Fachhochschule Münster
Fachbereich Elektrotechnik und Informatik

Bachelorarbeit

zur Erlangung
des akademischen Grades
Bachelor of Science (B.Sc.)
im Studiengang Informatik

Entwicklung eines Fuzzers für die UEFI/PI-Referenzimplementierung

Erstprüfer	Prof. Dr.-Ing. Sebastian Schinzel
Zweitprüfer	Hendrik Schwartke, M.Sc.
vorgelegt am	4. Oktober 2017
von	Jan Ewald
Matrikelnummer	

Inhaltsverzeichnis

Abstract

Das Unified Extensible Firmware Interface hat sich als Standard für Computer-Firmware weitgehend durchgesetzt und definiert mittlerweile in vielen Systemen elementare Funktionen wie den Bootvorgang. In der Vergangenheit wurden bereits mehrfach Schwachstellen gefunden, die zu einer vollständigen Kompromittierung führen können. Zugleich gibt es nur wenige Ansätze für eine systematische Sicherheitsanalyse. Diese Arbeit stellt einen Fuzzer vor, mit dem in der Referenzimplementierung automatisiert Schwachstellen gesucht werden können. Als Proof of Concept wird ein bekannter Fehler durch Fuzzing wiederentdeckt.

Abkürzungsverzeichnis

AL	After Life
BDS	Boot Device Selection
BIOS	Basic Input/Output System
BSP	Bootstrap Processor
BST	Binary Search Tree
BTS	Branch Trace Store
CFG	Control Flow Graph
DXE	Driver Execution Environment
EDK	EFI Development Kit
EFI	Extensible Firmware Interface
FUP	Flow Update
GUID	Globally Unique Identifier
HMP	Human Monitor Protocol
HOB	Hand-Off Block
ISA	Industry Standard Architecture
ISR	Interrupt Service Routine
JSON	JavaScript Object Notation
KVM	Kernel-based Virtual Machine
MSR	Model-specific Register
NVRAM	Non-Volatile Random-Access Memory
OVMF	Open Virtual Machine Firmware

PE Portable Executable
PEI Pre-EFI Initialization
PEIM Pre-EFI Module
PI Platform Initialization
PPI PEIM-to-PEIM Interface
PSB Packet Stream Boundary
PT Processor Trace

QEMU Quick Emulator
QMP QEMU Machine Protocol

RT Runtime
RVA Relative Virtual Address

SEC Security
SMI System Management Interrupt
SMM System Management Mode

TIP Target Instruction Pointer
TLV Type Length Value
TNT Taken/Not-taken
TSL Transient System Load
ToPA Table of Physical Addresses

UDK UEFI Development Kit
UEFI Unified Extensible Firmware Interface

VM Virtual Machine
VMCS Virtual Machine Control Structure
VMM Virtual Machine Monitor

1 Einleitung

Das Unified Extensible Firmware Interface (UEFI) hat das ursprünglich aus den 70er-Jahren stammende Basic Input/Output System (BIOS) inzwischen weitgehend verdrängt. UEFI-kompatible Firmware findet sich heute in PCs, Laptops und Servern. Während das BIOS auf elementare Funktionen beschränkt war, unterstützen moderne Systeme aufwendige grafische Benutzeroberflächen und eine Vielzahl von Hardwarekomponenten, Bootprotokollen und Konfigurationsmöglichkeiten.

Features wie Secure Boot dokumentieren zudem den gestiegenen Sicherheitsanspruch. Ist Secure Boot aktiviert, so müssen alle Bootloader sowie der Betriebssystemkern eine gültige kryptografische Signatur haben. Bei einer Manipulation schlägt diese Integritätsprüfung fehl, und der Bootvorgang wird abgebrochen.

Mit der Komplexität hat sich jedoch auch die Wahrscheinlichkeit für Schwachstellen erhöht. Außerdem bietet UEFI sowohl während des Bootvorgangs als auch gegenüber dem Betriebssystem eine große Angriffsfläche. Die Sicherheit UEFI-kompatibler Firmware hängt deshalb entscheidend davon ab, ob die Implementierung tatsächlich korrekt ist.

1.1 Motivation

Weil die Firmware die Grundlage des Systems bildet, sind Schwachstellen grundsätzlich kritisch und können zu einer vollständigen und dauerhaften Kompromittierung führen. Im Fall von UEFI wird dieses Risiko von mehreren Faktoren erhöht. Selbst Komponenten, die für den Betrieb des Systems nicht unbedingt notwendig sind, haben oftmals uneingeschränkten Lese- oder sogar Schreibzugriff auf die Firmware und den Arbeitsspeicher. Zugleich sind Manipulationen nur mit hohem Aufwand sowie spezieller Soft- und Hardware zu erkennen. Gewöhnliche Diagnosewerkzeuge wie Userspace-Programme oder Funktionen des Betriebssystemen können nicht direkt auf die Firmware zugreifen und sind zudem selbst durch die UEFI-Umgebung manipulierbar. Ein weiteres Problem ergibt sich daraus, dass Hersteller nach dem Bekanntwerden einer Sicherheitslücke den Patch mitunter verspätet bereitstellen. Vielen Nutzern ist außerdem die Notwendigkeit von Firmware-Updates nicht bewusst, sodass auch bekannte Schwachstellen eine weite Verbreitung haben [vgl. 1].

Eine Auswertung des offiziellen UEFI Security Advisories [2] zeigt, dass neben UEFI-spezifischen Schwachstellen vor allem typische Programmierfehler wie arithmetische Überläufe und Pufferüberläufe in der Vergangenheit immer wieder auftraten. Zugleich gibt es kaum Ansätze, diese systematisch und automatisiert zu suchen. So wurde etwa eine der schwerwiegendsten Schwachstellen erst im Rahmen eines mehrwöchigen Code-Audits gefunden [vgl. 3].

1.2 Bestehende Ansätze

Das CHIPSEC-Framework enthält einen einfachen Fuzzer, der zufällige UEFI-Variablen erzeugt und dadurch Fehler bei der Verarbeitung finden kann[1]. Diese unsystematische Suche eignet sich allerdings nicht für komplexere Funktionen, bei denen Fehler nur unter bestimmten Bedingungen auftreten. Zudem läuft das Programm auf demselben System wie die zu testende UEFI-Firmware, sodass bei einem Absturz auch der Fuzzing-Prozess abgebrochen wird und möglicherweise Daten verloren gehen.

In dem Excite-Projekt[4] von Intel werden die statische Analyse der Firmware und systematisches Fuzzing kombiniert, um ungültige Speicherzugriffe in so genannten System Management Interrupt (SMI) Handlern zu finden. SMI Handler führen geschützten Code in dem mit maximalen Privilegien ausgestatteten System Management Mode der CPU aus und sind somit besonders gefährdet. Zum Zeitpunkt dieser Arbeit gibt es jedoch keine veröffentlichte Implementierung. Ob sich der Ansatz verallgemeinern und auf andere Schwachstellen anwenden lässt, bleibt also unklar.

Der Fuzzer kAFL[23] hat bereits demonstriert, wie sich Fuzzing auch auf andere Ziele als Userspace-Programme anwenden lässt. Allerdings ist die Implementierung auf Betriebssysteme ausgelegt.

1.3 Ansatz dieser Arbeit

Im Rahmen dieser Arbeit wird ein generischer Fuzzer für die UEFI-Referenzimplementierung entwickelt. Weil der Beispielcode als Vorlage für Hersteller dient, besteht eine hohe Wahrscheinlichkeit, dass gefundene Schwachstellen auch in realen Produkten vorhanden sind. In der Vergangenheit konnten Defekte selbst dann ausgenutzt werden, wenn die Firmware die betroffenen Komponenten überhaupt nicht verwendete, sondern lediglich als toten Code enthielt [vgl. 5].

Der Fuzzer generiert Eingabedaten für eine virtuelle UEFI-Umgebung und erkennt verschiedene Fehlersymptome. Hierdurch ist es möglich, Schwachstellen wie etwa Pufferüberläufe zu entdecken. Im Vergleich zu der Analyse physischer Firmware hat die Virtualisierung mehrere Vorteile. So wird keine spezielle Hardware benötigt, die Fuzzing-Infrastruktur ist skalierbar, und physische Schäden durch Fehler lassen sich vermeiden.

[1] https://github.com/chipsec/chipsec/blob/master/chipsec/modules/tools/uefi/uefivar_fuzz.py

1.4 Aufbau der Arbeit

Der Hauptteil dieser Arbeit besteht aus vier Abschnitten.

(i) Das Kapitel 2 erläutert die Grundlagen von UEFI und gibt einen Überblick über die verwendeten Technologien, insbesondere die Virtualisierung mit QEMU/KVM und die Methode des Fuzzings. Kapitel 3 geht auf Bedrohungsszenarien für UEFI-Umgebungen ein.

(ii) Eine Beschreibung der Konzeption und Implementierung des Fuzzers findet sich in Kapitel 4, wobei die technischen Besonderheiten einer UEFI-Umgebung und deren Bedeutung für das Fuzzing herausgestellt werden. In Kapitel 5 wird mit dem Fuzzer eine spezifische UEFI-Komponente getestet. Die Wiederentdeckung eines bekannten Fehlers zeigt dabei die Funktionsfähigkeit der Software.

(iii) In den Kapiteln 6 und 7 werden die Ergebnisse ausgewertet und ein Fazit gezogen.

(iv) Kapitel 8 gibt einen Ausblick auf mögliche Weiterentwicklungen des Ansatzes.

2 Grundlagen

2.1 Unified Firmware Interface und Platform Initialization

2.1.1 Grundlagen

UEFI und Platform Initialization (PI) sind komplementäre Standards, die den Bootvorgang und die Schnittstellen zwischen einem Betriebssystem und der Geräte-Firmware definieren.

Noch in den späten 90er-Jahren benutzten Mainboardhersteller ausschließlich proprietäre Implementierungen des BIOS, das allerdings zunehmend technische und konzeptionelle Schwächen zeigte. So mussten etwa neuere 32-Bit-Betriebssysteme weiterhin im 16-Bit-Modus gestartet werden, und das Master-Boot-Record-Partitionsschema beschränkte die Größe der Bootpartition. Weitere Nachteile waren die fehlende Standardisierung und die Verwendung von Assemblercode anstelle von Hochsprachen. Während der Entwicklung der Itanium-Architektur entschied Intel deshalb, einen BIOS-Nachfolger zu konzipieren [vgl. 6, S. 8 ff.]. Das im Jahr 2000 vorgestellte Extensible Firmware Interface (EFI) löste nicht nur die technischen Probleme, sondern war zudem eine offene Spezifikation und plattformneutral. Eine breite Akzeptanz dieses deutlich moderneren Ansatzes führte dazu, dass im Jahr 2005 ein Zusammenschluss mehrerer Hersteller das Unified EFI Forum gründete und die Spezifikation unter der Bezeichnung UEFI weiterentwickelte. Der UEFI-Standard erschien erstmals im Jahr 2006 und wird seitdem laufend aktualisiert. Zum Zeitpunkt dieser Arbeit liegt die Version 2.7 vor.

UEFI ist jedoch eine reine Schnittstellendefinition, die lediglich beschreibt, wie innerhalb einer festgelegten Umgebung ein Betriebssystem gestartet wird und anschließend mit der Firmware interagieren kann. Die genauen Initialisierungsschritte für eine UEFI-kompatible Firmware sind in der separaten PI-Spezifikation definiert. Diese wird seit dem Jahr 2006 ebenfalls von einer Arbeitsgruppe des UEFI Forums betreut und hat momentan die Version 1.6.

2.1.2 Phasen

Eine UEFI-Umgebung durchläuft nach dem Systemstart eine Sequenz von sieben Phasen. Die ersten drei Phasen sind in der PI-Spezifikation definiert und sorgen für die Initialisierung der Hard- und Firmware. In den nächsten beiden Phasen wird ein Betriebssystem gesucht und gestartet. Ein Teil der UEFI-Umgebung verbleibt weiterhin im Speicher, sodass das Betriebssystem auf Funktionen der Firmware zugreifen kann. Die letzte Phase betrifft die Abschaltung des Systems.

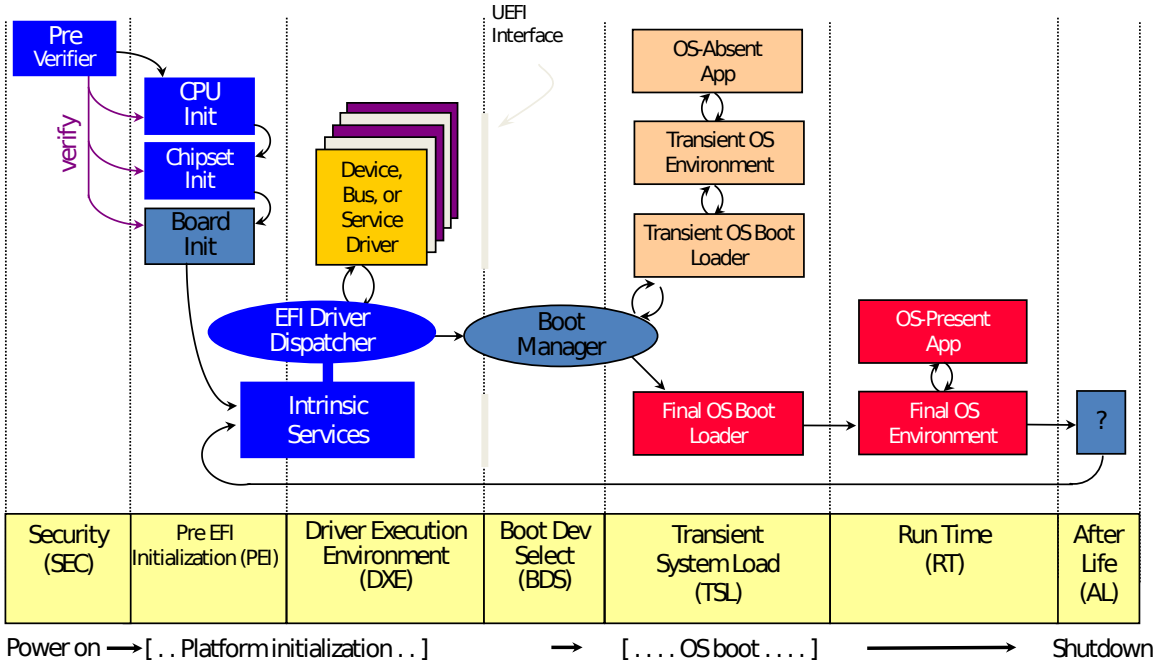

Abbildung 1: UEFI/PI-Phasen, Quelle: [7]

2.1.2.1 Security

Die Security-Phase (SEC) übernimmt die rudimentäre Initialisierung der Hardware und ist zudem der Root of Trust, also der Ausgangspunkt aller Sicherheitsmaßnahmen. Der SEC-Firmware selbst wird implizit vertraut.

Unmittelbar nach dem Start befindet sich die CPU im 16-Bit Real Mode mit lediglich einem aktiven Kern, der als Bootstrap Processor (BSP) bezeichnet wird. Der BSP beginnt die Dekodierung und Ausführung von Instruktionen bei dem Reset Vector. Diese architekturspezifische physische Adresse – 0xFFFFFFF0 im Fall von x86-Prozessoren – zeigt auf eine Sprunganweisung im ROM, die wiederum den Anfang des SEC-Codes als Ziel hat.

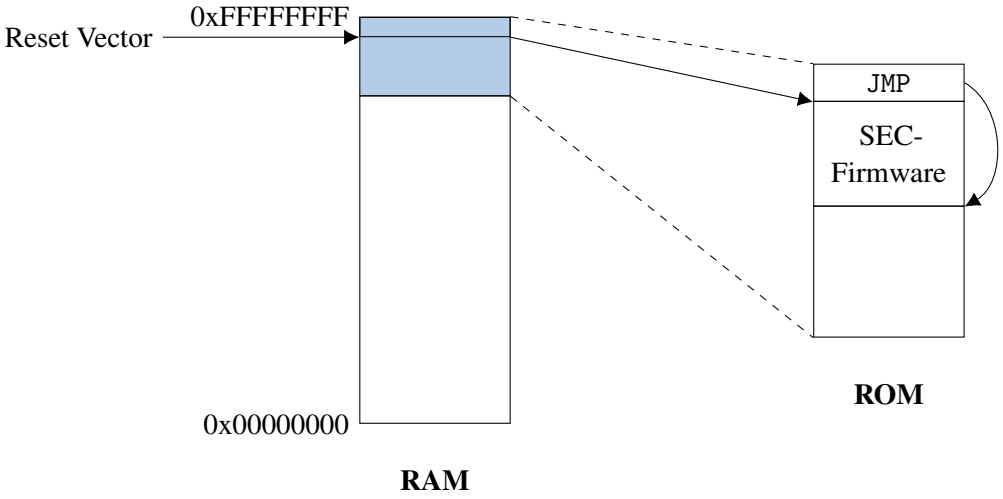

Abbildung 2: Reset Vector

In der SEC-Phase wird die CPU je nach Firmware in den 32-Bit oder 64-Bit Protected Mode versetzt. Weil der Hauptspeicher noch nicht konfiguriert ist, richtet die Firmware den CPU-Cache als temporären Speicher ein und erzeugt einen Callstack. Die SEC-Phase fragt außerdem den CPU-Status ab und kann optional die Integrität der Pre-EFI-Initialization-Firmware prüfen. Zuletzt werden alle relevanten Informationen wie die Größe des temporären Speichers sowie die Größe und Position des Stacks an die nachfolgende Phase übergeben.

2.1.2.2 Pre-EFI Initialization

Hauptaufgabe der Pre-EFI-Initialization-Phase (PEI) ist es, den Bootmodus zu ermitteln und alle nötigen Ressourcen für die komplexe Driver-Execution-Environment-Phase bereitzustellen. Insbesondere muss der Arbeitsspeicher konfiguriert werden.

Die PEI-Firmware ist unterteilt in eine Kernkomponente – die PEI Foundation – und mehrere Pre-EFI Initialization Modules (PEIMs), die spezifische Initialisierungsschritte übernehmen. PEIMs können über so genannte PEIM-to-PEIM Interfaces (PPIs) miteinander kommunizieren. Diese standardisierten Schnittstellen werden von den Modulen registriert und enthalten Funktionszeiger oder Daten. Der Dispatcher der PEI Foundation ermittelt die Abhängigkeiten zwischen den PEIMs anhand der PPIs und ruft die Module in der richtigen Reihenfolge auf, wobei das letzte Modul stets das DXE Initial Program Load PEIM (DXE IPL PEIM) ist, das die nächste Phase einleitet. Für die Übergabe von Daten wird eine verkettete Liste von Hand-Off Blocks (HOBs) benutzt. HOBs sind typisierte Container und beschreiben beispielsweise die CPU oder den physischen Speicher.

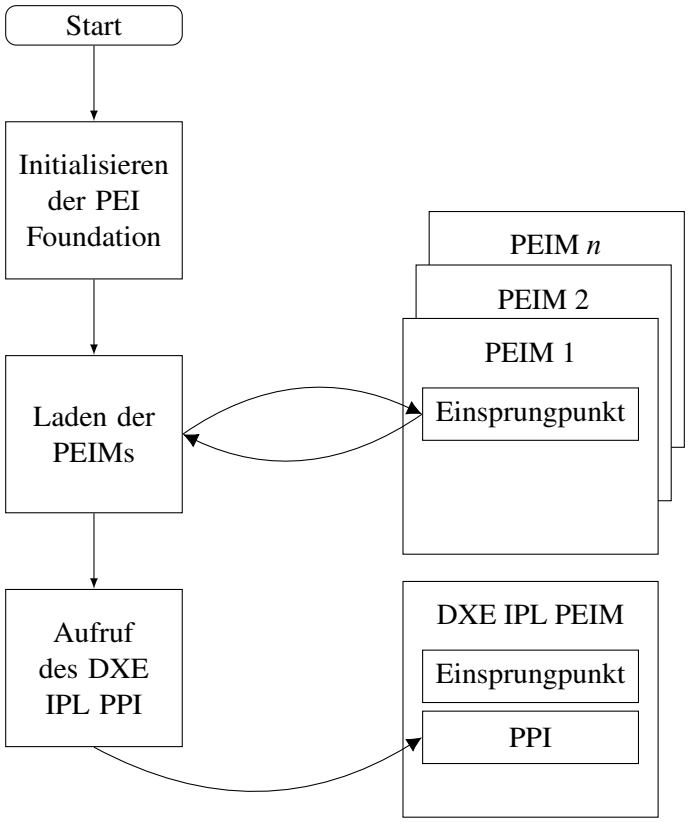

Abbildung 3: Ablauf der PEI-Phase, [vgl. 8, Vol. 1, Kap. 2.3, S. 11]

Die konkreten Schritte der PEI-Phase sind architekturabhängig. Weil allerdings kaum Ressourcen zur Verfügung stehen, werden die Module generell auf die wichtigsten Funktionen beschränkt.

2.1.2.3 Driver Execution Environment

Der größte Teil der Systeminitialisierung findet in der Driver-Execution-Environment-Phase (DXE) statt. Analog zu der PEI-Phase gibt es eine DXE Foundation und mehrere Treiber, die von einem Dispatcher aufgerufen werden. Während die DXE Foundation grundlegende Services bereitstellt, sind die Treiber dafür zuständig, alle für den Bootvorgang relevanten Hardwarekomponenten zu initialisieren. Dazu gehören beispielsweise die CPU, der Chipsatz, Bussysteme wie USB, Laufwerke oder Netzwerkkarten. Zudem implementieren DXE-Treiber wichtige Softwarefunktionen wie Netzwerkprotokolle oder kryptografische Algorithmen. Die Schnittstellen der Treiber werden als Protokolle bezeichnet und sind in Kapitel 2.1.6 genauer erläutert. Nachdem die DXE-Treiber geladen wurden, ruft die DXE Foundation den Boot Manager auf.

Zentrale Datenstruktur der DXE-Phase ist die UEFI-Systemtabelle, die den Zugriff auf alle Services sowie Konfigurationsdaten ermöglicht und auch noch in den nachfolgenden Phasen zur Verfügung steht.

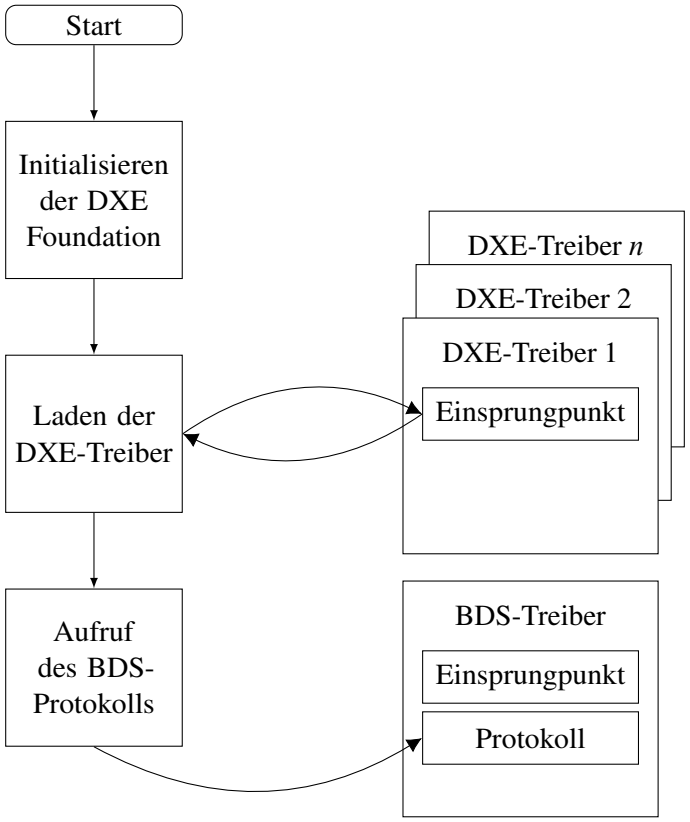

Abbildung 4: Ablauf der DXE-Phase, [vgl. 8, Vol. 1, Kap. 2.3, S. 11]

2.1.2.4 Boot Device Selection

Die Boot-Device-Selection-Phase (BDS) versucht eine gültige Bootoption zu finden. Dies ist typischerweise der Bootloader eines Betriebssystems; es ist aber auch möglich, andere Applikationen wie beispielsweise die UEFI-Shell oder eine Diagnoseanwendung zu starten. Bootoptionen werden als UEFI-Variablen im Non-Volatile RAM (NVRAM) des Systems gespeichert und haben einen Namen sowie einen Pfad. In der `BootOrder`-Variablen ist die Reihenfolge der Bootoptionen festgelegt. Der Boot Manager durchläuft die Optionen nacheinander und versucht die jeweilige Anwendung zu starten. Ist dies erfolgreich, beginnt die nachfolgende Phase. Bei einem Fehler übernimmt wieder die DXE-Phase die Kontrolle.

2.1.2.5 Transient System Load

Die letzte Phase des Bootvorgangs ist die Transient-System-Load-Phase (TSL). Je nachdem, welche Anwendung gestartet wurde, kann der Benutzer noch mit dem System interagieren und gegebenenfalls zurück in die BDS-Phase wechseln. Andernfalls schließt der Bootloader den Bootprozess mit dem Aufruf von `ExitBootServices` ab. Es verbleiben nur noch die UEFI-Systemtabelle und die Runtime-Services im Speicher.

2.1.2.6 Runtime

Mit dem Start eines Betriebssystems beginnt die Runtime-Phase (RT). Das Betriebssystem kann die UEFI-Services nutzen und diese über eigene Schnittstellen auch dem Userspace zur Verfügung stellen, um beispielsweise den Zugriff auf UEFI-Variablen zu ermöglichen.

2.1.2.7 After Life

Wird das Betriebssystem beendet, geht die UEFI-Umgebung in die After-Life-Phase (AL) über. Die PI-Spezifikation definiert kein bestimmtes Verhalten, es steht also jedem Hersteller frei, die AL-Phase für eigene Funktionen zu nutzen.

2.1.3 Images

UEFI-Images sind ausführbare Programme in einer Variante des Portable-Executable-Formats (PE32+). Jedes Image hat einen Einsprungpunkt, dem die UEFI-Systemtabelle und das Image-Handle[2] als Argumente übergeben werden. Es gibt drei verschiedene Typen von Images.

Eine Applikation wird von dem Boot Manager oder einer anderen Applikation aufgerufen und nach dem Programmende aus dem Speicher entfernt. Beispiele sind die graphische UEFI-Benutzeroberfläche und die EFI-Shell.

Bootloader sind spezielle UEFI-Applikationen, die ein Betriebssystem starten. Ist dies erfolgreich, wird mit dem Aufruf von `ExitBootServices` die RT-Phase eingeleitet, andernfalls kann der Bootloader mit `Exit` die Kontrolle wieder an den Aufrufer übergeben.

Treiber stellen Funktionen für die UEFI-Umgebung zur Verfügung und verbleiben langfristig im Arbeitsspeicher. Nur bei einem Fehler während des Ladevorgangs wird das Image aus dem Speicher entfernt.

2.1.4 Variablen

Variablen ermöglichen den Austausch von Daten zwischen UEFI-Komponenten oder zwischen der UEFI-Umgebung und einem Betriebssystem.

Jede Variable hat einen Vendor-GUID, einen Namen, Attribute und einen Wert. Der GUID (Globally Unique Identifier) ist ein 128-Bit langer Bezeichner, der den Namensraum der Variablen festlegt. Es gibt standardisierte GUIDs für elementare UEFI-Variablen, zudem können neue Namensräume definiert werden. Der Variablenname ist ein menschenlesbarer UTF-16-String und innerhalb des Namensraums eindeutig. Mit den Attributen werden Eigenschaften der Variablen definiert.

[2] siehe Kapitel 2.1.6

Attribut	Bedeutung
NON_VOLATILE	Die Variable wird persistent im NVRAM des Systems gespeichert.
BOOTSERVICE_ACCESS	Die Variable existiert während der Bootphasen.
RUNTIME_ACCESS	Die Variable existiert in der Runtime-Phase; dies impliziert BOOTSERVICE_ACCESS.
AUTHENTICATED_WRITE_ACCESS	Die Daten der Variablen müssen kryptografisch signiert sein.
TIME_BASED_AUTHENTICATED_WRITE_ACCESS	Die Daten sind signiert und zusätzlich mit einem Zeitstempel versehen.

Abbildung 5: Variablenattribute

2.1.5 Services

Services sind elementare Funktionen, die beispielsweise Speicher allozieren oder den Wert einer UEFI-Variablen ändern. Die Signatur und das Verhalten sind dabei in der UEFI-Spezifikation definiert. Boot-Services existieren nur während der Bootphasen, wohingegen Runtime-Services auch noch in der RT-Phase von dem Betriebssystem aufgerufen werden können.

Für den Aufruf wird die UEFI-Systemtabelle benutzt, die auf eine Boot-Services- und eine Runtime-Services-Tabelle verweist. Diese Tabellen enthalten wiederum Funktionszeiger für die einzelnen Services.

2.1.5.1 Boot-Services

Die Boot-Services lassen sich in fünf Kategorien einteilen:

- Services zur Erzeugung von Events oder Timern
- Speicherverwaltung
- Protokoll-Services
- Services zum Laden, Starten und Entfernen von Images
- verschiedene System- und Hilfsfunktionen

Eine besondere Rolle spielt der `ExitBootServices`-Service, der den Bootvorgang beendet und die Kontrolle an ein Betriebssystem übergibt.

2.1.5.2 Runtime-Services

Runtime-Services können wie Boot-Services von UEFI-Komponenten genutzt werden. Zusätzlich geben sie dem Betriebssystem die Möglichkeit, mit der UEFI-Umgebung zu interagieren, um Informationen abzufragen, Daten zu ändern oder das System zu steuern.

Service	Aufgabe
`GetVariable`	gibt den Wert und die Attribute einer Variablen zurück
`GetNextVariableName`	ermöglicht das Iterieren über alle Variablen
`SetVariable`	erstellt, ändert oder löscht eine Variable
`QueryVariableInfo`	liefert Informationen über den Variablen-Speicher
`GetTime`	gibt den Zeitstempel der Echtzeituhr zurück
`SetTime`	ändert das Datum oder die Zeit
`GetWakeupTime`	gibt den Zeitstempel des automatischen Starts zurück
`SetWakeupTime`	ändert den Zeitpunkt des automatischen Starts
`SetVirtualAddressMap`	aktiviert die virtuelle Adressierung
`ConvertPointer`	konvertiert eine physische in eine virtuelle Adresse
`GetNextHighMonotonicCount`	gibt den Wert eines globalen Zählers zurück
`ResetSystem`	löst einen Reset aus
`UpdateCapsule`	initialisiert ein Firmware-Update
`QueryCapsuleCapabilities`	überprüft ein Firmware-Update

Abbildung 6: Runtime-Services

2.1.6 Protokolle, Treiber und Handles

Der weit überwiegende Teil der UEFI-Funktionen wird über Protokolle bereitgestellt. Ein Protokoll ist eine Schnittstelle, die von anderen Komponenten genutzt werden kann, um beispielsweise mit der Hardware zu kommunizieren, kryptografische Berechnungen durchzuführen oder ein Netzwerkprotokoll wie TCP zu nutzen. Standardisierte Protokolle sind in der UEFI-Spezifikation festgelegt. Zusätzlich können neue Protokolle definiert werden, um die UEFI-Umgebung zu erweitern.

Technisch besteht ein UEFI-Protokoll aus einem GUID und einer optionalen Protocol Interface Structure mit Funktionszeigern und beliebigen Daten. Protokollinstanzen können außerdem private Daten haben, die ausschließlich intern genutzt werden und für den Protokollnutzer nicht sichtbar sind.

Implementiert werden Protokolle von Treibern, die konkrete Funktionen für die Protocol Interface Structure definieren und die Instanz an einen Handle koppeln. Handles repräsentieren physische Geräte oder Softwarekomponenten und können beliebig viele assoziierte Protokolle haben.

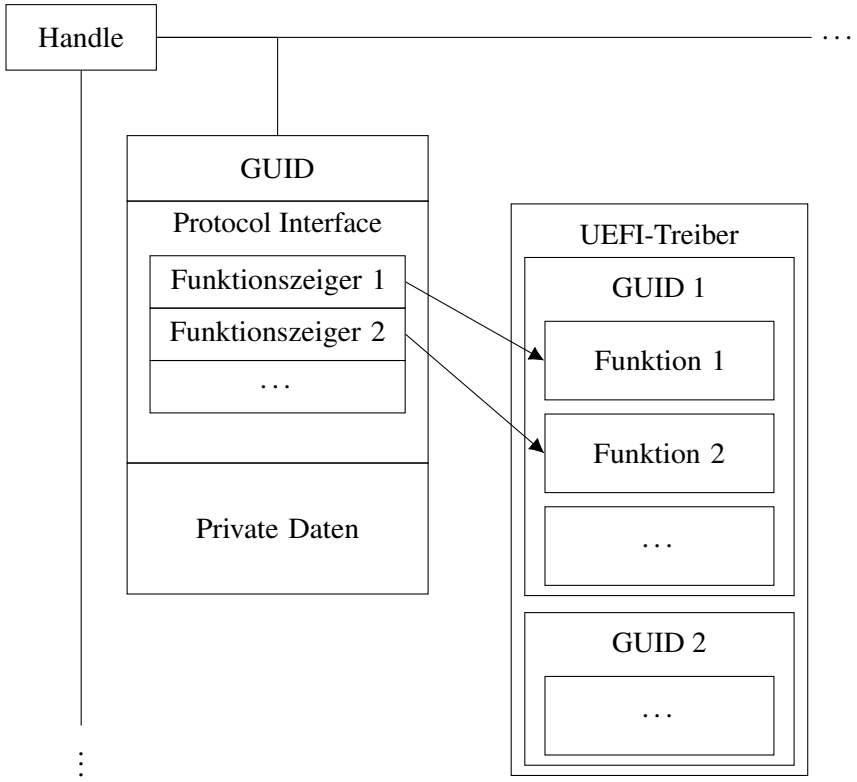

Abbildung 7: Schematische Darstellung eines UEFI-Protokolls, [vgl. 9, S. 19, Abb. 2.3]

Alle Handles sind in der Handle-Datenbank abgelegt, auf die mit den Boot-Services zugegriffen werden kann. `InstallProtocolInterface` und `UninstallProtocolInterface` registrieren beziehungsweise entfernen ein Protokoll. Mit `OpenProtocol` lässt sich ein Protokoll eines bestimmten Handles nutzen. Umgekehrt ist es möglich, mit dem `LocateHandleBuffer`-Service alle Handles zu suchen, die ein spezifisches Protokoll bereitstellen.

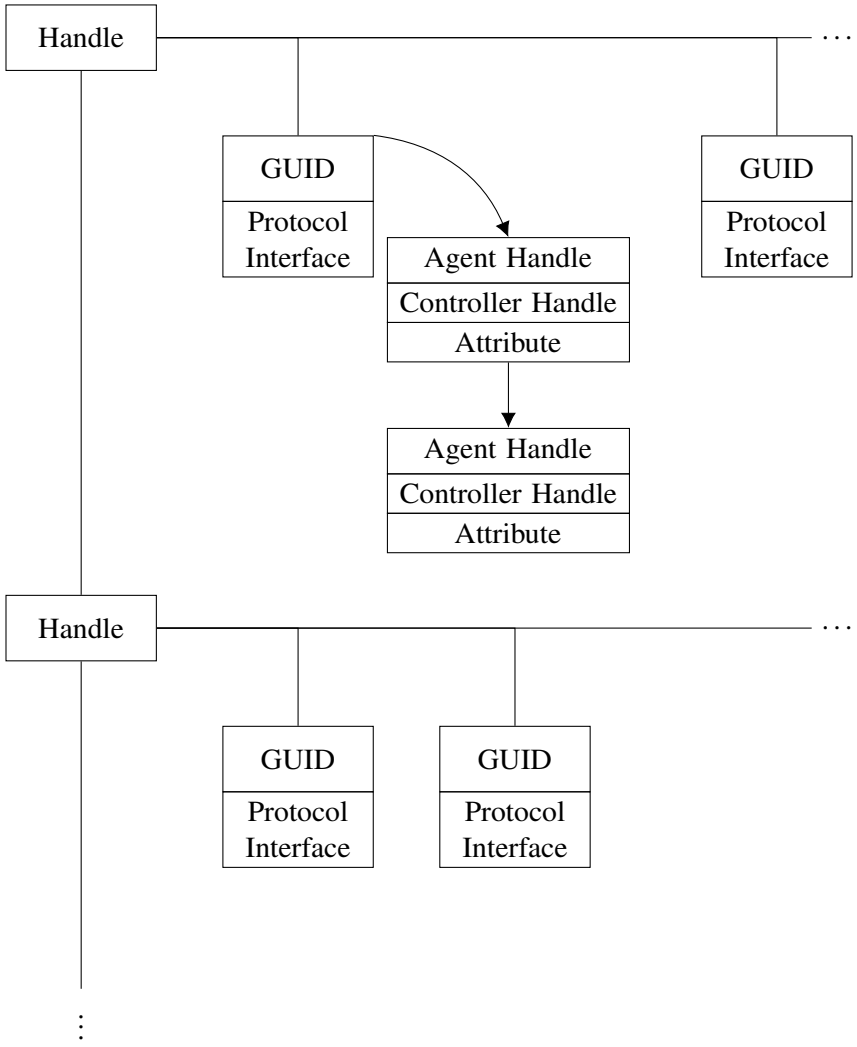

Abbildung 8: Handle-Datenbank, [vgl. 9, S. 17, Abb. 2.1]

2.1.7 Referenzimplementierung

Das EFI Development Kit II (EDK II) ist ein Open-Source-Projekt, das Referenzimplementierungen der UEFI/PI-Spezifikation für verschiedene Plattformen bereitstellt[3]. Die Codebasis wird ständig weiterentwickelt und kann unter den Bedingungen der BSD-Lizenz genutzt, verändert und erweitert werden.

In unregelmäßigen Abständen verifiziert Intel eine Teilmenge der jeweils aktuellen EDK II-Version und veröffentlicht diese als UEFI Development Kit (UDK). Das UDK ist produktiv einsetzbar und eignet sich beispielsweise als Firmware-Grundlage für Hardwarehersteller. Derzeit steht das UDK2017 zur Verfügung[4].

Ein Unterprojekt des EDK II ist die Open Virtual Machine Firmware (OVMF), die von Hypervisoren wie QEMU/KVM oder Xen als virtuelle UEFI-Umgebung genutzt werden kann[5].

2.2 Virtualisierung mit QEMU und KVM

2.2.1 QEMU

Der Quick Emulator (QEMU) ist eine Open-Source-Virtualisierungssoftware, die ein vollständiges System im Userspace emuliert. Dies schließt Peripheriegeräte wie Grafikkarten, Festplatten, CD-ROM-Laufwerke oder Netzwerkkarten ein.

QEMU wird über die Kommandozeile gestartet und kann mit zahlreichen Parametern konfiguriert werden. Der folgende Aufruf erzeugt beispielsweise ein x86-64-System mit 2 GiB Arbeitsspeicher, einer Festplatte und einer virtuellen Netzwerkkarte, die ein TAP-Device als Backend benutzt.

```
qemu-system-x86_64 \
        -kvm \
        -m 2G \
        -hda drive.img \
        -netdev tap,id=net0,ifname=tap0,script=no,downscript=no \
        -device virtio-net-pci,netdev=net0,romfile=
```

Obwohl QEMU in der Lage ist, auch CPUs softwareseitig zu emulieren, hat dies im Vergleich zu hardwarebasierten Lösungen negative Auswirkungen auf die Performance. Wenn die Architekturen von Host- und Gastsystem übereinstimmen, wird die CPU-Emulation deshalb typischerweise an das KVM-Modul delegiert.

[3] https://github.com/tianocore/tianocore.github.io/wiki/EDK-II
[4] https://github.com/tianocore/tianocore.github.io/wiki/UDK2017
[5] https://github.com/tianocore/tianocore.github.io/wiki/OVMF

2.2.2 KVM

Die Kernel-based Virtual Machine (KVM) ist ein Modul für den Linux-Kernel, das mithilfe der Virtualisierungsfunktionen moderner Intel- und AMD-CPUs einen Hypervisor implementiert. Damit ist es möglich, in einem Linux-Betriebssystem virtuelle Maschinen zu betreiben.

Neben dem eigentlichen `kvm`-Kernelmodul wird je nach CPU entweder das `kvm_intel`-Modul für die VT-x-Technologie oder das `kvm_amd`-Modul für AMD-V geladen. Weil der hier vorgestellte Ansatz Intel-spezifische Features verwendet, beziehen sich die folgenden Beschreibungen auf VT-x, allerdings gelten die grundsätzlichen Konzepte auch für AMD-CPUs.

Die `VMXON`-Instruktion aktiviert die Virtualisierungsfunktionen. Anfangs befindet sich die CPU im VMX-Root-Modus, der für das Hostsystem verwendet wird. Um eine virtuelle Maschine zu starten, legt der Host eine Virtual Machine Control Structure (VMCS) an. Mit dieser Datenstruktur wird der Wechsel zwischen Host und Gast konfiguriert und jeweils der Zustand der CPU gespeichert. Nach der Initialisierung lässt sich die CPU mit `VMXLAUNCH` in den VMX-Non-Root-Modus versetzen, in dem das Gastsystem läuft. Dieser Übergang heißt VM-Entry. Bestimmte Ereignisse wie Interrupts lösen einen VM-Exit aus, also die Rückkehr in das Hostsystem. Der Host kann jeweils mit `VMXRESUME` wieder in den VMX-Non-Root-Modus wechseln, bis die virtuelle Maschine nicht mehr benötigt wird. Für die vollständige Deaktivierung der Virtualisierungsfeatures sorgt `VMXOFF`.

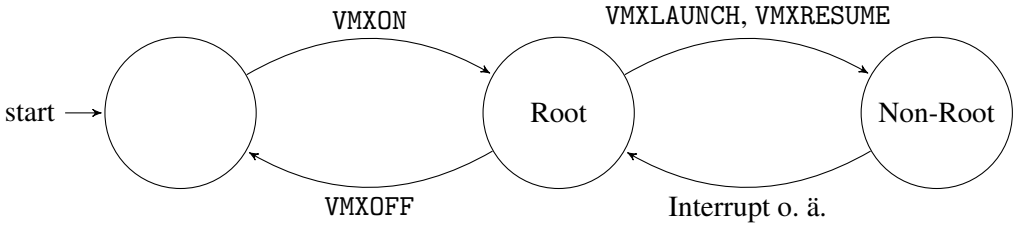

Abbildung 9: der VT-x-Zustandsautomat

2.2.3 QEMU-Monitor und QEMU Machine Protocol

Mit dem QEMU-Monitor können virtuelle Maschinen gesteuert werden. Dabei gibt es zwei verschiedene Betriebsmodi. Der Parameter `-mon mode=readline` bewirkt, dass QEMU das Human Monitor Protocol (HMP) verwendet und eine Konsole für Benutzereingaben bereitstellt. Beispielsweise lässt sich mit dem Befehl `system_powerdown` die Maschine herunterfahren. Wird QEMU mit dem Parameter `-mon mode=control` gestartet, benutzt der Monitor das QEMU Machine Protocol (QMP). QMP ist ein nachrichtenorientiertes Protokoll auf Grundlage des Datenformats JavaScript Object Notation (JSON) und ermöglicht es anderen Programmen, mit der virtuellen Maschine zu kommunizieren [vgl. 10]. Dies kann vollkommen automatisiert ohne jegliche Benutzerinteraktion geschehen.

Üblicherweise ist der Monitor mit der Standardeingabe und -ausgabe des QEMU-Prozesses verbunden. Die Kommunikation über Unix-Domain-Sockets oder ein Netzwerk ist aber ebenfalls möglich. Jede QMP-Session beginnt mit einem Handshake, bei dem der Monitor eine Nachricht mit der QEMU-Version und einer Liste von unterstützten Zusatzfunktionen sendet.

```
{
        "QMP": {
                "version": {
                        "qemu": {
                                "micro": 0,
                                "minor": 10,
                                "major": 2
                        },
                        "package": ""
                },
                "capabilities": [
                ]
        }
}
```

Im Fall einer leeren Capabilities-Liste sind nur Standardfunktionen verfügbar.

Nach der initialen Nachricht befindet sich der Monitor im Capabilities-Negotiation-Modus und wartet darauf, dass der Client die angebotenen Funktionen mit dem `qmp_capabilities`-Befehl aktiviert. Anschließend wechselt der Monitor in den Befehlsmodus. Der Client kann nun Anfragen oder Befehle als JSON-Dokumente senden und bekommt jeweils eine Antwort, die entweder die angeforderten Informationen, eine Bestätigung oder eine Fehlermeldung enthält.

Neben Befehlen gibt es im QMP-Modus auch asynchrone Ereignisse, die jederzeit von der virtuellen Maschine oder QEMU selbst ausgelöst werden können, um Fehler und Zustandsänderungen anzuzeigen. Dazu gehört beispielsweise ein Reset der Maschine.

```
{
        "event": "RESET",
        "timestamp": {
                "seconds": 1267041653,
                "microseconds": 9518
        }
}
```

2.2.4 pvpanic-Device

Das pvpanic-Device ist ein virtuelles ISA-Gerät, mit dem ein Gastsystem dem Host signalisieren kann, dass ein schwerwiegender Fehler wie beispielsweise eine Kernel-Panic des Betriebssystems aufgetreten ist [vgl. 11].

Wird das Device mit dem Kommandozeilenparameter `-device pvpanic` eingebunden, kann der Gast über den I/O-Port 0x505 jeweils ein Byte lesen oder schreiben, wobei jedes Bit eine Funktion des Geräts repräsentiert. Beim Lesen lassen sich die unterstützten Funktionen abfragen, beim Schreiben werden sie ausgelöst. Aktuelle QEMU-Versionen unterstützen lediglich das Bit 0, um einen Systemabsturz zu melden. In diesem Fall tritt das QMP-Ereignis `GUEST_PANICKED` auf.

2.2.5 Debugkonsole

Neben dem pvpanic-Device unterstützt QEMU eine Debugkonsole, mit der ein Gast Fehlermeldungen oder Debug-Informationen an den Host senden kann.

Wird QEMU mit den folgenden Kommandozeilenparametern gestartet, erscheint die Debugkonsole im Gastsystem als ISA-Gerät an I/O-Port 0x402. Die geschriebenen Daten werden auf der Host-Seite in die Datei `debug.log` umgeleitet.

```
-debugcon file:ovmf.log \
-global isa-debugcon.iobase=0x402
```

2.2.6 Migrationen

Migrationen sind eine Möglichkeit, den Zustand einer VM zu übertragen oder zu speichern [vgl. 12]. Dadurch lassen sich beispielsweise zwei Maschinen miteinander synchronisieren, sodass bei einem Ausfall die jeweils andere VM unmittelbar als Ersatz bereitsteht. Der Zustand kann mit einer so genannten Pseudo-Migration auch in einer Datei gespeichert und anschließend wiederhergestellt werden, um das zeitaufwendige Starten des Betriebssystems oder einzelner Programme zu überspringen. Dies ist eine Alternative zu Snapshots, bei denen ein QEMU-Image in einem geeigneten Format benötigt wird.

Im HMP-Modus sorgt der folgende Befehl dafür, dass QEMU den aktuellen Zustand der virtuellen Maschine in einer komprimierten Datei speichert.

```
migrate "exec: gz --stdout > vm-state.gz"
```

Um den Zustand wiederherzustellen, wird eine neue virtuelle Maschine mit dem folgenden Parameter gestartet.

```
-incoming "exec: gz --decompress --stdout vm-state.gz"
```

2.2.7 UEFI

QEMU verwendet als BIOS für virtuelle Maschinen standardmäßig die Open-Source-Firmware SeaBIOS [vgl. 13], allerdings lassen sich auch andere BIOS- oder UEFI-Implementierungen wie OVMF nutzen. Die Firmware muss dazu als Image vorliegen und wird mit dem Parameter `-drive if=pflash` als virtueller Flash-Speicher eingebunden.

Eine Besonderheit von OVMF ist die Unterteilung in zwei Images. `OVMF_CODE.fd` enthält die eigentliche Firmware und sollte schreibgeschützt sein, wohingegen `OVMF_VARS.fd` als Speicher für persistente UEFI-Variablen dient [vgl. 14]. Die Images werden direkt hintereinander in den physischen Arbeitsspeicher eingeblendet.

```
-drive if=pflash,format=raw,readonly,file=OVMF_CODE.fd \
-drive if=pflash,format=raw,file=OVMF_VARS.fd
```

2.3 Fuzzing

2.3.1 Grundlagen

Fuzzing ist ein Ansatz zur Fehlersuche in Software, bei dem automatisiert Eingabedaten erzeugt und als Input an das zu testende Programm übergeben werden. Anschließend beobachtet der Fuzzer das Programmverhalten, um Abstürze oder andere Fehlersymptome zu erkennen.

Durch Fuzzing sollen insbesondere ungewöhnliche Ausführungspfade getestet werden. In komplexer Software ist es oft nicht mehr möglich, jeden Pfad mit manuell geschriebenen Tests abzudecken, sodass Fehler möglicherweise unentdeckt bleiben. Ein Fuzzer kann diese Lücke schließen.

2.3.2 Fuzzing-Methoden

Es gibt zahlreiche Varianten von Fuzzern und Kombinationen mit anderen Techniken wie der statischen Codeanalyse. Hier soll jedoch nur ein grober Überblick gegeben werden.

2.3.2.1 Blackbox-Fuzzing

Bei dem Blackbox-Fuzzing werden die Eingabedaten „blind“ generiert, also ohne das Verhalten des Programms zu berücksichtigen. Der Input kann entweder rein zufällig sein oder eine vorher definierte Struktur haben. Fuzzer dieser Art sind einfach zu implementieren, allerdings ist die Effizienz gering. Weil keine gezielte Suche nach neuen Pfaden stattfindet, ist damit zu rechnen, dass typische Pfade immer wieder durchlaufen werden, wohingegen der Fuzzer komplexe, seltene Fälle kaum findet. Bei einem relativ kleinen Suchraum ist der Ansatz dennoch geeignet.

2.3.2.2 Graybox-Fuzzing

Graybox-Fuzzer benutzen Heuristiken, um eine möglichst hohe Codeabdeckung zu erreichen. Dazu erfasst der Fuzzer in jeder Iteration Kontrollflussinformationen und sucht gezielt nach neuen Testdaten, die entweder zu Variationen bekannter Pfade oder vollkommen neuen Pfaden führen. Diese Methode hat sich als sehr effektiv erwiesen und wird mittlerweile routinemäßig zur Fehler- und Schwachstellensuche eingesetzt.

Bei der Implementierung kommen häufig Mutations-Selektions-Verfahren zum Einsatz. Zuerst muss ein initialer Testkorpus generiert werden, der manuell oder automatisch erzeugte Eingabedaten enthält. In einer Schleife wählt der Fuzzer jeweils ein Testdatum aus, mutiert es und analysiert den Kontrollfluss. Wenn der neue Testfall zu einem bisher unbekannten Pfad geführt hat, wird das Datum in den Korpus aufgenommen, andernfalls verworfen. Die Codeabdeckung vergrößert sich also immer weiter.

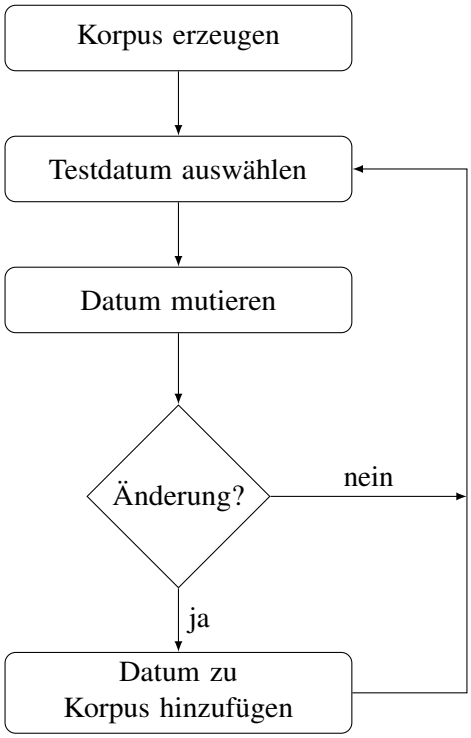

Abbildung 10: Schematische Darstellung von Graybox-Fuzzing

Ein typischer Ansatz bei Graybox-Fuzzern besteht darin, den Quelltext oder die ausführbare Datei zu instrumentieren. Dabei werden Instruktionen eingefügt, die den Kontrollfluss aufzeichnen und die Daten an den Fuzzer weiterleiten. Diese Methode kommt beispielsweise bei american fuzzy lop (afl) zum Einsatz [vgl. 15]. Alternativ kann die CPU selbst Kontrollflussinformationen liefern, sodass keine Manipulation des Programms nötig ist. Ein Beispiel für ein solches CPU-Feature ist das im nächsten Kapitel genauer beschriebene Intel Processor Trace.

Um den Kontrollfluss möglichst kompakt zu kodieren, werden nur die Knoten oder Kanten des Kontrollflussgraphen (CFG) betrachtet. Die Knoten – auch Basic Blocks genannt – geben Aufschluss darüber, welche Teile des Codes ausgeführt wurden. Bei den CFG-Kanten gibt die Richtung zusätzlich die Reihenfolge der Ausführung an.

2.4 Intel Processor Trace

2.4.1 Grundlagen

Processor Trace (PT) ist ein Feature aktueller Intel-CPUs, das umfassende Informationen über den Kontrollfluss von Programmen, Kontextwechsel, den Betriebsmodus der CPU, Virtualisierungsoperationen und zeitliche Abläufe aufzeichnen kann [vgl. 16, Vol. 3C, Kap. 35, S. 35-1]. Damit ist eine genaue Rekonstruktion des Systemverhaltens möglich.

Im Vergleich zu Vorgängertechniken wie Branch Trace Store (BTS) ist PT sowohl präziser als auch performanter. Während BTS lediglich Verzweigungen aufzeichnet und das Programm um den Faktor 40 verlangsamen kann [vgl. 17], liefert PT ein Gesamtbild der Systemabläufe und hat einen Overhead von etwa 5% [vgl. 18]. Dabei muss allerdings berücksichtigt werden, dass die anschließende Analyse der Daten sehr aufwendig sein kann.

Intel PT hat verschiedene Anwendungsbereiche und ist in mehrere Tools integriert. Der GNU Debugger verwendet PT seit Version 7.10 für das Process-Record-and-Replay-Feature, das einen Prozess zuerst aufzeichnet und dem Benutzer anschließend die Möglichkeit gibt, die Programmausführung beliebig oft abzuspielen und sogar „zurückzuspulen“ [vgl. 19]. In dem perf-System des Linux-Kernels wird PT zur Performanceanalyse eingesetzt [vgl. 20]. Ein Trace kann beispielsweise die Grundlage für das Profiling eines Programmes sein, bei dem besonders zeit- oder rechenintensive Funktionen identifiziert werden, um die Anwendung gezielt zu optimieren.

Spezifisch zur Coverage-Analyse wird PT bereits in WinAFL [21] und honggfuzz [22] eingesetzt. kAFL [23] benutzt PT zum Fuzzen von Betriebssystemen in virtuellen Maschinen.

PT speichert alle aufgezeichneten Informationen als Binärpakete im Arbeitsspeicher. Diese Pakete werden dekodiert und lassen sich anschließend in Kombination mit den Executables zur Kontrollflussanalyse verwenden. Je nach Anwendungsfall muss das Betriebssystem zusätzlich so genannte Sideband-Informationen wie eine Beschreibung der Prozesswechsel bereitstellen.

2.4.2 Register

Processor Trace wird mit modellspezifischen Registern (MSR) konfiguriert, gesteuert und überwacht.

Das MSR `IA32_RTIT_CTL` enthält Konfigurationsbits für den Ausgabebereich, verschiedene Filter und die Auswahl der Pakettypen. Durch Setzen und Löschen des `TraceEn`-Bits wird das Tracing aktiviert beziehungsweise deaktiviert. Informationen über den aktuellen Zustand sowie Fehler sind in dem MSR `IA32_STATUS_CTL` hinterlegt. Weitere wichtige Register werden in den nächsten Kapiteln erläutert.

2.4.3 Table of Physical Addresses

Eine Table of Physical Addresses (ToPA) definiert Bereiche im physischen Arbeitsspeicher, in denen die CPU die Pakete ablegen kann. Jede ToPA besteht dabei aus beliebig vielen 64 Bit großen Einträgen, die jeweils eine Kachelnummer, eine Größenangabe und die drei Flags `STOP`, `INT` und `END` haben.

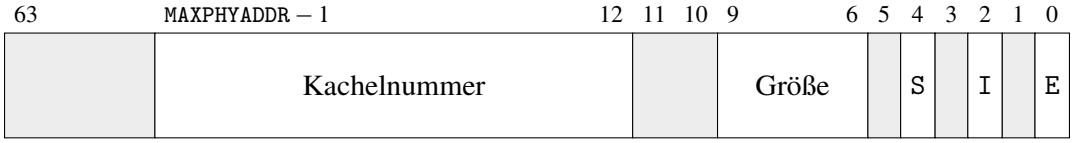

Abbildung 11: ToPA-Eintrag, [vgl. 16, Vol. 3C, S. 35-11, Abb. 35-2]

Ist bei einem Eintrag des `END`-Flag nicht gesetzt, dann zeigt die Kachelnummer auf einen nutzbaren Speicherbereich mit der angegebenen Größe. Logisch bilden alle Speicherbereiche einen zusammenhängenden Block.

Einträge mit gesetztem `END`-Flag markieren das Ende der Tabelle und zeigen auf die nächste ToPA. Auf diese Weise können mehrere Tabellen beliebig verkettet werden. Es ist auch möglich, die ToPA auf sich selbst verweisen zu lassen, um einen Ringpuffer zu implementieren.

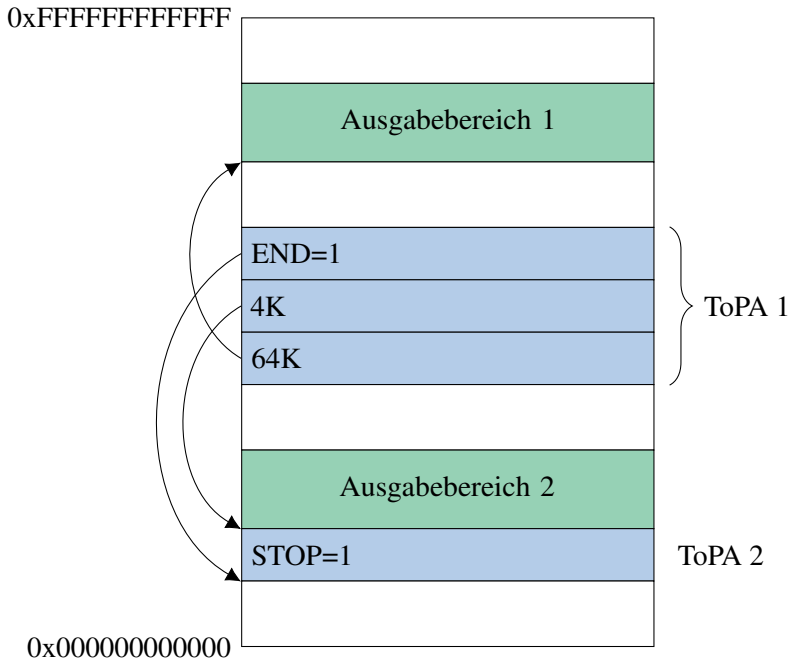

Abbildung 12: ToPAs im RAM, [vgl. 16, Vol. 3C, S. 35-10, Abb. 35-1]

Wenn das STOP-Flag gesetzt ist, beendet die CPU das Tracing, sobald der Puffer gefüllt ist. Bei gesetztem INT-Flag wird dagegen ein Interrupt ausgelöst. Der Interrupthandler kann den aktuellen Pufferinhalt verarbeiten und den Speicherbereich wieder für neue Pakete freigeben. Dies ist vor allem bei großen Traces sinnvoll, die nicht vollständig in den Arbeitsspeicher passen.

Um die aktuelle ToPA festzulegen, wird das MSR IA32_RTIT_OUTPUT_BASE auf den Anfang der Tabelle im physischen Speicher gesetzt. Wechselt die CPU während des Tracings auf eine andere ToPA, aktualisiert sie diesen Wert. Der aktuelle Tabelleneintrag wird als Offset zur Anfangsadresse angegeben und befindet sich im MSR IA32_RTIT_OUTPUT_MASK_PTRS. Dieses Register enthält außerdem die nächste Position im Ausgabepuffer, kodiert als Offset zur Kachelnummer.

Bei einer ungültigen ToPA setzt die CPU das Error-Bit des IA32_RTIT_STATUS-Registers.

2.4.4 Pakete

Processor Trace verwendet derzeit 25 verschiedene Pakettypen, von denen aber nur die Kontrollflusspakete für diese Arbeit relevant sind. Jedes Paket hat einen Opcode, mit dem der Typ des Pakets eindeutig identifiziert werden kann, und optionale Nutzdaten.

Ein Trace ist durch Packet-Stream-Boundary-Pakete (PSB) in voneinander unabhängige Abschnitte unterteilt. Die CPU erzeugt PSB-Pakete in regelmäßigen Abständen, wobei sich die Frequenz mit dem PSBFreq-Bit des IA32_RTIT_CTL-Registers konfigurieren lässt. Nach dem PSB-Paket folgt eine als PSB+ bezeichnete Sequenz von Paketen, die den Status der CPU beschreibt. Dazu gehören beispielsweise die Werte des CR3- und des TSC-Registers, mit denen der aktuelle Prozess und die Zeit ermittelt werden. Ein PSBEND-Paket schließt die Sequenz ab und markiert den Anfang der eigentlichen Aufzeichnung.

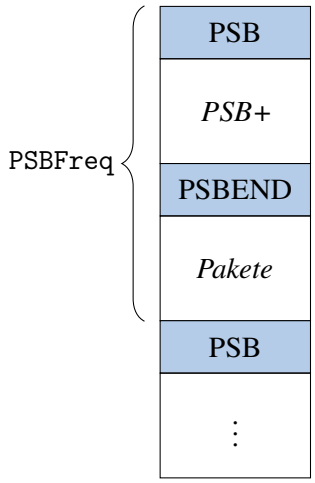

Abbildung 13: Struktur eines Traces

	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0
1	1	0	0	0	0	0	1	0
⋮	⋮							
14	0	0	0	0	0	0	1	0
15	1	0	0	0	0	0	1	0

Abbildung 14: PSB-Paket, [vgl. 16, Vol. 3C, Kap. 35.4.2.17, Tabelle 35-36]

	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0
1	0	0	0	0	0	0	1	1

Abbildung 15: PSBEND-Paket, [vgl. 16, Vol. 3C, Kap. 35.4.2.18, Tabelle 35-37]

2.4.4.1 Taken/Not-taken

Taken/Not-taken-Pakete (TNT) werden für bedingte Sprünge wie JZ erzeugt. Weil die Instruktion das Sprungziel als Offset enthält, müssen die Pakete lediglich angeben, ob der Sprung ausgeführt wurde. Diese Information ist als einzelnes Bit kodiert.

TNT-Pakete treten in zwei Varianten auf, die jeweils mehrere aufeinanderfolgende Sprünge zusammenfassen können. Kurze TNT-Pakete enthalten bis zu 6 Bits, lange Pakete bis zu 47 Bits. Die Bitfolge wird jeweils mit einem einzelnen Stop-Bit abgeschlossen.

	7	6	5	4	3	2	1	0
0	0/1	B_1	B_2	B_3	B_4	B_5	B_6	0

Abbildung 16: kurzes TNT-Paket, [vgl. 16, Vol. 3C, Kap. 35.4.2.2, Tabelle 35-16]

	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0
1	1	0	1	0	0	0	1	1
2	B_{40}	B_{41}	B_{42}	B_{43}	B_{44}	B_{45}	B_{46}	B_{47}
3	B_{32}	B_{33}	B_{34}	B_{35}	B_{36}	B_{37}	B_{38}	B_{39}
4	B_{24}	B_{25}	B_{26}	B_{27}	B_{28}	B_{29}	B_{30}	B_{31}
5	B_{16}	B_{17}	B_{18}	B_{19}	B_{20}	B_{21}	B_{22}	B_{23}
6	B_8	B_9	B_{10}	B_{11}	B_{12}	B_{13}	B_{14}	B_{15}
7	0/1	B_1	B_2	B_3	B_4	B_5	B_6	B_7

Abbildung 17: langes TNT-Paket, [vgl. 16, Vol. 3C, Kap. 35.4.2.1, Tabelle 35-16]

Ein Sonderfall sind `RET`-Instruktionen. Um diese möglichst kompakt zu kodieren, verwendet PT standardmäßig ein Verfahren namens Return Compression. Hierbei muss der Decoder die Rückkehradresse jedes Funktionsaufrufs auf einem Stack ablegen. Stimmt das Ziel einer `RET`-Instruktion mit der obersten Adresse des Stacks überein, so signalisiert PT dies mit einem einzigen TNT-Bit. Nur wenn die Rückkehradresse der Funktion zur Laufzeit überschrieben wurde und von dem erwarteten Wert abweicht, enthält der Trace ein Target-Instruction-Pointer-Paket mit der expliziten Adresse. Return Compression lässt sich deaktivieren, indem das `DisRETC`-Bit des `IA32_RTIT_CTL`-Registers gesetzt wird. In diesem Fall entfällt die Kodierung mit TNT-Paketen.

2.4.4.2 Target Instruction Pointer

Das Target-Instruction-Pointer-Paket (TIP) wird bei indirekten Sprüngen wie `JMP` (`FF /4`) und bei asynchronen Ereignissen generiert. Weil das Sprungziel nicht aus dem Maschinencode hervorgeht, sondern erst zur Laufzeit feststeht, enthält das Paket die explizite Adresse.

	7	6	5	4	3	2	1	0
0	IPBytes			0	1	1	0	1
1	TargetIP[7:0]							
2	TargetIP[15:8]							
3	TargetIP[23:16]							
4	TargetIP[31:24]							
5	TargetIP[39:32]							
6	TargetIP[47:40]							
7	TargetIP[55:48]							
8	TargetIP[63:56]							

Abbildung 18: TIP-Paket, [vgl. 16, Vol. 3C, Kap. 35.4.2.2, Tabelle 35-17]

Haben mehrere aufeinander folgende TIP-Pakete innerhalb eines PSB-Abschnitts denselben Adresspräfix, so können die Adressen komprimiert werden.In diesem Fall enthält nur noch das erste Paket den vollständigen Instruction Pointer, alle nachfolgenden Pakete lassen den Präfix weg, wobei die genaue Komprimierungsvariante in dem `IPBytes`-Feld festgelegt ist.

IPBytes-Wert	Bedeutung
0b000	Adresse unterdrückt
0b001	Adresse komprimiert mit 48-Bit-Präfix
0b010	Adresse komprimiert mit 32-Bit-Präfix
0b011	Vorzeichenerweiterung nötig, niedrigstwertige 48 Bits gegeben
0b100	Adresse komprimiert mit 16-Bit-Präfix
0b101	(reserviert)
0b110	keine Komprimierung
0b111	(reserviert)

Abbildung 19: IPBytes-Werte, [vgl. 16, Vol. 3C, Kap. 35.4.2.2, Tabelle 35-18]

Je nach Komprimierung variiert die Länge des Pakets. Die Adresse kann auch ganz unterdrückt sein, wenn sie sich außerhalb des aufgezeichneten Kontexts befindet. Dies tritt beispielsweise beim Verwenden von Filtern auf.

Es gibt zwei weitere Varianten von TIP-Paketen. Ein Packet-Generation-Enable-Paket (TIP.PGE) wird bei jedem Beginn des Tracings erzeugt und enthält die Adresse der ersten aufgezeichneten Anweisung. Bei einer Unterbrechung generiert die CPU ein Packet-Generation-Disable-Paket (TIP.PGD) mit der Adresse der nächsten Instruktion.

2.4.4.3 Flow Update

Flow-Update-Pakete (FUP) werden bei asynchronen Ereignissen wie Interrupts oder Exceptions erzeugt. Das FUP-Paket gibt an, an welcher Adresse im regulären Programmablauf das Ereignis aufgetreten ist. Danach folgt allgemein ein TIP-Paket mit der Adresse der Interrupt Service Routine, bei der die Ausführung fortgesetzt wird. FUP-Pakete sind ebenfalls komprimierbar.

	7	6	5	4	3	2	1	0
0	IPBytes			1	1	1	0	1
1	IP[7:0]							
2	IP[15:8]							
3	IP[23:16]							
4	IP[31:24]							
5	IP[39:32]							
6	IP[47:40]							
7	IP[55:48]							
8	IP[63:56]							

Abbildung 20: FUP-Paket, [vgl. 16, Vol. 3C, Kap. 35.4.2.6, Tabelle 35-22]

2.4.5 Filter

Standardmäßig zeichnet die CPU sämtliche Vorgänge auf, die während des Tracings stattfinden. Dazu gehören neben der Ausführung von Programmen im Userspace auch Systemaufrufe, Interrupts oder Virtualisierungsoperationen. Vollständige Traces werden deshalb in kurzer Zeit sehr groß – bis zu mehrere hundert MiB pro Sekunde können anfallen – und sind nur noch mit erheblichem Aufwand zu dekodieren. Zudem können sie die Kapazität oder Schreibleistung der Datenträger übersteigen, sodass Pakete verloren gehen.

Um das Tracing einzuschränken, unterstützt Processor Trace mehrere Filter. Zunächst ist es möglich, mit dem Current-Privilege-Level-Filter ausschließlich im Kernelmodus (Ring 0) oder Usermodus (Ring 3) zu tracen. Dazu wird entweder das `OS`-Bit oder das `User`-Bit des `IA32_RTIT_CTL`-Registers gesetzt.

Mit dem `CR3`-Filter lassen sich spezifische Prozesse aufzeichnen. Das `CR3`-Register der CPU enthält den für jeden Prozess eindeutigen Verweis auf ein Seitenverzeichnis oder eine Page-Directory-Pointer-Tabelle. Um den Filter zu aktivieren, werden der Zielwert in das `IA32_RTIT_CR3_MATCH`-Register geschrieben und das `CR3Filter`-Bit des `IA32_RTIT_CTL`-Registers gesetzt.

Noch präziser ist der Adressfilter, der das Tracing auf spezifische Programmabschnitte eingrenzen kann. Die Aufzeichnung beim Betreten eines bestimmten Adressbereichs abzuschalten ist ebenfalls möglich. Je nach CPU-Modell werden bis zu vier Bereiche unterstützt, deren Anfang und Ende jeweils in den Registern `IA32_RTIT_ADDR`*n*`_A` und `IA32_RTIT_ADDR`*n*`_B` festgelegt sind. Hat die Bitfolge `IA32_RTIT_CTL.ADDR`*n*`_CFG` den Wert 1, dann ist PT nur innerhalb dieser Region aktiv. Bei dem Wert 2 schaltet die CPU das Tracing ab, sobald der Wert des Befehlszählers in dem festgelegten Intervall liegt.

3 Bedrohungsszenarien

3.1 PEI- und DXE-Phase

Bei einem Soft-Reset greifen die PEI- und die DXE-Phase auf die Runtime-Variable `CapsuleUpdateData` zu, mit der das Betriebssystem ein Firmware-Update auslösen kann. Weil diese Routine eine Schnittstelle zwischen Ring 0 und dem mit maximalen Privilegien ausgestatteten System Management Mode (SMM) bildet, sind Schwachstellen besonders kritisch. In früheren EDK II-Versionen war es etwa aufgrund einer fehlerhaften Implementierung möglich, ein SMM-Rootkit zu installieren und das System vollständig und dauerhaft zu kompromittieren [vgl. 24].

3.2 BDS- und TSL-Phase

Die UEFI-Spezifikation definiert für die BDS- und TSL-Phase zahlreiche Protokolle, mit denen Images von Datenträgern oder Netzwerkhosts geladen werden können. Unterstützung gibt es beispielsweise für ATA/ATAPI- und SCSI-Laufwerke, USB-Devices, WLAN und Ethernet. Als Netzwerkprotokolle lassen sich unter anderem PXE, iSCSI, FTP, HTTP und HTTPS verwenden. Daneben werden Schnittstellen wie Bluetooth für Peripheriegeräte unterstützt.

Enthält ein Device-Treiber eine Schwachstelle, so kann es einem Angreifer mit physischem Zugriff gelingen, die UEFI-Umgebung zu kompromittieren. Im Fall eines fehlerhaften Netzwerktreibers ist sogar ein Angriff von einem anderen Host aus möglich. Ältere EDK II-Versionen waren beispielsweise anfällig für Pufferüberläufe bei der Verarbeitung von DHCP-Paketen [vgl. 2, Kap. 35, S. 34].

3.3 RT-Phase

Während der RT-Phase hat der Betriebssystemkern über die UEFI-Systemtabelle direkten Zugriff auf alle Runtime-Services. Viele Betriebssysteme bieten außerdem Userspace-Schnittstellen an, mit denen Ring-3-Prozesse zumindest einen Teil der Services nutzen können. So unterstützten aktuelle Linux-Versionen das efivarfs-Dateisystem, bei dem jede Datei eine Runtime-Variable repräsentiert [25]. Lese- und Schreiboperationen werden in `GetVariable`- beziehungsweise `SetVariable`-Aufrufe übersetzt. Ein lesender Zugriff ist uneingeschränkt möglich, Änderungen bleiben privilegierten Nutzern vorbehalten. Die Windows API stellt analog `GetFirmwareEnvironmentVariable` [26] und `SetFirmwareEnvironmentVariable` [27] zur Verfügung.

Hieraus ergeben sich folgende Angriffsszenarien:

- Ein Angreifer, der den Kernel kompromittiert hat, kann jeden Service direkt aufrufen und Schwachstellen ausnutzen.
- Je nach Betriebssystem ist auch ein Angriff über privilegierte Userspace-Accounts möglich.

4 Implementierung

4.1 Ansatz

Für diese Arbeit soll zunächst eine Infrastruktur entwickelt werden, mit der sich die UEFI-Services und einzelne DXE-Treiber innerhalb von OVMF fuzzen lassen. Eine geeigneter Ansatzpunkt ist eine UEFI-Applikation, die in der TSL-Phase läuft und dadurch vollständigen Zugriff auf alle Boot-Services, Runtime-Services und Treiber hat. Diese Methode ist auch insofern vorteilhaft, weil die VM kein Betriebssystem benötigt, sondern ausschließlich OVMF starten muss.

Die Applikation vermittelt zwischen dem Host, auf dem der Großteil der Fuzzing-Infrastruktur läuft, und der virtualisierten UEFI-Umgebung. Während der Host Testdaten generiert, die Codeabdeckung analysiert und Fehler aufzeichnet, leitet die Anwendung die Daten an den Service oder Treiber weiter, der getestet werden soll. Hiermit lässt sich ein großer Teil der obigen Szenarien abdecken. Beispielsweise können die Runtime-Services mit beliebigen Argumenten aufgerufen werden, um Fehler zu finden, die auch für einen Angreifer mit Ring-0-Privilegien ausnutzbar wären. Eine einzelne Komponente wie der Treiber des HTTP-Boot-Protokolls ist ebenfalls ein mögliches Ziel. Hier wird von einem Angreifer ausgegangen, der den als Boot-Option hinterlegten HTTP-Server kompromittiert hat oder in der Lage ist, den Netzwerkverkehr auf einen eigenen Server umzuleiten. Abläufe in der SEC- oder PEI-Phase können dagegen nicht oder nur eingeschränkt getestet werden, weil diese bei dem Übergang in die DXE-Phase bereits abgeschlossen sind.

Für die Coverage-Analyse wird PT verwendet. Um die Traces auf relevante Informationen zu beschränken, konfiguriert der Fuzzer den Adressfilter und zeichnet nur den Kontrollfluss innerhalb des spezifischen Treiberimages auf.

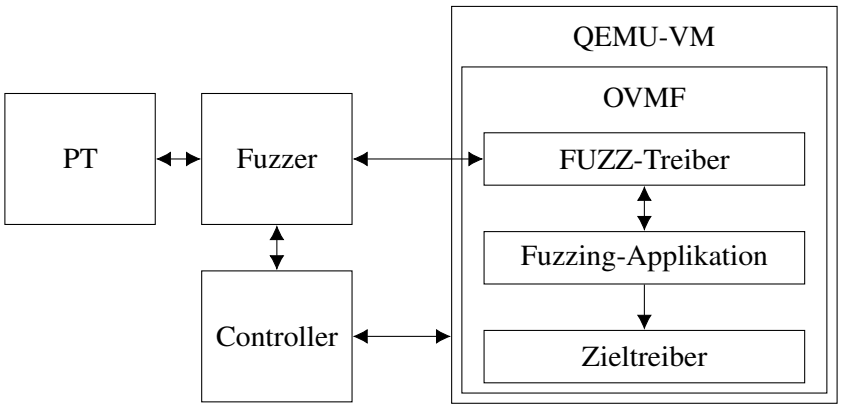

Abbildung 21: Komponenten des Fuzzers

4.2 Fuzzer

Das Google-Projekt honggfuzz[22] bildet die Grundlage der Fuzzing-Komponente. honggfuzz ist ein evolutionärer Fuzzer, der sich durch eine hohe Geschwindigkeit, Multithreading und die Unterstützung zahlreicher soft- und hardwarebasierter Feedbackmethoden auszeichnet. Eine rudimentäre PT-Anbindung auf Basis des perf-Systems ist bereits integriert.

Unterstützt werden unter anderem folgende Mutationstechniken.

- Manipulation zufällig gewählter Bytes durch Inkrementieren, Dekrementieren oder Negieren
- Einfügen von Zufallsdaten
- Vergrößern und Verkleinern des Inputs
- Verschieben von Datenbereichen
- Einsetzen spezieller Werte wie maximaler und minimaler Integer

Grundsätzlich ist honggfuzz darauf ausgelegt, Userspace-Prozesse über die Standardeingabe zu fuzzen. Durch einige Änderungen und einen Patch des `kvm_intel`-Moduls ist es jedoch möglich, den Fuzzer für die virtuelle UEFI-Umgebung anzupassen.

Als zusätzliche Argumente werden honggfuzz der Datei-GUID des Zielimages, der Pfad zu dem entsprechenden Executable und die IP-Adresse der UEFI-VM übergeben. Zudem wird die Kommunikation auf TCP umgestellt. Prinzipiell bietet honggfuzz bereits die Möglichkeit, die Testdaten an einen anderen Prozess zu senden, der diese wiederum über TCP an die VM weiterleiten könnte. Allerdings wird auch ein Rückkanal benötigt, um Daten der UEFI-Umgebung abzufragen.

Die Initialisierungsphase wird so angepasst, dass honggfuzz die Basisadressen der zu beobachtenden UEFI-Images ermittelt und den PT-Adressfilter entsprechend konfiguriert. Das Image wird disassembliert, um den Kontrollflussgraphen für die spätere Coverage-Analyse zu erzeugen [6].

Anschließend beginnt die Hauptschleife. In jeder Iteration aktiviert honggfuzz das Tracing, sendet die Daten über TCP an die VM und wartet auf das Ende des Tests. Sobald der Test abgeschlossen ist, wird PT deaktiviert, und die Auswertung der aufgezeichneten Daten kann erfolgen.

[6] siehe Kapitel 4.6

4.3 Anpassung des `kvm_intel`-Moduls

Aktuelle Intel-CPUs unterstützen das Tracen virtueller Maschinen, allerdings ist dieses Feature in dem Linux-Kernel derzeit nicht implementiert. Der `guest`-Filter des perf-Systems, der für die Einschränkung auf den Non-Root-Modus vorgesehen ist, hat im Fall von PT keine Funktion [vgl. 20]. Aus diesem Grund ist ein Kernel-Patch notwendig[7].

Damit die CPU ausschließlich die Operationen eines Gastsystems aufzeichnet, muss das `TraceEn`-Bit des `IA32_RTIT_CTL`-MSR bei jedem VM-Entry gesetzt und bei einem VM-Exit wieder gelöscht werden. Dies lässt sich implementieren, indem das Register in die MSR Load Lists eingetragen wird. Die VM-Entry Load List legt fest, welche MSR-Werte die CPU bei dem Übergang von dem Root-Modus in den Non-Root-Modus lädt; analog enthält die VM-Exit Load List die MSR-Werte für den Wechsel zurück in den Root-Modus.

Weil UEFI-Treiber eine relative geringe Komplexität haben und vollständige Traces bei Tests stets unterhalb der 1 MiB-Grenze geblieben sind, wird die ToPA so konfiguriert, dass der Ausgabebereich ein einziger logischer Block ist, der für den jeweiligen Anwendungsfall hinreichend groß zu wählen ist. Auf ein komplexes Layout mit `INT`-Einträgen wird verzichtet.

Als Schnittstelle zwischen dem Kernelmodul und dem Userspace dient ein neues Miscellaneous Device, das mit der `ioctl`-Funktion gesteuert werden kann. Das Device unterstützt folgende Operationen.

- `TRACE_INIT` alloziert Arbeitsspeicher für den PT-Ausgabereich und erzeugt eine ToPA.
- `TRACE_START` setzt ein Flag, sodass PT bei dem nächsten VM-Entry mittels der Load List aktiviert wird.
- `TRACE_STOP` löscht das Flag.
- `TRACE_GET_OUTPUT_SIZE` gibt die Größe des aktuellen Traces zurück.
- `TRACE_SET_ADDR_LIMITS` setzt die minimale und maximale Adresse für den Filter.

Zur Verhinderung von Race Conditions sind die Operationen mit den VM-Entries und VM-Exits synchronisiert. Der Modul-Patch benutzt hierfür eine Completion[8], die vor jedem VM-Entry zurückgesetzt wird und nach dem VM-Exit den Abschluss der Virtualisierungsphase signalisiert.

[7] siehe auch [23]

[8] https://www.kernel.org/doc/Documentation/scheduler/completion.txt

4.4 Virtuelle UEFI-Umgebung

4.4.1 FUZZ-Protokoll

Weil die technischen Details des Datenaustauschs für die einzelnen Anwendungen keine Rolle spielen sollen, wird ein neues UEFI-Protokoll als Abstraktionsschicht verwendet. Das FUZZ-Protokoll definiert eine Schnittstelle zwischen der Applikation und dem Fuzzer, ohne eine spezifische Implementierung vorzugeben. Nachträgliche Optimierungen der Fuzzing-Infrastruktur oder der Umstieg auf einen anderen Kommunikationsmechanismus sind damit jederzeit möglich. Es muss lediglich der Treiber ausgetauscht werden, der das Protokoll implementiert.

Das FUZZ-Protokoll enthält drei Funktionszeiger: `Init`, `StartIteration` und `StopIteration`.

```
typedef
EFI_STATUS
(EFIAPI *FUZZ_INIT) (
  IN  FUZZ_PROTOCOL  *This
  );

typedef
EFI_STATUS
(EFIAPI *FUZZ_START_ITERATION) (
  IN   FUZZ_PROTOCOL  *This,
  OUT  GUID           *Guid,
  OUT  UINT8          **Data,
  OUT  UINT32         *DataLen
  );

typedef
EFI_STATUS
(EFIAPI *FUZZ_STOP_ITERATION) (
  IN  FUZZ_PROTOCOL  *This
  );
```

Aufgabe der `Init`-Funktion ist es, den Kommunikationsendpunkt für den Fuzzer zu erstellen, die UEFI-Umgebung auf das Fuzzing vorzubereiten und die Basisadressen aller Zielimages zu übermitteln, damit die Instruction Pointer in den Traces interpretiert werden können. Die `StartIteration`-Funktion nimmt die generierten Daten des Fuzzers entgegen und leitet sie an die Applikation weiter. Mit `StopIteration` wird das Ende der Iteration signalisiert, sodass der Fuzzer die aufgezeichneten Kontrollflussdaten auswerten kann.

4.4.2 Treiber

Um das FUZZ-Protokoll zu implementieren, wird ein UEFI-Treiber erstellt, der über TCP/IP mit dem Fuzzer kommuniziert.

Das UEFI-Treibermodell[9] schreibt vor, dass Treiber beim Start zunächst nur das `EFI_DRIVER_BINDING`-Protokoll an ihrem Image-Handle registrieren. Dieses Hilfsprotokoll stellt die Funktionen `Supported`, `Start` und `Stop` bereit, mit denen die DXE-Firmware die eigentliche Protokolle des Treibers an den richtigen Handles installieren kann. Die `Supported`-Funktion überprüft, ob ein Handle mit dem Treiber kompatibel ist. Falls dies der Fall ist, ruft die Firmware die `Start`-Funktion auf, die alle Protokollinstanzen erzeugt und initialisiert. Entsprechend soll `Stop` diese Protokolle wieder entfernen.

UEFI definiert das EFI TCPv4-Protokoll für TCP über IPv4[10], zudem enthält EKD II die TcpIo-Bibliothek mit verschiedenen Wrapperfunktionen zur Vereinfachung des Netzwerkcodes.[11] Der Treiber prüft deshalb in der `Supported`-Funktion, ob das übergebene Handle das `EFI_TCP4_SERVICE_BINDING`-Protokoll unterstützt, und installiert mittels `Start` eine Instanz des FUZZ-Protokolls mit einem TcpIo-Member als privatem Datum. `Stop` baut die TCP-Verbindung ab und löscht die Instanz.

Für die Kommunikation werden binäre Type-Length-Value-Pakete (TLV) verwendet.

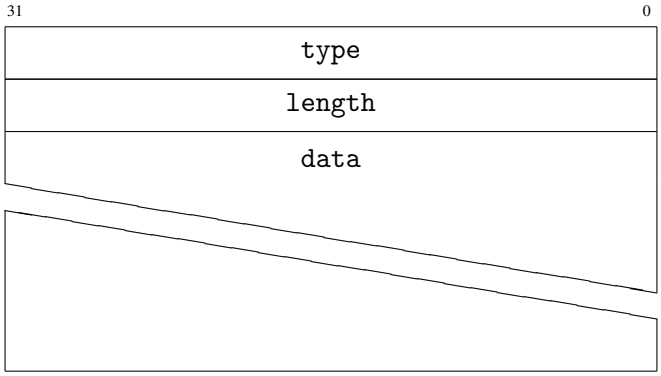

Abbildung 22: Paketformat

[9] vgl. UEFI-Spezifikation S. 367
[10] vgl. UEFI-Spezifikation S. 1385
[11] siehe https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Library/DxeTcpIoLib

Die `Init`-Implementierung des Treibers schaltet zuerst den Watchdog-Timer aus, der zum Schutz vor fehlerhaften Bootloadern nach fünf Minuten einen Reset auslösen würde. Außerdem wird eine Interrupt Service Routine registriert, um CPU-Exceptions während des Fuzzings zu behandeln.[12] Die Funktion erzeugt anschließend mit TcpIo einen TCP-Socket und wartet darauf, dass der Fuzzer eine Verbindung aufbaut. Zu diesem Zeitpunkt wird eine Pseudo-Migration angelegt, damit sich neue Instanzen der UEFI-Umgebung starten lassen, die unmittelbar betriebsbereit sind und nicht mehr den Bootvorgang zu durchlaufen brauchen. Nachdem der Fuzzer ein `INIT_REQUEST`-Paket mit dem Datei-GUID des Zielimages gesendet hat, durchsucht die `Init`-Funktion alle Handles, die das `EFI_LOADED_IMAGE`-Protokoll unterstützen. Konnte das Image gefunden werden, überträgt die Funktion die Basisadresse mit einem `INIT_RESPONSE`-Paket, andernfalls signalisiert ein `INIT_ERROR`-Paket, dass der GUID ungültig ist.

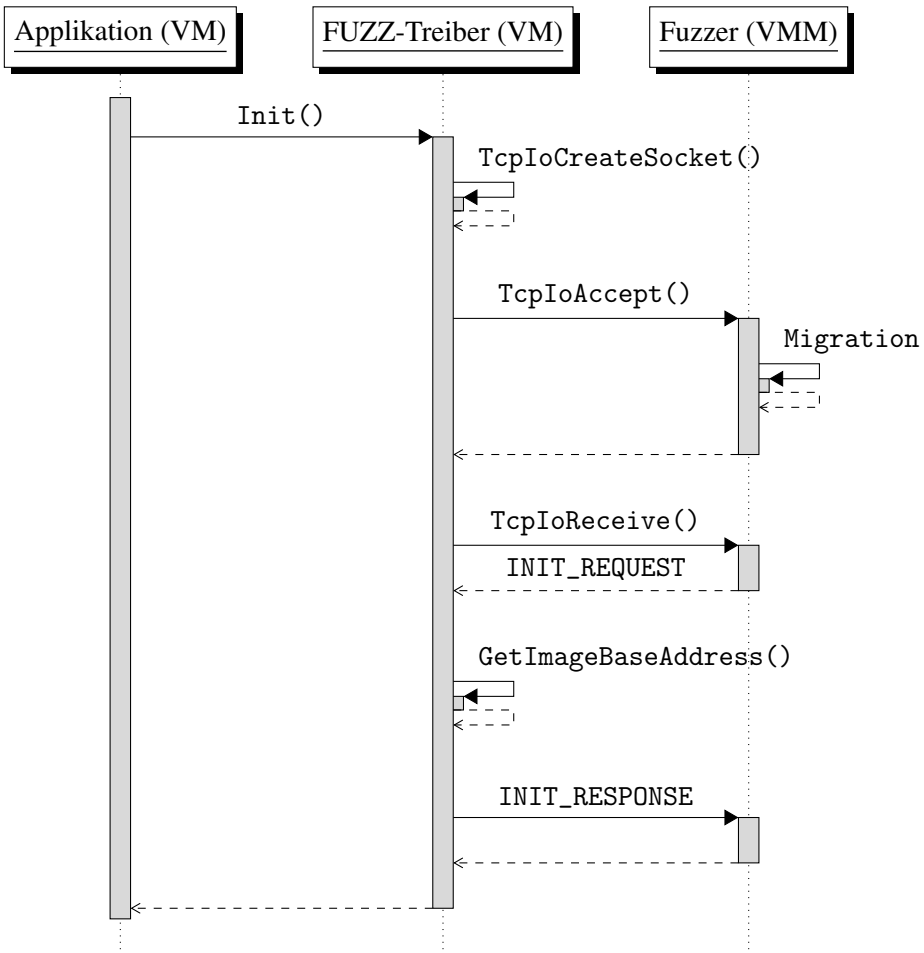

Abbildung 23: Sequenzdiagramm der Initialisierung

[12] siehe das Kapitel Fehlerbehandlung

Die `StartIteration`-Implementierung wartet auf ein `DATA`-Paket, das die Daten des Fuzzers sowie einen GUID zur eindeutigen Kennzeichnung des Testfalls enthält, und reicht diese über die Ausgabeparameter an die Applikation weiter. Die `StopIteration`-Funktion sendet ein `STOP`-Paket an den Fuzzer, der daraufhin das Tracing ausschaltet und die Coverage-Analyse der aufgezeichneten Daten durchführen kann.

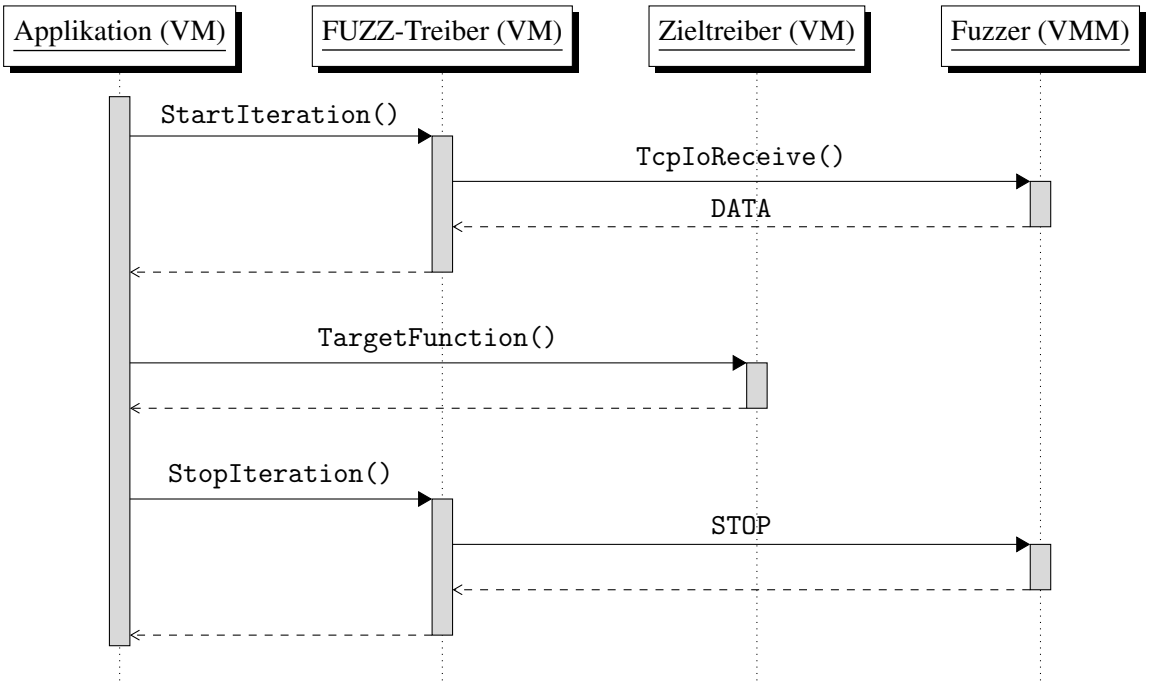

Abbildung 24: Sequenzdiagramm einer Iteration

4.4.3 Applikation

Eine Fuzzing-Anwendung sucht mit dem `LocateHandleBuffer`-Service einen Handle, der das Fuzz-Protokoll unterstützt, und erhält mit `OpenProtocol` die Protokollinstanz.

Nach der Initialisierung kann die Applikation in einer Schleife jeweils mit `StartIteration` die nächsten Testdaten anfordern. Diese werden für den konkreten Anwendungsfall interpretiert und an den zu testenden Code durchgereicht. `StopIteration` beendet die aktuelle Iteration.

Die kompilierten Anwendungen werden über ein virtuelles CD-ROM-Laufwerk eingebunden und entweder als Bootoption eingetragen oder über die UEFI-Shell gestartet.

```
Fuzz->Init (Fuzz);

while (TRUE) {
  Status = Fuzz->StartIteration (Fuzz, &TestGuid, &Data, &DataLen);
  if (EFI_ERROR (Status)) {
    goto Error;
  }

  if (DataLen >= 4
          && Data[0] == 'f'
          && Data[1] == 'u'
          && Data[2] == 'z'
          && Data[3] == 'z') {
    crash();
  }

  FreePool (Data);

  Status = Fuzz->EndIteration (Fuzz);
  if (EFI_ERROR (Status)) {
    goto Error;
  }
}
```

Abbildung 25: Beispielanwendung

4.5 Coverage-Analyse

4.5.1 Kontrollflussgraph

Um die von PT bereitgestellten Kontrollflussinformationen möglichst effizient auszuwerten, bietet es sich an, zunächst den Kontrollflussgraphen (CFG) zu konstruieren. Die CFG-Knoten ergeben sich dabei aus den Basic Blocks, also Sequenzen von Instruktionen, die stets bei der ersten Anweisung beginnen und bei der letzten enden[13]. Mit dem folgenden Schema kann die jeweils erste Instruktion eines Basic Blocks – auch Leader genannt – identifiziert werden [vgl. 28, Kap. 8.4.1, S. 526].

- Die erste Anweisung des Programms ist ein Leader.
- Jedes Sprungziel ist ein Leader.
- Die Instruktion, die unmittelbar auf den Sprung folgt, ist ein Leader.

Ein einzelner Basic Block umfasst den Leader sowie alle Instruktionen bis zu dem nächsten Leader.

Anhand der Beziehungen zwischen den Blöcken lassen sich im nächsten Schritt die CFG-Kanten ermitteln. Endet der Basic Block mit einer bedingten Verzweigung wie `JE`, so führt eine Kante zu dem Sprungziel und eine zweite Kante zu dem nachfolgenden Block. Bei einer direkten Verzweigung wie `JMP` (`E9`) gibt es eine einzige Kante zu dem Sprungziel. Enthält ein Basic Block keine Verzweigung, wird er mit dem nächsten Basic Block verbunden; dies kann als impliziter direkter Sprung aufgefasst werden. Im Fall einer indirekten Verzweigung wie `JMP` (`FF /4`) oder eines Interrupts ist das Sprungziel erst zur Laufzeit bekannt und nicht mit einer Kante darstellbar.

Ein Spezialfall sind `CALL`- und `RET`-Instruktionen. Ist die Return Compression aktiviert, muss bei jeder `CALL`-Anweisung die jeweilige Rücksprungadresse auf einen Stack abgelegt werden. Bei einer `RET`-Instruktion wird die oberste Adresse entfernt und als das nächste Ziel verwendet.[14]

[13] Diese Definition bezieht sich ausschließlich auf die statische Analyse und berücksichtigt keine Kontrollflussänderungen zur Laufzeit.

[14] Es sei denn, ein TIP-Paket gibt eine abweichende Adresse an; siehe Kapitel 2.4.4.1.

Kontrollflussänderung	Instruktionen
bedingter Sprung	JA, JAE, JB, JBE, JC, JCXZ, JECXZ, JRCXZ, JE, JG, JGE, JL, JLE, JNA, JNAE, JNB, JNBE, JNC, JNE, JNG, JNGE, JNL, JNLE, JNO, JNP, JNS, JNZ, JO, JP, JPE, JPO, JS, JZ, LOOP, LOOPE, LOOPNE, LOOPNZ, LOOPZ
direkter Sprung	JMP (E9, EB) oder implizit
indirekter Sprung	JMP (FF /4), INT3, INTn, INTO, IRET, IRETD, IRETQ, JMP (EA, FF /5), CALL (9A, FF /3), RET (CB, CA), SYSCALL, SYSRET, SYSENTER, SYSEXIT, VMLAUNCH, VMRESUME
direkter Aufruf	CALL (E8)
indirekter Aufruf	CALL (FF /2)
Rücksprung	RET (C3, C2)

Abbildung 26: Klassifizierung der Knoten anhand von Kontrollflussinstruktionen, [vgl. 16, Vol. 3C, Kap. 35.2.1, Table 35-1]

Weil eine statische Codeanalyse den Kontrollfluss nicht vollständig erfassen kann, muss es während der Auswertung eines Traces die Möglichkeit geben, an eine beliebige Adresse zu springen. Dies lässt sich implementieren, indem die Knoten des CFG zusätzlich zu einem balancierten binären Suchbaum (BST) verbunden werden, wobei die Leader-Adressen die Schlüssel sind. Eine geeignete BST-Variante ist der AVL-Baum, der sich nach jeder Einfügeoperation so balanciert, dass die Höhendifferenz zwischen je zwei Kindbäumen höchstens 1 ist. Aufgrund dieser Eigenschaft hat die Suche selbst im schlechtesten Fall logarithmische Komplexität.

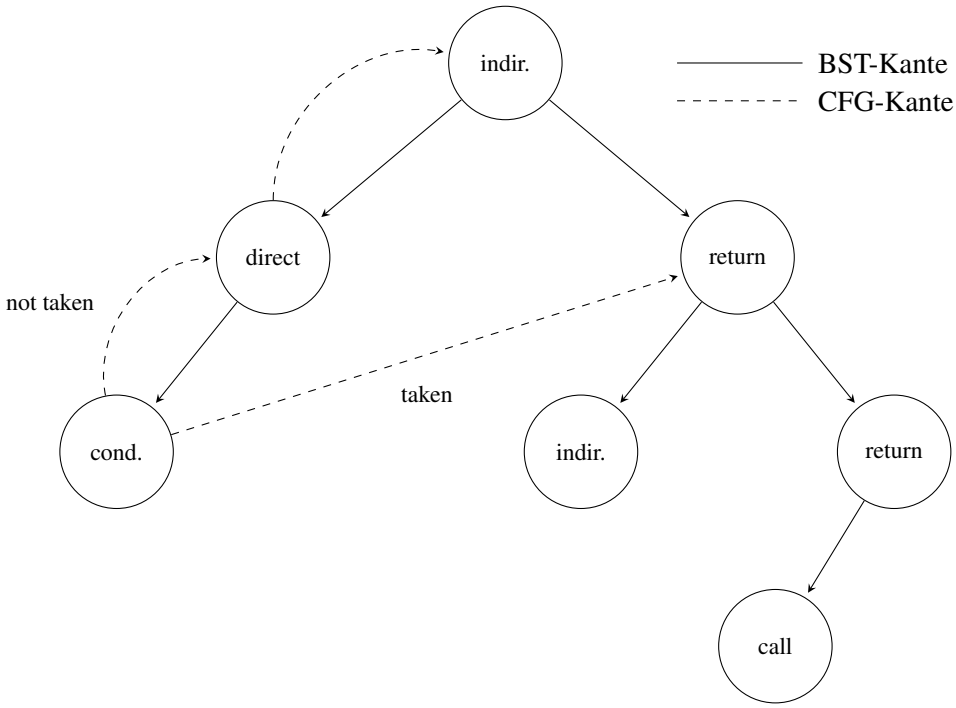

Abbildung 27: Kombination aus Kontrollflussgraph und Suchbaum

Um aus einer PE32+-Datei den zugehörigen Kontrollflussgraphen zu konstruieren, wird zunächst die .text-Sektion mit dem eigentlichen Programmcode extrahiert. In dem Sektionsheader ist außerdem die Relative Virtual Address (RVA) angegeben, die den Offset zur Basisadresse des geladenen Images festlegt. Weil die Basisadresse bereits in der Initialisierungsphase ermittelt wurde, lässt sich die physische Adresse jeder Instruktion berechnen.

$$instruction\_address = image\_base\_address + text\_rva + instruction\_offset$$

Anschließend wird der Maschinencode disassembliert. Eine erste Schleife durchläuft die Instruktionen, fügt die Basic Blocks in den Suchbaum ein und sammelt alle Sprunganweisungen. Anhand der Sprünge lassen sich dann die Knoten klassifizieren und die CFG-Kanten eintragen.

4.5.2 Verarbeitung der Pakete

Bei der Kontrollflussanalyse sind lediglich TIP-, TNT- und FUP-Pakete relevant. Informationen wie der CR3-Wert brauchen nicht berücksichtigt zu werden, weil UEFI das Flat-Memory-Modell verwendet und direkt mit physischen Adressen operiert.

Als Metrik für die Coverage dient die Anzahl der durchlaufenen Basic Blocks. Um dies möglichst effizient umzusetzen, speichert honggfuzz die jeweiligen Leader-Adressen mit einem Bloomfilter aus einem 16 MiB großen Bitfeld und der folgenden Hashfunktion.

$$h(x) = 1 \ll (x \,\&\, 0\text{x}7\text{FFFFFF})$$

TNT-Pakete werden bitweise verarbeitet. In jeder Iteration durchläuft das Programm zunächst alle CFG-Knoten, deren ausgehende Kante einen direkten Sprung repräsentiert. Endet der nächste Basic Block mit einem bedingten Sprung, so wird je nach Wert des TNT-Bits entweder die Kante zu dem Sprungziel oder die zu dem Basic Block mit der nächsthöheren Leader-Adresse verfolgt. Andernfalls müssen die verbleibenden Bits zur späteren Verarbeitung zwischengespeichert werden. Aus Effizienzgründen kann die CPU bei der Reihenfolge der Pakete von dem realen Kontrollfluss abweichen und mehrere Sprünge, die eigentlich nicht aufeinanderfolgen, in einem einzigen TNT-Paket zusammenfassen.

Kombinationen aus FUP- und TIP-Paketen beschreiben asynchrone Ereignisse wie Interrupts. Weil diese für die Coverage-Analyse keine Rolle spielen, werden die Pakete ignoriert. Ein einzelnes TIP-Paket gibt dagegen das Ziel eines indirekten Sprungs oder Funktionsaufrufs an. Auch hier müssen zuerst alle direkten Sprünge verfolgt werden. Anschließend ermittelt das Programm mit einer Binärsuche den Basic Block, in dem das Sprungziel liegt, und zeichnet die Zieladresse auf.

4.6 Fehlererkennung

Standardmäßig beobachtet honggfuzz den zu fuzzenden Prozess mit dem `ptrace`-Systemcall und erkennt Fehler anhand von Signalen wie `SIGSEGV`, die das Betriebssystem an den Prozess sendet. Dieser Mechanismus ist bei der virtualisierten UEFI-Umgebung nicht anwendbar, weil diese aus Sicht des Gastes nicht im Kontext eines Betriebssystem läuft, und sich Fehler innerhalb der VM auch nicht unmittelbar auf den QEMU-Prozess auswirken.

Im einfachsten Fall führen fehlerhaft implementierte UEFI-Komponenten zu CPU-Exceptions. Ist das entsprechende Bit in der Exception-Bitmaske der VMCS gesetzt, so wird ein VM-Exit ausgelöst, und das KVM-Modul kann den Fehler behandeln. Alternativ kann dies die VM selbst übernehmen und über das pvpanic-Device den QEMU-Prozess informieren. Um einen weitere Anpassung des Kernels zu vermeiden, wird der zweite Ansatz gewählt.

Der Code der Referenzimplementierung enthält außerdem zahlreiche Assertions, die Programmierfehler zur Laufzeit aufdecken können. Assertions werden typischerweise nur während der Entwicklung genutzt und sind im Release-Modus inaktiv, allerdings lassen sie sich durch Anpassung der `PcdDebugPropertyMask`-Bitmaske bei der Kompilierung aktivieren. Hierzu muss das Bit `DEBUG_PROPERTY_DEBUG_ASSERT_ENABLED` gesetzt sein. Die Assertion kann wiederum einen Breakpoint-Interrupt (`INT 3`) auslösen, der beim Fuzzing als Fehlerindikator dient. Das entsprechende Bit in der Maske ist `DEBUG_PROPERTY_ASSERT_BREAKPOINT_ENABLED`.

Um die Fehlererkennung zu verbessern, können auch zusätzliche Assertions in den Treibercode oder die Applikation eingefügt werden.

4.6.1 Interrupt-Service-Routine

Um die Fehler an den Host weiterzureichen, wird mit der `RegisterInterruptHandler`-Funktion des `CPU`-Protokolls eine neue Interrupt-Service-Routine (ISR) registriert, die folgende Exceptions behandelt.

Interruptvektornummer	Bedeutung
0	Divisionsfehler
3	Breakpoint-Interrupt, ausgelöst durch Assertion
6	ungültiger Opcode
8	Double Fault
13	General Protection Fault

Die ISR setzt bei dem `pvpanic`-Device an I/O-Port 0x505 das Bit 0, sodass der Host einen `GUEST_PANICKED`-Event erhält. Zudem sendet sie die Fehlerinformationen an die QEMU-Debugkonsole. Der Host loggt diese Daten für die spätere Fehleranalyse.

4.7 Controller-Prozess

Aufgabe des Controllers ist es, die Fuzzing-Infrastruktur zu starten, die UEFI-Umgebung zu überwachen und im Fall einen Fehlers die VM in den Ursprungszustand zu versetzen, damit das Fuzzing fortgesetzt werden kann.

Der Controller lädt zuerst die Pseudo-Migration, die in der Initialisierungsphase des FUZZ-Protokolls angelegt worden ist, und verbindet sich über ein Socketpair mit der Standardeingabe und -ausgabe der QEMU-VM. Für die Kommunikation wird QMP verwendet. Sobald die VM betriebsbereit ist, startet der Controller den Fuzzer und wartet in einer Ereignisschleife auf QMP-Nachrichten. Bei einem `GUEST_PANICKED`-Event wird die aktuelle VM mit dem `quit`-Befehlt kontrolliert heruntergefahren. Danach kopiert der Controller die ursprüngliche `OVMF_VARS.fd`-Datei, um eventuelle Änderungen der UEFI-Variablen rückgängig zu machen, und erzeugt eine neue VM auf Basis der Pseudo-Migration. Der Fuzzer kann sich erneut verbinden und das nächste Testdatum senden.

5 Fuzzing der Variablenservices

Die Runtime-Services und dabei insbesondere der `SetVariable`-Service sollten vorrangig untersucht werden, weil sie direkt aus dem Kernel des Betriebssystems und teilweise auch indirekt aus dem Userspace aufrufbar sind. Der `SetVariable`-Service ist zudem komplex und hatte in der Vergangenheit bereits mehrere Schwachstellen, unter andere arithmetische Überläufe [vgl. 2].

5.1 Speicherformat

Variablen mit dem `NON_VOLATILE`-Attribut sind in einem Firmware-Volume des NVRAM gespeichert, temporäre Variablen dagegen in einem Variablen-Store im RAM. Der Store besteht aus einem Header, der Liste der Variablen sowie einem Pufferspeicher, der die größtmögliche Variable aufnehmen kann. Die Größe ist dabei plattformabhängig und wird mit dem `PcdMaxVariableSize`-Konfigurationseintrag definiert.

Jeder Variableneintrag hat einen Header, in dem Daten wie die Attribute und der Vendor-GUID hinterlegt sind. Danach folgen der Name und der Wert, wobei die Größe jeweils variabel ist.

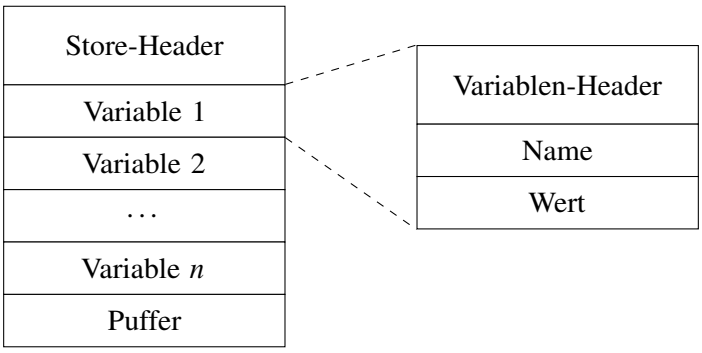

Abbildung 28: Variablen-Store, [vgl. 29, S. 19, Abb. 12]

Sowohl beim Erstellen als auch beim Ändern von Variablen wird zunächst eine neue Variablenstruktur im Pufferspeicher angelegt. Ist die Variable temporär, so wird die Struktur anschließend an die Liste angehängt. Persistente Variablen werden dagegen in mehreren Schritten in den NVRAM kopiert. Dies ist notwendig, um die Atomizität des Vorgangs sicherzustellen und zu verhindern, dass eine Unterbrechung zu einer unvollständigen Variablen führt.

5.2 Fuzzing von `SetVariable`

Die UEFI-Spezifikation definiert den folgenden Prototypen für den `SetVariable`-Service.

```
EFI_STATUS
SetVariable (
  IN CHAR16   *VariableName,
  IN EFI_GUID *VendorGuid,
  IN UINT32   Attributes,
  IN UINTN    DataSize,
  IN VOID     *Data
  );
```

Weil der Service die Variablen nicht inhaltlich verarbeitet, erscheint es sinnvoll, das Fuzzing auf die Attribute und den `DataSize`-Parameter zu beschränken. Der Name, der GUID und der Wert werden beliebig festgelegt.

Je nachdem, mit welchen Optionen OVMF kompiliert und gestartet wird, wählt die Firmware einen von drei möglichen Treibern aus.

- Unterstützen sowohl die CPU als auch OVMF den SMM, wird das VariableSmm-Modul verwendet, das die Variablen vor direktem Zugriff schützt.
- Bei fehlender SMM-Unterstützung lädt die Firmware das VariableRuntimeDxe-Modul.
- Steht kein Speicher für persistente Variablen zur Verfügung, nutzt OVMF ersatzweise das EmuVariableRuntimeDxe-Modul, bei dem Variablen ausschließlich im RAM abgelegt werden.

Für einen ersten Test wird eine bekannte Integer-Overflow-Schwachstelle aus einer früheren EDK II-Version [vgl. 2, Kap. 9, S. 13] in die `SetVariable`-Implementierung des VariableRuntimeDxe-Moduls eingefügt. Der Fehler betrifft die Größenvalidierung der Argumente und kann dazu führen, dass der Variablenwert über die Grenzen des Variablen-Store hinaus geschrieben wird.

```
-  if ((UINTN)(~0) - DataSize < StrSize(VariableName)){
-    //
-    // Prevent whole variable size overflow
-    //
-    return EFI_INVALID_PARAMETER;
-  }

-    if (StrSize (VariableName) + DataSize >
-        PcdGet32 (PcdMaxVariableSize) - sizeof (VARIABLE_HEADER)) {
+    if ((DataSize > PcdGet32 (PcdMaxVariableSize)) ||
+        (sizeof (VARIABLE_HEADER) + StrSize (VariableName) + DataSize >
+        PcdGet32 (PcdMaxVariableSize))) {
       return EFI_INVALID_PARAMETER;
     }
```

Abbildung 29: Integer-Overflow, Quelle: [30]

Der initiale Testkorpus besteht aus gültigen Argumenten für den `SetVariable`-Service, die von honggfuzz abhängig von der Coverage gezielt manipuliert werden. Nach 10 Minuten findet der Fuzzer eine Kombination, die den Integer-Overflow auslöst und zu einer fehlgeschlagenen Assertion bei dem Kopieren des Variablenwerts in den Pufferspeicher führt.

```
ASSERT edk2/MdePkg/Library/BaseMemoryLibRepStr/CopyMemWrapper.c(56):
       (Length - 1) <= (0xFFFFFFFFFFFFFFFFULL - (UINTN)DestinationBuffer)
```

6 Auswertung

Weil die Infrastruktur nicht noch optimiert ist und für eine umfassende Analyse zunächst weitere Applikationen entwickelt werden müssen, kann zum jetzigen Zeitpunkt nur eine vorläufige Bewertung des Ansatzes vorgenommen werden. Es ist damit zu rechnen, dass sich die Performance noch steigern lässt.

6.1 Performance

Als Benchmark wurden 100.000 Iterationen des Variablentests ausgeführt. Hierbei konnten im Durchschnitt 148 Tests pro Sekunde erreicht werden. Die verwendete CPU war ein Intel Core i3-7100.

6.2 Grenzen

Derzeit kann der Fuzzer nur Funktionen testen, die in der TSL-Phase zur Verfügung stehen, also Runtime-Services, Boot-Services und DXE-Treiber. Die DXE-Phase selbst oder die PEI-Phase sind nicht zugänglich. Dies ist jedoch keine prinzipielle Einschränkung des Ansatzes, sondern gilt lediglich für die aktuelle Implementierung. Mit geeigneten Anpassungen der Infrastruktur kann das Fuzzing auf weitere Phasen angewendet werden [15].

[15] siehe Kapitel 8.3

7 Fazit

In dieser Arbeit konnte gezeigt werden, dass die automatisierte Schwachstellensuche mittels Fuzzing auch bei UEFI-Umgebungen möglich ist. Der vorgestellte Ansatz erlaubt es, einen großen Teil der DXE-Treiber und Services systematisch zu testen. Da sich Graybox-Fuzzing bereits in verschiedenen Anwendungsbereichen bewährt hat, ist diese Methode auch bei UEFI-Firmware vielversprechend und kann zu einer effizienteren Sicherheitsanalyse beitragen. Die Ergebnisse sind dabei nicht auf ein spezifisches Produkt oder einen Hersteller beschränkt, sondern gelten potenziell für alle Implementierungen, die von dem Referenzcode abgeleitet sind.

Um die Funktionsfähigkeit zu demonstrieren, wurde eine Testapplikation entwickelt, die einen bekannten Fehler in dem `SetVariable`-Service innerhalb weniger Minuten findet. Hierbei waren weder eine genaue Analyse der Implementierung noch eine spezielle Testmethode notwendig. Es genügte, geeignete Parameter auszuwählen und einen initialen Korpus aus gültigen Daten zu erzeugen. Dies unterstreicht die Flexibilität des Fuzzing-Ansatzes.

Die Modularität der Infrastruktur ermöglicht zudem nachträgliche Optimierungen und Erweiterungen. Alle Komponenten – der Fuzzer, das FUZZ-Protokoll, der Treiber sowie die Applikationen – sind weitgehend unabhängig voneinander und lassen sich gegebenenfalls austauschen.

Fuzzing sollte jedoch nicht als Ersatz für manuelle Audits oder die statische Code-Analyse betrachtet werden. Zum einen deckt der gewählte Ansatz nicht den gesamten UEFI-Code ab, zum anderen hat die Methode des Fuzzings grundsätzliche Grenzen. Da ein Fuzzer auf eindeutige Fehlersymptome angewiesen ist, lassen sich subtile Fehler, die lediglich zu einem anderen, formal gültigen Verhalten führen, kaum finden. Die sinnvollste Herangehensweise wäre deshalb, verschiedene Methoden zu kombinieren.

8 Ausblick

8.1 Fuzzing-Applikationen

Neben den Variablenservices sollten weitere kritische UEFI-Komponenten untersucht werden. Dies können etwa die besonders exponierten Netzwerktreiber oder die Treiber der kryptografischen Protokolle sein. Hierfür müssen lediglich neue Applikationen implementiert werden.

8.2 Performance

Die Fuzzing-Infrastruktur bietet zu diesem Zeitpunkt noch keine optimale Performance, sodass die Schwachstellensuche viel Zeit benötigt. Eine mögliche Optimierung ist der Austausch des TCP-basierten Protokolls durch einen effizienteren Kommunikationsmechanismus. So setzt etwa kAFL[23] Hypercalls ein, mit denen die VM bestimmte Funktionen des VMM direkt aufrufen kann, um Daten anzufordern oder das Tracing zu steuern. Die Verzögerung, die sich aus dem Senden und Empfangen der Pakete ergibt, würde hierdurch wegfallen.

8.3 Phasen

Um möglichst viele UEFI-Komponenten abzudecken, ist es wünschenswert, den Fuzzer auch in der DXE- oder sogar PEI-Phase verwenden zu können. Mit Hypercalls ist dies prinzipiell möglich, weil für den Datenaustausch keine – virtuellen – Geräte verwendet werden, sondern ausschließlich CPU-Register und der Arbeitsspeicher. Ein DXE-Treiber beziehungsweise PEIM könnte also analog zu der UEFI-Applikation die Testdaten des Fuzzers anfordern und an eine andere Komponente der Phase senden. Auf diese Weise ließe sich etwa der Firmware-Update-Mechanismus testen.

Literatur

[1] C. Kallenberg und X. Kovah. (2015). How Many Million BIOSes Would you Like to Infect?, Adresse: http://legbacore.com/Research_files/HowManyMillionBIOSesWouldYouLikeToInfect_Whitepaper_v1.pdf (besucht am 10. 08. 2017).

[2] (2017). Security Advisory for the UEFI Open Source Community, Adresse: https://www.gitbook.com/download/pdf/book/edk2-docs/security-advisory (besucht am 10. 08. 2017).

[3] (2015). Vulnerability Note VU#552286: UEFI EDK2 Capsule Update vulnerabilities, Adresse: https://www.kb.cert.org/vuls/id/552286 (besucht am 10. 08. 2017).

[4] O. Bazhaniuk, J. Loucaides, L. Rosenbaum, M. R. Tuttle und V. Zimmer. (2015). Symbolic execution for BIOS security, Adresse: https://www.usenix.org/system/files/conference/woot15/woot15-paper-bazhaniuk.pdf (besucht am 10. 08. 2017).

[5] C. Kallenberg und X. Kovah. (2015). BIOS Necromancy: Utilizing "Dead Code" for BIOS Attacks, Adresse: http://www.legbacore.com/Research_files/BIOSNecromancy.pdf (besucht am 10. 08. 2017).

[6] R. Bowles, Hrsg., *Intel Technology Journal*, Bd. 15, Intel Corporation, 2011.

[7] (2015). Unified Extensible Firmware Interface (UEFI) Framework UEFI Overview, Adresse: https://software.intel.com/sites/default/files/m/d/4/1/d/8/Unified_2BExtensible_2BFirmware_2BInterface_2B_28UEFI_29_2BFramework_2BOverview.pdf (besucht am 10. 08. 2017).

[8] (2017). Platform Initialization (PI) Specification, Version 1.6, Adresse: http://www.uefi.org/sites/default/files/resources/PI_Spec_1_6.pdf (besucht am 10. 08. 2017).

[9] V. Zimmer, M. Rothman und S. Marisetty, *Beyond BIOS: Developing with the Unified Extensible Firmware Interface*. Walter de Gruyter Inc., 2017, ISBN: 978-1-5015-1478-4.

[10] (2017). QEMU Machine Protocol, Adresse: https://wiki.qemu.org/Documentation/QMP (besucht am 10. 08. 2017).

[11] (2017). PVPANIC DEVICE, Adresse: https://github.com/qemu/qemu/blob/master/docs/specs/pvpanic.txt (besucht am 10. 08. 2017).

[12] (2017). Migration, Adresse: https://www.linux-kvm.org/page/Migration (besucht am 10. 08. 2017).

[13] (2017). SeaBIOS, Adresse: https://www.seabios.org/SeaBIOS (besucht am 10. 08. 2017).

[14] (2017). README, Adresse: https://github.com/tianocore/edk2/blob/master/OvmfPkg/README (besucht am 10. 08. 2017).

[15] (2017). american fuzzy lop – README, Adresse: http://lcamtuf.coredump.cx/afl/README.txt (besucht am 10. 08. 2017).

[16] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, 325462-063US. 2017.

[17] M. L. Soffa, K. R. Walcott und J. Mars. (2017). Exploiting Hardware Advances for Software Testing and Debugging, Adresse: http://web.eecs.umich.edu/~profmars/wp-content/papercite-data/pdf/soffa11icse.pdf (besucht am 10. 08. 2017).

[18] A. Kleen und B. Strong. (2017). Intel® Processor Trace on Linux, Adresse: https://www.halobates.de/pt-tracing-summit15.pdf (besucht am 10. 08. 2017).
[19] (2017). Recording Inferior's Execution and Replaying It, Adresse: https://sourceware.org/gdb/onlinedocs/gdb/Process-Record-and-Replay.html (besucht am 10. 08. 2017).
[20] (2017). Intel Processor Trace, Adresse: https://github.com/torvalds/linux/blob/master/tools/perf/Documentation/intel-pt.txt (besucht am 10. 08. 2017).
[21] A. Allievi und R. Johnson. (2017). Harnessing Intel Processor Trace on Windows for Vulnerability Discovery, Adresse: https://recon.cx/2017/brussels/resources/slides/RECON-BRX-2017-Harnessing_Intel_Processor_Trace_on_Windows_for_Vulnerability_Discovery.pps (besucht am 10. 08. 2017).
[22] (2017). Feedback-driven fuzzing, Adresse: https://github.com/google/honggfuzz/blob/master/docs/FeedbackDrivenFuzzing.md (besucht am 10. 08. 2017).
[23] S. Schumilo. (). kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels, Adresse: https://www.usenix.org/system/files/conference/usenixsecurity17/sec17-schumilo.pdf (besucht am 10. 08. 2017).
[24] C. Kallenberg u. a. (2017). Extreme Privilege Escalation On Windows 8/UEFI Systems, Adresse: https://www.blackhat.com/docs/us-14/materials/us-14-Kallenberg-Extreme-Privilege-Escalation-On-Windows8-UEFI-Systems-WP.pdf (besucht am 10. 08. 2017).
[25] (2017). efivarfs – a (U)EFI variable filesystem, Adresse: https://www.kernel.org/doc/Documentation/filesystems/efivarfs.txt (besucht am 10. 08. 2017).
[26] Microsoft Corporation. (2017). GetFirmwareEnvironmentVariable function, Adresse: https://msdn.microsoft.com/en-us/library/windows/desktop/ms724325(v=vs.85).aspx (besucht am 10. 08. 2017).
[27] ——, (2017). SetFirmwareEnvironmentVariable function, Adresse: https://msdn.microsoft.com/en-us/library/windows/desktop/ms724934(v=vs.85).aspx (besucht am 10. 08. 2017).
[28] A. V. Aho, M. S. Lam, R. Sethi und J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Pearson Education, Inc., 2007, ISBN: 0-321-48681-1.
[29] J. Yao und V. Zimmer. (). A Tour Beyond BIOS – Implementing UEFI Authenticated Variables in SMM with EDKII, Adresse: https://firmware.intel.com/sites/default/files/resources/A_Tour_Beyond_BIOS_Implementing_UEFI_Authenticated_Variables_in_SMM_with_EDKII.pdf (besucht am 10. 08. 2017).
[30] (2015). Fix potential overflow for SetVariable interface, Adresse: https://sourceforge.net/p/edk2/code/14305 (besucht am 10. 08. 2017).

Eidesstattliche Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen und ist noch nicht veröffentlicht worden. Ich bin mir bewusst, dass eine unwahre Erklärung rechtliche Folgen haben wird.

Münster, 4. Oktober 2017, _____

Jan Ewald