-A006-11D4-B
CFA-0080C73C8881,00000000)
Press ESC in 5 seconds to skip startup.nsh or any other key to continue.
Shell> _
```

图 5-9 NT32 运行进入 UEFI SHELL

Fig.5-9 NT32 run into UEFI SHELL

首先我们需要通过如下指令来编译和运行 NT32 模拟开发平台，-p 是指定要编译的平台，-a 是指定要编译的架构，-t 是指定编译用的工具链。

编译： build -p Nt32Pkg\Nt32Pkg.dsc -a IA32 -t VS2008x86

运行： build -p Nt32Pkg\Nt32Pkg.dsc -a IA32 -t VS2008x86 RUN

接着 NT32 模拟开发平台会运行进入 UEFI SHELL，如图 5-9 所示。

然后我们就可以利用 `setvar` 命令来模拟对 NV Variable 及 NV Variable 区域进行攻击了，看新的保护和恢复机制-UEFI 变量检查是否能够抵御这些攻击。

1) 篡改 Variable 的属性。

现在的 `setvar` 命令并不支持改变 Variable 的属性，在修改 `setvar` 源代码和重新编译 UEFI SHELL 之后，我们可以利用以下命令来试着修改启动选项 Variable L"Boot0000"的属性为 NV+BS。

```
setvar Boot0000 -nv -bs =00
```

命令运行的结果如下，改变 Variable 属性的操作会被拒绝并返回无效的参数状态，这是因为基于 UEFI 的 NULL 库实例 VarCheckUefiLib 利用变量特性设置接口设置了启动选项 Variable 的属性必须为 NV+BS+RT。

```
Variable Check Attributes(0x00000007 to 0x00000003) fail Invalid Parameter - 8BE4DF61-93CA-11D2-AA0D-00E098032B8C:Boot0000
```

2) 篡改 Variable 的数据大小。

我们可以利用以下命令来试着修改超时 Variable L"Timeout"的数据大小为 1 个字节。

```
setvar Timeout =00
```

命令运行的结果如下，改变 Variable 数据大小的操作会被拒绝并返回无效的参数状态，这是因为基于 UEFI 的 NULL 库实例 VarCheckUefiLib 利用变量特性设置接口设置了超时 Variable 的数据大小必须为 2 个字节。

```
Variable Check DataSize fail(0x1 not in 0x2 - 0x2) Invalid Parameter - 8BE4DF61-93CA-11D2-AA0D-00E098032B8C:Timeout
```

3) 篡改 Variable 的数据。

我们可以利用以下命令来试着修改控制台输入 Variable L"ConIn"的数据。

```
setvar ConIn =0123456789012345
```

命令运行的结果如下，改变 Variable 数据的操作会被拒绝并返回无效的参

数状态，这是因为控制台输入 Variable 的数据格式必须是 EFI_DEVICE_PATH_PROTOCOL，基于 UEFI 的 NULL 库实例 VarCheckUefiLib 利用注册 SetVariable 检查处理程序接口注册的检查处理程序会对控制台输入 Variable 的数据格式进行语法检查。

```
UEFI Variable Check function fail Invalid Parameter - 8BE4DF61-93CA-11D2-AA0D-00E098032B8C:ConIn
```

利用以下命令来试着修改 MyEfiVar Variable 的 SubmittedCallback 域为 0x3412。

```
setvar MyEfiVar -guid A04A27F4-DF00-4D42-B552-39511302113D =0000000000001234
```

命令运行的结果如下，改变 Variable 数据的操作会被拒绝并返回违反安全状态，这是因为基于 HII 的 NULL 库实例 VarCheckHiiLib 利用注册 SetVariable 检查处理程序接口注册的检查处理程序会利用基于 HII 的变量检查数据库对 MyEfiVar Variable 进行语义检查，第 5.3 节中分析过 MyEfiVar Variable 的 SubmittedCallback 域的有效值范围为 0 到 255，0x3412 会被认为是非法的值。

```
VarCheckHiiVariable - MyEfiVar:A04A27F4-DF00-4D42-B552-39511302113D with Attributes = 0x00000003
Size = 0x8
VarCheckHiiQuestion fail: Numeric mismatch (0x3412)

00000000: 07 09 06 00 02 00 00 FF-00          *.....*

Variable Check handler fail Security Violation - A04A27F4-DF00-4D42-B552-39511302113D:MyEfiVar
```

利用以下命令来试着修改超时 Variable L”Timeout”的值为 0x000B。

```
setvar Timeout =0B00
```

命令运行的结果如下，改变 Variable 数据的操作会被拒绝并返回违反安全状态，这是因为基于 PCD 的 NULL 库实例 VarCheckPcdLib 利用注册 SetVariable 检查处理程序接口注册的检查处理程序会利用基于 PCD 的变量检查数据库对超时 Variable 进行语义检查，第 5.4 节中说明过可以通过 MdePkg 的包描述文件 MdePkg.dec 中的 @ValidRange 描述指定 PcdPlatformBootTimeOut 对应的超时 Variable 的有效值范围就为 0 到 10，0x000B 会被认为是非法的值。

```

VarCheckPcdVariable - Timeout:8BE4DF61-93CA-11D2-AA0D-00E098032B8C with Attributes = 0x00000007
Size = 0x2
VarCheckPcdValidData fail: ValidRange mismatch (0xB)

00000000: 02 09 00 00 02 00 00 0A-00                *.....*

Variable Check handler fail Security Violation - 8BE4DF61-93CA-11D2-AA0D-00E098032B8C:Timeout

```

4) DoS 攻击。

NT32 模拟开发平台通过 PcdFlashNvStorageVariableSize 指定了 NV Variable 区域的大小为 0x0000c000， 我们可以在 NT32 的平台描述文件 Nt32Pkg.dsc 中通过 PcdMaxUserNvVariableSpaceSize 来指定用户 NV Variable 空间的最大大小为 0x1000， 然后利用下面的 SHELL 脚本来试图耗尽 NV Variable 空间。

```

For %A in 0 1 2 3 4 5 6 7 8 9 a b c d e f
    For %B in 0 1 2 3 4 5 6 7 8 9 a b c d e f
        For %C in 0 1 2 3 4 5 6 7 8 9 a b c d e f
            For %D in 0 1 2 3 4 5 6 7 8 9 a b c d e f
                setvar  %A%B%C%D  -guid  FFFFFFFF-FFFF-FFFF-FFFF-FFFFFFFFFFFFFF -nv -bs -rt
            =%A%B%C%D
        endfor
    endfor
endfor
endfor

```

脚本运行的结果如下，脚本中设置的 Variable 会被认为是用户 NV Variable，当用户 NV Variable 空间的使用到达限制的时候，错误会被记录到 Variable L"VarErrorFlag"中。

```
RecordVarErrorFlag (0xFE) 004f:FFFFFFFF-FFFF-FFFF-FFFF-FFFFFFFFFFFFFF - 0x00000007 - 0x2C
```

在下次启动的时候，平台 BDS 会读取 Variable L"VarErrorFlag"的数据，在发现是用户 NV Variable 空间的使用到达了限制之后，会利用平台变量清理库提供如图 5-10 所示的配置界面列出所有的用户 NV Variable 来让用户选择删除哪些

用户 NV Variable， 用户可以选择删除所有的用户 NV Variable， 也可以选择只删除上面 SHELL 脚本创建出来的用户 NV Variable， 从而对 NV Variable 空间进行清理及恢复。



图 5-10 用户 NV Variable 清理配置界面
Fig.5-10 User NV Variable cleanup setup user interface

5.7 平台应用

UEFI 变量检查具有很好的适用性和可扩展性， 它不依赖于特定的 Variable 驱动的实现， 可以被用于包括服务器、 台式机、 笔记本和嵌入式设备等各种平台。 但为了很好地应用 UEFI 变量检查， 平台开发人员还是需要非常深入的了解自身平台对 Variable 及 Variable 区域的使用， 特别是对 NV Variable 及 NV Variable 区域的使用。

平台开发人员需要在平台描述文件和 FLASH 声明文件中进行如下配置来使 DXE Variable 驱动与变量检查库及 VarCheckUefiLib、 VarCheckHiiLib 和 VarCheckPcdLib 链接， 同时还需要在平台描述文件中对 PcdMaxUserNvVariable SpaceSize 和 PcdBoottimeReservedNvVariableSpaceSize 进行配置。另外平台开发人员还可能要实现特定的平台 NULL 库实例 PlatformVarCheckLib 来对一些 Variable 进行变量检查、 限制特定平台的同类型 Variable 的数量以及可变长 Variable 的数据大小和使用平台变量清理库。

```

*.dsc:

PCD_VAR_CHECK_GENERATION          = TRUE

MdeModulePkg/Universal/Variable/RuntimeDxe/VariableRuntimeDxe.inf {

  <LibraryClasses>

    VarCheckLib |MdeModulePkg/Library/VarCheckLib/VarCheckLib.inf

    NULL|MdeModulePkg/Library/VarCheckUefiLib/VarCheckUefiLib.inf

    NULL|MdeModulePkg/Library/VarCheckHiiLib/VarCheckHiiLib.inf

    NULL|MdeModulePkg/Library/VarCheckPcdLib/VarCheckPcdLib.inf

}

*.fdf:

INF RuleOverride = VARCHCKPCD MdeModulePkg/Universal/Variable/RuntimeDxe/VariableRuntimeDxe.inf

[Rule.Common.DXE_RUNTIME_DRIVER.VARCHCKPCD]

FILE DRIVER = $(NAMED_GUID) {

  DXE_DEPEX  DXE_DEPEX Optional  $(INF_OUTPUT)/$(MODULE_NAME).depex

  RAW  BIN  $(WORKSPACE)/$(OUTPUT_DIRECTORY)/$(TARGET)_$(TOOL_CHAIN_TAG)/FV/PcdVar
Check.bin

  PE32  PE32  $(INF_OUTPUT)/$(MODULE_NAME).efi

  UI  STRING=$(MODULE_NAME)" Optional

  VERSION  STRING=$(INF_VERSION)" Optional BUILD_NUM=$(BUILD_NUMBER)

}

```

如果一个平台实现了自己的 Variable 驱动，这个 Variable 驱动也可以使用 UEFI 变量检查，而 UEFI 变量检查的接口及实现则不需要做任何改动。

5.8 本章小结

本章首先研究了变量检查模型的建立，接着分别研究了基于 UEFI、基于 HII 和基于 PCD 的变量检查以及配额管理，然后利用 NT32 模拟开发平台对新的保护和恢复机制-UEFI 变量检查进行了验证测试，测试结果反映出 UEFI 变量检查能很好地抵御通过 UEFI SHELL setvar 命令模拟的对 NV Variable 及 NV Variable 区域进行的攻击，最后简要地说明了一个平台如何应用 UEFI 变量检查。

第六章 总结与展望

6.1 全文总结

基于 UEFI 和 PI 规范实现的 UEFI BIOS 一般会存储在一个 NOR 非易失性块存储设备（如 SPI FLASH）中。EDK2 作为一个现代、功能丰富且跨平台的 UEFI BIOS 固件开发环境已经被业界广泛使用。NV Variable 区域会占用 NOR FLASH 的几个区块，其存储容量有限，但其存储的数据却非常重要。UEFI Variable 运行时服务定义了各部件访问 Variable 的接口。NV Variable 的非易失性和 UEFI Variable 服务的运行时特性使得存储重要信息的 NV Variable 及 NV Variable 区域易受恶意程序的攻击，因此对 NV Variable 的保护显得尤为重要。

本文分析了现有的保护和恢复机制-Variable Lock 和平台配置 Variable 默认值恢复，以及 Variable 接口的安全性。针对各种可能的恶意攻击，很多情况下现有的保护和恢复机制并不能被使用，我们提出了新的保护和恢复机制-基于 EDK2 的 UEFI 变量检查来对 NV Variable 及 NV Variable 区域进行更好地保护。

本文研究和实现了新的保护和恢复机制-UEFI 变量检查，它利用 UEFI 和 PI 规范定义的访问 Variable 的各种服务及接口和 Variable 区域及 Variable 本身的特性，借鉴现代操作系统的磁盘空间配额管理方法，实现变量检查库和变量检查协议，通过属性检查、数据大小检查、语法检查、语义检查、配额管理、错误记录和恢复，来保证从 UEFI Variable 运行时服务的 SetVariable 接口传入的 Variable 属性、数据和数据大小的合法性，从而更好地对 NV Variable 及 NV Variable 区域进行保护。利用 NT32 模拟开发平台的验证测试结果反映出新的保护和恢复机制-UEFI 变量检查能很好地抵御抵御恶意程序的攻击。本文主要的工作具体如下：

- 1) 阐述了 UEFI 和 PI 规范及 UEFI BIOS 的启动流程，现代、功能丰富且跨平台的 UEFI BIOS 开源固件开发环境 EDK2。
- 2) 分析了 Variable 的相关接口，基于 EDK2 实现的 Variable 存储区格式和 Variable 模块。

- 3) 详细地分析了现有的保护和恢复机制-Variable Lock 和平台配置 Variable 默认值恢复, 以及 Variable 接口的安全性。针对各种可能的恶意攻击, 很多情况下现有的保护和恢复机制并不能被使用, 我们提出了新的保护和恢复机制-基于 EDK2 的 UEFI 变量检查来对 NV Variable 及 NV Variable 区域进行更好地保护。
- 4) 研究了变量检查模型的建立, 基于 UEFI、基于 HII 和基于 PCD 的变量检查以及配额管理, 利用 NT32 模拟开发平台对新的保护和恢复机制-UEFI 变量检查进行了验证测试, 最后简要地说明了一个平台如何应用 UEFI 变量检查库。

6.2 展望

通过新的保护和恢复机制-UEFI 变量检查, 加上现有的保护和恢复机制-Variable Lock 和平台配置 Variable 默认值恢复, 我们已经可以对 NV Variable 及 NV Variable 区域进行很好的保护了。平台开发人员需要非常深入的了解自身平台对 Variable 及 Variable 区域的使用, 特别是对 NV Variable 及 NV Variable 区域的使用, 才能很好地运用这些机制。平台开发人员还可能要实现特定的平台 NULL 库实例 PlatformVarCheckLib 来对一些 Variable 进行变量检查。由于 NV Variable 区域的存储容量一般很有限, 因此对 NV Variable 的使用需要遵循保守、节约和谨慎的原则。另外 Variable 运行时属性的使用也需要谨慎, 只有在某个 Variable 确实需要在操作系统运行时被访问的时候, Variable 运行时属性才应该被使用。

基于 EDK2 实现的 UEFI BIOS 会应用的越来越广泛, 针对现在的恶意攻击研究和实现的新的保护和恢复机制-UEFI 变量检查, 如果不能很好地抵御将来可能出现的新的对 NV Variable 及 NV Variable 区域的恶意攻击, 我们将需要利用变量特性结构体、变量检查库和变量检查协议的可扩展性对 UEFI 变量检查进行进一步的完善。

参考文献

- [1] <http://firmware.intel.com/learn/uefi/about-uefi>
- [2] Vincent Zimmer, Michael Rothman, Robert Hale, Beyond BIOS Developing with the Unified Extensible Firmware Interface, Intel Press, 2006, 9 184 297-298
- [3] Flash memory, https://en.wikipedia.org/wiki/Flash_memory
- [4] Denial of service attack, https://en.wikipedia.org/wiki/Denial-of-service_attack
- [5] 沈美明, 温冬婵, IBM-PC 汇编语言程序设计 (第二版), 清华大学出版社, 2001
- [6] 邢卓媛, 基于 UEFI 的网络协议栈的研究与改进, 华东师范大学硕士学位论文, 2011, 6-7
- [7] 张远虎, 基于 uEFI BIOS 的多线程研究与实现, 上海交通大学硕士学位论文, 2013, 5-6 19-21 2.4
- [8] 唐文彬, 视跃飞, 陈嘉勇, 统一可扩展固件接口攻击方法研究, 计算机工程, 2012, 1
- [9] 倪兴荣, 基于 UEFI 技术的 API 性能分析设计与实现, 南京信息工程大学硕士学位论文, 2010, 9-11
- [10] 王晓箴, 于磊, 刘宝旭, 基于 EDK2 平台的数据备份与恢复技术, 计算机工程, 2011, 1
- [11] 付思源, 刘功申, 李建华, 基于 UEFI 固件的恶意代码防范技术研究, 计算机工程, 2012, 1
- [12] 戴正华, UEFI 原理与编程, 机械工业出版社, 2015, 15-16
- [13] Intel Corporation, Extensible Firmware Interface Specification 1.01, 2002
- [14] 谭浩强, C 程序设计 (第四版), 清华大学出版社, 2010
- [15] 高云岭, 庄克良, 丁守芳, UEFI+BIOS 全局配置数据库的设计与实现, 物联网技术, 2014, 1-2
- [16] <http://www.uefi.org/workinggroups>
- [17] Unified EFI, Inc., Unified Extensible Firmware Interface Specification 2.6, 2016

- http://www.uefi.org/sites/default/files/resources/UEFI%20Spec%202_6.pdf, 96-97 3.3
7.2 1908 30.6.1 31.3.8
- [18] edk2 open source code, <https://github.com/tianocore/edk2>
- [19] 曾宇睿, 基于 EDKII 的 PCI 总线驱动管理技术的设计与开发, 华中科技大学硕士学位论文, 2011, 7
- [20] 胡侠情, 基于 EDKII 框架的固件生成工具设计与实现, 华中科技大学硕士学位论文, 2014, 14-16 29
- [21] UEFI - PI Overview, <http://firmware.intel.com/learn/uefi/uefi-training-materials>
- [22] Unified EFI, Inc., VOLUME 1: Platform Initialization Specification Pre-EFI Initialization Core Interface 1.4, 2015, 65 8.2.3
- [23] Unified EFI, Inc., VOLUME 2: Platform Initialization Specification Driver Execution Environment Core Interface 1.4, 2015, 12.11 12.12
- [24] Unified EFI, Inc., VOLUME 3: Platform Initialization Specification Shared Architectural Elements 1.4, 2015, 9-10 16 3.2.1 54 58 61 122
- [25] Unified EFI, Inc., VOLUME 4: Platform Initialization Specification System Management Mode Core Interface 1.4, 2015
- [26] Unified EFI, Inc., VOLUME 5: Platform Initialization Specification Standards 1.4, 2015
- [27] Unified EFI, Inc., Advanced Configuration and Power Interface Specification 6.1, 2016
- [28] Distributed Management Task Force, Inc. (DMTF), System Management BIOS (SMBIOS) Reference 5 Specification 3.0, 2015
- [29] Intel Corporation, Intel® 64 and IA-32 Architectures Software Developer's Manual 055, 2015
<http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>
- [30] Intel Corporation, EFI Developer Kit (EDK) Getting Started Guide 0.41, 2005
- [31] Intel Corporation, EDKII User Manual 0.7, 2010, 10, 9-12
- [32] 哲思社区, 可爱的 Python, 电子工业出版社, 2009
- [33] Intel Corporation, EDK II Module Information (INF) File Specification 1.25,

2016

[34] Intel Corporation, EDK II Package Declaration (DEC) File Format Specification 1.25, 2016

[35] Intel Corporation, EDK II Platform Description (DSC) File Specification 1.26, 2016

[36] Intel Corporation, EDK II Flash Description (FDF) File Specification 1.26, 2016

[37] Intel Corporation, EDK II Module Writer's Guide 0.7, 2010, 45-48

[38] PEI and DXE and FV Overview,

<http://firmware.intel.com/learn/uefi/uefi-training-materials>

[39] Fault tolerance, https://en.wikipedia.org/wiki/Fault_tolerance

[40] Microsoft Corporation, Microsoft Portable Executable and Common Object File Format Specification 8.3, 2013

[41] Intel Corporation, Intel® Firmware Configuration Editor (Intel® FCE) Quick Start Guide 1.5, 2014, 9

[42] Windows Hardware Certification Requirements for Client and Server Systems, <https://msdn.microsoft.com/en-us/library/windows/hardware/jj128256.aspx>

[43] Disk quota, http://en.wikipedia.org/wiki/Disk_quota

[44] Unified EFI, Inc., UEFI Shell Specification 2.2, 2016

附 录

致 谢

光阴如箭，岁月如梭，转眼间研究生生活已经过去三年多了，期间还迎来了我的宝贝女儿。从开始研究生阶段的学习到本文的完成，许多老师、同学、领导、同事和家人都对我给予了无私的关心和帮助。值此论文完成之际，向他们表示由衷的感谢。

首先感谢我的导师黄林鹏教授，在我进行课题研究和论文撰写的过程中，对我的论文进度给予了密切的关注，提出很多宝贵意见，让我受益匪浅。另外感谢我的同学们，他们认真学习的态度和细微的帮助，使我深受鼓舞。

其次感谢英特尔亚太研发 SSG PSI 组的领导和同事们，特别是卢炬、田鹤、姚颀文和蔺杰在工作和学习上的关心和帮助，我的论文的完成离不开他们的指导和支持。

最后感谢我的父母家人长期以来的关怀和支持，他们的关爱和对我的要求是我取得成绩的前提和前进的动力。

攻读学位期间发表的学术论文

- [1] 曾令静, 基于EDK2 的容错写机制的研究, www.csmoe.sjtu.edu.cn, 2016

上海交通大学

学位论文原创性声明

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

学位论文作者签名：曾令静

日期：2016 年 5 月 13 日

上海交通大学 学位论文版权使用授权书

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

保密 ☐ 在 ____ 年解密后适用本授权书。

本学位论文属于

不保密 ☒。

(请在以上方框内打“√”)

学位论文作者签名：

曾令静

指导教师签名：

苏小明

日期：2016年5月13日

日期：2016年5月13日